Second International Workshop on Variability Modelling of Software-intensive Systems

Heymans, Patrick; Kang, Kyo-Chul; Metzger, Andreas; Pohl, Klaus (Eds.)

In: ICB Research Reports - Forschungsberichte des ICB / 2008

This text is provided by DuEPublico, the central repository of the University Duisburg-Essen.

This version of the e-publication may differ from a potential published print or online version.

DOI: https://doi.org/10.17185/duepublico/47116

URN: urn:nbn:de:hbz:464-20180920-085059-2

Link: https://duepublico.uni-duisburg-essen.de/servlets/DocumentServlet?id=47116

License:
As long as not stated otherwise within the content, all rights are reserved by the authors / publishers of the work. Usage only with permission, except applicable rules of german copyright law.

Source: ICB-Research Report No. 22, Januar 2008

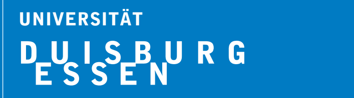
Patrick Heymans
Kyo-Chul Kang
Andreas Metzger
Klaus Pohl (Eds.)

Second International Workshop on Variability Modelling of Software-intensive Systems

22

ICB-RESEARCH REPORT

Proceedings



ICB-Research Report No.22
Januar 2008

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

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

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

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

Proceedings

Edited by:

Patrick Heymans

University of Namur, Belgium

phe@info.fundp.ac.be

Kyo-Chul Kang

Pohang University of Science and Technology, Korea

kck@postech.ac.kr

Andreas Metzger
Klaus Pohl

University Duisburg-Essen, Germany

andreas.metzger@sse.uni-due.de
klaus.pohl@sse.uni-due.de

ICB Research Reports

Edited by:

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

Managing Assistant and Contact:

Jonas Sprenger

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

icb@uni-duisburg-essen.de

ISSN 1860-2770

Abstract

This ICB Research Report constitutes the proceedings of the Second International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'08), which was held from January 16–18, 2008 at the University of Duisburg-Essen.

Table of Contents

1 MESSAGE FROM THE ORGANIZERS 1
2 ORGANIZATION 2
3 WORKSHOP FORMAT 3
4 TECHNICAL PAPERS 5

1 Message from the Organizers

Welcome to VaMoS'08 – the Second International Workshop on Variability Modelling of Software-intensive Systems!

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.

To facilitate interactions, VaMoS'08 will adopt the organization structure of the first VaMoS workshop, which was held in Limerick, Ireland in 2007. 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'08 has attracted 23 submissions from 10 countries. Each submission was reviewed by at least three members of the programme committee. Based on the reviews, 17 submissions have been accepted.

The accepted papers address a wide range of topics relevant to variability modelling and management. In detail, the following topics are covered:

- Product derivation and configuration
- Variability modelling for automotive systems and service-based systems
- Aspects
- Requirements variability and elicitation
- Quality
- Handling complexity
- Variability in Behaviour
- Dynamic Variability

We like to extend our gratitude to all the people who spent time and energy to make VaMoS a success. VaMoS'08 would not have been possible without their efforts and expertise. We like to 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 Maike Uhlig, André Heuer, Andreas Classen and Arnaud Hubaux. Finally, we thank the sponsors of VaMoS: The University of Duisburg-Essen and the University of Namur.

Enjoy VaMoS 2008 and your stay in Essen, Germany!

The VaMoS organizers

Patrick Heymans	Kyo-Chul Kang	Andreas Metzger	Klaus Pohl

2 Organization

Organizing Committee

Patrick Heymans, University of Namur, Belgium

Kyo-Chul Kang, Pohang University of Science and Technology, Korea

Andreas Metzger, University of Duisburg-Essen, Germany

Klaus Pohl, University of Duisburg-Essen, Germany & Lero, Limerick, Ireland

Programme Committee

David Benavides, University of Seville, Spain

Jürgen Börstler, Umeå University, Sweden

Pascal Costanza, Free University of Brussels, Belgium

Krzysztof Czarnecki, University of Waterloo, Canada

Ulrich Eisenecker, University of Leipzig, Germany

Hasan Gomaa, George Mason University, USA

Paul Grünbacher, Johannes Kepler Universität Linz, Austria

Jilles van Gurp, Nokia Research, Finland

Øystein Haugen, University of Oslo & SINTEF, Norway

André van der Hoek, University of California, Irvine, USA

Jean-Marc Jezequel, IRISA, France

Tomoji Kishi, Japan Advanced Institute of Science and Technology

Charles Krueger, BigLever Software, USA

Frank van der Linden, Philips, The Netherlands

Roberto Lopez-Herrejon, University of Oxford, UK

Tomi Männistö, Helsinki University of Technology, Finland

Kim Mens, Catholic University of Louvain, Belgium

Dirk Muthig, Fraunhofer IESE, Germany

John Mylopoulos, University of Toronto, Canada

Linda Northrop, SEI, USA

Camille Salinesi, University of Paris 1-Sorbonne, France

Pierre-Yves Schobbens, University of Namur, Belgium

Vijay Sugumaran, Oakland University, USA

Steffen Thiel, Lero, Limerick, Ireland

Matthias Weber, Carmeq GmbH, Germany

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 Papers

Interactive Visualisation to Support Product Configuration in Software Product Lines
Ciarán Cawley, Daren Nestor, André Preußner, Goetz Botterweck, Steffen Thiel 7

Towards an Automatic PL Requirements Configuration through Constraints Reasoning
Olfa Djebbi, Camille Salinesi 17

Integrated Product Line Model for Semi-Automated Product Derivation Using Non-Functional Properties
Norbert Siegmund, Martin Kuhlemann, Marko Rosenmüller, Christian Kaestner, and Gunter Saake 25

Increasing Reliability of Model-driven Software Family Engineering and Product Configuration
Frank Grimm 33

Dealing with Changes in Service-Oriented Computing Through Integrated Goal and Variability Modelling
Roger Clotet, Deepak Dhungana, Xavier Franch, Paul Grünbacher, Lidia López, Jordi Marco, Norbert Seyff 43

Weaving Aspect Configurations for Managing System Variability
Brice Morin, Olivier Barais, Jean-Marc Jézéquel 53

Model-Based Implementation of Meta-Variability Constructs: A Case Study using Aspects
Klaus Schmid, Holger Eichelberger 63

Value-Based Elicitation of Product Line Variability: An Experience Report
Rick Rabiser, Deepak Dhungana, Paul Grünbacher, Benedikt Burgstaller 73

Tracing from Features to Use Cases: A Model-Driven Approach
Edward Alférez Salinas, Uirá Kulesza, Ana Moreira, João Araújo, Vasco Amaral 81

Svamp – An Integrated Approach for Modeling Functional and Quality Variability
Mikko Raatikainen, Eila Niemelä, Varvana Myllärniemi, Tomi Männistö 89

How complex is my Product Line? The case for Variation Point Metrics
Roberto Lopez-Herrejon, Salvador Trujillo 97

A Multiple Views Model for Variability Management in Software Product Lines
Rabih Bashroush, Ivor Spence, Peter Kilpatrick, Charles Gillan, Thomas Brown 101

Understanding Decision Models – Visualization and Complexity Reduction of Software Variability
Thomas Forster, Dirk Muthig, Daniel Pech, ... 111

Variability Management on Behavioral Models
Patrick Tessier, David Servat, Sébastien Gérard 121

Statecharts and Variabilities
Nora Szasz, Pedro Vilanova 131

Reflective Component-based Technologies to Support Dynamic Variability
Nelly Bencomo, Gordon Blair, Carlos Flores, Pete Sawyer 141

Towards Visualisation and Analysis of Runtime Variability in Execution Time of BDD Systems
Ildefonso Montero, Joaquín Peña, Antonio Ruiz-Cortés 151

Interactive Visualisation to Support Product Configuration in Software Product Lines

Ciarán Cawley[1], Daren Nestor[1], André Preußner[2], Goetz Botterweck[1], Steffen Thiel[1]

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

[2]BTU Cottbus
Institute of Computer Science
Cottbus, Germany
apreussn@informatik.tu-cottbus.de

Abstract

Software Product Line engineering allows companies to realise significant improvements in time-to-market, cost, productivity, and system quality. One major difficulty with software product lines is that within industry there may exist thousands of variation points in a single product line. This scale of variability can become extremely complex to manage resulting in a product configuration process that bears significant costs. This paper presents a feature configuration meta-model and introduces a prototype tool that employs visualisation and interaction techniques to provide feature configuration functionality.

1. Introduction

Software Product Line (SPL) engineering is a paradigm to develop software applications using platforms and mass customisation. This is achieved through the identification and control of the applications' commonality and variation. Developing using a product line allows companies to build a variety of systems with a minimum of technical diversity and to realise significant improvements in time-to-market, cost, productivity and quality [1]. The management of such a product line's variability is fundamentally key to its success. Particularly in the area of feature modelling and product configuration, variability management can greatly impact the complexity that is involved when producing a new product from existing product line assets [2].

Within industry, product lines exist with thousands of variation points and configuration parameters that need to be managed in order to customise a product [3]. Managing this level of variability is extremely complex and can be very costly [4]. Furthermore, in cases such as these where there are a large number of variants, appropriate techniques are required to allow particular stakeholders to perform their specific tasks [5].

One technique that can be applied beneficially in this context is visualisation. Visualisation takes abstract data and transforms it into a format that is useful for presentation to humans. In doing this, human cognition is enhanced and understanding is afforded. In the area of software product line variability management, visualisation can be used to amplify cognition of the large, complex data sets that can exist in industrial SPL engineering.

This paper presents a meta-model and a prototype tool for feature configuration. The tool implements various visualisation and interaction techniques that can support stakeholders in the process of product configuration for software product lines.

The remainder of this paper is organised as follows: in Section 2 we summarize a meta-model for feature configuration in software product lines; in Section 3 we introduce our visual prototype tool (VISIT-FC) which is based on the meta-model and discuss the tool's architecture design; in Section 4 we explain some of the visualisation techniques implemented in VISIT-FC and how they help to address the challenges of high variability in large feature models. Section 5 provides an illustrating example of a feature configuration using VISIT-FC. Section 6 discusses related work in visual feature configuration and Section 7 outlines future work. Finally, Section 8 concludes the paper.

2. Feature Modelling

A key aspect of the Software Product Line engineering approach is the modelling of the variability of the supported product line features. Such a feature model can support the derivation of a product allowing the

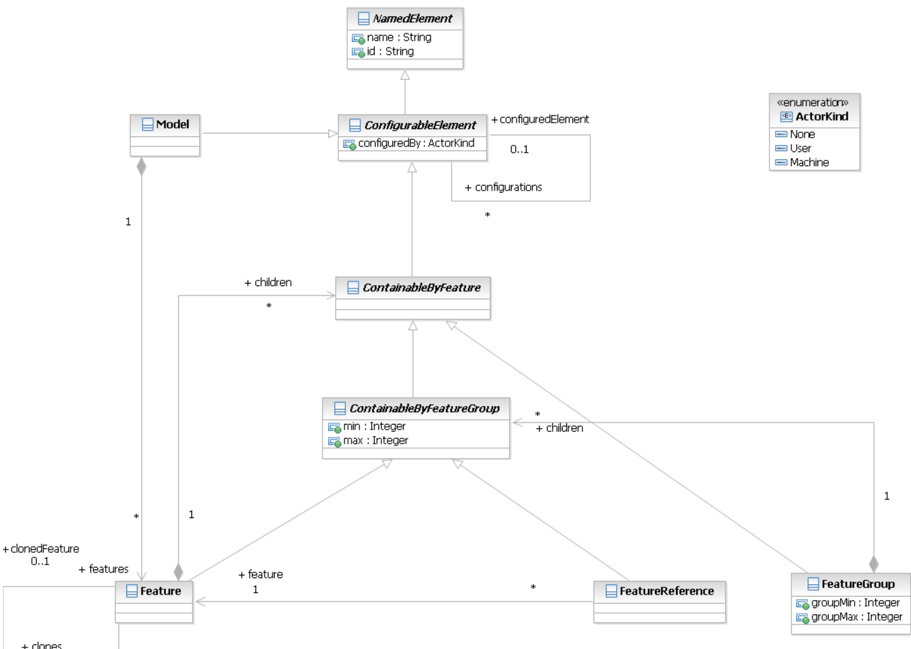


Figure 1. Basic structure of the feature model

inclusion and exclusion of various features and variants so that a valid feature configuration is produced. A feature model can also act as a guide for product configuration and can be used to validate a particular configuration for conformance.

We summarise a meta-model which can be used to describe feature models and which forms the basis of the prototype tool presented in sections 3, 4 and 5.

There are a number of suggested feature modelling languages in existence [6-8] but we have chosen to extend and modify Czarnecki's meta-model [8] (see Figure 1). The following are the reasons behind our extensions and modifications.

- To reduce complexity, explicit reference to `SolitaryFeatures`, `GroupFeatures` and `RootFeatures` was removed and there is no separation between elements that can be contained by a `Feature` and a `FeatureGroup`. These aspects of the meta-model presented in [8] are not required for our purposes.

- We needed enhanced support for the product configuration process and so increased the options available for relating features with architecture and for supporting inter feature dependencies.

- Support for the cloning of features within a feature group was added.

The following sub-sections describe the main characteristics of our meta-model.

2.1. Basic Model Structure

The model structure is designed to support a staged configuration approach where a model may be loaded, partially configured/constrained and saved in iterations. This allows a product to be gradually configured with each stage extending on the previous until all feature variability has been resolved and an end product has been configured. This is supported through the subclassing of `ConfigurableElement` which can contain many `configurations` and can itself be a configuration of one `configuredElement`.

The model supports a hierarchy of features and feature groups where a `Feature` can contain `Features`, `FeatureGroups` and `FeatureReferences`. By using a generalisation / specialisation association between `ContainableByFeature` and `FeatureGroup` and a composition association between `FeatureGroup` and `ContainableByFeatureGroup` we enforce that a

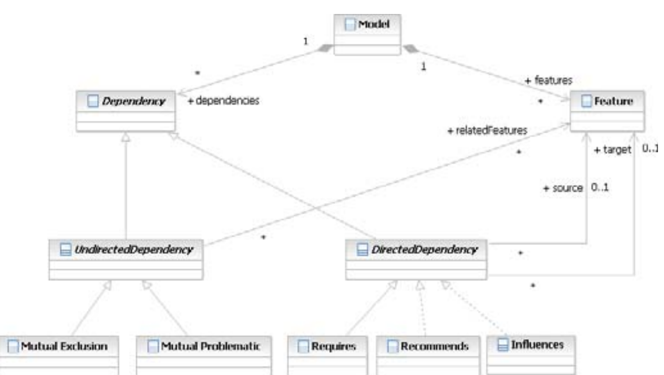


Figure 2. Dependencies among features

`FeatureGroup` cannot contain other `FeatureGroups` but can contain `Features` and `FeatureReferences`.

2.2. Cardinalities

Element selection and elimination is modelled using cardinalities. A `Feature` and `FeatureReference` have a minimum and maximum denoting the number of times they occur [min, max]. This allows us to model optional features as [0,1], mandatory features as [1,1] and eliminated features as [0,0].

A `FeatureGroup` has a `groupMin` and a `groupMax` attribute denoting the minimum and maximum number of elements that can be contained within them. As an example, a `FeatureGroup` containing a set of alternative features would be modelled as `groupMin=1`, `groupMax=1` and each `Feature` within the `FeatureGroup` would have their `min` and `max` atributes set to [0,1].

2.3. Dependencies

The meta-model supports two types of feature relationships (Figure 2), an `UndirectedDependancy` and a `DirectedDependancy`. Two concrete implementations of a `DirectedDependancy` are `Requires` and `Recommends`. As the names suggest, a `Requires` dependency denotes that if a source feature is selected then the target feature *must* also be selected. A `Recommends` dependency denotes that if the source feature is selected then the target feature *should* also be selected.

Two concrete implementations of an `UnDirectedDependancy` are `MutualExclusion` and `MutualProblematic`. `MutualExclusion` denotes that if any one of the set of features is selected then the other feature(s) *must not* be selected. `MutualProblematic` denotes that if any one of the set of features is selected then all other features *should preferably not* be selected.

3. Tool Prototype

Based on the meta-model presented in Section 2 we developed VISIT-FC, a Visual and Interactive Tool for Feature Configuration. Well known visualisation and interactive techniques were employed to attempt to fulfil MacKinlay's [10] expressiveness criteria. This criteria states that a set of facts is *expressible* if all the facts in the set, and only the facts in the set, are expressed. To this end, the VISIT-FC tool strives to display all the information that is required for a particular stakeholder without showing that which can lead to incorrect interpretations through mis-associations.

Visualisation has been described as an "adjustable mapping from data to visual form" [11]. The term "map shock" describes a phenomenon whereby a perceiver has an audible reaction to a visual form that displays an overly complex diagram. Visualisations (and VISIT-FC) aim to relate as much relevant information as possible while avoiding such a reaction from a perceiver. In addition, VISIT-FC adds interactive functionality allowing clear exploration and manipulation of the data.

An instance of the meta-model presented in Section 2 has been created and is used to illustrate the visualisation and interactive techniques employed by VISIT-FC to support the product configuration functionality. The feature model instance introduced represents the Restraint System Control Unit (RESCU) product line. This product line contains features of electronic control units (ECUs) for automotive restraint systems such as airbags and seatbelt tensioners.

The following two subsections give a breakdown of the design and architecture of the tool to show the underlying model and how it supports our interactive visualisation approach.

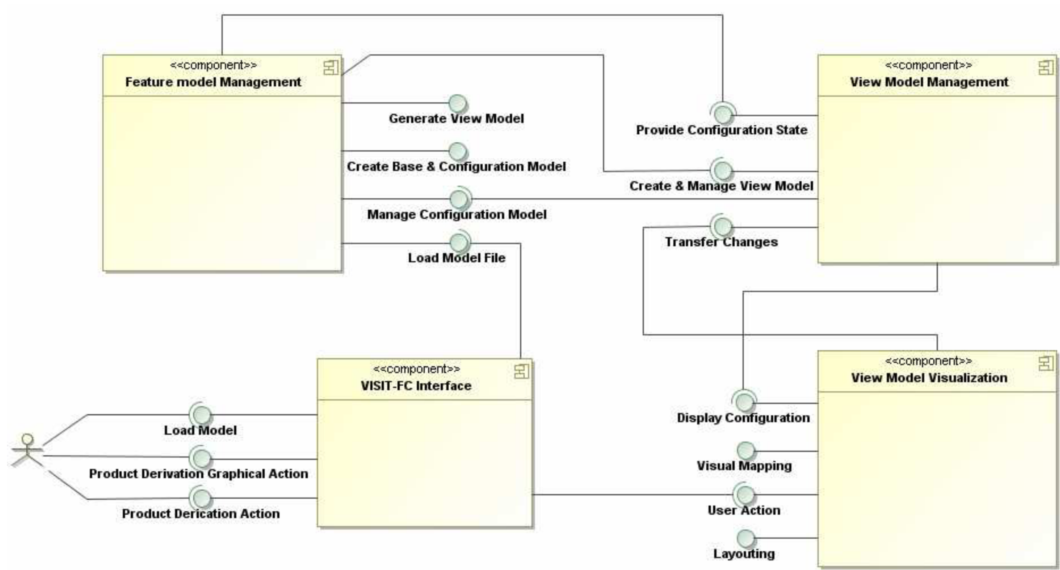


Figure 4. VISIT-FC component diagram

3.1. Design Concept

The primary tasks of the VISIT-FC tool are the visualisation of Software Product Line information and feature configuration. The underlying concept of the design is the separation of the various concerns. Feature configuration is divided between the base *Feature Model* and the *View Model* that acts as a broker between the *Feature Model* and a third *Visualisation* component (see Figure 3).

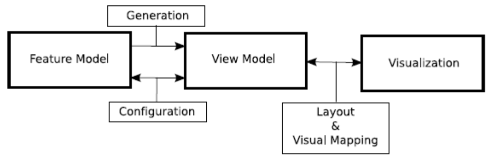


Figure 3. VISIT-FC design overview

As a stakeholder interacts with the *Visualisation*, the *View Model* transfers the information between it and the *Feature Model* and vice versa.

This design affords a number of advantages. As long as a transformation of the *Feature Model* into the *View Model* can be delivered, the *Visualisation* can operate independently of the implementation details of the *Feature Model*. This protects the *Visualisation* against changes to the *Feature Model* and also can allow different feature meta-models to be used. The *View Model* can hide the complexity of the *Feature Model* and provide a simplified view of it.

At a stakeholder's request, the *Feature Model* component loads an XML file containing the product line feature model. A copy of the model (the configuration model) is generated and each feature is linked between the base model and the copy. The copy is used to record and represent the new state of the feature model at its current stage. The *View Model* brokers the configuration data flow between the *Feature Model* and the *Visualisation*. The *Visualisation* is controlled by layout and mapping mechanisms that use defined information for visualising the contents of the *View Model* and providing interactive functionality to the stakeholder.

3.2. Software Architecture

Figure 4 shows the component diagram for the VISIT-FC tool. The tool is comprised of four components.

Firstly, the Feature Model Management component facilitates the loading of the model source XML file, the creation of a *base* and *configuration* model, the generation of the *View Model* and provides the functionality that manages the changes to the model that occur during the product configuration process.

Secondly, the View Model Management component allows the creation of a new *View Model.* It maintains the *View Model* reflecting the product configurations being undertaken, transfers information concerning the

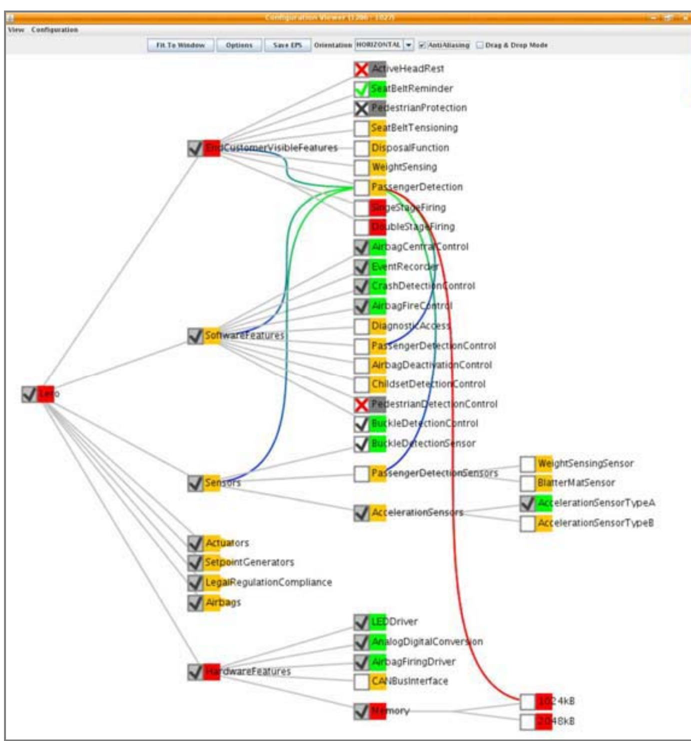


Figure 5. VISIT-FC configuration viewer showing features of the RESCU product line

configuration model to the View Model Visualisation component and synchronises the modifications performed by the stakeholder with the Feature Model Management component.

Thirdly, the View Model Visualisation component stores and maintains all the pertinent information related to the graphical representation. It manages the layout algorithms and facilitates graphical manipulations by the stakeholder.

Finally the VISIT-FC User Interface component allows the actual screen events to take place providing functionality such as feature selection, pan and zoom and node manipulation.

To load a model file, functionality within the Feature Model Management component is invoked. The Feature Model component creates a base and configurable model and also generates the *View Model* through a call to the View Model Management component. Finally, a call is made to the View Model Visualisation component resulting in the rendering of the *View Model* visually.

When product configuration actions take place, functionality within the View Model Visualisation component is invoked that produces a transfer of information between the View Model Visualisation component and the Feature Model Management component through the View Model Management component. When the Feature Model Component receives this information it applies the modification on the configuration model and invokes the View Model Visualisation component via the View Model Management component to display the modifications.

4. Visualisation & Interactive Techniques

The VISIT-FC tool (see Figure 5) aims to employ visualisation and interactive techniques that facilitate the following goals: provide a compact, interactive representation of large feature hierarchies; provide facilities to restrict the view to feature model parts of interest; allow feature configuration with automatic constraint propagation and provide hints for configuration problems and open decisions, see also [5]. This section describes the techniques that are utilised and the specific functionality that is then possible allowing the fulfilment of the specific goals.

4.1. Explicit Representation

The VISIT-FC tool uses Explicit Representation as opposed to Implicit Representation. Explicit Representation refers to drawing methods which display the hierarchy as links between nodes, e.g. [12]. Implicit drawing methods represent the hierarchy by a special arrangement of nodes, e.g. containment or overlapping. Examples of implicit graph drawing are tree-maps [13], or the information cube [14]. Figure 5 shows a screenshot of the main visualisation of the RESCU product line feature model in VISIT-FC.

4.2. Horizontal Linear Tree Layout

Advanced layouts exist for explicit tree-drawings such as cone-trees [15] or space-trees [12]. However, for the purpose of this prototype, a 2D visualisation was chosen, and therefore a simple non-radial tree layout [16] was adopted. The horizontal orientation is preferable over the vertical orientation although the tool does allow the stakeholder to view the model in vertical tree layout. The non-radial (linear) layout and horizontal orientation combine to provide the optimal use of screen space to allow the display of the kinds of data related to a product line feature model. As an example, displaying the names of features on screen with a radial or vertical tree layout would result either in large amounts of overlapping or a zoomed out view (to avoid overlapping) both of which would obscurely render the visualisation.

The combination of an Explicit Representation and a Horizontal Linear Tree gives us the opportunity to encode a significant amount of information on screen utilising the restricted space in an efficient manner. VISIT-FC uses an explicit horizontal linear tree layout where the nodes represent features and the edges represent the relationships between those features. Straight edges indicate parent-child relationship and curved edges represent dependency relationships.

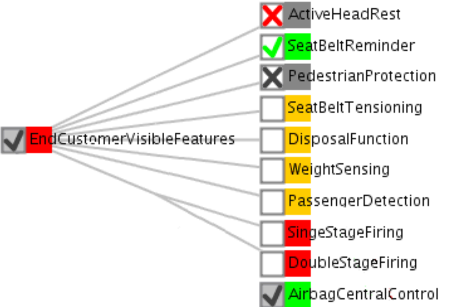


Figure 6. Information encoding in VISIT-FC

Figure 6 shows a portion of the RESCU feature model illustrating the information that is encoded in quite a small screen area. Colour coding of the features adds another layer of information to this basic node link tree structure.

The colours indicate the configuration status of the selected features and their sub-features, that is, a `FeatureGroup` is colour-encoded *mandatory but not configured* if its sub-features are not resolved. There are four levels of colour encoding, one for each of the feature states, which are *selected (green)*, *eliminated (grey)*, *optional (amber)* and *mandatory but not configured (red)*. These colour codes allow a quick overview of the feature model and its state, for instance to see if a valid product configuration exists. Further information is encoded by use of graphical symbols (tick or cross). A tick indicates selection, a cross indicates elimination. To this, another layer of information is encoded through the use of additional colour coding. If the box is shaded, then the feature has been preconfigured or eliminated at an earlier stage of configuration and is no longer changeable. If the box is not shaded but the icon is not coloured, then the feature was selected or eliminated based on a dependency.

Information encoded at this low level of visual representation is processed pre-attentively [11] [17] by the human graphical system. Therefore once the colour encoding becomes familiar, a stakeholder would be able to interpret large representations rapidly.

4.3. Details on Demand

Details on Demand refer to the facility whereby the stakeholder can choose to display additional detailed information at a point where this data would be useful. Information such as cardinalities can be displayed through the use of a "mouse-over" (see Figure 7) and feature names can be displayed or removed through viewing configuration options.

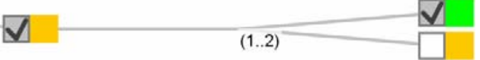


Figure 7. Relationship showing cardinalities

VISIT-FC also provides the facility to choose a specific feature and show all sub features and dependent features while hiding all other features that are neither sub features nor dependent in any way on the chosen feature (see Figure 8).

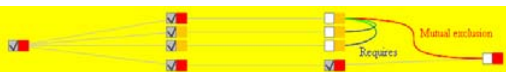


Figure 8. Contextual view

This allows the stakeholder to focus on the relevant data for a particular feature while temporarily removing irrelevant data. This function is easily accessible and removable by use of a keyboard shortcut and mouse movement.

4.4. Incremental Browsing

Incremental browsing is a form of information filtering, where only limited sections of the visualised structure are displayed. The rest is hidden and can be visualised when needed.

In VISIT-FC the feature model visualisation starts with displaying only the high-level features, and the stakeholder can then explore the feature hierarchy by unfolding the sub-features of features in which the stakeholder is interested in. The stakeholder is thus able to perceive the feature structure step by step, and is not overwhelmed by the complete model.

Figure 9 is a simple illustration where only one feature has been unfolded. The triangular extension of the colour coded feature indicates further unfolding is possible.

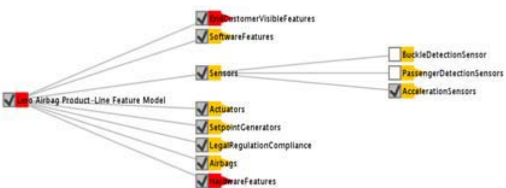


Figure 9. Support for incremental browsing

4.5. Focus+Context

Focus+Context refers to the ability to focus on a particular aspect or portion of the visualisation while not losing the context in which that aspect or portion resides e.g. [18]. The advantage of Focus+Context is that the stakeholder does not get lost when zooming into a large structure, or exploring the details of certain features. They are always able to see where they came from, and are not required to keep this in memory. This can be useful, e.g., for the visualization of search results or to see dependent feature nodes in distant parts of a large feature model within the context of the whole feature structure.

Pan, Zoom and Degree of Interest in combination are powerful techniques that allow the stakeholder to move around the visualisation, zoom and highlight a particular area of interest. VISIT-FC provides these facilities and also allows selective zooming of a specific chosen portion of the feature tree focusing on the area of interest and allowing the non-relevant area to remain in view but to a lesser degree. Figure 10 shows a simplified version to illustrate the split zooming facility. It shows certain user selected features that have been "zoomed out" because they are of lesser interest while keeping them in view which maintains the overall context. Different sets of feature nodes can be "zoomed in" or "zoomed out" to varying degrees to allow an optimum view for the task at hand.

5. Feature Configuration Example

To illustrate the use of VISIT-FC and its benefits, this section describes the steps that a stakeholder would undertake to configure a specific aspect of interest within the RESCU product line. In this instance a stakeholder wishes to explore the "Software Features" of a product configuration that is in progress and add configuration for Diagnostic Access.

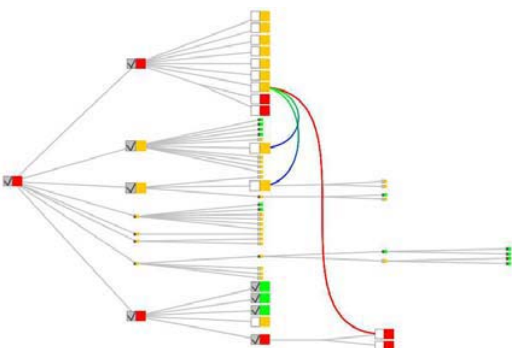


Figure 10. Focus+Context, Degree of Interest and Details on Demand

On start up VISIT-FC shows the root node of the feature model, which can be expanded to show the next level of tree nodes (see Figure 11). Immediately the stakeholder can see that the model has been through a previous configuration stage, that there are eight feature sets, two of those are mandatory and that none of the feature sets have been fully configured as yet. By expanding the "Software Features" node, the stakeholder can explore this section of the product line while keeping all other sections out of the way but still in context. The stakeholder can see that four of the ten sub-features have been configured at a previous stage and that six remain to be resolved.

In this scenario, the stakeholder is interested in configuring "Diagnostic Access" and can see it has not been previously configured (see the corresponding green node in Figure 12). By clicking on the Diagnostic Access node, the stakeholder can select this feature for the product being derived. On selection, the application

automatically configures two other features in the product line by selecting the feature “CAN Bus Interface” (a sub-feature of “Hardware Features”) and eliminating the “1024KB Memory” variant. These dependent features are highlighted through increased node size notifying the stakeholder of the automatic actions. If a dependent node is not currently displayed at the point of automatic selection / elimination of the feature, then it is made visible at that time. The stakeholder can then distinctly display the dependencies using curved colour coded links. By use of split zooming and panning, the stakeholder modifies the display for even further clarity.

If desired by the stakeholder, he can display all other features that are connected in a dependent fashion providing a useful view of connected parts of the product being derived. Moreover, he or she can switch the view to the dependency context mode (Figure 8) temporarily removing all data from the screen except that which is directly connected to the feature being configured.

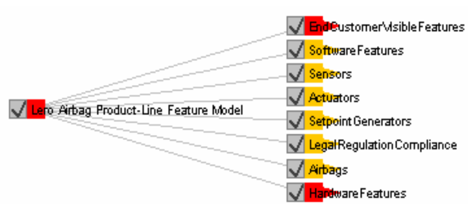


Figure 11. Initial Feature Model View

6. Related Work

FeaturePlugin [19], pure::variants [20], COVAMOF [21] and Gears [22] are examples of other feature modelling tools that employ a visual component to aid product configuration and variability management.

FeaturePlugin is an Eclipse IDE plug-in that supports feature modelling. It uses the Eclipse Modelling Framework (EMF) to generate the editors that facilitate the modelling aspect and provides a rich set of modelling functionality. Nested lists and a tree layout are employed as techniques to support product configuration using the FODA style. Some of the drawbacks of FeaturePlugin are that the lists can be difficult to navigate as the focus+context display implementation is not very effective. It is also difficult to comprehend the dependencies as constraints are shown as unsorted lists.

pure::variants was developed by pure-systems GmbH and is a software package that provides similar functionality to FeaturePlugin. It supports various views which provide different approaches for different stakeholder tasks but does not support cardinality. Using the built in automatic layout, can adversely affect the tree layout which as it is can be confusing. Large industrial product lines could easily lead to information overload.

A suite of tools exist that provides the implementation of the requirements of the ConIPF Variability Modelling Framework, COVAMOF. Even though significant functionality exists, understanding of the overall state of the configuration can be difficult due to the separate and disconnected window views.

7. Future Work

The development of the VISIT-FC prototype is based on the utilisation of well understood but non-complex visualisation and interaction techniques. It has shown an avenue down which the challenges faced by stakeholders during product configuration can be addressed. Even simple information encoding through colour schemes suggests an increase in the speed at which product configurations can be interpreted. More in depth research into visualisation techniques and their applicability to and usability for, variability management tasks is planned.

Development of the tool to implement further functionality provided by the meta-model is also planned, such as implementation of the `FeatureReference` entity, cloning of features and linking of the asset base, feature model and realisation artefacts. This would provide an end-to-end visual support for an interactive product derivation tool.

The possibility of providing this prototype tool as an Eclipse plug-in will also be explored.

8. Conclusions

We have presented a feature configuration meta-model and introduced a prototype tool that utilises that meta-model and employs a variety of visualisation and interaction techniques. We suggest that targeted use of these techniques in combination with a suitable meta-model can provide significant aid to product configuration stakeholders.

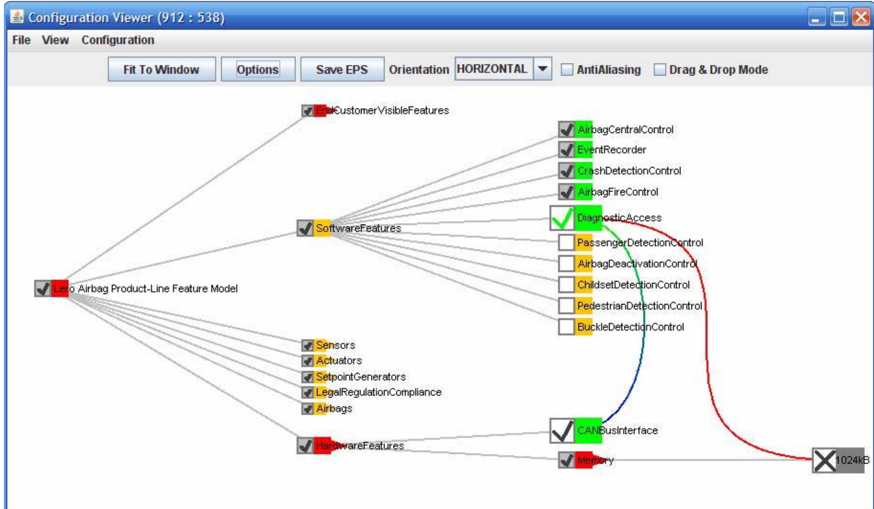


Figure 12. Staged feature configuration example

In the authors' opinion, further research into the applicability of various visualisation and interaction techniques to variability management could significantly lessen the challenges faced by stakeholders and greatly increase the efficiency of a number of tasks that require fulfilment when undertaking product line engineering. Furthermore, a configurable visualisation toolkit could replace the dependence on a small number of experts and allow software product line engineers execute their tasks with much greater autonomy.

9. Acknowledgements

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

10. References

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

[2] M. Sinnema and S. Deelstra, "Classifying variability modeling techniques," *Information and Software Technology,* vol. 49, pp. 717-739, 2007.

[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] 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.

[5] D. Nestor, L. O'Malley, A. Quigley, E. Sikora, and S. Thiel, "Visualisation of Variability in Software Product Line Engineering," in *VaMoS-2007*, Limerick, Ireland, 2007.

[6] K. Kang, S. Kim, J. Lee, K. Kim, S. E., and M. Huh, "FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures," *Annals of Software Engineering,* vol. 5, pp. 143-168, 1998.

[7] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," SEI, Carnegie Mellon University CMU/SEI-90-TR-21, November 1990.

[8] K. Czarnecki, S. Helsen, and U. W. Eisenecker, "Staged Configuration Using Feature Models," in *Proceedings of SPLC 2004*, 2004, pp. 266-283.

[9] T. Bednasch, "Konzept und Implementierung eines konfigurierbaren Metamodells für die Merkmalmodellierung," in *Fachbereich Informatik* Zweibrücken, Germany: Fachhochschule Kaiserslautern, 2002.

[10] J. Mackinlay, "Automating the design of graphical presentations of relational information," *ACM Trans. Graph.,* vol. 5, pp. 110-141, 1986.

[11] S. K. Card, J. D. MacKinlay, and B. Shneiderman, *Readings in Information Visualization - Us-*

ing Vision to Think, 1st ed. San Francisco: Morgan Kaufmann Publishers, 1999.

[12] C. Plaisant, J. Grosjean, and B. B. Bederson, "SpaceTree: Supporting Exploration in Large Node Link Tree, Design Evolution and Empirical Evaluation," in *Proceedings of the IEEE Symposium on InformationVisualization (InfoVis 2002)*, 2002.

[13] B. Johnson and B. Shneiderman, "Tree-maps: a space-filling approach to the visualization of hierarchical information structures," in *Proceedings of the 2nd Conference on Visualization*, 1991, pp. 284–291.

[14] J. Rekimoto and M. Green, "The information cube: Using transparency in 3d information visualization," in *Workshop on Information Technologies & Systems*, 1993, pp. 125-132.

[15] G. G. Robertson, J. D. Mackinlay, and S. K. Card, "Cone trees: Animated 3D visualizations of hierarchical information," *Proceedings of CHI 9,* pp. 189-194, 1991.

[16] E. M. Reingold and J. S. Tilford, "Tidier Drawings of Trees," *IEEE Transactions on Software Engineering,* vol. 7, pp. 223-228, 1981.

[17] C. Ware, *Information Visualisation: Perception for Design*, 2nd ed.: Morgan Kaufmann, 2004.

[18] S. K. Card and D. Nation, "Degree-of-interest trees: A component of an attention-reactive user interface," *Advanced Visual Interface,* pp. 231-245, 2002.

[19] 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.

[20] pure-systems GmbH, "Variant Management with pure::variants," http://www.pure-systems.com, Technical White Paper, 2003-2004.

[21] M. Sinnema, O. d. Graaf, and J. Bosch, "Tool Support for COVAMOF," in *SPLC 2004, Workshop on Software Variability Management for Product Derivation* Boston, MA, USA, 2004.

[22] Biglever Software, Gears,: http://www.biglever.com.

Towards an Automatic PL Requirements Configuration through Constraints Reasoning

Olfa Djebbi[12], Camille Salinesi[1]
[1] *CRI, Université Paris 1 – Sorbonne, 90, rue de Tolbiac, 75013 Paris, France*
[2] *Stago Instruments, 125 avenue Louis Roche, 92230 Gennevilliers, France*
olfa.djebbi@malix.univ-paris1.fr, Camille.salinesi@univ-paris1.fr, odjebbi@stago.fr

Abstract

Requirements Engineering (RE) processes are of great interest in Software Product Line (SPL) development. Several variability approaches were developed to plan requirements reuse, but only little of them actually address the issue of configuring products. This paper presents an approach that intends to support requirements configuration in SPL. Its goal is to deliver products people really want.

Three main characteristic of the approach are noteworthy: 1) it is user-oriented, 2) it guides product requirements elicitation and configuration as a matching activity offering a coherent global view on the product line, and 3) it provides systematic and interactive guidance assisting analysts in taking decisions about requirements.

1. Introduction

Software Product Line Engineering is emerging as a viable and important development paradigm allowing companies to realize order-of-magnitude improvements in time to market, cost, productivity, quality and flexibility [1]. This is attributed to the capitalization on the commonality and the management of the variations among products in the product line. As a result, the main effort to design a product from the product line is due to the variations analysis and the impact of the choices made for the required product.

In this particular context, Requirements Engineering (RE) processes have two goals: (i) to define and manage requirements within the product line and (ii) to coordinate requirements for the single products. To achieve the latter goal, existing approaches rely on a commonly named 'derivation' process (figure 1) that consists in retrieving from the product line requirements some collection specifying the single product to build.

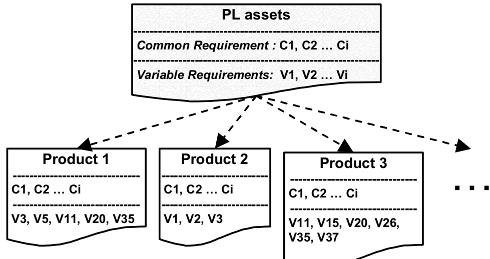


Figure 1. **Requirements derivation as checklist choices**

This way of working have several serious limitations concerning in particular the lack of representation of stakeholders' requirements, and the RE process itself.

Indeed, selecting requirements among pre-defined product line requirements models influences stakeholders and skews their choices. Experience with this approach in other domains such as COTS selection or ERP implementation shows that stakeholders naturally establish links between their problem and the pre-defined solutions, adopt features with marginal value, and naturally forget about important requirements that are not present in the PL requirements model [2] [3]. As a result, the focus is on model elements that implement the solution rather than on the expression of actual needs. While this approach supports reuse, it generates products that finally lack of attractiveness, or even worse usefulness. Each important requirement missing leads to unsatisfied final users and customers.

Besides, while the RE process should foster creative thinking within the system requirements, selecting among predefined requirements restricts considerably creativity and search for innovative ways to deal with

problems, and hence reduces the added value of the new products to be developed.

Moreover, analysts are most often on their own to elicit the requirements for new products. As shown in previous publications [4] [5], existing approaches and tools provide little guidance (notation, process, rules, impact analysis) to assist them in eliciting consistent product requirements. They are neither guided in adding new requirements to the PL requirements model to support more complex evolutions of the product line.

On the other hand, an approach in which stakeholders would come up with completely new requirements, specifying these independently from the PL requirements model would be difficult to handle and can become very inefficient. Indeed, retrieving correspondences between stakeholders' requirements and PL requirements can be time taking, error prone, and implies to face difficult issues such as inconsistent levels of abstraction, inconsistency in the way similar requirements are expressed, and the need for large amount of details to decide whether stakeholders' requirements are satisfied. We strongly believe that a systematic guidance is needed to facilitate this activity and, most importantly to check the consistency of product requirements with PL and stakeholders' requirements models.

To deal with requirements configuration avoiding aforementioned limits, our proposal is to handle it as a matching issue substituting the classic transformation issue [6]. Thereby, the design of a product is no longer a stepwise derivation carried out from the alternatives offered by the product line models predefining possible solutions, but a matching between the requirements regarding products, and the requirements that the product line is designed to meet (figure 2). The obtained configuration balance then the definition of the solution between:

(i) Initial requirements (As-wished) that can be satisfied in a standard manner by the PL (Might-be);

(ii) Initial requirements that, on the contrary, can not be satisfied by the PL as it is defined. Nevertheless, they should be included in the final product configuration (To-be);

(iii) and PL assets that should be implemented in the final configuration even though they have not been expressed originally as requirements.

As shown in the figure 2, the matching between stakeholders' initial requirements and PL requirements allows eliciting consist collections of requirements relating to different products that are likely to be considered. Arbitration is then needed to elect one of these collections.

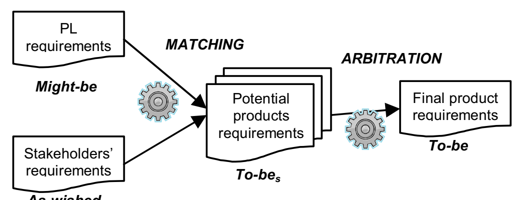


Figure 2. **Requirements Configuration as a matching issue**

One important challenge to undertake product specification in this context is obviously to efficiently handle both the constraints underlying the PL model and the constraints expressed by users for the product under development. We believe that a 'good' requirements configuration approach should enable:

- considering both PL capabilities and stakeholders' requirements,
- arbitrating on possible requirements sets in an interactive manner by analyzing decisions impacts,
- reasoning automatically on variability so as to deal with complexity, to be scalable, to be supported by CASE tools, …
- integrating easily new requirements.

This paper presents our first ideas towards a systematic approach for product requirements specification in a PL context. It lies on the consideration of the requirements configuration process as a resolution of assembly constraints rather than a resolution of their variability points.

The remaining of the paper is structured as followed. Section 2 presents our approach to deal with the matching issue. Section 3 outlines the language used to conduct the approach. Section 4 is dedicated to the related works. And finally, conclusions and future works are reported in section 5.

2. Constraints based configuration

Referring to the configuration framework introduced in the previous section, matching techniques are required in order to concord PL and stakeholders' requirements. They should consider two specificities of the SPL context:

(i) The diversity of variability notations [5] [7] based on features, use cases, aspects, etc.

(ii) The multi-facet PL environment and the multi-point-of-view on it due to the multiplicity of the stakeholders. Several stakeholders (customers,

legislation, hardware, organization, executives, developers, distributors, etc.) could express differently their requirements, goals, constraints, restrictions … using different notations.

To deal with matching, different approaches have traditionally been employed in requirements engineering; among them similarity techniques are the most studied and applied [8] [9] [10] [11].

In a PL configuration context, and beyond the popular belief that these techniques are time taking and error prone, they prove to be inadequate for many reasons.

Indeed, retrieving syntactic or semantic correspondences implies to face a conceptual mismatch given that the various requirements are not expressed by the same languages and may even be structured on several abstraction levels. Dependency relationships between PL requirements add further complexity to the matching process. Moreover, inconsistency in the way similar requirements are expressed can also be observed so that there is a need for a large amount of details to decide whether stakeholders' requirements are satisfied. Besides, in no case similarity analysis facilitates new requirements integration. Furthermore, the retrieved PL requirements that are 'similar' to stakeholders' requirements do not necessarily constitute the right solution since (i) arbitration is still needed if PL capabilities include alternatives to satisfy some requirements, and (ii) some stakeholders' requirements that are constraints on other requirements values or on some requirements combinations will not be 'similar' with PL requirements and then will not be considered in specifying a satisfactory product configuration [12].

It clearly appears that an intermediate ambivalent language is required to deal with all requirements, whatever is the used language, in a unique global view. In the other hand, we strongly believe that a systematic approach is needed for both matching and arbitrating on the requirements in the same time. It would be possible by analyzing not only domain variabilities but also variabilities of strategic requirements that are expressed about the PL variability to carry out in single products.

The approach should be formal in order to facilitate this activity and most importantly to check automatically the consistency of product, PL and stakeholders' requirements, the one with respect to the other. It would be also useful if the approach formalism is flexible enough to allow to users querying on requirements before adopt them so as it provides a kind of an interactive guidance during configuration process. Our investigation among domain experts shows that in practice the interactivity is preferable to a ready-made solution so as constraints are claimed and decisions are made with a good view on their impacts. An important point to pick up then is that we are not only interested in solving completely one instance. We are mostly interested in staged configuration, in dealing with a partial solution and in finding several solutions.

At this aim, our proposal is a formal language that abstracts the variability expression and guides the configuration process by means of constraint structures. It appears that Constraints are the most adequate paradigm to fulfill all identified requirements. Constraints seem to be an intuitive language to express several stakeholders' requirements as well as PL capabilities. It can be also directly mapped into a powerful programming language for Constraint Satisfaction Problems (CSP), in instance the Constraint Programming (CP) [13] [14] [15] [16] [17]. Besides, Constraint paradigm is flexible enough to be able to deal with new needs such as the introduction of new types of requirements dependencies, new kinds of constrains, or richer way to express stakeholders' requirements (e.g. "we want at most 3 occurrences of feature X in a final product").

In a previous work [13], we demonstrated how to use constraints for (i) the definition of consistency verification rules and (ii) the specification of configuration requests that may help users take decisions. Our intention grows now to define a language based on the Constraint concept, named Constraint Language (CL), that formalizes these specifications.

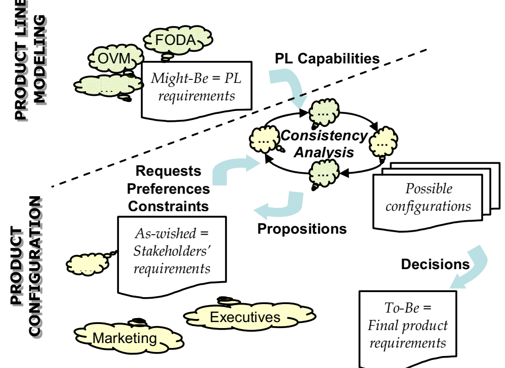


Figure 3. **Constraint Language mechanisms overview**

As shown in the figure 3, the language should allow to express PL requirements and to configure products. Stakeholders' requirements, constraints and preferences should be taken into account within an interactive decision process under the form of requests. Users should receive propositions showing explicitly some choices impacts before take firm decisions.

A first step in this ongoing research is then to specify a subset of the language grammar that will allow to express variability for PL requirements, as well as the transformation patterns from the several variability notation to the Constraint Language. A second step will be to complete CL grammar in order to describe the possible product configurations, and to define transformation rules from an As-wished & a Might-be state into To-be product(s).

The next section develops the first step. For the second step, some language elements are elaborated but not refined enough to be presented in this stage.

3. Constraint Language

Let have this terminology:

- R: requirement
- a: requirement's attribute
- D: domain of attribute

The language grammar is composed of two main structures:

1. Induction

A relationship 'induce' between a requirement R_1 (having a set of attributes a_{j1}) and a set of requirements R_i (having each a set of attributes a_{ji}) specifies that the selection of some requirements R_i, satisfying some function f involving one or more attributes of R_i and one or more attributes of R_1, implies the selection of R_1 (with specific attribute(s) value(s)).

induce $(\{<R_i, a_{ji}>\}, <R_1, a_{j1}>, f\ Da_{ji} \rightarrow Da_{j1} \rightarrow (boolean))$

For example, let consider a library system; one can specify that when the requirement 'Loan books' having as attribute 'max number books', and the requirement 'Loan journals' having as attribute 'max number journals', are both selected, so that the function 'max number books + max number journals > 4' is verified, then the requirement 'Notify for restitution' is also selected, and must have the value '(max number books + max number journals)*2' for its attribute 'notice deadline'.

When the 'induce' relationship concerns two requirements independently from their respective attributes (i.e. f is always true), then its syntax can be shortened as following:

$R_1 >> R_2$

2. Restriction

The relationship 'restrict' allows to express restrictions on requirements attributes. These restrictions are defined by a mathematical function on these attributes that can contain aggregations, logic operators, etc.

restrict $(\{<R_i, a_{ji}>\}, f(\{<a_{ji}>\}))$

The cardinality of a requirements set is a particular case of this relationship where restrictions are put on the maximum number and the minimum number of requirements that can be selected among the set. The 'restrict' relationship can then be shortened as following:

$\{R_i\} : [\ card_{min}]\text{-}[\ card_{max}]$

If minimal and maximal cardinalities are equal to the requirements number in the set, so these requirements are mandatory. The relationship is then shortened under this form:

$\{R\} : n\text{-}n \equiv R$

If the minimal cardinality of a requirement is equal to zero and its maximal cardinality is equal to one, then this requirement is optional. The relationship is shortened as following:

$\{R\} : 0\text{-}1 \equiv [R]$

When minimal and maximal cardinalities of a set of two requirements are equal to zero and one respectively, then this relationship express an exclusion between these two requirements. The relationship is then shortened as following:

$\{R_1, R_2\} : 0\text{-}1 \equiv R_1 >< R_2$

This section explains how variability notations can be expressed through the two structures of our Constraint Language. The table bellow presents transformation patterns for Feature based variability notations.

Table 1. **Transformation patterns for Feature notation**

Feature notation	*CL transformation*
F0 *(F0 is the root)*	F0
F1 F2 *(composition)*	F1 >> F2 F2 >> F1
F1 F2 *(option)*	[F2] F2 >> F1
F1 Card min..max F2 F3 F4 *(alternative)*	{F2, F3, F4} : $card_{min}$-$card_{max}$ F2 >> F1 F3 >> F1 F4 >> F1
F1 « requires » F2 *(requires)*	F1 >> F2
F1 « mutex » F2 *(mutex)*	F1 >< F2
F1 F2 [p1] [P2] F11 F12 F21 F22 m *(OCL-like constraints♣)* **context** F11, F12, F22 **def** p11: F11.p1[int] P22: F22.p2[int] **inv** if(selected(F12) and selected(F11) and selected(F22)) then (p11 <= 5*p22);	induce (F11, F12, F22, p1<=5*p2)

♣ Example extracted from FORE notation. [D. Streitferdt, 'Family-Oriented Requirements Engineering', PhD Thesis, Technical University Ilmenau, 2004.]

The rightness of the transformation patterns is verified by their semantic equivalence. Indeed, products that can be configured from a feature model are comparable to those that can be configured from its equivalent expressed in our Constraint Language.

Further works will expose transformation patterns for other variability approaches families.

4. Related works

Different methods interested in constructing SPL assets are available in the literature. Product configuration methodologies are on the contrary rather scarce and still reduced to technical levels. Regarding RE in an SPL context only little work is done [18] [19] [20]. Mostly, configuration methods are restricted to the selection of a PL requirements subset. However, industry experience suggests that simply having the right assets is not sufficient to facilitate its selection and assembly. Our approach is motivated by two goals that make its originality: (i) a product specification is not just a choice made on alternatives offered by the PL capabilities, but also a response to the several stakeholders' requirements and constraints (customers, executives, legal environment, marketing …), (ii) the requirements configuration is guided as a complex and stepwise decision making activity.

For the latter goal, there are already some proposals on automating analysis of feature models in order to support retrieving information from PL models.

The greatest number of works is based on propositional logic where first-order logic is mainly used to check validity of feature models [21] [22] [23] [24] [25]. This technique does not really fit in our context. It is well appropriate to compute complete solutions, but we are also interested in incrementality dealing with a partial solution and in finding several solutions. Besides, we want a technique that is flexible enough to be able to deal with new needs such as the introduction of new types of requirement dependencies, new kinds of constrains …. It appears that the CSP is the most adequate paradigm to fulfill all these requirements.

Benavides [14] [26] [27] works precisely on Constraint Programming. In his study, he proposes a mapping from feature models to a CP notation, and introduces some operations on PL models in order to extract information like all the possible products of the PL, their number, the optimum product, etc. One operation is precisely important because it can assist PL engineers take decisions, and that indeed justifies the use of CP technique. It is the filter operation that consists on applying a constraint on the attributes

values within the PL model, so that it restricts derived products.

While the focus of Benavides is on translating feature models into a CSP notation to allow extract automatically information from PL models, we are interested in instructing requirements configuration processes that originate from users needs, and involve users choices while tacking decisions; which is not typically available in general derivation approaches.

Furthermore, from this perspective, we are not interested only in feature models, but on all other variability notations. Our aim concerning the modeling language is twofold: (i) to be abstract enough to be ambivalent, (ii) to be not bulky but with a high expression capability.

We are interested also in finding and defining new decisions to make in product configuration and improve the ones considered in [26]. Example of such requests is: the next choices to make with respect to a first requirements selection / exclusion of some undesired requirements, etc. [13].

5. Conclusion

The requirements configuration in a SPL context is an ongoing research area in the RE field. It did not yet have sufficient interest and it is restricted to the selection of a requirements subset from the PL assets.

This paper sets for a matching framework to deal with this issue. The idea behind the proposed approach is that: (i) all stakeholders should be involved in specifying product requirements, and (ii) the configuration process should dispose of formal processes and automatic means to be able to assist efficient decisions about product requirements to build.

For that purpose, an interactive decision process is outlined. It is conducted by a Constraint based Language (CL) that allow expressing stakeholders' requirements, as well as reasoning on them to enable efficient decisions. We demonstrate that Constraint paradigm is the most adequate to fulfill this.

Our work is still in progress. Its extension will include the development of the Constraint Language, its transformation patterns and its transformation rules. A further goal will focus on the validation of our approach in an industrial environment. We intend also to implement the tool supporting it and that can be interfaced with existing PL management tools [4] [13].

6. References

[1] http://www.sei.cmu.edu/productlines/

[2] I. Zoukar, C. Salinesi, "Matching ERP Functionalities with the Logistic Requirements of French railways - A Similarity Approach", *6th International Conference on Enterprise Information Systems, ICEIS' 2004*, Porto, Portugal, 2004.

[3] C. Rolland, N. Prakash, "Matching ERP System Functionality to Customer Requirements", *Proceedings of the 5th International Symposium on Requirements Engineering, RE'01*, Toronto, Canada, 2001, pp. 66-75.

[4] O. Djebbi, C. Salinesi and G. Fanmy, "Industry Survey of Product Lines Management Tools: Requirements, Qualities and Open Issues", *International Requirements Engineering Conference, RE'07*, India, 2007.

[5] O. Djebbi, and C. Salinesi, "Criteria for Comparing Requirements Variability Modeling Notations for Product Lines", *Comparative Evaluation in Requirements Engineering, in RE'06 (CERE)*, Minneapolis, USA, September 2006, pp 20 - 35.

[6] C. Salinesi, C. Rolland, "Fitting Business Models to Systems Functionality Exploring the Fitness Relationship", *proceedings of CAiSE'03*, Velden, Austria, 16-20 June 2003.

[7] P.Y. Schobbens, , P. Heymans, , J.C. Trigaux and Y. Bontemps, "Generic semantics of feature diagrams", *Computer networks*, February 2007.

[8] S. Castano, V. De Antonellis, B. Zonta, "Classifying and Reusing Conceptual Components", *Proceedings of the 11th International Conference on Conceptual Modelling (ER'92)*, Karlsruhe, 1992, pp. 121-138.

[9] L.L. Jilani, R. Mili, A. Mili, "Approximate Component Retrieval: An Academic Exercise or a Practical Concern?", *Proceedings of the 8th Workshop on Institutionalising Software Reuse (WISR8)*, Columbus, Ohio, March 1997.

[10] J. Natt och Dag, B. Regnell, P. Carlshamre, M. Andersson, J. Karlsson, "Evaluating Automated Support for Requirements Similarity Analysis in Market-driven Development", *Proceedings of REFSQ'01, 7th International Workshop on Requirements Engineering : Foundations of Software Quality,* Interlaken, Switzerland, June 2001.

[11] I. Zoukar, "MIBE : Méthode d'Ingénierie des Besoins pour l'implantation d'ERP", *Thesis of Université Paris I*, 18 April 2005.

[12] O. Djebbi, and C. Salinesi, "RED-PL, a Method for Deriving Product Requirements from a Product Line Requirements Model", *International Conference on Advanced information Systems Engineering (CAISE'07)*, Springer Verlag, Trondheim, Norway, June 2007.

[13] O. Djebbi, C. Salinesi, and D. Diaz, "Deriving Product Line Requirements: the RED-PL Guidance Approach", *Asian Pacific Software Engineering Conference (APSEC)*, IEEE Computer Society, Nagoya, Japan, December 2007.

[14] D. Benavides, S. Segura, P. Trinidad and A. Ruiz-Cortés, "Using Java CSP Solvers in the Automated Analyses of Feature Models", *Post-proceedings Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'05)*, Braga, Portugal. 2005.

[15] P. Codognet and D. Diaz, “Simple and Efficient Consistency Techniques for Boolean Solvers in Constraint Logic Programming”, *Journal of Automated Reasoning*, Vol. 17, No. 1, 1996.
[16] J. Jaffar and M. J. Maher, “Constraint logic programming: A survey”, *Journal of Logic Programming*, Vol. 19/20, 1994.
[17] D. Diaz and P. Codognet, “Design and Implementation of the GNU Prolog System”. *Journal of Functional and Logic Programming (JFLP)*, Vol. 2001, No. 6, October 2001.
[18] T. von der Massen and H. Lichter, “Requiline: A requirements engineering tool for software product lines”, *In Proceedings of the Fifth International Workshop on Product Family Engineering (PFE-5)*, LNCS 3014, Siena, Italy, 2003. Springer Verlag.
[19] Ø. Haugen, B. Møller-Pedersen, J. Oldevik, A. Solberg, “An MDA®-based framework for model-driven product derivation”, *Software Engineering and Applications*, USA, 2004.
[20] G. Chastek, J.D. McGregor, “Guidelines for developing a product line production plan”, *Software Engineering Institute*, Technical Report CMU/SEI-2102-TR-006, 2002.
[21] A. Metzger, P. Heymans, K. Pohl, P.-Y. Schobbens, G. Saval, “Disambiguating the Documentation of Variability in Software Product Lines: A Separation of Concerns, Formalization and Automated Analysis", *15th IEEE International Conference on Requirements Engineering (RE 07)*, 15 October 2007, New Delhi, India. IEEE, September 2007, 243-253.
[22] M. Mannion, “Using First-Order Logic for Product Line Model Validation”, *Proceedings of the Second Software Product Line Conference (SPLC2)*, LNCS 2379, Springer, San Diego, CA, 2002, pp. 176–187.
[23] W. Zhang, H. Zhao and H. Mei, “A propositional logic-based method for verification of feature models”, *ICFEM 2004*, volume 3308, Springer–Verlag, 2004, pp. 115–130.
[24] D. Batory, “Feature models, grammars, and propositional formulas”, *In Software Product Lines Conference*, LNCS 3714, 2005, pp. 7–20.
[25] K. Czarnecki, A. Wasowski, “Feature Diagrams and Logics: There and Back Again”, *11th International Software Product Line Conference (SPLC 07)*, 10 September 2007, pp. 23 – 34.
[26] D. Benavides, P. Trinidad and A. Ruiz-Cortés, “Automated Reasoning on Feature Models”, *The 17th Conference on Advanced Information Systems Engineering (CAiSE'05)*, Porto, Portugal. 2005.
[27] D. Benavides, S. Segura, P. Trinidad and A.Ruiz-Cortés, “A first step towards a framework for the automated analysis of feature models”, *Managing Variability for Software Product Lines: Working With Variability Mechanisms (SPLC'06)*, Baltimore. 2006.

Integrated Product Line Model for Semi-Automated Product Derivation Using Non-Functional Properties

Norbert Siegmund, Martin Kuhlemann, Marko Rosenmüller, Christian Kaestner, and Gunter Saake
University of Magdeburg
39106 Magdeburg, Germany
{nsiegmun,mkuhlema,rosenmue,ckaestne,saake}@ovgu.de

Abstract

Software product lines (SPLs) allow to generate tailor-made software products by selecting and composing reusable code units. However, SPLs with hundreds of features and millions of possible products require an appropriate support for semi-automated product derivation. We envision this derivation to be extended by non-functional properties that are associated to code units and domain features. Code units and domain features are commonly organized in different models and connected via complex mappings, what make automation difficult. We propose a model that integrates features and code units in order to allow semi-automated product derivation using non-functional properties.

1 Introduction

Software product lines (SPLs) aim at providing variability for a family of similar software products tailored to individual user needs [12]. Variation points of an SPL, i.e., the functional differences between different product line members [4], are analyzed and modeled during domain analysis as features inside a *feature model* [20, 14]. Based on feature models SPLs are implemented using reusable and modular *code units* that are organized in an *implementation model*. Product line members are *derived* by composing such code units.

In small SPLs, it is usually simple to derive a product by selecting the required features manually. However, as the size of SPLs grows – large SPLs in industry may contain over 1000 features [31, 25] – the derivation process of selecting these features becomes more tedious and difficult, because many decisions are necessary, each requiring knowledge of the SPL's domain and maybe of implementation.

Product derivation becomes further complex in the presence of non-functional constraints, e.g., in domains like embedded systems where resources are restricted. There are many relevant non-functional constraints [19], for example, a generated database management product should have a maximum footprint size of 48 KB to fit on an embedded device and must be capable of handling a throughput of 10 transactions per second (T/s) because input is provided at this rate. To derive a product by configuring hundreds of variation points, that additionally has to adhere to non-functional constraints is difficult and often results in a trial-and-error approach, which is tedious and error-prone.

We envision tool support that assists developers in selecting features to support the product derivation process. For example, tools can automatically hide variation points that are irrelevant because of constraints and features selected earlier inside configuration process. In the following, we refer to this process as *semi-automated derivation (SAD)* [32, 5, 36]. We argue that SAD is particularly promising in the presence of non-functional constraints. For example, tool support could check which features cannot be selected because they would violate a footprint constraint.

SAD tools require domain specific information about the SPL, that come solely from the feature model in existing approaches (i.e., features and constraints between features). However, to define non-functional constraints we need additional information. While some non-functional properties, like development time, can be directly attached to the feature model [5], others, like performance, binary code size, and in-memory size, depend on the implementation and can be associated with code units [36]. Therefore, we have to consider both, feature model and implementation model, for SAD.

Current approaches to SPL development typically use a mapping between feature model and implementation model which makes SAD with non-functional constraints difficult because the intermediate result of selecting code units using non-functional properties in the implementation model must be propagated back to the feature model used for configuration. In this paper, we suggest an *integrated software*

product line model (ISPLM), that combines both, feature model and implementation model, to overcome problems in the SAD process with two models. This model should provide the basis for creating an SAD tool that supports the user in deriving products based on non-functional constraints. In our long term vision, this model enables an adequate handling of large and complex SPLs in resource constrained environments.

2. Background

In this section, we give an overview of feature modeling and current approaches for implementing and configuring SPLs.

Feature Modeling. *Feature-oriented domain analysis* (FODA) [20] is the process of identifying and collecting information relevant for a stakeholder that describe the features of a concrete domain. These features might be modeled with additional information like attributes or annotations [14] and are integrated into a feature model with further domain constraints. Features can be mandatory or optional and may have relations or constraints to other features, e.g., two features can be alternative. The feature model is typically visualized by a feature diagram that is a hierarchical representation of all features of an SPL.

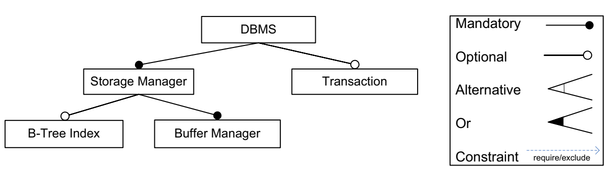


Figure 1. Simple Feature Diagram.

Figure 1 depicts a sample feature diagram of an SPL for a *database management system (DBMS)*. The diagram consists of the base concept DBMS as the root node which represents the core functionality of the DBMS and additional nodes that represent features of the product line. The feature diagram shows that only feature *Storage Manager* is mandatory for every product because of the required storage functionality for every DBMS instance provided by this feature. Feature *B-Tree Index*, which represents a special data structure for accessing data in a DBMS, is optional, i.e., a stakeholder has to decide, whether this special feature should occur in a product. Further relations between features are possible, e.g., *excludes* and *implies*, but not shown in the Figure.

SPL Implementation. Code units implement the features of an SPL [14]. A common practice is the realization of code units using components [12]. A mapping assigns features to code units that implement the according functionality. In general, code units can implement multiple features and crosscutting features may map to several code units [8]. All code units and constraints between them form the implementation model (a.k.a. architecture model [23]).

An important difference between common SPLs and SPLs in the embedded systems domain are alternative implementations. Alternative implementations are required for fine-grained adjustments of non-functional properties by providing equivalent functionality. Figure 2 depicts two different implementations of a B-Tree feature. Component *B-Tree small* implements basic functionality and is optimized for binary code size at the costs of performance. In contrast, component *B-Tree fast* uses special algorithms, e.g., lazy deletion [37, 18], that increases performance at the costs of binary code size. Such a need for specialized algorithms is common in embedded systems [10, 35]. The *Buffer Manager* functionality (cf. Figure 1) provides support for different storage types. It similarly has two variants, one for a simple data handling without any specialized memory structures (*Minimal Buffer Manager*) and one for performance optimized data handling (*Unrestricted Buffer Manager*). The developer has to decide which code unit is optimal for a given environment. As shown in Figure 2 there are constraints between code units as well.

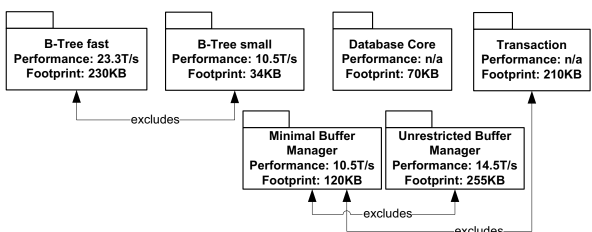


Figure 2. Implementation Model.

Product Derivation. To derive a product of an SPL, the stakeholder decides which features to include. Feature selection is usually realized based on a feature model [21, 1, 2, 27, 29]. During feature selection, tools can check the current configuration against existing constraints that are defined in the feature model and implementation model.

3 Semi-automated Derivation with Non-Functional Constraints

SAD is an approach to assist a user during configuration of a large SPL with many features. This support can be realized by automatically hiding features that cannot be selected at the current state of configuration due to existing

constraints [2, 15, 11, 7]. Other approaches guide the user through the configuration space and further visualize dependencies between features [27, 29]. Our vision goes beyond this derivation process based only on the feature model, i.e., we also want to include non-functional constraints that have to be fulfilled in the derived product.

We envision an extension to the concept of SAD by an automated selection of features and code units according to *non-functional requirements*. Often non-functional properties depend on how a code unit is implemented. Therefore, it should be possible to present hints to a stakeholder during configuration of an SPL which show how a selection affects the properties of the final software. To start the SAD process, a user defines constraints, e.g., *Footprint* $<$ *48KB AND Performance* $>$ *10T/s*, for the resulting software which may already exclude certain features from the configuration space. As a next step, the user selects needed functionality of the SPL. After every decision the SAD tool supports the user by giving hints or automatically selecting features or code units according to the constraints. Thus, the SAD process requires information from the feature model (e.g., features, domain constraints) as well as from the implementation model (e.g., non-functional properties of code units like binary code size and reliability). The *measurement* of such non-functional properties of code units is in the focus of our research, but outside the scope of this paper[1].

4 Problem Statement

In the following we present problems we found that result from the separation of feature model and implementation model.

SAD Tools. During our development of an SAD tool, we observed several problems. When evaluating a user selection, we have to proof this selection against domain constraints defined in the feature model. Afterwards, the tool has to map the current configuration to the implementation model. Again, we have to proof the same user selection of the feature, but now against the implementation model, because of implementation constraints, e.g., it has to be validated if *excludes* relations are violated. Moreover, additional requirements and constraints may result in an automatic selection of required code units that map back to a feature selection. This task is already complex, but the SAD tool, which supports non-functional constraints, has to evaluate the respectively actual configured implementation against the existing non-functional properties. These

[1] As part of the FAME-DBMS project (funded by the German Research Foundation, project no. SA 465/32-1), we work on the derivation of non-functional properties by composing products and measuring the resulting properties.

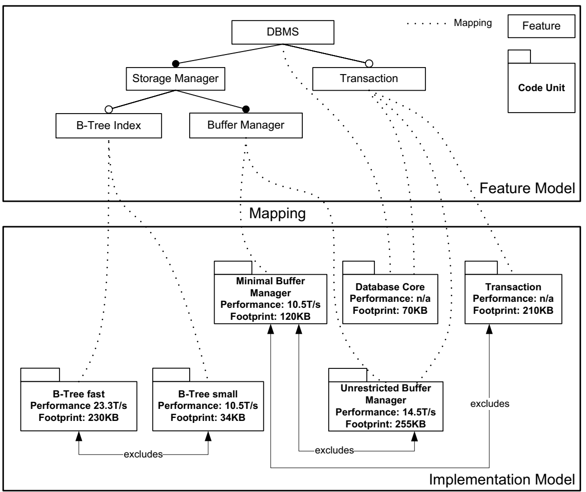


Figure 3. Problems of separated Models.

non-functional constraints can raise conflicts in the implementation selection and therefore, in the feature selection.

Figure 3 shows a simplified abstraction of a mapping between a feature model and an implementation model from our DBMS product line. If we define performance and footprint constraints and select the feature *Transaction*, an SAD tool has to map the selection to three different components. The tool has to check the *excludes* constraints of components *Transaction* and *Unrestricted Buffer* which results in a verification of the incomplete feature selection. Furthermore, the SAD tool has to check that the non-functional constraints are not violated. In this example, the configuration of component *Minimal Buffer Manager* might be changed to the selection of component *Unrestricted Buffer Manager* which leads to a change of the non-functional properties of the current configuration.

The reason for for this complex derivation process lies in a complex interaction of two separated models of *one* SPL that are typically connected via an intricate mapping [9, 34, 23] and have to be consistent. This complexity makes the development of SAD tools costly and time consuming and the SAD process expensive.

Interacting Code Units. Assigning non-functional properties to elements of one model can be difficult if these properties vary depending on the remaining module selection. Reasons for the changing values are mainly code interactions. The interaction code, a.k.a. *derivatives* [24] or *lifters* [26], arise if one feature crosscuts another feature. For example, the code units of *B-Tree fast* and *Unrestricted Buffer Manager* interact by including extra code if they occur in the same product. This is not shown in the diagram, because we use components as code units that usually in-

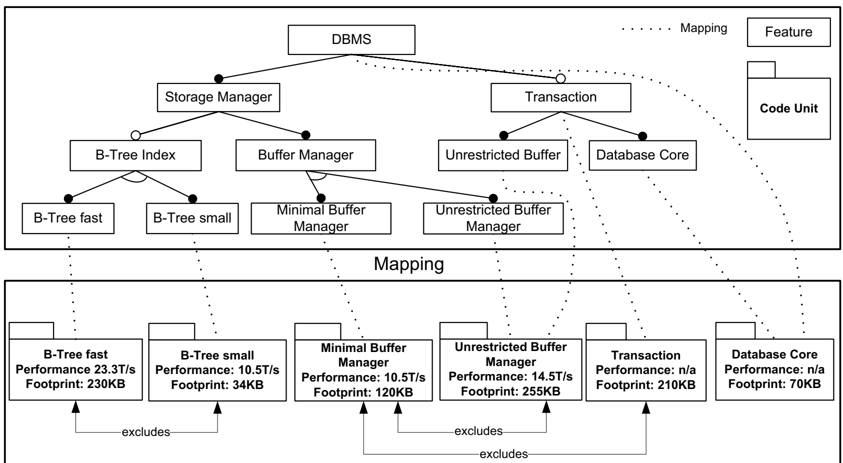


Figure 4. Redundant Representation of Code Units.

clude the whole interaction code so that it does not occur modularly. This additional code may lead to increased binary code size or affect performance. However, interaction code may change non-functional properties significantly, e.g., a *Transaction* interaction code can reduce the performance of the *Buffer Manager* by locking data in the memory. For an SAD tool, it is problematic to derive required non-functional properties if derivatives are not modeled explicitly. Because of the common integration of interaction code in one already existent code unit [22], it is not modeled separately in the implementation model. This results in a lack of expressiveness for the SAD process and therefore, it is a problematic investigation of the varying non-functional properties of interaction code.

Consistency. The problems of SAD tools and consistency described above, typically occur with a complex mapping. A possible solution for selecting alternative implementations during configuration, is the redundant representation of code units of the implementation model in the feature model. For example, the feature *B-Tree Index* could be modeled by two alternative subfeatures (cf. Figure 4) which represent the two alternative *B-Tree* components. This transformation, however, results in a mixture and duplication of both models which raises consistency problems and is error-prone. Additionally, SAD becomes more time consuming because it has to validate the code units twice. Considering changes in one model, like it is common during software evolution, the maintenance of the models becomes difficult. This is caused by the evolution of feature models which is separated from the evolution of implementation models and leads to an increasing mismatch between both models as already investigated by Tesanovic et al. [34].

5 Integrated Software Product Line Model

In the following we present our approach for an *integrated software product line model (ISPLM)*. In particular, we integrate code units into a feature model to improve SAD of an SPL.

5.1 Overview

In Figure 6 we show a meta model for our approach. The ISPLM of Figure 5 consists of one root feature, like feature *DBMS*. The feature *DBMS* has subfeatures that are connected with different relations, e.g., feature *Storage Manager* is mandatory and feature *Transaction* is optional. Our syntax for these constraints is equivalent to the FODA representation of the DBMS domain. We integrate code units into the feature model to represent the features' implementation, e.g., the code unit *Database Core* implements feature *DBMS*. Features and code units in the ISPLM can have non-functional properties as well as relations (*excludes* and *implies*).

The integration of code units into the feature model requires two conditions. First, code units can only be child elements of features or other code units. We do not allow to model a code unit as a root node or as a parent of a feature node because features are defined during the domain analysis which precedes the implementation phase (the feature model written once is solely extended but not changed). Second, we need an additional relation to represent the interaction between code units (i.e., derivatives, cf. Section 4) because of the interacting code units. The *Interaction* relation allows an SAD tool to automatically include the target code unit (filled rectangle) when all interaction sources are configured. In Figure 5, this is the case for the code unit

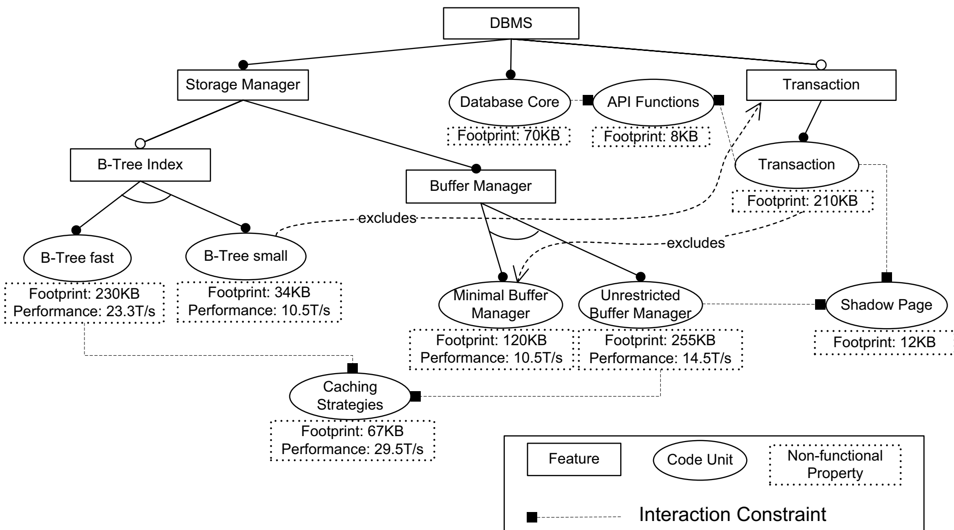


Figure 5. Diagram of the integrated Software Product Line Model.

Caching Strategies. The code unit and its non-functional properties influence the derivation process only if code unit *B-Tree fast* and *Unrestricted Buffer Manger* are selected. Relations can also exist between code units and features, e.g., consider the *excludes* relation between code unit *B-Tree small* and feature *Transaction*. With the ISPLM, it is possible to constrain the variation points of the domain space dependent on a selected code unit.

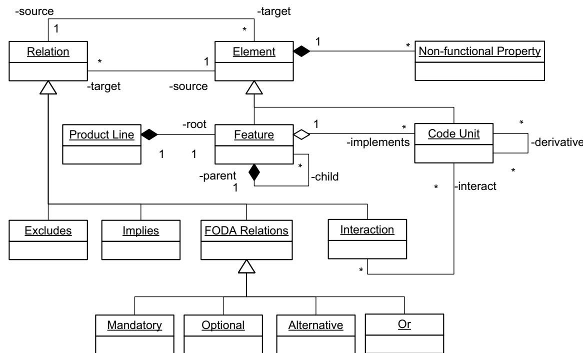


Figure 6. UML Metamodel of the SAD Model.

5.2 Benefits of ISPLM

Semi-automated Derivation. The integration of code units into the feature model allows to perform SAD without the effort of creating and maintaining two consistent models. It simplifies the implementation of SAD tools because information do not have to be propagated between both models. Moreover, representing interactions of code units inside the ISPLM provides an improved SAD because these interactions can be responsible for changes of non-functional properties.

User Benefits. The ISPLM allows a simplified visualization of non-functional properties in a diagram because no abstract and confusing mapping between feature model and implementation model is needed. We reason that this model can provide a basis for collaborating domain engineers and software engineers because the interaction between domain features and code units is apparent with one integrated model. We argue that maintaining one model is less error-prone than two different because both engineers use the same model which enhances the communication during software development. In particular, this additional information is needed for stakeholders who require implementation knowledge in deeply embedded systems and software engineers who require domain knowledge when implementing features.

Furthermore, the ISPLM explicitly allows to configure all variation points (features and code units) instead of only features.

Additional Benefits. Having an integrated model provides consistent changes in the domain as well as in the implementation. When restructuring the domain model the implementation is automatically adjusted, thus software evolution needs less effort. For instance, if the exemplified DBMS must run on an embedded device with feature *Transaction* but cannot fulfill the binary code size constraint, a

new code unit for feature *Transaction* might be needed. A software developer solely has to attach the new code unit to the feature *Transaction* and may define additional constraints to other code units. Additionally, maintenance costs for only one model instead of two independent models might decreased.

5.3 Discussion

The proposed model raises several discussion points, because it contradicts the well known separation of domain model and implementation model.

Separation of SPL Models. The separation of domain model and implementation model has the advantage of supporting independent implementation models by having a constant domain model. Ideally, the stakeholder and domain engineer should not need implementation knowledge nor be restricted by implementation issues. This distinction already blurs during the implementation phase where programming depends on domain modeling. Furthermore, this strict separation cannot be held up because the stakeholder obviously is interested in implementation dependent information, he at least wants to choose out of different implementations. In fact, embedded systems' SPLs need to mix the domain requirements with implementation requirements. Moreover, source code structures (e.g., variables, classes, etc.) could be directly generated from the domain model and their values are set up in the product derivation phase (e.g., configured numeric values). We argue that our SAD model is an appropriate model in areas where implementation requirements are relevant for a stakeholder.

Model Complexity. The integration of two models may increase complexity and size of the ISPLM compared to feature model and implementation model. Large SPLs with hundreds of features and a similar number of code units degrade usability of the whole model. To handle this problem we recommend views that filter only needed information. For example, one view could only represent the features and their constraints (the common feature model) and another view could show all code units including their relations (implementation model). In contrast to separated models the implementation view could contain parts of the domain model that are needed to understand the implementation. We enable the support of views by identifying the source and the target of relations and restrict the position of code units and features (e.g., a feature can only occur as a child of another feature).

6 Related Work

Several researchers aim at simplifying product derivation using SAD [2, 1, 15, 11, 27, 29]. Some researchers also include non-functional constraints for automated reasoning in extended feature modules. A prominent example is the work of Benavides et al. [5, 3, 6, 7], where non-functional properties, e.g., costs of a feature or its development time, are assigned to features. Automated product derivation strongly relates to the well known *constraint satisfaction problem*. This approach laid the basis for our work on SAD.

The next step for SAD is to include non-functional properties of product line code units. White et al. [36] published the tool *Scatter* that integrates non-functional properties into the product derivation process. In particular, *Scatter* includes the binary size of non-code data files (pictures). Products are derived using an extended constraint satisfaction solver presented in [5]. White et al. also investigated that code units can be modeled similar to feature models in the product line architecture model they proposed. In contrast to this approach, we go further to allow SAD with any kind of non-functional property that is related to code units of the SPL. Scatter handle only non-code data files. We evaluate code units of the resulting product instance. Moreover, we enable to represent changing properties of interacting code units (cf. Section 5.1).

Feature modeling gains much attention in recent research. Different feature models and extensions have been proposed, typically for tree-like diagram representation [21, 15, 16], in UML [17, 13], or using the object constraint language [30, 33] to improve domain modeling and domain reasoning, e.g., by adding cardinality and attributes to features. Extensions with attributes can also represent non-functional properties. However, our vision goes beyond just domain modeling and includes code units and their properties (potentially derived automatically) therefore we need an integrated software product line model.

A closely related approach by Reiser et al. [28] formalizes a unified feature model that includes features and code units in the same model. They argue that the heterogeneity of the SPL development process, methods, and tools is difficult to manage. They propose a framework to model artifacts and code units of a product line as an *artifact product line*. In other words the global product line consists of many small product lines which can be further decomposed in even smaller product lines. This approach models the implementation and makes the implementation selectable by configuring these small artifact product lines. However, this framework does not consider any non-functional properties, hence it does not allow the SAD process using non-functional properties.

Ziadi et al. [38] propose the use of UML to derive

products. They translate feature models into an equivalent UML product line architecture model. Based on this architecture model they allow the configuration of product instances. In contrast to our approach they do not consider non-functional properties neither the interactions of code units.

7 Conclusion and Further Work

In this paper we outlined our vision of *semi-automated derivation (SAD)* using non-functional properties of SPL products and discussed difficulties caused by the typical distinction between feature model and implementation model. Product derivation is a complex task if an SPL has hundreds of features and the implementation of features varies. To configure such an SPL with many variation points we propose to use non-functional constraints for supporting a user in product derivation.

We have shown that traditional approaches of modeling domain and implementation separately are insufficient. They do not consider alternative implementations of one feature nor non-functional properties completely. SAD tools have to validate the feature model and the implementation model to enable a configuration which is not always possible because of complex mappings between both models. In contrast, we presented a new *integrated software product line model (ISPLM)* that integrates code units and their non-functional properties into the feature model. We argue that the ISPLM reduces the effort for the SAD process and it improves consistency and evolution management.

In further work, we will continue to develop an SAD tool and implement the ISPLM. This will allow us to analyze and evaluate the resulting implementation effort and benefits in contrast to existing models in a case study.

Acknowledgments

Norbert Siegmund and Marko Rosenmüller are funded by German Research Foundation (DFG), Project SA 465/32-1. The presented work is part of the FAME-DBMS project[2] a cooperation of Universities of Dortmund, Erlangen-Nuremberg, Magdeburg, and Passau funded by DFG.

References

[1] M. Antkiewicz and K. Czarnecki. FeaturePlugin: feature modeling plug-in for Eclipse. In *Proceedings of the 2004 OOPSLA Workshop on Eclipse Technology eXchange*, pages 67–72. ACM Press, 2004.

[2] D. Batory. Feature Models, Grammars, and Propositional Formulas. In *Proceedings of the International Software Product Line Conference (SPLC)*, volume 3714 of *Lecture Notes in Computer Science*, pages 7–20. Springer, 2005.

[3] D. Batory, D. Benavides, and A. Ruiz-Cortes. Automated Analysis of Feature Models: Challenges Ahead. *Communications of the ACM (CACM)*, 49(12):45–47, 2006.

[4] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering (TSE)*, 30(6):355–371, 2004.

[5] D. Benavides, A. Ruiz-Cortés, and P. Trinidad. Automated Reasoning on Feature Models. *Advanced Information Systems Engineering: International Conference (CAiSE)*, 3520:491–503, 2005.

[6] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Corts. A first Step towards a Framework for the Automated Analysis of Feature Models. In *Managing Variability for Software Product Lines: Working With Variability Mechanisms*, pages 45–53, 2006.

[7] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Corts. FAMA: Tooling a Framework for the Automated Analysis of Feature Models. In *Proceeding of the First International Workshop on Variability Modelling of Software-intensive Systems (VAMOS)*, 2007.

[8] D. Beuche. pure::variants Eclipse Plugin. User Guide., 2004.

[9] D. Beuche, H. Papajewski, and W. Schröder-Preikschat. Variability Management with Feature Models. *Science of Computer Programming*, 53(3):333–352, 2004.

[10] C. Bobineau, L. Bouganim, P. Pucheral, and P. Valduriez. PicoDBMS: Scaling Down Database Techniques for the Smartcard. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 11–20, 2000.

[11] G. Botterweck, D. Nestor, A. Preuer, C. Cawley, and S. Thiel. Towards Supporting Feature Configuration by Interactive Visualization. In *International Workshop on Visualisation in Software Product Line Engineering (ViSPLE)*, pages 125–131, 2007.

[12] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[13] L. M. Cysneiros and J. C. S. do Prado Leite. Nonfunctional requirements: From elicitation to conceptual models. *IEEE Transactions on Software Engineering (TSE)*, 30(5):328–350, 2004.

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

[15] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Staged configuration using feature models. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 266–283, 2004.

[16] K. Czarnecki, C. H. P. Kim, and K. T. Kalleberg. Feature Models are Views on Ontologies. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 41–51, 2006.

[17] H. Gomaa. Designing Software Product Lines with UML. Addison-Wesley, 2004.

[18] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[2]http://wwwiti.cs.uni-magdeburg.de/iti_db/research/FAME

[19] International Organization for Standardization (ISO). ISO 9126 Software engineering – Product quality. ISO/IEC 9126-0, 2006.
[20] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.
[21] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. FORM: A feature-oriented reuse Method with domain-specific Reference Architectures. *Annals of Software Engineering (ASE)*, 5:143–168, 1998.
[22] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, 1997.
[23] U. Kulesza, V. Alves, A. Garcia, A. C. Neto, E. Cirilo, C. Lucena, and P. Borba. Mapping Features to Aspects: A model-based generative Approach. In *Workshop On Early Aspects (EA)*, 2007.
[24] J. Liu, D. Batory, and C. Lengauer. Feature-Oriented Refactoring of Legacy Applications. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 112–121. ACM Press, 2006.
[25] F. Loesch and E. Ploedereder. Optimization of Variability in Software Product Lines. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 161–160. IEEE Computer Society, 2007.
[26] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 419–443. Springer, 1997.
[27] R. Rabiser, D. Dhungana, and P. Grnbacher. Tool Support for Product Derivation in Large-Scale Product Lines: A Wizard-based Approach. In *International Workshop on Visualisation in Software Product Line Engineering (ViSPLE)*, pages 119–124, 2007.
[28] M.-O. Reiser, R. Tavakoli, and M. Weber. Unified Feature Modeling as a Basis for Managing Complex System Families. In *Proceeding of the First International Workshop on Variability Modelling of Software-intensive Systems (VAMOS)*, pages 79–86, 2007.
[29] D. Sellier and M. Mannion. Visualizing Product Line Requirement Selection Decision. In *International Workshop on Visualisation in Software Product Line Engineering (ViSPLE)*, pages 109–118, 2007.
[30] P. Sochos, I. Philippow, and M. Riebisch. Feature-Oriented Development of Software Product Lines: Mapping Feature Models to the Architecture. In *Net.ObjectDays*, pages 138–152, 2004.
[31] M. Steger, C. Tischer, B. Boss, A. Müller, O. Pertler, W. Stolz, and S. Ferber. Introducing PLA at Bosch Gasoline Systems: Experiences and Practices. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 34–50, 2004.
[32] D. Streitferdt, M. Riebisch, and I. Philippow. Details of Formalized Relations in Feature Models Using OCL. *Engineering of Computer-Based Systems (ECBS)*, 00:297–304, 2003.
[33] D. Streitferdt, P. Sochos, C. Heller, and I. Philippow. Configuring Embedded System Families Using Feature Models. In *NetObjectsDay*, 2005.
[34] A. Tesanovic and M. de Jonge. Exploring Effects of Feature Mismatch to Evolution of Product Lines with Components and Aspects. In *GPCE Workshop on Aspect-Oriented Product Line Engineering (AOPLE)*, 2007.
[35] R. Vingralek. GnatDb: A Small-Footprint, Secure Database System. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 884–893, 2002.
[36] J. White, D. C. Schmidt, E. Wuchner, and A. Nechypurenko. Automating Product-Line Variant Selection for Mobile Devices. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 129–140, 2007.
[37] B. Zhang and M. Hsu. Unsafe operation in B-trees. *Acta Informatica*, 26(5):421–438, 1989.
[38] Ziadi, Tewfik and Jézéquel, Jean-Marc and Fondement, Frédéric. Product Line Derivation with UML. In *Proceedings Software Variability Management Workshop (SVM)*, 2003.

INCREASING THE RELIABILITY OF MODEL-DRIVEN SOFTWARE FAMILY ENGINEERING AND PRODUCT CONFIGURATION FOR AUTOMOTIVE CONTROLLER SOFTWARE

Frank Grimm
Software Systems Research Centre, Bournemouth University, UK
Email: fgrimm@bournemouth.ac.uk

Abstract

When software product family techniques [5, 16] are applied to software modelling for electronic controller units (ECUs) used in automotive applications, problems arise when concrete software products are to be extracted from models. This so-called configuration process is unsafe and error-prone because software engineers have to decide which classes comprise a certain product and select the correct classes from class repositories. Creating wrong combinations of functionalities has massive negative consequences on safety and reliability of the software itself and may compromise human safety. These configuration problems are discussed in detail in this paper, and a proposal for model extensions that help to constrain class combinations and allow for a safer, more reliable and unambiguous configuration process is made. These model extensions are used to describe formal modelling rules that are used to validate models and configurations based on these models. Thus, dependencies between functionalities of a software product can be fulfilled more reliably. Validated models and configurations can then be used to automatically generate source code that is safer and more reliable than code generated from non-validated models and configurations respectively.

1. Introduction

Cars contain an ever-increasing number of electronic components. Applications of these components are manifold; they range from safety-critical systems such as anti-lock braking systems (ABS) and airbag control to convenience features like parking aid and entertainment systems. All these electronic controller units (ECUs) are controlled by software. Today's cars may contain more than 50 ECUs, and this number is likely to increase in future car generations. Due to the vast number of ECUs, software systems are very complex and will become more complex as even more software-controlled applications are likely to appear in cars. Safety and reliability of today's cars depend to a large extend on safety and reliability of software.

In addition to the engineering complexity that is inherent in automotive software development—automotive electronics are tightly coupled and software controlling them can be looked at as a huge distributed real-time system—, it is becoming even more challenging as time-to-market schedules are becoming tighter and more feature-rich software has to be developed in shorter production cycles.

Another problem that has to be addressed is variability in automotive software systems. Most automotive software manufactures supply not just one but many customers. Albeit customers may use the same type of ECU, their controller software is likely to vary due to different customer requirements. For instance, gentle gear shifting is preferred for one type of cars, but more sportive shifting for another one. To face the variability problem, software product line techniques are commonly used in automotive software engineering [13, 15]. Software product lines consider related products, their commonalities and variability. Variability makes the main difference when comparing product lines with single systems [15]. The ability to include variability in software design models is important to meet the challenging software engineering requirements and customer needs described above.

2. Problem Statement

The approach to model-based software family engineering discussed in this paper is OMOS [2, 3, 11]. OMOS is an industrial approach to software product family engineering in the automotive domain. It is used to model (on the design level) the static structure of automotive software system produced by a global electronics company. It is an object-oriented approach based on UML. UML class diagrams are leveraged to create OMOS models. OMOS models include variation points [12, 13, 14, 17] that allow for the definition of arbitrary variations of certain functionalities that have

to be realised by an ECU software product. In OMOS, variation points are expressed using UML class composition. Creating a composition link between a class representing the whole and a class representing the part (part-class) introduces a new variation point belonging to the part-class. This class introduces the base variant of a specific functionality [4, 5]. By using inheritance to create sub-classes, new variations can be introduced. These sub-classes can make use of functionalities provided by their base class, overwrite these functionalities, and add new functionalities. For example, the model shown in Figure 1 has a variation point called *Wheel* that provides functionality associated with a car's wheel, e.g., measure its revolutions per minute. There is a specialised version of the wheel called *ASRWheel* that provides additional functionality for Anti-Slipping Regulation (ASR). An OMOS model contains all variants of a certain software system. The variants are different in terms of customer requirements, but the overall goal, i.e., to control an ECU, and principal software structure, is the same for all variants.

Analyses of a typical software component that is responsible for just a part of functionality a single ECU showed that even this component has more than one hundred classes representing base variants, and more than three hundred classes in total. Hence, there are three variants for each variation point in average. A configured software product only includes those classes that represent variants required for the specific product. Different variants are included in different products, but all the products are based on the same model that contains all the variants. Thus, variants of a specific functionality are likely to contradict each other.

Problems arise during the product derivation process, i.e., when a specific product is configured. The term *product* is used to refer to the software that meets the requirements of a specific customer for a specific ECU. During product configuration, specific required class variants have to be selected. Selecting the right combination of variants for a certain product is error-prone because of the huge number of variants. Knowledge about dependent variants is currently not explicitly included in models. When a variant of a specific variation point implies a specific variant of another variation point, these dependencies have to be solved by relying on the knowledge of software engineers who have to be aware of implicit dependencies between variants. It is important for software manufactures to be sure that a delivered software product fulfils the customer's requirements and to be able prove this to customers. A more reliable software development process would help to create safer and more reliable software. Hence, restricting the combination of variants in OMOS models by defining explicit dependencies would help to make the configuration process more reliable by reducing ambiguity [6] and by making the rationale behind dependencies explicit.

Unmet dependencies can appear for two reasons. The first reason is variation points that are not directly related, i.e., there exist no associations between them. When looking at the example model shown in Figure 1, selecting *ASRAxle* as the concrete instance of variation point *Axle*, *ControllerWithASR* is needed in order to enable the ASR functionality of the system. Despite of not being directly associated, i.e., operations of other variants are not directly called, unrelated variants may

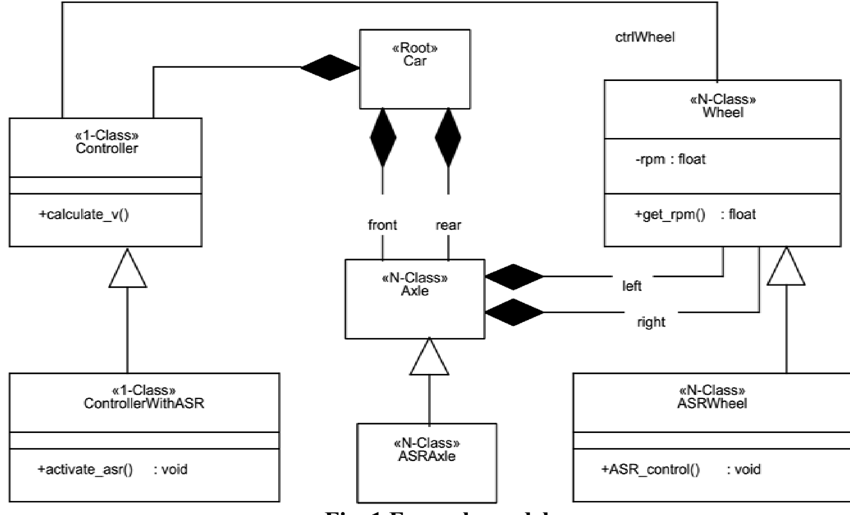


Fig. 1 Example model

implicitly depend on each other because they are connected indirectly. Even if variants are not directly related, the implementation of one variant's behaviour may affect another variant that expects different behaviour. These errors are hard to find because they do not result in compile- or link-time errors. They first appear during run-time, and result in erroneous system behaviour. They are hopefully detected when the system is tested.

The example in Figure 1 is used to explain the second reason why unmet dependencies arise. There are two variation points which are associated, *Controller* and *Wheel*. That is, *Controller* depends on *Wheel* because it calls *Wheel*'s operations. Selecting wrong variants of these dependent variations when a product is configured cannot be prevented by the current OMOS approach. For instance, when selecting the more specialised variant *ControllerWithASR,* which inherits the association to variation point *Wheel,* it needs to communicate with an instance of variant *ASRWheel*, not just *Wheel*, because *ControllerWithASR* uses specific functionality that only the more specialised *ASRWheel* provides. Such configuration errors may be discovered during compile or link time when interface mismatches of dependent classes appear, e.g., *ControllerWithASR* calls operations that the configured *Wheel* does not provide. But interface mismatches do not appear when only implementations differ, so these errors may even be first detected as run-time errors when performing system tests.

Another kind of run-time errors originating from faulty configurations appear when references to communication partners are not configured. In a model, communication between variants is expressed using associations. For example, *Controller* references *Wheel* since an association exists between these classes shown in Figure 1. As references have to be configured on the instance level, models alone do not provide sufficient information because they deal with classes and not with instances. Thus references have to be defined at configuration time specifying the instances of the required variants that are related to each other. For example, a *Controller* instance and four *Wheel* instances are created for a specific product, and one specific *Wheel* instance has to be selected to be referenced by the *Controller* instance. When references are not configured, uninitialized references on the code level will be the result. This in turn may lead to undefined run-time behaviour.

To clarify the impact of these kinds of errors, it is important to understand that parts of ECU software system implementations are automatically generated from their corresponding OMOS models using code generation templates. OMOS models contain classes that represent functionalities, and contain relationships between these classes. A model is used to automatically generate infrastructure code that corresponds with the given model. For each class its corresponding code representation is generated including operations, attributes, and references to other classes (i.e., variants) it communicates with. To create a specific product, its corresponding configuration is used to generate code that creates concrete instances of configured classes, initialises attributes with the values, and initialises references to other instances. All these aspects are defined by a product configuration. So a model and a configuration that is based on this model form the basis for generating code for ECU software systems by using code generation templates.

While code generation templates help to increase code quality due to consistent code structures that are defined by domain experts, and contain their knowledge, templates cannot prevent the error issues addressed above, because they are committed during the configuration process. Since OMOS is a model-driven approach to ECU software engineering, these problems, which are unacceptable in every kind of software system, especially in high-safety environments like automotive systems need to be addressed at the modelling level and not at the source code level. As discussed in this section, OMOS models do not provide enough information to enable a level of safety within models and configurations that allows configuring reliable ECU software products. OMOS models and configurations are transformed into C source code. Thus, it could be argued that using a more reliable and (type-) safe implementation language than C, e.g., Ada, could help to detect more errors during compilation time. But the vast majority of errors results from combining class variants that do not work together because they realise behaviour different from the required variants. These errors won't even be detected when using a different implementation language because this simply cannot be done at compile time.

Two specific meta-models for OMOS models and their configurations are suggested in this paper. By extending OMOS to allow for explicit modelling of dependencies between variants, these meta-models address the aforementioned issues that were identified in current OMOS models and configurations. Based on these meta-models, tools to validate OMOS models and configurations can be developed. These tools allow checking for the correctness of configurations according to the models they are based on. Therefore checking rules can be specified based on theses meta-models. These rules can be used to check for semantic

errors, such as missing communication partners or connecting wrong variants.

Since large parts of the ECU software code are automatically generated based on OMOS models and configurations, quality, reliability, and safety of the software products can be increased because a development process can be established that allows for the validation of models and product configurations. Then source code generation is based on validated models and configurations.

In the following section both meta-models are discussed in detail and it is explained how they help to avoid the problems described above.

3. Meta-Model-Based Approach

As suggested above, specific meta-models for OMOS class models and configuration were created. The meta-model for class models as shown in Figure 2 describes the concepts that are used in the ECU software engineering domain, thus it is a domain-specific meta-model. For example, it contains meta-classes for 1- and N-classes. 1-classes are instantiated at most once in an ECU system, whereas N-classes may have arbitrary instances. 1-classes are common in ECU software systems which usually have limited resources. When compared to N-classes, 1-classes allow for implementation optimisations that result in better run-time characteristics and memory footprint.

The class meta-model also contains meta-classes that describe relationships between classes. Possible relationships are: variation point definition (class composition), variant creation (class inheritance), and class communication. In addition, the meta-model contains meta-classes to specify attributes and operations belonging to classes.

Since the OMOS meta-model is a domain-specific one, it is different to the UML meta-model [9, 10], but an UML profile can be applied to create OMOS

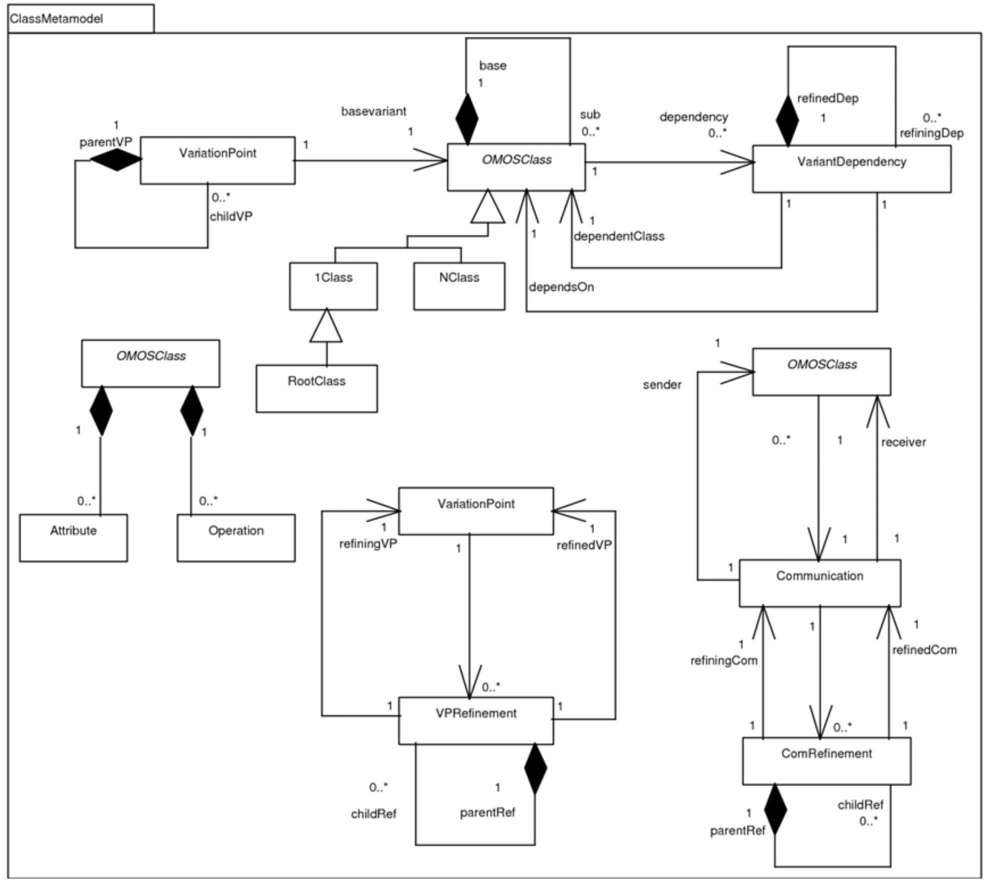


Fig. 2 Class meta-model

models using conventional UML CASE tools. As described in [1] meta-models used to describe specific domain vocabularies can be automatically transformed into UML profiles and vice versa.

The configuration meta-model shown in Figure 3 specifies the entities of the product configuration domain. For example, it contains an element for variation points and associations to describe parent-child relationships between variation points. Since a configuration, i.e., an instance of the configuration meta-model, is based on a class model, i.e., an instance of the class meta-model, elements from the class meta-model are related to the configuration meta-model and re-appear there. Both meta-models include concepts to explicitly manage variability in ECU software systems.

To cope with the first type of configuration errors, i.e., when wrong class variants are included in a product, the class meta-model is extended. It now includes concepts to allow for a variant of a variation point to specify that it depends on a specific variant of another variation point. Thus, engineers are able to specify dependencies between variants in a fine-grained manner. Dependencies can be refined by sub-variants, i.e., sub-classes, to redefine existing dependencies. For a sub-variant it can be specified that it depends on a different, possibly more specific variant than its base variant, which originally defined the dependency. Defining dependencies and refining them is in principal also possible in UML. Hence, these concepts can be implemented in a UML profile that can be created from the OMOS class meta-model.

An extended example, shown in Figure 4, demonstrates the refinement as follows:

1. Variant refinement: A first refinement is introduced to refine the variation point defined by *Wheel* which belongs to *Axle*. When one of the *Axle* variation points, *front* or *rear*, is of concrete type *ASRAxle*, *ASRWheel* has to be instantiated as well. This refinement is represented by the two refined variation points connected to its base variation point using the dotted vertical directed dependency links. Both refinements are instances of the meta-model class *VPRefinement*.

2. Communication refinement: *Controller* is associated with *Wheel*, i.e., it can call operations of *Wheel* or more specified sub-classes, e.g., *ASRWheel*. When *ControllerWithASR* is instantiated, it needs to be associated with an instance of *ASRWheel* to work correctly. This communication refinement is expressed by creating a new association from *ControllerWithASR* to *ASRWheel*. This refinement is linked to the original base association between *Controller* and *Wheel* as shown by the dashed directed dependency link. This dependency link is an instance of meta-class *CommRefinement*.

3. Refinement of implicitly related variation points: When using variant *ControllerWithASR* in a product that requires ASR functionality, *ASRAxle* has to be used as well. Dependencies between variants whose base variants and variation points were not directly related are modelled by creating directed dependency links between the dependent variants. These dependency links are instances of meta-class *VariantDependency*.

Variant and communication refinement could also be derived automatically without explicitly defining such refinements in an OMOS model. This would be possible by interpreting aggregation (whole-part) and

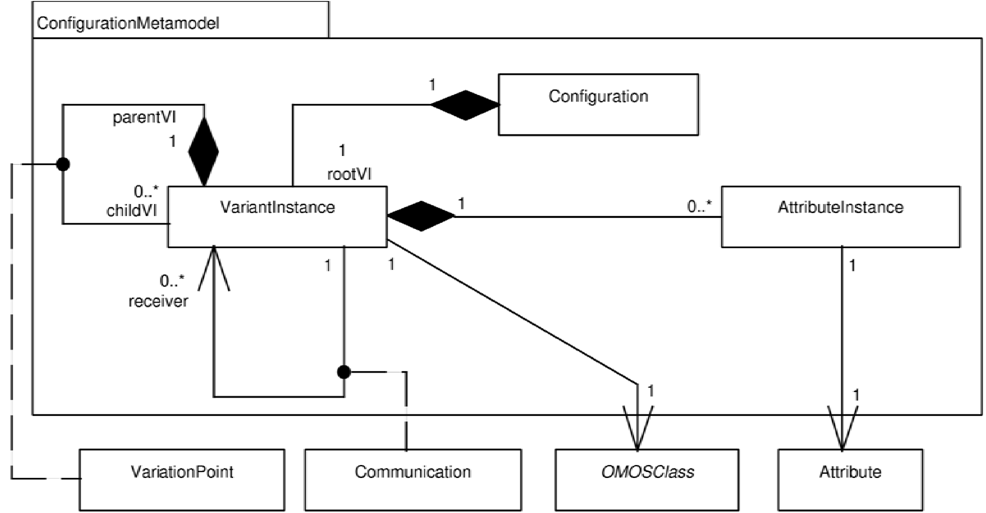


Fig. 3 Configuration meta-model

association relationships in the following way: when a sub-class (e.g., *ASRAxle*) owns an aggregation/association end with the same name (*left*/*right*) as one of its base classes (*Axle*) and the aggregation/association end points to a class (*ASRWheel*) that is a sub-class of the class associated by the base class (*Wheel*), then the sub-class overrides this aggregation/association. Such an overriding can be seen as an implicit definition of a variant and communication refinement, i.e., an implicit dependency definition. However, defining such dependency in the graphical way proposed in this paper has the advantage that these dependencies and refinements are made obvious and users do not need to detect overridden relationships, i.e., implicit refinements, manually.

When compared to the general UML meta-model [10] using domain-specific meta-models has several advantages:

Advantage 1: The model elements represent concepts which originate from the ECU software engineering domain. Thus, engineers, i.e., domain experts by definition, can think, model, and exchange knowledge using the terminology of the domain they are working with every day.

Advantage 2: Both meta-models are moderate in size, and they are not as complex as the UML meta-model. This allows domain experts to thoroughly understand the concepts they use to model ECU software systems and which are described in the meta-models.

The meta-models described above help to solve the problems of variant-rich software systems described above in several ways. First, based on the meta-model entities which represent ECU software system concepts, formal rules can be defined that describe and constrain the usage of these entities. Second, code generation templates that are used to automatically produce code for ECU software systems described by OMOS models can be based on domain entities. As these entities are understood by domain experts, creating templates becomes significantly easier.

Rules can be defined in terms of the domain using considerable smaller meta-models than the UML meta-model. Since domain practitioners understand these meta-models, rules are more expressive for them as they are based on domain entities. Since the class meta-model can be transformed into an UML profile, such rules can be transformed in order to be applied to profile elements as well.

Rules can be defined using the Object Constraint Language (OCL) [8]. Since the Meta Object Facility (MOF) [7] is used as the meta-meta-language of the meta-models, OCL is a perfect fit because it is based on MOF as well.

Rules can be used to check for semantics such as, e.g., naming rules, composition of 1- and N-class types, communication, inheritance, and rules that apply to the signatures of operations. For example the following

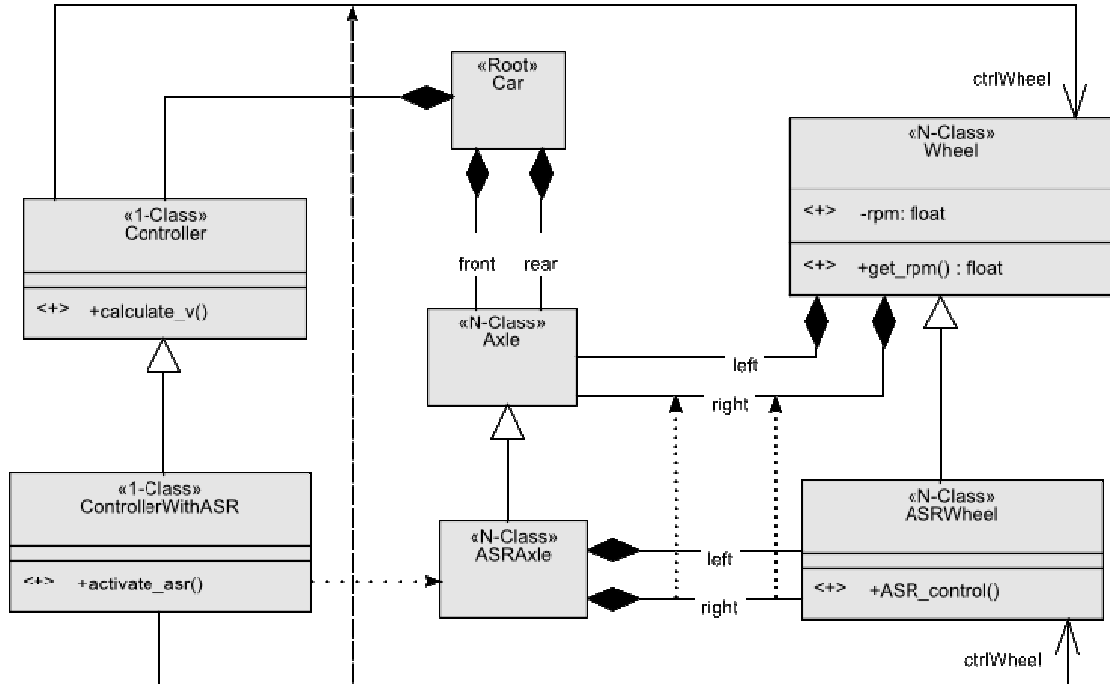


Fig. 4 Example model showing refinement specifications

rule states that communication partners need to be initialised[1]

```
Context ConfigurationMetamodel::
VariantInstance inv:
self.omosclass.communication.receiver
  ->forAll( rcv|self.communication
    [receiver]->size() = 1 and
    self.communication[receiver]
     ->oclIsKindOf(rcv) )
```

This rule applies to (instances of) meta-class *VariantInstance* from the configuration meta-model and navigates to (instances of) meta-class *OMOSClass* from the class meta-model. Hence, both meat-models help to overcome the second type of errors described above, i.e., uninitialized references to communication partners which can lead to undefined runtime behaviour.

4. Tools for validating models and product configurations

Based on both meta-models and on the formal rules that were specified using elements of both meta-models, validation tools can be developed that help to avoid modelling errors. One scenario could be to directly include rules related to modelling into a CASE tool. This would allow for in-place model checking. So errors can be avoided and detected early during the modelling phase. Another kind of validation tool could be a model checker that checks whether models are well-formed before products are actually configured.
After configuration, i.e., before code generation, a configuration checker can be used to verify whether configurations stick to the configuration rules.

Creating validation tools by directly using the OCL rules defined using the meta-models has the advantage, that tools can immediately uses these formal rules. This means that no intermediate step is required to interpret these rules and implement them manually in the tool. Transforming rules manually would be error-prone and could lead to misinterpretations and loss of rules.

5. Related Work

There exist several approaches to software product family/line engineering which focus on the usage of UML to model the static structure by introducing variability concepts [16, 17, 18, 19, 20, 21]. According to [16] only three of these approaches take the process of product derivation into account [16, 19, 20]. In [16] product derivation is realised as a UML model transformation to transform a product line model into a model for a specific product. This approach is based on a UML profile for software product line modelling proposed in [17]. The other two approaches that take product derivation into account do not provide such a detailed definition of product derivations. The variability concepts proposed in [16] and [17] can be applied to UML classes, packages, attributes, and operations. In contrast, the variability concepts presented in this paper can also be applied to whole-part (aggregation) and communication (association) relationship elements. Expressing variability for these constructs is necessary since whole-part relations are very important in automotive systems represented by OMOS models since they are used to define the system's class hierarchy. The class hierarchy is essential during product derivation process. Instances of classes belonging to the model that the product under configuration is based on are created based on the whole-part relationships (aggregations) defined in this model. Starting at the class marked with stereotype <<*root*>>, an instance is created for each aggregated class (part-class) of an aggregation. This process is repeated for all aggregations of each part-class that is selected for the product under configuration.

In [17], UML classes presenting variation points are explicitly marked with the <<*variation*>> stereotype. Sub-classes of such classes can be marked with the <<*variant*>> stereotype to become a variant of the respective variation point. Sub-classes that do not have the <<*variant*>> stereotype are mandatory in all products. A different approach was chosen for OMOS: every class that has sub-classes automatically becomes a variation point in OMOS and every sub-class becomes a variant. While classes presenting variation points in [18] are abstract and thus cannot be included in a product, variation point classes in OMOS do not have to be abstract and can be included in products. In OMOS, a class is made optional by becoming a part-class of a subclass (the whole-class). Since this whole-class represents a variation, it is per se optional; hence, its part classes are optional as well. Due to the strict hierarchical structure of OMOS models, it is possible to define classes as variation points that are part-classes of variation point classes.

The OCL constraints presented in Section 3 are so-called generic constraints [16] because they apply to all OMOS models and present general modelling rules all OMOS models have to obey. Constraints that are specific for a certain model, i.e., product family or line, are expressed within the model using the variability refinement (dependency) elements also discussed in Section 3. Including the constraints directly in an

---

1 The sample rule does not handle refinement and inheritance of communication.

OMOS diagram and defining them in a graphical form is different from the approach taken in [16] where specific constraints for a certain model are expressed by textual OCL constraints. While the textual approach allows for more detailed constraints, e.g., exclusion of certain classes when a certain variant is included in a product, the (graphical) refinement elements presented in this paper could easily be enabled to express such detailed constraints as well.

5. Conclusions

Meta-models that describe elements used to create structural models for ECU software product families in the automotive controller software domain were defined in this paper. These models explicitly contain elements that allow to model for variability which is required to re-use the same model for several software products. These products belong to the same software product family, i.e., they differ in their implementations but have significant commonalities since all of the derived software products are used to control the same type of ECU that performs the same kinds of tasks. By explicitly defining meta-model elements that represent variability in models, and by defining meta-model elements that allow to refine variability by constraining the types of compatible variants, the risk of combining incompatible variants is decreased. Formal rules for the semantics of modelling ECU software systems can be defined based on these meta-models. These rules increase reliability and safety of ECU software products. Since these rules apply on the model level, code that is generated from these models is safer and more reliable than code generated from models that have not been validated. Hence, meta-models and rules based on them allow for software validation in early development phases and strengthen the use of model-driven software engineering for automotive software systems.

6. References

[1] A. Abouzahra, J. Bézivin, M. D. D. Fabro, and F. Jouault. A practical approach to bridging domain specific languages with UML profiles. In *Proceedings of the Best Practices for Model Driven Software Development at OOPSLA 05*, 2005.

[2] W. Hermsen and K.-J. Neumann. Object-oriented modeling concept for software of electronic control units in vehicles. *it+ti - Informationstechnik und Technische Informatik*, 5, 1999.

[3] W. Hermsen and K.-J. Neumann. Application of the object-oriented modeling concept OMOS for signal conditioning of vehicle control units. Technical report, SAE 2000 World Congress, March 2000.

[4] M. Jaring and J. Bosch. Representing variability in software product lines: A case study. In *Proceedings of the Second International Conference on Software Product Lines (SPLC)*, pages 15–36, 2002.

[5] M. Jazayeri, A. Ran, and F. van der Linden. *Software Architecture for Product Families: Principles and Practice*. Addison-Wesley, 2000.

[6] P. Knauber and S. Thiel. Session report on product issues in product family engineering. In *Reviced Papers of the Fourth International Workshop on Software Product-Family Engineering (SPFE)*, pages 3–12, 2001.

[7] OMG. Meta Object Facility (MOF) Core Specification Version 2.0. OMG Document formal/2006-01-01.

[8] OMG. Object Constraint Language (OCL). OMG Document ptc/03-10-14.

[9] OMG. UML Superstructure Specification Version 2.0. OMG Document formal/05-07-04.

[10] OMG. UML Version 2.0 Meta-model. OMG Document ptc/04-10-05.

[11] M. Schweizer and M. Benkel. Development of product families - an example from the automobile industry. Third Workshop on Object-oriented Modeling of Embedded Real-Time Systems (OMER3), 2005.

[12] M. Sinnema, S. Deelstra, J. Nijhuis, and J. Bosch. Covamof: A framework for modeling variability in software product families. In *Proceedings of the Third International Conference on Software Product Lines (SPLC)*, pages 197–213, 2004.

[13] S. Thiel and A. Hein. Modeling and using product line variability in automotive systems. *IEEE Software*, 19(4):66–72, 2002.

[14] S. Thiel and A. Hein. Systematic integration of variability into product line architecture design. In *Proceedings of Second International Conference on Software Product Lines (SPLC)*, pages 130–153, 2002.

[15] J. van Gurp, J. Bosch, and M. Svahnberg. On the notion of variability in software product lines. In *Proceedings of the 2001 Working IEEE / IFIP*

Conference on Software Architecture (WICSA 2001), pages 45–54, 2001.
[16] T. Ziadi et al. Product Line Engineering with the UML: Deriving Products. In *Software Product Lines*. ISBN: 978-3-540-33252-7. Springer Verlag, 2006.
[17] T. Ziadi, L. Hélouët, and J.M. Jézéquel. Towards a UML profile for software product lines. In *Proceedings of the 5th International Workshop on Product Family Engineering (PFE-5)*. Lecture Notes in Computer Science, vol 3014. Springer, Verlag, pages 129–139, 2003.
[18] M. Clauß. Generic modeling using UML extensions for variability. In *Workshop on Domain Specific Visual Languages* at OOPSLA 2001, Tampa Bay, FL, USA, 2001.
[19] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practices*, 1st edn (Addison-Wesley, Reading, MA 1998).
[20] O. Flege. System family architecture description using the UML. In *Technical report, IESE-report no. 092.00/E*, IESE (December 2000).
[21] H. Gomaa. Object oriented analysis and modeling for families of systems with UML. In: *IEEE International Conference for Software Reuse (ICSR6)*, ed by Frakes, W.B., June 2000, pages 89–99, 2000.

Dealing with Changes in Service-Oriented Computing Through Integrated Goal and Variability Modelling

Roger Clotet[1] Deepak Dhungana[3] Xavier Franch[1] Paul Grünbacher[2,3] Lidia López[1] Jordi Marco[1] Norbert Seyff[2]

Universitat Politècnica de Catalunya (UPC)[1]
Barcelona, Spain

Johannes Kepler Universität[2]
Institute for Systems Engineering and Automation
Linz, Austria

Johannes Kepler Universität[3]
Christian Doppler Laboratory for Automated Software Engineering
Linz, Austria

Abstract

Variability modelling and service-orientation are important approaches for achieving both the flexibility and adaptability required by stakeholders of software systems. In this paper we present an approach that integrates domain models captured in the i modelling framework with variability models to support runtime monitoring and adaptation of service-oriented systems. We believe that approaches integrating goal-oriented modelling and variability management are needed to build, operate, and evolve such systems. We illustrate our approach using two scenarios and present a tentative tool architecture based on an existing product line engineering tool suite.*

1. Introduction

Software-intensive systems are characterized by the heterogeneity of the platforms and networks they operate on; the diversity of stakeholders with changing needs; and the dynamicity of their operating environment [7]. Stakeholders of these systems demand properties such as flexibility and adaptability [27] to allow rapid evolution caused by requirements changes, service performance changes, service updates, new (types of) stakeholders, new regulations, etc. Often, these systems cannot be fully specified in advance and are under constant development, so as to be continuously adjusted and adapted to emerging and evolving requirements from their various stakeholders [3]. The Service-Oriented Computing (SOC) paradigm offers a powerful technological solution for conceiving such flexible and evolvable systems. Services are open components that support rapid and low-cost composition of distributed applications.

Adapting a complex software system to different environments and contexts is also a prime goal of variability modelling in product line engineering [23][24]. Variability modelling is an approach fostering software reuse and rapid customization of systems. Not surprisingly, researchers have started to explore the integration of service-oriented systems and variability modelling to support run-time evolution and dynamism in different domains [18][20][26][27]. Integrating variability modelling and service-orientation is seen promising to achieve both flexibility and adaptability.

Dealing properly with changes in large systems, as software-intensive systems are, demands a thorough knowledge of the rationale of decisions, alternatives considered, as well as traceability between stakeholders' goals and technical solution elements. Goal-oriented approaches [19] have been recognized as a powerful technique in this direction [6][13]. Among several existing approaches, the *i** framework [28] is gaining popularity to model service-oriented and agent-based systems [22]. Researchers have started to explore new ways to enlarge the framework with variability modelling capabilities [21].

Pursuing similar goals, we have been using the *i** framework to model a service-oriented multi-stakeholder distributed system in the travel domain [7]. The goal was to validate the usefulness of *i** in this context and to gain a deeper understanding of the dependencies between goal modelling and variability modelling. In an earlier workshop paper [17] we have presented some initial results. In this paper we show how variability can be modelled on different levels of abstraction. We present a set of rules that allow to identify variability in *i** models. These rules, formulated over corresponding metamodels, help to convert *i** models into variability models which can then be refined to allow monitoring and adaptation of service oriented systems. We illustrate the approach using examples and present a tool architecture based on our existing work on meta-tools for variability modelling.

2. Change Scenarios

Our fictitious example is a distributed system provided by Travel Services Inc. (TSI), a company offering services to travellers to search for and book trips online. While some of these services are developed by TSI, most are provided by third party Service Providers. Services range from very simple (e.g., currency conversion) to highly complex ones offering large functionality. For example, the system relies on a payment service provider offering payment services to TSI. Further, a number of travel services, e.g., for booking flights or checking the availability of hotel rooms are used. Various Travel Agencies (TA) contract TSI's software solution to offer a customized online travel platform to their customers.

It is obvious that changes play an important role in such a system. Changes such as new requirements, new types of stakeholders or changes in the environment (e.g., new regulations) occur "top-down" while other changes such as service performance variations or service updates happen "bottom-up".

The following scenarios highlight a top-down and a bottom-up change and illustrate subsequent system adaptation. The first scenario highlights how changing needs of stakeholders make it necessary to adapt the system.

Top-down stakeholder-driven change: TSI has so far been used only by European travel agencies, which rely on a payment service for money transfer within Europe. TSI has won a new TA Transworld Travels that interacts with clients in all continents and therefore needs a worldwide payment solution offering more options despite expected higher costs.

1. The new requirement of Transworld Travels might provoke a change affecting other customer TAs serviced by TSI.

2. TSI's software architect analyses whether the new global payment service is interfering with the intentions of other TA's. He finds out that other TAs and the Payment Service Provider (PSP) are affected by this request.

3. TSI negotiates the change with the other customer TAs. The affected TAs and the Payment Service Provider are invited for discussions. The other TAs concur that they are not interested in the new global service, but accept to use a new PSP if it will not increase their costs.

4. The World-Wide Payment System Berne (WWPS-Berne) is willing to provide a more advanced service to satisfy the requested needs and Transworld Travels is willing to pay the extra cost for this new service.

5. Based on this decision, TSI's software architects confirm that Transworld Travels will use WWPS-Berne. All other TAs will also be switched to WWPS-Berne, but for them the service will only provide reduced functionality. The architect updates the system configuration to address this change.

The second scenario illustrates how a system change can be trigged by a highly dynamic software environment. We assume that customer TAs are either Austrian or Spanish.

Bottom-up monitoring-driven change: In order to ensure high availability of the travel services to its customers, TSI has decided to use two world-wide available Amadeus travel services instances, a Spanish and an Austrian Amadeus server. The system configuration specifies that each Spanish TA normally redirects customer requests to the Spanish Amadeus server and Austrian TAs primarily use the Austrian Amadeus Server. According to the system configuration requests of Austrian customers can be redirected to the Spanish Amadeus if necessary and vice versa.

1. The following day Austria classifies for the Football World Cup to be held at Barbados next summer. All Austrian TAs are experiencing high load on their website as many Austrian customers are eager to book trips. As a result the Austrian Amadeus server is under heavy load and the average response time is increasing considerably. Finally, monitoring tools detect that the average response time is higher than documented by the Service Level Agreement (SLA) between TSI and its customer TAs.

2. The TSI operator is automatically informed that the system is not satisfying the specification and that it is recommended to automatically redirect some Austrian traffic to the Spanish Amadeus server.

3. After a confirmation by the TSI operator the system automatically updates its configuration and redirects some Austrian traffic to the Spanish server.

4. Four hours later, the monitoring tools detect that the load on the Austrian Amadeus server is again stabilized and that its response time satisfies the SLA.

5. The TSI operator is automatically informed by the monitoring system that the redirection of traffic is no longer needed. After the operator's confirmation the system again redirects all traffic to the Austrian Amadeus server.

The two scenarios show that TSI software architect needs support to deal with top-down and bottom up changes. For instance, he needs to understand the common and specific goals of different TSI customers. He also needs to know the current service configuration of customers. In order to determine the best way to proceed, the architect relies on information about alternatives (e.g., the alternative for a certain service in case it fails). The architect also needs traceability information to comprehend the dependencies among goals, service types, and service instances. An up-to-date and detailed representation of the system at different levels, from stakeholders' needs down to the concrete system configuration, is needed to ensure a speedy and correct reaction after top-down or bottom-up changes [7].

We will show that the integrated use of *i** and variability modelling techniques allow to model these different layers together with information traceability and variability information to facilitate system adaptation. In the next section we thus provide the necessary background about *i** and variability models.

3. Background

3.1 The *i** Framework

In [7] we have shown the use of the *i** framework to model all the levels of our distributed system: stakeholder needs, software architecture and running system. The different models are built with similar constructs, although emphasis is different in each case. Figure 1 provides a simplified view of the *i** metamodel [1] only depicting the elements that are relevant for variability modelling.

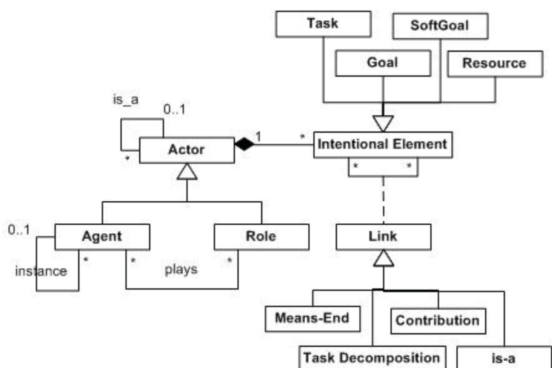


Figure 1. Part of *i metamodel relevant for variability modelling.**

The *i** framework supports the modelling of systems as a network of actors together with their rationale. The network of actors is described using a Strategic Dependency (SD) diagram which shows how actors depend on each others, whilst the rationale is described through a Strategic Rational (SR) diagram for each actor.

Our simplified metamodel describes two kinds of actors: *Roles* are used to represent the system stakeholders and the different "parts" of the system. *Agents* are used to represent the real software components such as services and their instances. Agents can *play* roles and can be *instances* of other agents. There is also an *is-a* relation to create actor hierarchies.

An actor is composed by its *intentional elements*, which are responsible to describe the actor's needs/requirements or responsibilities. There are four kinds of intentional elements: *goals*, *softgoals*, *tasks* and *resources*. The intentional elements inside an actor can be related between them using different kinds of links: *means-end*, *task decomposition* and *contribution* (these links will be used depending on the intentional element type). We allow a fourth link type when there are two actors related by an is-a relation: when the actor *A* is-a actor *B*, some intentional elements of actor *A* can be related to intentional elements of actor *B* using is-a link (see [8] for a precise definition).

3.2 Variability Modelling

Variability modelling is a key technique in product line engineering to define how various products in a product line can be distinguished from each other. Orthogonal variability modelling [2] is based on complementing existing models and artifacts with variability information rather than using specific notations or languages for variability modelling. For instance, John and Schmid [24] have proposed a decision-oriented approach that supports orthogonal variability modelling for arbitrary artifacts independent from a specific notation. Their work is based on earlier work in the Synthesis project [25]. The benefits of decision-oriented approaches are the flexibility gained and traceability established by using one variability mechanisms for different artifacts at the requirements, design, architecture, implementation, application, and runtime level. Our work is based on the meta-meta-model presented in [10] which uses decision and assets as key modelling elements (see Fig. 2). An *asset model* describes the system elements and their dependencies. The decision model describes the variability of the system through a set of decision variables that are used to adapt a system, i.e., to derive a product from the product line. The *included-if* relationship determines which assets will be part of the system depending on the values of the decision variables.

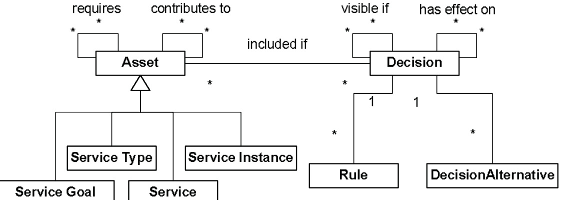


Figure 2. Metamodel for Variability Management.

The first step of the approach presented in [10] is the development of a domain-specific metamodel by identifying the relevant assets and dependencies among them. For this context we identified the following asset types: service goal, service type, service, and service instances: A *service goal* establishes the objective of a service (e.g., "Offer travels"). Different *services types* contribute to fulfilling these goals (e.g., "Travel services provider"). Available services realizing a service type are modelled as

a *service* (e.g., “Amadeus”). Finally, available runtime implementations of services can be modelled as *service instances* (e.g., “Spanish Amadeus Server”). We also identified two kinds of relationships between the assets: The *requires* relationship is used whenever the selection of a certain asset leads to the selection of another asset. This can be a result of logical dependencies between goals, conceptual relationships between service types, relationships between services or functional dependencies between service instances. The *contributesTo* relationship is used to capture structural dependencies between assets of different levels. Service instances for example contribute to services. Services contribute to service types which contribute to goals. It is however also possible for goals to be split up into sub-goals. Such compositional relationships between goals can also be modelled using the *contributesTo* relationship.

A *decision model* is used to model the variability of the system and to describe dependencies between the variation points (cf. Table 1). Decision alternatives describe the range of available options when taking a decision. For example, decision alternatives can be an enumeration of available services. The event of taking a decision triggers the evaluation of attached rules. This includes checking relevant conditions and identifying actions which have to be executed. For example, deciding which travel service shall be used could depend on the average response time of the available services. A condition checking whether the average response time is higher than a predefined threshold could influence the service selection and the identification of actions relevant for service configuration.

Table 1. Partial Decision Model.

Decision Variable	*Decision Alternative*	*Has-effect-on Rule*
typeOfCustomer-Assistence	synch, asynch	
typeOfTravelPayment	credit card, transfer, worldwide	
typeOfService Travel-Provider	flight, hotel, camping	**if** (typeOfService TravelProvider == camping) **then** whichTravel Service:=Amadeus
whichTravelService	Amadeus, Vivaldi	
whichCredit CardSer-vice	CheapCard, Securitas, NorbSecureCredit, FastAndCheap	**if** (whichCredit CardService == Securitas) **then** typeOfIdentifica tion:=FingerPrint
whichAmadeusService	Austrian, Spanish	
AustrianAmadeus-AverageResponseTime	Metric [ms]	**if** AustrianAmadeus AverageRespose Time > 200 **then** whichAmadeus Service:=Spanish

4. Modelling the Variability of Service-Oriented Systems with *i**

In the previous section we have presented two different metamodels that describe the main concepts of the *i** framework and decision modelling. We use each approach to model the same system capturing overlapping but also different information. To guarantee model consistency and traceability we first integrate the two metamodels and then introduce the rules for identifying candidate variation points in *i** models and their transformation to decision models.

4.1 Metamodel Integration

Our decision-oriented variability modelling approach has two main elements: *assets* and *decisions*. Assets are included in the deployed system depending on the decisions that the user has taken, i.e., the values of the decision variables. The different kinds of assets have a direct relation with *i** model elements which is shown in the rest of the section. The decision model is not connected to the *i** metamodel since it is dealing with variability which is not kept in the *i** model (cf. Table 2 this information).

Table 2. Integration of Metamodels.

Variability meta-model element	*i* metamodel element*	*Constraints*
Service Goal	Intentional Element	It is not a resource
Service Type	Roles	Software role
Service	Agents	Is playing a role and is not an instance of another agent
Service In-stance	Agents	Is an instance of other agent

4.2 Identification of Candidate Variation Points

The *i** framework has not specific constructs for modelling variability. Some authors have tackled this issue by extending *i** with explicit constructs (see Sections 6 and 7). Others consider variability as implicit in *i** models depending on the types of modelling constructs used. We adhere to this perspective and aim at identifying candidate variation points by analyzing the very structure of the *i** model. From our analysis of the Travel Agency example, we have identified six different cases of candidate variation points. We present next the different cases classified by the type of *i** construct.

4.2.1 Means-end variability

A means-end link is used to describe different ways to achieve a goal or a task, thus it may be describing a varia-

tion point, usually related to external variability [23], i.e., the variability of artifacts that is visible to customers.

Means-end links are composed as an OR, so at least one of the means should be attained to achieve the end. This OR can be interpreted as a variation point when the customer can decide the way she wants the system to achieve his goal. For instance, customer assistance can be either provided using asynchronous or synchronous support (see Fig. 3).

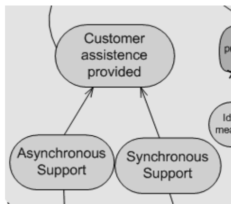


Figure 3. Variation point: Means-end case.

4.2.2 Plays variability

Agents allow modelling real services or components, and these agents play roles. At this point we can find some internal variation points [23] (i.e., the variability of domain artifacts that remains hidden from customers), because the architect can decide among the different services (agents) that can play a role. This variability is represented using the *plays* link. In Figure 4 we find the Amadeus and Schubert agents playing the same role (Travel Services Provider) which means that the architect has to choose between them when deploying the system.

4.2.3 Instance variability

The *i** framework also allows modelling which services instances can be deployed. The example shown in Figure 4 highlights that there are two agents – the Spanish and the Austrian Amadeus Server – as instances of the agent Amadeus. The variability is modelled using the link *instance*.

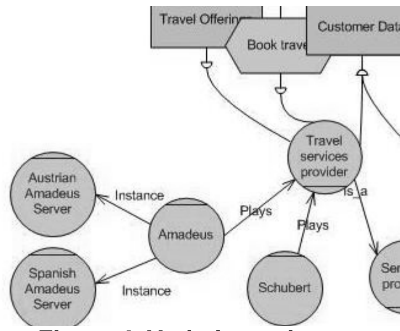


Figure 4. Variation point: Plays and instance cases.

4.2.4 Role inheritance variability

Variability related with architectural features is found in the relationship between actors. In an *i** model, actors can represent the different roles our system has to include. These roles can be classified as a hierarchy using the *is_a link*. For example, Fig. 5 shows a classification for *Travel Payment* role. This example is also an external variation point because the TAs will be able to select a kind of payment they will provide to their customers.

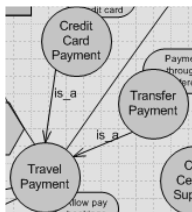


Figure 5. Variation point: Inheritance case.

4.2.5 Intentional element inheritance variability

At a finer-grained scale, a superactor may have different intentional elements refined onto the subactor. This inheritance relationship may take different forms (see [8] for further details). In the case of having more than one heir, the intentional element in the superactor becomes a variation point. For instance, Figure 6 shows how the task *Select Destination* (which is a subtask of task *Buy Travel*) in the general actor *Customer* is refined into its heirs as *Select Destination Country* in *Family* heir and *Select Destination Conference* in *Researcher* heir. In this case there are 2 ways of achieving the task *Select Destination*.

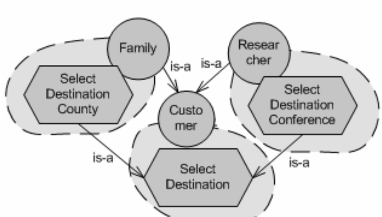


Figure 6. Variation point: Intentional element inheritance.

4.2.6 Softgoal variability

Since the satisfaction of a softgoal is not uniquely defined, we may imagine several criteria acceptable at different moments or contexts. For instance, **Fehler! Verweisquelle konnte nicht gefunden werden.** shows the softgoal *Secure*.

Table 3. Rules for identifying variability in *i models.**

Rule identifier	*Type of variation point*	*Formulation*	*Additional Restrictions*	*Decision Variable*	*Decision Name Prefix*	*Decision Alternatives*	*Cardinality*	*Asset Type*
ME-VP	means-end	$\{x_1, ..., x_n\}$ are means of y	$n > 1$ AND (is-goal(y) OR is-task(y)) AND (not is-resource(x_i))	y	TypeOf y	$\{x_1, ..., x_n\}$	Min: ≥ 0 Max: $\leq n$	Service Goal
P-VP	play	$\{a_1, ..., a_n\}$ play r	$n > 1$ AND is-role(r) AND is-agent(a_i))	r	Which r	$\{a_1, ..., a_n\}$	Exactly 1	Service
I-VP	instance	$\{a_1, ..., a_n\}$ instance a	$n > 1$ AND is-agent(a) AND is-agent(a_i))	a	Which a	$\{a_1, ..., a_n\}$	Exactly 1	Service Instance
RI-VP	role inheritance	$\{r_1, ..., r_n\}$ is-a r	$n > 1$ AND is-role(r) AND is-role(r_i))	r	TypeOf r	$\{r_1, ..., r_n\}$	Min: ≥ 0 Max: $\leq n$	Service Type
IEI-VP	IE inheritance	$\{x_1, ..., x_n\}$ is-a y	$n > 1$ AND is-IE(y) AND is-IE(x_i)	y	TypeOf y	$\{x_1, ..., x_n\}$	Exactly 1	Same as inherited element
SG-VP	softgoal	is-softgoal(y)		y	LevelOf y	Metrics available as fit criterion	Exactly 1	Service Goal

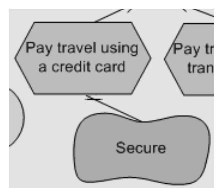


Figure 7. Softgoal variability.

There may be several strategies for satisfying this soft goal. One option could be use a website that implements SSL v3 to transfer credit card information. Another option could be to make a phone call to a TA operator and provide the credit card information. If we choose SSL, it may be sufficient to use an 80-bit key. Advances in cryptology could make 80-bit keys obsolete.

4.3 Building decision tables from *i** models

We have identified a set of rules to generate the corresponding excerpt of a decision model from variation points found in the *i** model. We define this correspondence in terms of the metamodel. The result is summarized in Table 3. Also, in table 1 (see section 3.2) we compile the excerpts of decision models built up from the *i** models presented in 4.2.

Means-end variability rule (ME-VP). This rule is applicable when a goal or a task is the end for more than one means-end link. Only goals, softgoals and tasks are taken into account to know if there are more than one means; we consider that a resource as a means is an information needed to attain the end, not one way to achieve it.

Plays variability rule (P-VP). This rule is applicable when a role is played by several agents in the model.

Instance variability rule (I-VP). This rule is applicable if an agent is instantiated by several other agents in the model. This means that there exist different deployments of the same type of service that may be selected, typically according to SLA clauses.

Role inheritance variability rule (RI-VP). This rule is applicable when there is a classification of roles. This means that there are different kinds of agents, representing the same role. So one or more of the heirs will be chosen depending on the user characteristics.

Intentional element inheritance variability rule (IEI-VP). This rule is applicable for actor classifications using inheritance if some inherited intentional elements are modified in their heirs. In the three cases of inheritance identified in [8] (extension, refinement and redefinition), the intentional element placed in the parent has different ways to be achieved. In the case of extension, the new features are considered as alternatives to the parent. In the other cases, each intentional element declared as an heir is considered a way to achieve the intentional element in the parent.

Softgoal variability rule (SG-VP). This rule is applicable for every softgoal of the *i** model. Since softgoals are high-level concepts, we need here some more concrete fit criterion, e.g. metrics or qualitative reasoning arguments for the particular softgoal. A catalogue of such metrics and techniques would be helpful, and then the different items of the catalogue would be the possible decisions.

It is important to mention that in all of the cases, the rule will be applied only when the decision is relevant (concept of relevance of a variation point). For the sake of an example, consider a role R played by two agents A and B, and assume that the agent A has two instances C and D. Then, the rule I-VP over A is applied only if the decision variable Which R equals A.

5. Tool Architecture

We are currently developing a set of tools for supporting the type of scenarios discussed in Section 2 using the introduced concepts. We are using software components that are part of the DOPLER product line tools suite [11] and plan to extend them with software components for service monitoring and service adaptation. We have shown that we use *i** to create a domain model of our service oriented system. Such domain models can be encoded using *i*StarML, an emerging XML-based standard allowing model exchange among existing tools for *i** [5].

We assume that monitoring and adaptation of a service-oriented system should never be fully automated as user feedback will be required in most domains. This means that we need to provide an interface that enables system administrators to (i) manually modify a service-oriented system by exploiting the known variability or to (ii) confirm changes suggested by the reasoning capabilities of our tool architecture (e.g., when replacing one service with another). Instead of automating the procedure for taking decisions, the proposed tool architecture therefore provides a user console which is able to perform the discussed tasks in an intuitive manner. Similar to work reported in [27] we are adopting the DOPLER component ConfigurationWizard for this purpose.

5.1 Architecture overview

The *variability management engine* is at the heart of our tool architecture. The encoded *i** domain model is used as input for DecisionKing [9], a meta-tool for variability modelling. We are adopting DecisionKing in our architecture to support variability modelling as described in Section 3 [10]. DecisionKing allows managing a variability model of available service instances, services, and stakeholder goals together with trace links. Decisions are used to represent variation points. The rule engine JBoss Rules[1] used by DecisionKing supports reasoning needed to compute the impact of changes stemming from monitored service behaviour or user-triggered adaptations. It is important to note that we do not aim at creating a complete domain-service variability model from an existing *i** model. We can however create candidate variation points based on the rules described in Section 4. These initial rules can be refined using DecisionKing's rule language to express dependencies and constraints more precisely.

The *Monitor component* acquires the monitoring parameters from the variability management engine. It provides measure instruments (MI) for measuring and monitoring different runtime parameters such as response time or availability of services. The *Adaptor component* performs the changes that are required based on the actions carried out by the user. It performs the actual update of the service-oriented system. Both monitors and adaptors are unaware of the concrete technologies used to develop and compose the service-oriented system. These software components define generic extension points that can be implemented to address technology-dependent aspects. Plug-ins are used to connect the generic monitoring and adaptation components to concrete service implementation technologies (e.g., BPEL, WSDL) We are currently developing initial examples of these plug-ins.

The *Services* represents existing deployed services of the service-oriented system. It is important to note that these services are often developed, maintained, and deployed by 3rd party service providers.

Suggestions for actions are presented to the user for dealing with changes in the services. The suggestions are inferred from the variability management engine, which is constantly updated with current monitoring parameters.

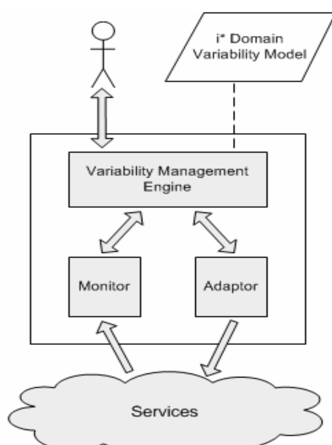


Figure 8. Tool Architecture Overview.

5.2 Revisiting the scenarios

Let's consider the impact of the framework and the usage of the tool by revisiting the bottom-up monitoring-driven change scenario presented in Section 2. We explain in more detail how the framework and tools work.

[1] http://www.jboss.com/products/rules

1. *In order to ensure high availability of the travel services to its customers TSI uses two world-wide available Amadeus travel services instances concurrently (only one is used at the time, each TA has a primary one it is using*)
   a. The technology independent information about the Spanish Amadeus server and Austrian Amadeus server (e.g., service location, service name, etc) is defined in the *i** Domain Variability Model (see Figure 8).
   b. The architect configures the Variability Management Engine (VME) by adding information about these two services.
   c. The architect configures the Monitor by specifying rules and deploying Measure Instruments (MI) to check the running system. The decision model with rules for load balancing is integrated in the Monitor and also in the Adaptor.
2. *The day after Austria classifies for the Football World Cup to be held at Barbados next summer, all Austrian TAs are experiencing high load on their website as many Austrian customers are eager to book trips. As a result the Austrian Amadeus server is under heavy load and the average response time is increasing considerably.*
   d. The corresponding MI inside the Monitor detects a sharp increase of the response time.
   e. The Monitor discovers that the average response time is higher than the threshold established in the SLA.
   f. The Monitor sends a notification to the VME.
   g. The VME revaluates all rules after the notification to compute the effects of the change. The VME suggests and "automatic" switch from Austrian to Spanish Amadeus server for Austrian TA.
3. *The TSI operator is automatically informed by the system about the suggested automatic switching of the service. The operator confirms the switch in his configuration console.*
   h. The TSI operator confirms the change.
   i. The Adaptor is reported to make the change.
   j. The current configuration is automatically updated by the Adaptor to address the service switch. Some of the new requests are redirected to Spanish Amadeus.
   k. In a general case, we may need to inform to Analyzer about new services or new rules for monitoring.
4. *Four hours later, the Austrian Amadeus server is again stabilized and its response time is satisfying the SLA.*
   l. The Monitor, following low-level MI measures, reports a decrease of the response time of the Austrian Amadeus server.
   m. The Monitor realises that the average response time is now below the level established in the SLA.
   n. The Monitor sends a notification to the VME.
   o. The VME revaluates all rules after the notification to compute the effects of the change. The VME suggests an "automatic" switch from Spanish to Austrian Amadeus server.
5. The TSI operator is automatically informed by the system the service switching is no longer needed.
   p. The TSI operator confirms the change to the VME. (He might also decide to update the VME rules to better react to the situation. In this case an updated list of rules is automatically sent to the Analyzer).
   q. The Adaptor is notified to perform the change.
   r. The current configuration is automatically updated by the Adaptor to address the service switch. Every new request is redirected to Austrian Amadeus server.

6. Related work

Managing variability in *i** models is an emergent research issue. Some authors are interested in including the variability information in *i** models, while others focus on analyzing the implicitly captured variability in these models to create the variability model:

For instance, Bibian *et al.* [4] include decision boundaries for the easy identification of goals and the corresponding features. Yu *et al.* [29] include new constructors to distinguish the different types of features (Mandatory, Optional, Alternative and Or). Some authors also include some information about variability constraints by adding new constructs to the *i** language. Specifically, Liaskos *et al.* [21] add new links to model for representing constrains among goals, at level of satisfaction.

Other approaches suggest analyzing models to discover variability. For example, Baxuali *et al.* [15] show how studying the OR-trees goals and the contribution to softgoals can be used for the study of variants (functional behavior) depending on customer priorities (non-functional attributes). Liaskos *et al.* [21], besides introducing constraints to the language, show how goal models can be used to capture variability, constructing variability frames studying the goals semantics.

Even, Baxuali *et al.* present a new idea about using aspect-orientation concepts in goal models to deal with variability [16]. In this paper they also add new constructs such as crosscutting links to the *i** language.

Numerous researchers from different areas have developed approaches and tools contributing to runtime adaptation of systems: In the area of requirements engineering, researchers have explored runtime deviations of systems

from original requirements. In [12] an approach based on goal models specified in the formal language KAOS is presented. The approach adopts a set of agents to monitor runtime behaviour of systems and to suggest either automated or runtime adaptation of the systems. Variability is expressed via alternative refinements in goal models. In [3] different levels of requirements engineering for dynamic adaptive systems have been explored. Their aim is to provide a general framework bridging human-centred requirements and machine-centred adaptation mechanisms. Other approaches combine product line engineering and runtime adaptation of systems. For instance, [18] presents a feature-oriented approach for dealing with runtime adaptation which is based on identifying binding units in feature models that serve as the basis for later reconfiguration. The work of [26] shows how product line architectures can be used to support feature adaptation in the area of Web system personalization.

7. Conclusions and future work

Understanding the dependencies and interactions between goal modelling and variability modelling is important to support runtime adaptation of service-oriented systems. This paper presents how a variability model can be obtained from a goal model without including new language constructs to the goal model to avoid unnecessary complexity.

We have explored the possibility of including the variability explicitly in these kinds of models for instance by adding some graphical information to show which groups corresponds to variation points. Similar to the OR and AND labels for means-end links in Tropos [14] the modeler could include the label VP for variation points. We realized however that it is impossible to add variability information without negative effects on model comprehensibility. We thus decided to complement *i** models with an orthogonal variability modelling technique based on decision models. Some variability can be found in *i** models although it is not explicitly modelled. In our approach we thus decided not to include variability in the goal model but to use a set of rules to extract initial variability models by discovering variability in goal models. While other authors analyze the variability inside actors [15][21][16] our approach focuses on variability stemming from actor relations which are particularly important in service-oriented systems. We present a preliminary set of rules for identifying the variation point candidates. Most of these rules apply to links between actors (is-a, plays, instance). The rules allow deriving an initial variability model (mainly decisions and their alternatives). To define a complete variation model we need to manage dependencies between decisions, rules, and cardinalities which we manage in a separate model outside *i**. Model synchronization is supported by iStarML, a XML-based exchange format for goal models [5].

We are currently developing different elements of the tentative tool architecture presented in Section 5. A critical task in the near future is to validate the architecture using reasonably complex examples. This will also require the development of service monitors for different quality aspects. An interesting challenge is to make the framework technology-independent such that it can be used with different service technologies and monitoring environments.

Acknowledgements

This work has been supported in part by the ACCIONES INTEGRADAS program HU2005-0021 supporting bilateral scientific and technological cooperation between Austria and Spain; and the Spanish projects TIN2007-64753 (ADICT) and SODA FIT-340000-2006-312 (PROFIT programme). The development of the DOPLER tools has been supported by the Christian Doppler Laboratory for Automated Software Engineering.

References

[1] Ayala, C.P., Cares, C., Carvallo, J.P., Grau, G., Haya, M., Salazar, G., Franch, X., Mayol, E., and Quer, C. "A Comparative Analysis of i*-Based Goal-Oriented Modeling Languages". In: Proceedings of The Seventeenth International Conference on Software Engineering and Knowledge Engineering (SEKE'05). 14-16 July, 2005. Taipei, Taiwan, Republic of China. Pages: 43-50.

[2] Bachmann, F., Goedicke, M., Leite, J., Nord, R., Pohl, K., Ramesh, B. and Vilbig, A., "A Meta-model for Representing Variability in Product Family Development", in *Lecture Notes in Computer Science: Software Product-Family Engineering*. Siena, Italy: Springer Berlin / Heidelberg, 2003, pp. 66–80.

[3] Berry, D., B. Cheng, and J. Zhang, "The Four Levels of Requirements Engineering for and in Dynamic Adaptive Systems", 11th International Workshop on Requirements Engineering: Foundation for Software Quality (REFSQ'05), Porto, Portugal, June 2005.

[4] Bidian, C., Yu, E.S.K. "Towards Variability Design as Decision Boundary Placement". Anais do WER07 - Workshop em Engenharia de Requisitos, Toronto, Canada, May 17-18, 2007, pp 139–148.

[5] Cares, C., Franch, X., Perini, A., Susi, A. "Introduction to iStarML". Research report ITC/IRST, 2007.

[6] Castro, J., Kolp, M., Mylopoulos J. "Towards Requirements-Driven Information Systems Engineering: The Tropos Project". Information Systems, vol. 27, 2002.

[7] Clotet, R., X. Franch, P. Grünbacher, L. López, J. Marco, M. Quintus, and N. Seyff, "Requirements Modeling for Multi-Stakeholder Distributed Systems: Challenges and Techniques", 1st Int. Conf. on Research Challenges in Information Science

(RCIS), Ouarzazate, Apr. 23-26, 2007.

[8] Clotet, R., Franch, X., López, L., Marco, J., Seyff, N., Grünbacher, P., The Meaning of Inheritance in i*. 17th International Workshop on Agent-oriented Information Systems (AOIS-2007), Trondheim, Norway, June 11, 2007.

[9] Dhungana, D., Grünbacher, P., Rabiser, R., "DecisionKing: A Flexible and Extensible Tool for Integrated Variability Modeling.", 1st International Workshop on Variability Modelling of Software-intensive Systems, Limerick, Ireland, 2007.

[10] Dhungana, D., Grünbacher, P., Rabiser, R. “Domain-specific Adaptations of Product Line Variability Modeling”, IFIP WG 8.1 Working Conference on Situational Method Engineering: Fundamentals and Experiences, Geneva, Sept. 2007.

[11] Dhungana, D., Rabiser, R., Grünbacher, P., Lehner, K., and Federspiel, C., “DOPLER: An Adaptable Tool Suite for Product Line Engineering”, 11th International Software Product Line Conference (SPLC 2007), Kyoto, Japan, Sep. 10-14, 2007.

[12] Feather, M. S., Fickas, S. van Lamsweerde, A., and Ponsard, C., ”Reconciling System Requirements and Runtime Behavior“, Proceedings of the 9th international Workshop on Software Specification and Design, Washington, DC, April 1998.

[13] Franch, X., Maiden, N.A.M. “Modeling Component Dependencies to Inform their Selection”. In: Proceedings 2nd International Conference on COTS-Based Software Systems (ICCBSS), Lecture Notes on Computer Science 2580, Springer, 2003.

[14] Fuxman A., Liu L., Mylopoulos J., Pistore M., Roveri M., Traverso P. “Specifying and Analyzing Early Requirements in Tropos”. *Requirements Engineering Journal*, 9 (2), 2004.

[15] González-Baixauli, B., Leite, J.C.S.P., and Mylopoulos, J. “Visual Variability Analysis with Goal Models”. Proc. of the RE’2004. Sept. 2004. Kyoto, Japan. IEEE Computer Society, 2004, pp. 198–207.

[16] González-Baixauli, B., Laguna, M.A., Leite, J.C.S.P. “Using Goal Models to Analyze Variability”. First International Workshop on Variability Modelling of Software-intensive Systems (VaMoS 07), Volume 2007-01, pp. 101–107, Jan. 2007.

[17] Grünbacher, P., Dhungana, D., Seyff, N., Quintus, M., Clotet, R., Franch, X., López, L., Marco, J.: Goal and Variability Modeling for Service-oriented System: Integrating i* with Decision Models. In: Proceedings of Software and Services Variability Management Workshop: Concepts Models and Tools. Helsinki, Finland, 19-20 April 2007, pp. 99–104.

[18] Hallsteinsen, S., Stav, E., Solberg, A., and Floch, J., “Using Product Line Techniques to Build Adaptive Systems”, Proceedings of the 10th international on Software Product Line Conference, Washington, DC, Aug. 21-24, 2006, pp. 141–150.

[19] van Lamsweerde, A. “Goal-Oriented Requirements Engineering: A Guided Tour”. In *Proceedings 5th IEEE International Symposium on Requirements Engineering (RE 2001)*, Toronto (Canada), 2001.

[20] Lee, J., and K.C. Kang, “A Feature-Oriented Approach to Developing Dynamically Reconfigurable Products in Product Line Engineering”, Proceedings of the 10th International Conference on Software Product Line, Washington, DC, Aug. 21-24, 2006, pp. 131–140.

[21] Liaskos, S. Yu, Y., Yu, E., Mylopoulos, J. “On Goal-based Variability Acquisition and Analysis”. Proc. 14th IEEE Int’l Requirements Engineering Conference (RE'06) (Sept 11-15, 2006). IEEE Computer Society.

[22] Penserini, L., Perini, A., Susi, A., Mylopoulos, J. "From Stakeholder Needs to Service Requirements". Proceedings of the 2nd International Workshop on Service-Oriented Computing: Challenges on Engineering Requirements (SOCCER), 2006.

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

[24] Schmid, K., John, I., "A Customizable Approach to Full-Life Cycle Variability Management". Journal of the Science of Computer Programming, Special Issue on Variability Management, vol. 53(3), pp. 259–284, 2004.

[25] Software Productivity Consortium, *Reuse-driven Software Processes Guidebook* (SPC-92019-CMC, Version 02.00.03), Herndon, VA, November 1993.

[26] Wang, Y., Kobsa, A., van der Hoek, A. and J. White, “PLA-based Runtime Dynamism in Support of Privacy-Enhanced Web Personalization”, Proceedings of the 10th international on Software Product Line Conference, Washington, DC, Aug. 21-24, 2006, pp. 151-162.

[27] Wolfinger R., Reiter S., Dhungana D., Grünbacher P., and Prähofer H.: Supporting Runtime System Adaptation through Product Line Engineering and Plug-in Techniques. 7th IEEE International Conference on Composition-Based Software Systems (ICCBSS), February, 25-29, 2008, Madrid, Spain.

[28] Yu, E. Modeling Strategic Relationships for Process Reengineering, PhD Thesis, Toronto, 1995.

[29] Yu, Y., Mylopoulos, J., Lapouchnian, A., Liaskos, S., Leite, J.C.S.P.”From stakeholder goals to high-variabilitysoftware designs”. Technical Report CSRG-509, University of Toronto, 2005. Available at: ftp://ftp.cs.toronto.edu/csrgtechnical-reports/509.

[30]

Weaving Aspect Configurations for Managing System Variability

Brice Morin, Olivier Barais[1] and Jean-Marc Jézéquel[1]
IRISA Rennes - Equipe Projet INRIA Triskell
[1]Université de Rennes 1
Campus de Beaulieu
F-35 042 Rennes Cedex

E-mail: {bmorin | barais | jezequel}@irisa.fr

Abstract

Variability management is a key concern in the software industry. It allows designers to rapidly propose applications that fit the environment and the user needs, with a certain Quality-of-Service level, by choosing adapted variants. While Aspect-Oriented Programming has been introduced for managing variability and complexity at the code level, the Software Product-Line community highlights the needs for variability in the earlier phases of the software lifecycle, where a system is generally described by means of models. In this paper, we propose a generic approach for weaving flexible and reusable aspects at a model level. By extending our generic Aspect-Oriented Modeling approach with variability, we can manage variability and complexity in the early phases of the software lifecycle.

1 Introduction

Variability management is a key concern in the software industry. It allows designers to rapidly propose a wide range of applications by choosing adapted variants and options. These customized systems will fit the environment and the user needs, with a certain Quality-of-Service level. In order to improve traceability, reliability and maintainability, variability should be explicitly modeled.

The Aspect-Oriented Software Development (AOSD) paradigm proposes to separate distinct concerns into different aspects *e.g.*, security, logging or persistency, and finally compose them into the base system. It first appeared at the code level [15] and has more recently gained attention in the earlier steps of the software life-cycle [4, 5, 8, 17, 28]: requirement, architecture, design, leading to the creation of numerous ad-hoc Aspect-Oriented Modeling (AOM) approaches, and a dispersion of effort in their tooling, documentation and adoption.

In order to manage variability, recent works [1, 2, 3, 13, 23] discuss the use of Aspect-Oriented Programming (AOP) for implementing Software Product Lines (SPL). At the code level, AOP offers mechanisms to encapsulate (optional) cross-cutting features. In contrast, with more traditional mechanisms like conditional compiling, these features would be tangled and scattered across the program.

Meanwhile, the SPL community points out the needs for managing the variability during the entire software lifecycle [22, 33], this in order to trace variability from requirements to implementation and even execution. The Model-Driven Engineering (MDE) paradigm proposes to consider models as first class entities during the entire life cycle. For example, requirement models [7] represent the user needs, class and component diagrams specify the structure of the system, scenarios and state machines specify its behavior, and runtime models [6] monitor the running system. MDE techniques allow to automate the transition between the different steps of the life cycle. All these models conform to different metamodels, and are generally described by Domain Specific Modeling Languages (DSML), or metamodels.

We argue that Aspect-Oriented Modeling (AOM) can help users to design optional and variant parts of a model, like AOP does at the code level. By weaving incrementally aspects into a base model it is possible to construct a final product step-by-step. But, to be able to weave aspect into different kinds of model, users have to adapt to numerous ad-hoc AOM approaches. Indeed, AOM approaches [4, 5, 8, 17, 28] often propose domain-specific mechanisms to represent aspects. We tackle this issue by automatically generating domain-specific AOM frameworks that all rely on the same concepts. Thus, designers do not need to adapt to a new AOM framework for all the domain metamodels they have to deal with. Weaving aspect

represents the first variability dimension of our approach.

Moreover, AOM approaches are often said flexible and reusable, but actually not enough. Using these approaches, it is often impossible to weave an aspect into a base model if it does not exactly propose what the aspect expects. Additionnaly, when it is possible to weave an aspect into the base model, it is always composed the same way. Based on previous work [18], we propose to integrate variability mechanisms into aspects themselves to tackle the issue of the limited reusability of aspects. These mechanisms turn standard aspects into configurable aspects, more reusable and flexible. Aspect configuration represents the second variability dimension of our approach.

The remainder of this paper is organized as follows. Section 2 presents our generic approach for aspect weaving. Section 3 details our 2-dimension approach for managing variability of software systems. Finally, Section 4 presents related works and Section 5 concludes and discusses future works.

2 Our Generic Model-Driven Approach for Aspect Weaving

This section presents our generic model-driven approach for aspect weaving. It briefly introduces the notion of metamodeling, with a simple running example, and introduces the notion of Aspect-Orientation. Then, it details our approach for automatically generating Aspect-Oriented Modeling frameworks, for any domain metamodel.

2.1 Metamodeling, AOP and AOM

Metamodeling A domain metamodel describes all the concepts of a particular domain of interest, and their relations. To illustrate our approach, we introduce a simple domain metamodel **MM** for state machines, illustrated in Figure 1. A region contains several vertices and transitions, that are the main elements of a state machine. Note that the `Vertex` meta-class is abstract and cannot be instantiated in a model, but is extended by two concrete meta-classes: `PseudoState` and `State`. A transition must declare a source and a target vertex.

This metamodel allows designers to represent state machines, with any number of transitions and vertices, in any configuration. Then, it is possible to simulate models or generate other artifacts, using Model-Driven Engineering techniques and dedicated tools like Kermeta [26], an open-source environment[1] for metamodel engineering. There exists metamodels for state machines, components, scenarios, class diagrams, etc.

[1]available at www.kermeta.org/download

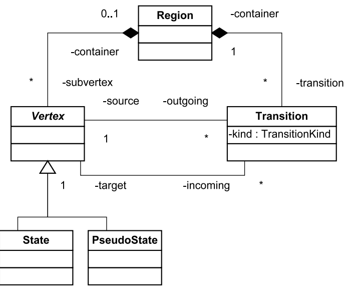


Figure 1. A Domain-Metamodel (MM)

Aspect-Oriented Programming (AOP) The Aspect-Oriented paradigm first appeared at the code level [15] and has been popularized with the AspectJ [14] programming language. AspectJ extends Java with the following concepts:

1. **Join Point**: point of interest in a program *e.g.*, method execution/call, attribute reading/writing.

2. **Pointcut**: it defines a set of join points where the aspect will intervene *e.g.*, all the calls to a given method.

3. **Advice**: it specifies the additional behavior that will modify the base program. It is executed in all the join points identified by a pointcut.

AOP allows users to encapsulate cross-cutting concerns into advice, and implicitly weave them into a base program, in all the join points identified by a pointcut. AOP significantly reduces the complexity of softwares a the code-level, by limiting the scattered and tangled code.

Aspect-Oriented Modeling (AOM) At a model level, AOM approaches [5, 10, 18, 29, 32] propose to encapsulate cross-cutting and reusable concerns. AOM concepts are comparable to AOP ones. But, as opposed to AOP, AOM mainly focus on the composition of structural and behavioral models, in the early phases of the software lifecycle, before implementation.

Template models represent what the aspect expects from the base model *i.e.*, the model elements needed to be able to weave the aspect into the base model, and their relations. Template do not need to be consistent models, for example, it can only be composed of a single operation, without representing its containing class, that is normally mandatory. Templates could be assimilated as pointcuts.

Then, aspect are woven into a base model. This is similar to advice weaving in AOP. On the one hand, symmetric AOM approaches [5, 10, 29, 32], that do not differentiate aspect and base, propose to systematically merge

all the corresponding concepts, and specify how to introduce non-shared ones. On the other hand, asymmetric approaches [18, 24, 30], that clearly differentiate aspect and base, propose to specify how to integrate the aspect. Generally, symmetric composition is a better way to compose homogeneous views of a given system, using a partially automated procedure, whereas asymmetric composition is a better way to introduce new concerns into models, and often offers better reusability, but the composition protocol must be explicited.

In the remainder of this paper, our running example focuses on state machines. However, our asymmetric approach is completely independent from any domain metamodel.

2.2 Generating the pointcut language

The previously introduced domain metamodel allows users to design consistent state machines, but it is too restrictive for designing aspects. For example, a template model might only be composed of a region with a vertex (whatever its type), a final state and a transition that links the vertex to the final state. We may also want the vertex to be an indirect source of the transition *i.e.*, it is possible to fire the (dashed) transition from vertex, directly or not. This model does not conform to **MM** because the `Vertex` meta-class cannot be instantiated and **MM** does not consider the semantic notion of indirect source. This template model is illustrated in Figure 2. Moreover, we want to be able to declare some elements as roles *i.e.*, elements that must be substituted by actual base model elements, whereas other elements can be seen as structural constraints that the pattern must respect. For example, if we want to modify all the *quitTransitions* from the region, we will declare the transition and the region as roles to be able to manipulate them. The other elements are just structural constrains: the transition must link a vertex to a final state.

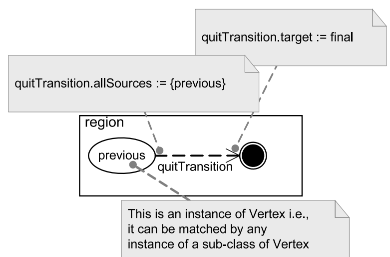


Figure 2. A Simple Template Model

In order to be able to describe more easily target models, we construct on demand a more flexible metamodel [27] **MM'**, using a model transformation written in Kermeta [26]. **MM'** is equivalent to **MM**, except that:

1. No invariant or pre-condition is defined in **MM'**;
2. All features of all meta-classes in **MM'** are optional;
3. **MM'** has no abstract element.

This model transformation is generic because instead of manipulating the domain elements (Vertex, Transition, . . .), it manipulates higher-level concepts provided by ECore, MOF or EMOF for describing metamodels. Consequently, **MM'** can be generated for any input metamodel **MM**. Figure 3 illustrates the result of this transformation applied to the metamodel for state machines (Figure 1).

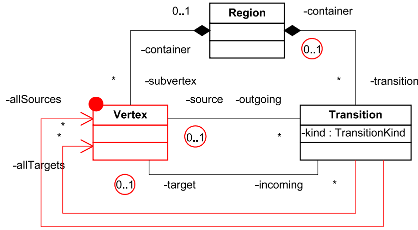


Figure 3. The Unconstrained Metamodel MM'

In **MM'** (Figure 3), we can see that a transition can declare no source/target vertex and can be instanciated without its containing region. Moreover, if a user wants to match a vertex, whatever its real type, he can now instantiate the `Vertex` meta-class. Additionally, we introduce two semantic associations (`allTargets` and `allSources`) to represent states (in)directly after or before a given transition. We also weave these associations as derived properties into **MM** to be able to compute all the state before/after a transition. Note that weaving derived properties does not change the metamodel, it only adds semantic.

In order to ease the detection of model elements that can match roles, we use a Prolog-based pattern matching engine [27], implemented in Kermeta [26]. The domain metamodel is automatically mapped onto a Prolog knowledge base. Then, patterns with roles are transformed into a Prolog queries over this knowledge base. Finally, the Prolog results are converted back into a Kermeta data-structure. This process is totally hidden from the user who only designs model snippets like the one presented in Figure 2.

2.3 Generating Adaptations

The second step of an Aspect-Oriented approach is the weaving process. It consists in composing aspects into the base model, at the places identified by the template model. The key concept is the adapter [18, 19], that describes the aspect structure (**what** will be woven), a template model with roles(**where** it will be woven) and a composition protocol (**how** it will be woven). The composition protocol is

described by adaptations, that are weaving operations manipulating the concepts of the domain. For example, in the context of class diagrams, an adaptation can add a super class to another class, introduce methods or attributes in a class, etc. These concepts are structured in the adaptation metamodel illustrated in the top-part of Figure 4.

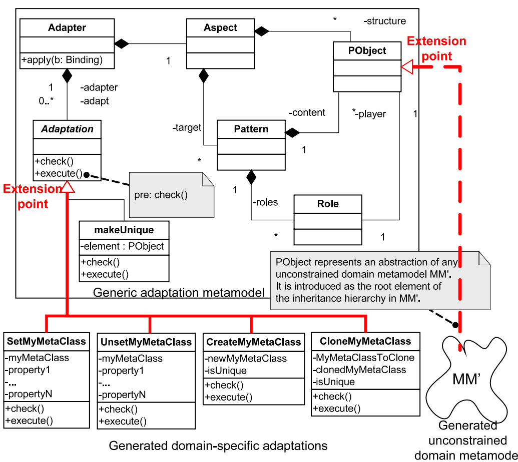


Figure 4. A Framework for Aspect Weaving

This metamodel is composed of three parts: *i)* a generic part describing the concept of adapter, adaptations and aspect (structure and target), *ii)* the unconstrained metamodel **MM'** that is linked to the generic part by introducing the meta-class `PObject` as the root element of **MM'**, and *iii)* domain specific adaptations extending the generic meta-class `Adaptation` (bottom-part of Figure 4).

We propose a systematic way to generate domain-specific adaptations. For each meta-classes `MyMetaClass` of a metamodel **MM**, we generate four adaptations:

1. **SetPropertiesOfMyMetaClass**: this adaptation allows user to set or update (addition) any property of `MyMetaClass`. For example, **SetPropertiesOfRegion** allows designers to add states and transitions in a region.

2. **UnsetPropertiesOfMyMetaClass**: this adaptation allows user to unset or update (removal) any property of `MyMetaClass`. Similarly, **UnsetPropertiesOfRegion** allows designers to remove states and transitions in a region.

3. **CreateMyMetaClass**: this adaptation allows user to create a new instance of `MyMetaClass`. It is generated only if `MyMetaClass` is concrete. For example, **CreateState** allows designers to create a new state, that can be manipulated in the remainder of the composition protocol.

4. **CloneMyMetaClass**: this adaptation allows user to clone an existing instance of `MyMetaClass`. It is generated only if `MyMetaClass` is concrete. Similarly, **CloneState** allows designers to clone an existing, and manipulate it.

The generation of these adaptations is also generic and can be done for any metamodel **MM**: we navigate the meta-classes and their properties and use Kermeta Emitter Template (KET) to generate all the above adaptations, specific to a domain metamodel. In our approach, we use KET to generate Kermeta files, but we can generate any kind of files such as Java code or textual documentation, by defining template. A template describes the structure of the output files (Kermeta, Java, text, etc), and the navigation is written in Kermeta, encapsulated in specific marks[2].

All these generated adaptations can manipulate elements from the template model or from the aspect structure *i.e.*, composition protocols written with these adaptations are totally independent from any base model, and can be reused in different contexts.

This section briefly exposed the principles of our generic model driven approach for aspect weaving. Our approach can be customized for any domain metamodel, to obtain a domain specific AO framework, through two extension points (Figure 4):

1. **PObject**: represents an abstraction of any (unconstrained) domain metamodel that allows us to describe the adaptation metamodel with no domain concepts. When we specialize the framework for a given domain, `PObject` is automatically introduced as the root meta-class of all the element of **MM'**, with a model transformation written in Kermeta [26].

2. **Adaptation**: represents an abstraction of any domain-specific weaving operation. All the domain-specific adaptations must extend this meta-class, declare some attributes, and implement the *execute* method that describes a composition between some model elements. We automatically generate some basic adaptations, but designers can create some additional adaptations that extends `Adaptation`, or modify existing ones.

3 Two-Dimension Variability Management

In the previous section we present our approach for generating Aspect-Oriented Modeling frameworks, for any domain with a well defined metamodel. In this section, we extend these AOM frameworks and describe our 2-dimension approach for managing the variability of software systems.

[2] similarly to Java code encapsulated in JSP or JET

The main idea is that each aspect is considered as a variability dimension *i.e.*, aspects integrate variability mechanisms to make them configurable and reusable in different contexts. Then, different configurations of an aspect can be woven (or not) in order to propose different variants of the system.

3.1 Variability Mechanisms for Aspects

The variability mechanisms we propose to integrate in the aspects are inspired by SPL approaches [31, 34][3]:

- **Alternatives/Variants**: specify that there exist several possible ways to compose the aspect (composition variability) and/or several different places where to compose it (targeting variability). All the variants are exclusive *i.e.*, we can only choose exactly one variant per alternative.
- **Options**: specify that some adaptations may be executed or not, and that some elements from the template model are not mandatory *i.e.*, they may be present or not in the base model where we want to weave the aspect.
- **Constraints**: control the variability mechanisms and limit the number of derived aspects to sensible ones. Without constraints, the number of possible combinations may become huge, and most of them would not be sensible. For example, we can easily imagine that some options or variants require (dependency) or exclude (mutual exclusion) some others.

We propose variability both for the composition protocol and for the targeting. For composition variability, we only need to apply the above concepts on adaptations, and integrate them in the adaptation metamodel. For the targeting variability, **MM'** does not allow designers to propose the full possible range of variability in their snippets because it is not possible to propose variants on certain features that have for example a [0..1] cardinality. For example, we can imagine that we want to instantiate a transition that targets either a pseudo-state or a state, and not simply a vertex, because the composition protocol uses adaptations that are specific to pseudo-state or state, in two distinct variants of an alternative. In order to propose variability in the target model, we propose to generate the maximum metamodel **MM"** that is equivalent to **MM'**, except that all the features can be multiple *i.e.*, all the upper bound are set to * (possibly infinite).

In order to allow composition and target variability, we extend the adaptation metamodel (Figure 4) presented in Section 2 with the following key concepts (see Figure 5):

[3] see http://www.sei.cmu.edu/productlines/ and http://www.splc.net

- **Derivable Adapter**: a derivable adapter is an adapter that contains variability *i.e.*, alternatives, options and constraints. It proposes both composition and targeting variability.
- **Adapter Element**: an adapter element is an element that can be optional or involved in an alternative: adaptation, target, alternative, conjunction (group of adapter elements). It is introduced as a super metaclass for all these elements.
- **Alternative**: an alternative describes several possible variants that are mutually exclusive. Each variant is an adapter element.
- **Constraints**: a constraint describes either a dependency or a mutual exclusion between some adapter elements. A dependency specifies that a source element requires some other elements, and an exclusion specifies that some elements are mutually exclusive *i.e.*, two elements cannot be present at the same time, after derivation.
- **Derivation**: a derivation allows designers to derive a derivable adapter *i.e.*, to fix variability. It allows designers to select options, and choose one variant, for each alternative.
- **Conjunction**: a conjunction is a block of dependent adapter elements. It allows to define optional blocks and variant blocks in an alternative.

To illustrate some of the variability mechanisms, the generated adaptations and the target model specific to state machines, we will describe an aspect that adds a *Log* state before reaching the final state *e.g.*, for logging errors. Optionally, we propose to come back to a previous vertex after logging an error, instead of reaching the final state. The target model and the aspect structure are the model snippets shown in Figure 6.

In the target model, the containing region, the final state, the transition that targets the final state are mandatory *i.e.*, they must be matched by actual base model elements before weaving the aspect into a base model. An option specify that we can target any vertex before the transition. All the elements of the target model are associated to roles, because we want these elements to be bound to base model elements, in order to modify the base model.

Now, we need to define the composition protocol that will describe how the structure will be woven into any base model. Note that the composition protocol is totally defined with elements from the target model and from the aspect structure *i.e.*, it does not reference elements from any base model. This protocol is illustrated in Figure 7.

The composition protocol describes the operations needed for integrating the aspect. In this example, all the

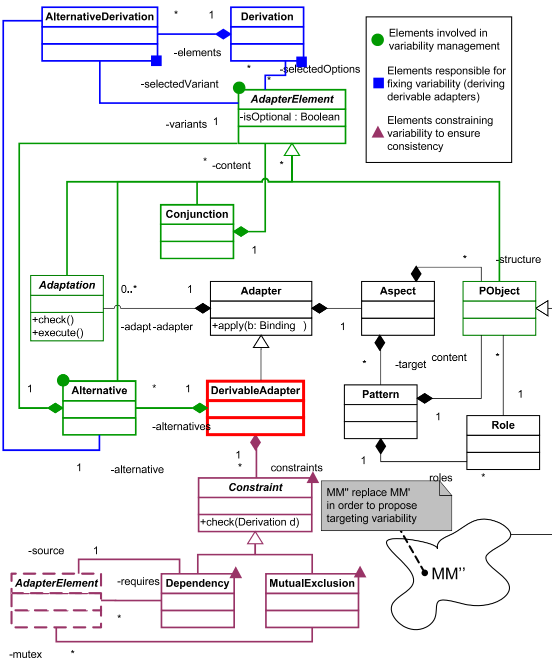


Figure 5. Extended Adaptation Metamodel

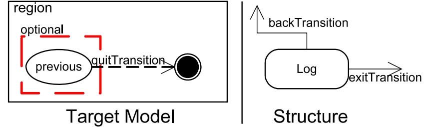


Figure 6. Target Model and Structure

adaptations are `Set*` adaptations because the aspect only adds model elements that exist in the aspect structure.

The concrete syntax we propose for adaptations is very basic. For example, the first adaptations (Line *01*) is called **introduceStruct** and its real type is `SetRegion`. Its first parameter is the region to set (Line *01-a*), and all the following parameters refer to the element we want to introduce in the targeted region: some subvertices (Line *01-b*) and transitions (Line *01-c*). The three following adaptations aims at connecting the transitions (Lines *2* and *3*) and renaming the *Log* state (Line *04*) to fit its context. Finally, we declare an optional conjunction (Lines *5*) that aims at introducing and connecting the *backTransition*.

Note that MDE tools like Sintaks[4] [25] can easily bridge abstract syntax (metamodel) and concrete syntax (text), by parsing texts into models, and transforming models into texts, according to rules defined in a Sintaks model.

[4]available at http://www.kermeta.org/sintaks

```
01 Adaptation introduceStruct SetPropertiesOfRegion
  a - region: target.region
  b - subvertex: {struct.Log}
  c - transition: {struct.exitTransition}
02 Adaptation setExitTrans SetPropertiesOfTransition
  a - transition: struct.exitTransition
  b - target: target.finalState
03 Adaptation setQuitTrans SetPropertiesOfTransition
  a - transition: target.quitTransition
  b - target: struct.Log
04 Adaptation setLogName SetPropertiesOfState
  a - state: struct.Log
  b - name: target.exitTransition.name + ``Log''
05 Conjunction backToPrev is optional {
  06 Adaptation introduceBack SetPropertiesOfRegion
    a - region: target.region
    b - transition: {struct.backTransition}
  07 Adaptation setBack SetPropertiesOfTransition
    a - transition: struct.backTransition
    b - target: target.previous
  08 TargetRef target.previous
}
```

Figure 7. Composition Protocol

3.2 Weaving Aspect Configurations

The previous sub-section details the first variability dimension *i.e.*, the integration of variability mechanisms into aspects. This sub-section details the second variability dimension: the configuration, or derivation of aspects and the weaving process.

The aspect presented in the previous sub-section (Figures 6 and 7) can be configured in two different ways, and consequently there are three possible variants:

- **Variant 1**: Do not weave the aspect
- **Variant 2**: Just add the *Log* state
- **Variant 3**: Variant 2, and we add a transition back to a previous state

If we consider several aspects, we can easily propose many different variants of the system by configuring aspects and weaving them, or not, into the base system. The derivation process can be summarized as follows:

1. Constraints: we check that the derivation *d* provided by the user respects all the constraints of the derivable adapter. We just call the *check(d)* method for all the constraints of the adapter, that is implemented directly in the adaptation metamodel (see `Constraint` in Figure 5), with Kermeta. If one constraint is not reached, the framework raises an exception telling the user that his derivation is not well-formed.
2. Adaptations: the composition protocol (adaptations) of the derived adapter is built in a positive way *i.e.*, selected options and variants are added into the derived adapter.
3. Target Model: the target model is (un)built in a negative way *i.e.*, the model elements that are not selected

(non-chosen options and variants) are deleted from the target model.

4. Post-condition: after derivation, the target model must conform to **MM'**, and not only to **MM"**. Otherwise it means that a cardinality is over the maximum bounds, and consequently the target model cannot be matched by any model snippet.

When an aspect is successfully configured, it can be woven into a base model, following this process:

1. Binding phase: the user provides a binding that links target model elements to actual base model elements. Note that bindings can automatically be found/checked using the pattern matching framework of Ramos *et al.* [27], to guide the user.

2. Weaving phase: for each binding selected by the user, we apply the composition protocol. In the adaptations, the target model elements are substituted with their corresponding actual base model elements, according to the binding. Between each binding, some elements of the aspect structure, or cloned/created elements (`Clone/Create*` adaptations), can be cloned, or remain unchanged. This choice depends on whether the user wants to use the same instances or introduce new instances, for each binding.

3. Post-condition : after composition, the modified base model must conform to **MM**, and not only to **MM'** or **MM"**. Otherwise it would mean that the composition protocol violates some constraints (*e.g.*, it removes mandatory features), or adds too many elements. In this case, we roll back to the initial base model.

The process can be applied several times and is potentially infinite and/or nondeterministic: if we consider that the process has been applied (n-1) times, we denote resp. *configuration*^{n}, *binding*^{n}, *weaving*^{n}, resp. the aspect configuration, the chosen binding and the result after weaving, for the n-th time. We have: *binding*^{n}=f(*configuration*^{n}, *weaving*$^{n-1}$) and *weaving*^{n}=g(*configuration*^{n}, *binding*^{n})=h(*configuration*^{n}, *weaving*$^{n-1}$). The configuration of an aspect may change the target model, and the previous weaving modify the base model, and potentially adds/removes possible targets, so the binding is dependent from the configuration and the previous weaving. The weaving depends on the aspect configuration (the selected adaptations) and on the selected binding, and consequently, it depends on the previous weaving. For this reason, the process is not fully automated: the user configures the aspect, then he chooses the binding and the composition protocol is applied. Next, he can reconfigure the aspect, choose another binding, etc.

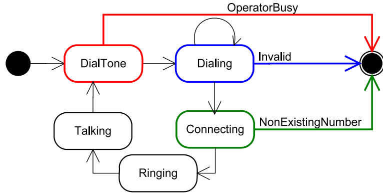


Figure 8. Basic behavior of a phone

Figure 8 illustrates a base model representing the behavior of a simple phone.

Figure 9 illustrates the composition of the aspect when no option is selected. In this case, we only introduce a *Log* state before reaching the final state.

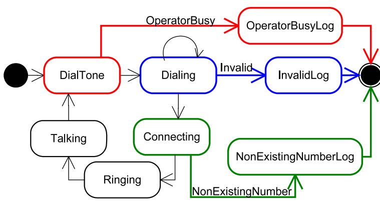


Figure 9. Behavior of a phone with error logging

Finally, Figure 10 illustrates the composition of the aspect when the option is selected. In this case, we also introduce a *Log* state before reaching the final state. Additionally, we introduce a roll-back transition that targets a previous state.

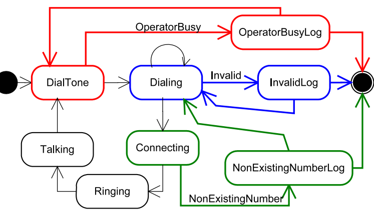


Figure 10. Behavior of a phone with error logging and roll-back

Note that is possible to combine different combination of the aspect to exactly fit the user needs.

4 Related Works

Our approach extends the SMARTADAPTERS approach [18, 19] by *i)* generalizing its concepts to any domain metamodel [24] (not only Java programs and UML

class diagrams), and *ii)* representing targets as model snippets [27], instead of declaring targets and constraints one by one. In [18], we introduce variability mechanisms in the base SMARTADAPTERS approach for class diagrams. In our generic approach, we also integrate these mechanisms (Section 3), in a slightly different way. Thus, we can propose configurable aspect, and weave them into models conforming to any domain metamodel.

Recent works discuss the use of Aspect-Oriented Programming (AOP) for managing variability at the code level, and implementing Software Product Lines (SPL). Some of these approaches advocate AOP for managing optional and variants cross-cutting features [2], or extracting and evolving SPL from a single application [1], and propose different variants, while some approaches like [13], insist on limitations and drawbacks of AspectJ for SPL implementation: code readability and maintainability, pointcut fragility making aspect weaving difficult. Moreover they point out that most of the mechanisms specific to AspectJ are not useful in most of the cases. Mezini *et al.* [23] point out the limitations of feature-oriented approach and AspectJ, especially its pointcut mechanism, and propose to use CaesarJ for resolving these problems. AOP is an interesting but still immature technology for managing variability. The combination of Aspect-Oriented Modeling (AOM) and Model-Driven Engineering (MDE) makes our approach more abstract and independent from problems inherent to the source code level. Unlike AspectJ pointcuts, our target models are totally independent from any base models and our aspect can be reused in different contexts, by binding target model elements to actual base model elements. There is no need for modifying the target model (pointcut), the aspect structure or the composition protocol (advice). Finally, AOP approaches for managing variability only propose one variability dimension and do not propose variability into the aspect itself, as we do.

In [21], Loughran *et al.* propose an approach that combines notions from AOP, frame technology and Feature-Oriented Domain Analysis (FODA). AOP aims at modularizing cross-cutting concerns and frame technology provide some means to configure aspects and make them context-independent and thus, more reusable. Using our approach, designers can also define context independent aspects using targets and adaptations that only reference elements from the aspect template or structure, and not directly base model elements. They use the variability mechanisms (alternative and options) of FODA models to represent the whole system *e.g.*, a generic cache. Then, they can delineate framed aspects and implement them in a reusable way using the frame technology. Frame is a fine mechanism to parameterize for example, the name and the type of attribute, method, parameters. In our approach, we use alternatives, options and constraints inside the aspect itself, for managing the different possible configurations. Frames are similar to our target model: both framed parameter and target model elements are substituted with actual elements from the base program/model, using bindings. Framed-aspect do not really propose internal variability, only configuration. Finally, both approach propose two variability dimensions, but they mainly focus on the system variability while we mainly focus on the aspect variability.

In [30], Schauerhuber *et al.* propose a common reference architecture for Aspect-Oriented Modeling. The concepts they identify are quite similar to the ones identified by Lahire *et al.* in the SMARTADAPTERS approach [18, 19], that we leverage to generalize the concepts of AOM to any domain metamodel. The approach of Schauerhuber *et al.* is also language-independent and may be applied for any domain metamodel. But, they do not propose means to generate the pointcut language nor domain-specific adaptations. Our generative approach, based on MDE techniques, allows designers to automatically specialize our framework, for any domain, by generating an unconstrained domain metamodel for designing target models (pointcuts), and generating domain-specific adaptations dealing with updating (addition/removal), creating and cloning elements. Moreover, they do not propose variability mechanisms, whereas we introduce mechanisms inspired by Software Product Line approaches.

In [16], Kim *et al.* combine this reference AOM architecture with a component-based SPL architecture. They propose to model variability using aspects, as we do in this paper. The variability mechanism is the *variability point* that is equivalent to our *alternatives* and *options*. In their architectures, they do not reify the notions of *constraints*, and do not really explicit how variants are selected, with their *variability point bindings*. In our metamodel, *constraints* and *derivation* clearly specify the dependencies between variants, and how to derive variants.

In [12], Whittle *et al.* propose the MATA (Modeling Aspects Using a Transformation Approach) tool for composing features in UML models (class diagrams, state charts and scenarios), based on graph rewriting. MATA allows user to describe the composition using stereotypes directly in feature models. The stereotypes they propose for composing features are similar to our Create/Set/Unset adaptations, but we also propose cloning adaptations. This can be useful, for example to implement a proxy, where all the operation needs to duplicate. Their notion of variable is equivalent to our notion of role *i.e.*, elements that can be substituted. With **MM'** and **MM"**, we can create more generic pattern by instantianting abstract elements and defining unconstrained models. Moreover, we propose variability mechanisms both for the matching and the composition whereas they only propose one variability dimension.

In [10], Fleurey *et al.* generalize the *Composition Di-*

rectives approach [29] and propose "a generic approach for automatic model composition", that can be adapted to any metamodel. This approach is based on signature matching and systematic merging of model elements. Their symmetric approach aims at merging different views of the same system *e.g.*, marketing and management views in order to obtain an integrated view of the system, using an automated weaving process that can be customized. Our asymmetric approach is different and aims at composing aspects, that can be considered as reusable patterns, into different base models, using parameterized composition protocols. Fleurey *et al.* do not propose variability mechanisms, but users can customize the matching by defining the signature of model elements, and customize the merging with context-specific composition directives. They do not propose alternatives, options and constraints for managing all the possible variants and consequently designers have to define as many aspects as possible configurations. Our approach allows designers to model an aspect per concern, with all the possible configurations. Then users select the most appropriate configurations to weave into their models.

In [11], Heidenreich *et al.* propose to extend the "Aspect Orientation for Your Language of Choice". Their generic approach is based on the Invasive Software Composition (ISC). Both base model and aspect model elements are annotated with *Slot*, *Hook* and *Anchor*. A slot indicates that a base element can be replaced by an aspect element with an anchor whereas a hook indicates a place in the base model where some anchored elements from the aspect can be added. They illustrate their approach on a UML class diagram and a Java program. Our approach is also generic but do not need to modify base models to make them aspect aware, letting base model oblivious of the aspect. We only use a binding mechanism before composition.

5 Conclusion

In this paper, we have presented our generic model-driven approach for aspect weaving: for any metamodel describing a given language or domain, we can generate both the targeting language and some weaving instructions that allow users to design reusable aspects. Then, we have extended this generic approach with variability mechanisms, and presented our 2-dimension approach for variability management. After deriving an aspect by choosing most appropriate variants and options, aspect configurations can be woven into base models, to integrate new features and propose different variants of the system.

In future work, we will extend our 2-dimension approach for variability management to runtime models [6], in the context of self-adaptive systems. The main idea is to use aspects at a model level, to adapt the running system, instead of hard-coding the adaptation logic at the platform level. Then, using a causal connection, modifications on the runtime model should be reflected on the running system. Moving models from design-time to runtime will reduce the complexity of runtime adaptations, by providing a higher level of abstraction. We are currently working on the implementation of the causal for the Fractal [20] component model. However, our causal link is not Fractal-specific and may be applied to other platforms like OpenCOM [9].

References

[1] V. Alves, P. M. Jr., L. Cole, P. Borba, and G. Ramalho. Extracting and Evolving Mobile Games Product Lines. In J. H. Obbink and K. Pohl, editors, *SPLC'05: 9th International Conference on Software Product Lines*, volume 3714, pages 70–81, Rennes, France, 2005.

[2] M. Anastasopoulos and D. Muthig. An evaluation of aspect-oriented programming as a product line implementation technology. In *ICSR'04: 8th International Conference on Software Reuse: Methods, Techniques and Tools*, pages 141–156, Madrid, Spain, 2004.

[3] S. Apel, T. Leich, and G. Saake. Aspectual Mixin Layers: Aspects and Features in Concert. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 122–131, New York, NY, USA, 2006. ACM Press.

[4] J. Araújo, J. Whittle, and D. K. Kim. Modeling and Composing Scenario-Based Requirements with Aspects. In *RE'04: Proceedings of the 12th IEEE International Conference on Requirements Engineering*, pages 58–67, Washington, DC, USA, 2004. IEEE Computer Society.

[5] E. Baniassad and S. Clarke. Theme: An Approach for Aspect-Oriented Analysis and Design. In *ICSE'04: Proceedings of the 26th International Conference on Software Engineering*, pages 158–167, Washington, DC, USA, 2004. IEEE Computer Society.

[6] N. Bencomo. Proceedings of the Models@run.time (at MoDELS) workshops. www.comp.lancs.ac.uk/ bencomo/MRT06/
www.comp.lancs.ac.uk/ bencomo/MRT07/.

[7] E. Brottier, B. Baudry, Y. L. Traon, D. Touzet, and B. Nicolas. Producing a Global Requirement Model from Multiple Requirement Specifications. In *EDOC'07: Proceedings of the 11th Enterprise Computing Conference*, Annapolis, Maryland, USA, 2007.

[8] T. Cottenier, A. van den Berg, and T. Elrad. Joinpoint Inference from Behavioral Specification to Implementation. *ECOOP'07: Proceedings of the 21st European Conference on Object-Oriented Programming*, 2007.

[9] G. Coulson, G. S. Blair, M. Clarke, and N. Parlavantzas. The Design of a Configurable and Reconfigurable Middleware Platform. *Distrib. Comput.*, 15(2):109–126, 2002.

[10] F. Fleurey, B. Baudry, R. France, and S. Ghosh. A Generic Approach For Automatic Model Composition. In *AOM@MoDELS'07: 11th International Workshop on Aspect-Oriented Modeling*, Nashville TN USA, Oct 2007.

[11] F. Heidenreich, J. Johannes, and S. Zschaler. Aspect-Orientation for Your Language of Choice. In *AOM@MoDELS'07: 11th International Workshop on Aspect-Oriented Modeling*, Nashville TN USA, Oct 2007.

[12] P. Jayaraman, J. Whittle, A. Elkhodary, and H. Gomaa. Model Composition in Product Lines and Feature Interaction Detection Using Critical Pair Analysis. In *MoDELS'07: Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems*, LNCS, pages 151–165, Nashville TN USA, Oct. 2007. Vanderbilt University, Springer-Verlag.

[13] C. Kästner, S. Apel, and D. Batory. A Case Study Implementing Features using AspectJ. In *SPLC'07: 11th International Software Product Line Conference*, September 2007.

[14] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An Overview of AspectJ. In *ECOOP'01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.

[15] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *ECOOP'97: Proceedings of the 11th European Conference on Object-Oriented Programming*, volume 1241, pages 220–242, Berlin, Heidelberg, and New York, 1997. Springer-Verlag.

[16] Y. Kim, M. Moon, and K. Yeom. An Aspect-Oriented Approach for Reprensenting Variability in Product Line Architecture. In *VaMoS'07: 1st International Workshop on Variability Modelling of Software-intensive Systems*, 2007.

[17] J. Klein, F. Fleurey, and J. Jézéquel. Weaving Multiple Aspects in Sequence Diagrams. *To appear in Transactions on Aspect-Oriented Software Development (TAOSD)*, 2007.

[18] P. Lahire, B. Morin, G. Vanwormhoudt, A. Gaignard, O. Barais, and J. M. Jézéquel. Introducing Variability into Aspect-Oriented Modeling Approaches. In *MoDELS'07: Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems*, LNCS, pages 498–513, Nashville TN USA, Oct. 2007. Vanderbilt University, Springer-Verlag.

[19] P. Lahire and L. Quintian. New Perspective To Improve Reusability in Object-Oriented Languages. *Journal Of Object Technology (JOT)*, 5(1):117–138, 2006.

[20] M. Leclercq, A. E. Ozcan, V. Quema, and J.-B. Stefani. Supporting Heterogeneous Architecture Descriptions in an Extensible Toolset. In *ICSE'07: Proceedings of the 29th International Conference on Software Engineering*, pages 209–219, Washington, DC, USA, 2007. IEEE Computer Society.

[21] N. Loughran and A. Rashid. Framed Aspects: Supporting Variability and Configurability for AOP. In *ICSR'04: 8th International Conference on Software Reuse: Methods, Techniques and Tools*, volume 3107 of *Lecture Notes in Computer Science*, pages 127–140, Madrid, Spain, 2004. Springer.

[22] N. Loughran, A. Sampaio, and A. Rashid. From Requirements Documents to Feature Models for Aspect Oriented Product Line Implementation. In *MoDELS Satellite Events*, pages 262–271, 2005.

[23] M. Mezini and K. Ostermann. Variability Management with Feature-Oriented Programming and Aspects. *SIGSOFT Software Engineering Notes*, 29(6):127–136, 2004.

[24] B. Morin, O. Barais, J. M. Jézéquel, and R. Ramos. Towards a Generic Aspect-Oriented Modeling Framework. In *3rd International ECOOP'07 Workshop on Models and Aspects - Handling Crosscutting Concerns in MDSD*, Berlin, Germany, August 2007.

[25] P. Muller, F. Fleurey, F. Fondement, M. Hassenforder, R. Schneckenburger, S. Gérard, and J. Jézéquel. Model-Driven Analysis and Synthesis of Concrete Syntax. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *MoDELS'06 : 9th International Conference on Model Driven Engineering Languages and Systems*, volume 4199 of *Lecture Notes in Computer Science*, pages 98–110, Genova, Italy, 2006. Springer.

[26] P. Muller, F. Fleurey, and J. M. Jézéquel. Weaving Executability into Object-Oriented Meta-languages. In *MoDELS'05: Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems*, volume 3713 of *Lecture Notes in Computer Science*, pages 264–278, Montego Bay, Jamaica, Oct 2005. Springer.

[27] R. Ramos, O. Barais, and J. M. Jézéquel. Matching Model Snippets. In *MoDELS'07: Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems*, LNCS, page 15, Nashville TN USA, Oct. 2007. Vanderbilt University, Springer-Verlag.

[28] A. Rashid, A. Moreira, and J. Araújo. Modularisation and Composition of Aspectual Requirements. In *AOSD'03: Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, pages 11–20, New York, NY, USA, 2003. ACM Press.

[29] Y. R. Reddy, S. Ghosh, R. B. France, G. Straw, J. M. Bieman, N. McEachen, E. Song, and G. Georg. Directives for Composing Aspect-Oriented Design Class Models. *Transactions on Aspect-Oriented Software Development I*, LNCS 3880:75–105, 2006.

[30] A. Schauerhuber, W. Schwinger, E. Kapsammer, W. Retschitzegger, and M. Wimmer. Towards a Common Reference Architecture for Aspect-Oriented Modeling. In *AOM'06@AOSD: 8th International Workshop on Aspect-Oriented Modeling at AOSD*, 2006.

[31] M. Sinnema, S. Deelstra, J. Nijhuis, and J. Bosch. COVAMOF: A Framework for Modeling Variability in Software Product Families. In R. L. Nord, editor, *SPLC'04: 3rd International Conference on Software Product Lines*, volume 3154 of *Lecture Notes in Computer Science*, pages 197–213, Boston, MA, USA, 2004. Springer.

[32] G. Straw, G. Georg, E. Song, S. Ghosh, R. B. France, and J. M. Bieman. Model Composition Directives. In T. Baar, A. Strohmeier, A. Moreira, and S. Mellor, editors, *UML'04: Proceedings of the 7th Conference on the Unified Modeling Language*, volume 3273 of *LNCS*, pages 84–97. Springer, Oct 2004.

[33] J. Van Gurp, J. Bosch, and M. Svahnberg. On the Notion of Variability in Software Product Lines. In *WICSA '01: Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, page 45, Washington, DC, USA, 2001. IEEE Computer Society.

[34] T. Ziadi and J. Jézéquel. *Families Research Book*, chapter Product Line Engineering with the UML: Products Derivation, pages 557–588. LNCS. Springer Verlag, 2006.

Model-Based Implementation of Meta-Variability Constructs: A Case Study using Aspects

Klaus Schmid, Holger Eichelberger
University of Hildesheim, Institute of Computer Science
Marienburger Platz 22
D-31141 Hildesheim, Germany
+49 5121 883 761/
{schmid, eichelberger}@sse.uni-hildesheim.de

Abstract

In this paper, we introduce the concept of meta-variability, i.e., variability with respect to basic variability attributes like binding time or constraints. While the main focus of this paper is on the introduction of the concept, we will also illustrate the concept by providing a case study.

The case study will feature a simple implementation environment based on aspect-oriented programming and will include an example that will exhibit some key characteristics of the envisioned production process.

1. Motivation

Product line engineering has become increasingly recognized in the last few years as a successful approach to dramatically reduce costs, reduce time-to-market and improve quality. It has also achieved significant acceptance in industry [12].

Along with the increasing recognition of product line engineering in industry, various approaches to variability modeling were proposed. In particular, feature modeling concepts are widely discussed and partially used in industry [6, 9, 10, 16], but a large range of other approaches have been proposed as well, e.g. [4, 14, 17, 19, 20].

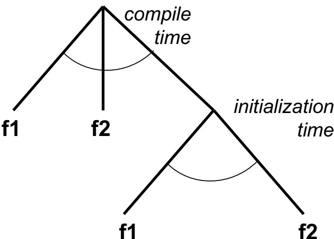
So far all these approaches share (at least) one commonality as they focus on variability as variation of specific attributes of the final product (e.g., functional or non-functional properties). Thus, they neglect the variability that may occur with respect to the characteristics of a specific variability itself. We term such variability of a variability attribute *meta-variability*.

Thus, meta-variability relates to variability that may occur with respect to production processes of subsets of the product line. For example, for some products a specific variability may be bound at compile time, while other products still support a sub-range of variability and the final binding happens during product initialization. Thus, the binding time itself may vary. Similar examples can be given for other variability attributes as well.

At first glance, the issue of meta-variability may seem very esoteric; however, it is firmly grounded in industrial practice.

A simple example, which we observed in one company, was that some variation was relevant to both high-end and low-end products. While the low-end product was produced in high volume, the high-end product was produced only in low volume. As a consequence, their production processes were different due to economic reasons. Some variations were common to these products, but for the low-end product, which also had less memory and processing power was reduced, the variation had to be bound at compile time. This way, each variant was produced independently and as this did save resources in the final variant and thus production costs, this was cost-effective.

On the other hand, for the high-end product the variability had to be bound at initialization time (prior to sales, but after shipping to country offices). The reason for this was that sales personal in the various countries could determine the final product variant in order to adapt to fluctuations in demand. As the volume of these high-end products was low, the added production costs were not relevant in comparison with the increased flexibility.



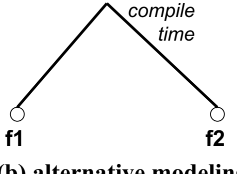


Figure 1 (a) Reification of binding time **(b) alternative modeling**

This episode shows a clear need for variation with respect to binding time. More precisely, two different binding times could be selected alternatively for the same variability.

In the following sections we will first discuss different approaches to dealing with meta-variability. In Section 3, we will then describe some basic concepts of a prototypical production environment that we used as a basis for supporting meta-variability. In Section 4, we describe a simple case study. Finally, in Section 5 we will provide our conclusions and illustrations of possible future work.

2. Dealing with Meta-Variability

To our knowledge, the problem of meta-variability has so far not been explicitly addressed. Only few approaches explicitly allow multiple binding times for a single variability. One example is [19], however, the approach does not provide a precise interpretation and semantic foundation of multiple binding times. An early case study that used this concept is described in [18].

While the problem sketched above is not uncommon in industry, so far we have not seen it being fully addressed. Instead an approach is typically taken that can be interpreted as *reification*: the meta-variability is represented as a different variability in the form shown in Figure 1a.

As this figure shows, reification leads to duplication of the respective variant information. In practice, this is often handled implicitly, by modeling the variation as something like Figure 1b and handling the case that both variants are selected in a special manner, by providing additional initialization code, without actually modeling this as variability. However, this actually means that variability is only partially modeled. In current industrial practice, where variability is usually not modeled at all, or at least not completely, this is not yet an issue, but if the underlying goal is to move towards more systematic (and automatic) variant-based software production processes, this becomes a major problem. Thus, we propose to deal with meta-variability explicitly and to accept it as a first class modeling element. This implies that variation of variability attributes must be modeled explicitly. This approach in turn provides the advantage that the various product instantiations can be produced automatically. We will describe this approach below and discuss it based on a case study.

3. Concepts of a Production Environment

In this paper, we provide a vision of how future software production environments that explicitly support variability can look like. We do not yet present a full-fledged product derivation environment.

We sketch a production environment that supports the automatic product derivation and instantiation of variability. The core idea is to use the variability model, together with the current variant selection to produce the binding of the variant parts. Depending on

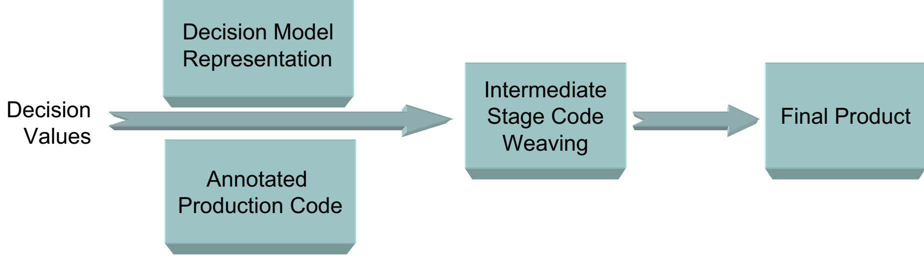


Figure 2 The Production Process

the binding time, the variant parts are bound in a different way. Thus, variation in binding time directly influences the model-based production process and leads to the production of different code. This is depicted in Figure 2.

The specific form of the production environment and of the production process depends on numerous parameters, including the type of artifacts supported, the programming languages used, etc. Here, we will focus on a particular example based on Java, which we tried to keep as simple as possible for illustration purposes. Of course, the implementation could be combined with context-oriented or feature-oriented programming [5, 13], instead of the straight-forward approach we use here. However, the key issue we address is not the implementation technique, but the need of communicating meta-variability in the product instantiation process between product developer and the final product. This goes beyond the existing approaches.

In order to evaluate the feasibility and appropriateness of this idea, we constructed a simple, prototypical production environment. The core idea of this environment is the stringent separation of functional code and the variability implementation, which we achieved in a very simple way for our example, a more sophisticated approach could include ideas from feature-oriented or context-oriented programming.

3.1 Domain Engineering

The approach to product line modeling which is used in our case study is based on decision modeling, an approach initially devised in the Reuse-Driven Software Process Guidebook [20]. The approach has been later extended in several ways, e.g., [4, 19]. Here, we will build in particular on the extensions as described in [19]. However, we believe the basic approach is not specifically influenced by the choice of variability modeling approach and could be integrated in a similar way with feature modeling or other approaches.

In accordance with the decision modeling approach, we capture the decisions that are relevant for deciding about the product characteristics. Further, we allow that for a single decision multiple appropriate binding times can be recorded. So far, we have not yet extended the modeling mechanism as far as enabling to define constraints on binding times, thus, the semantics is simply (as was initially defined in [19]) that during instantiation any of the previously specified binding times can be chosen for a concrete decision.

Besides the variability model (here the decision model) the basic information about the product line must be modeled. In general, this can happen in an arbitrary modeling language, respectively, by a combination of multiple modeling approaches. There is no specific restriction with respect to forms of modeling that can be used. In the case study, that we will discuss in Section 4 we will restrict our domain modeling actually to a Java implementation as this is a widely used and well-known implementation language. Of course, different types of artifacts may only allow for certain variability attributes. For example, a runtime adaptation of a UML model does not make too much sense.

Finally, a relation must be established between the various decisions and the artifacts that are impacted by these decisions. A high-level categorization of mechanisms to realize artifact variability was given in [20]:

- Physical Separation, i.e. to represent variant elements as physically distinct entities, e.g. as separate files.
- Target-Language specific mechanisms, e.g. templates, generics, alternatives in combination with constants, etc.
- Metaprogramming mechanisms to superimpose a language for handling variations on top of the target language.

In our example, we will exclusively rely on mechanisms that are yet supported by the target language and in related IDEs, i.e. existing editing and refactoring mechanisms or well-known add-ons e.g. by using available IDE plugins and libraries.

As one interesting approach aspect-orientation lends itself to variability implementation. Thus, we decided to use AspectJ as part of the production environment. Of course different choices would have been possible as our case study does not depend on specific realization techniques. The realization of variability will be done by referring to the production code with pointcuts as defined in aspect oriented programming [11] and by the use of special purpose variables. The use of special purpose variables is a target-language specific mechanism that, in particular, relies on the target language compiler (e.g. static evaluation of constants, code elimination and inlining). Aspect oriented programming may appear as a metaprogramming mechanism (e.g. additional keywords are introduced as in AspectJ versions prior to version 5) or as a mechanism that relies on meta-information represented by constructs of the target language (support of Java annotations since AspectJ 5). Combined with physical separation and conditional packaging of the resulting binaries, aspect oriented mechanisms act in our example as a tool to easily realize binding at startup time and runtime. An alternative, probably with some more architectural effort, could be applying layers of collaborations or polymorphic selection and default objects [15].

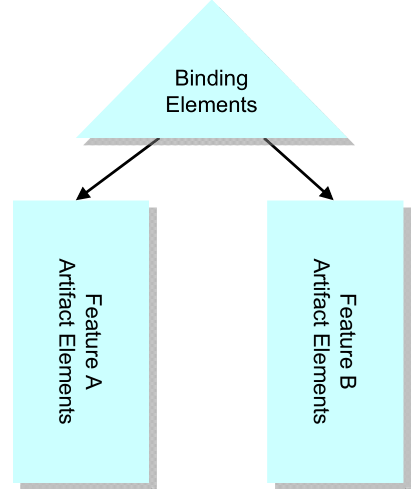


Figure 3 Instantiated Artifact Elements Model

As stated earlier, the specific form of the realization of the instantation technique depends on a number of parameters, including the programming language. Other programming languages will require explicit preprocessing mechanisms (categorized as metaprogramming in [20]) like the well known C preprocessor. As we will rely on Java in our example, such preprocessing mechanisms are substituted by the use of if-clauses with constant expressions, which are defined in Java to be equivalent to preprocessing. We combine this with target-language specific mechanisms facilities of the core language like special purpose variables and aspect oriented programming as described above.

3.2 Application Engineering

The aim of application engineering is to derive the final product. Thus, based on values for the various decisions (including the determination of the binding time), feature artifact elements from the product line model must be selected and combined (cf. Figure 3). So far, this is very much the standard approach as it has been realized in numerous other product line modeling tools (e.g., [3]). This becomes more of an issue as soon as binding times that involve runtime decisions must be addressed. In this case, the code that binds together the various feature implementations should be generated.

In our approach, we generate variability code from the decision model information and combine it with a simple runtime part provided by the production environment. Thus, if values and binding time are assigned to a decision so that full instantiation is possible at development time, the necessary artifact elements are combined. If runtime decision making is required (e.g., during start-up), this decision must be modeled on the level and in the representation of the artifact in question. In our case study, as we will only deal with Java Code, this will relate to the actual activation code.

This concept is shown in Figure 3. This figure shows two possible artifact elements. They are related by binding elements. These binding elements may take different forms, depending on the kind of decision taken and the binding time:

- In case the decision is taken to have only one of those elements (and this is valid at development time), the binding element simply needs to integrate the artifact element with the remainder of the model.
- In case the decision is taken to have both elements present at runtime (and this decision is taken at development time), the binding element must become a connector that connects both elements simultaneously.
- In case the decision is taken to have one of the elements, but the final decision which one is taken at runtime, this needs to turn into a connector that is evaluated at runtime.

Though this discussion is still rather abstract at this point, it will probably become somewhat clearer as we illustrate it in the next section based on a case study.

4. Case Study

In order to analyze the possibilities and implications of making meta-variability explicit and treating it as a first class citizen, we conducted a case study based on an existing software system, with which we were already well acquainted: the SVNControl system [2].

4.1 Prototype Realization of the Production Environment

The prototypical production environment contains

- the domain modeling view, an Eclipse plugin, which maintains the decision definition table from domain engineering (see section 4.3). The editor allows creating, editing and deleting domain decisions. In particular, for each domain decision the allowed binding times, i.e. the binding range, and a value range (currently boolean values, arbitrary integer ranges and arbitrary enumerations are supported) constraining the instantiation of the decision can be specified. The domain definition table is stored as a file in XMI format.
- the product derivation view, also part of the Eclipse plugin used to specify the values of con-

crete decisions (as described in section 4.4). The derivation view allows to provide values to non-instantiated decisions, and the editing and deleting of the concrete decisions. The concrete decision values are stored as a file in XMI format, which is linked to the domain definition table file.

- the code generator (as described in section 4.3) and build management support, i.e. appropriate ANT [1] tasks to
  - Configure and run the code generator by specifying the product decisions file and the names of the classes to be generated.
  - Optionally execute a Java specific C-style preprocessor based on the values of the product decisions. Currently, the preprocessor is intended to prepare the environment for other target languages.
  - Clean up empty class files that result from the execution of a preprocessor.
- the runtime core to be included into the final product if decisions are left to the user and must be made e.g. during startup or runtime. The runtime core contains a default mechanism for user decision making and the implementation of value range types in order to validate the user input.

The individual parts of the production environment will be described along with the case study in the next section.

4.2 The Base System

Our case study is based on the SVNControl system, which provides a network based management interface to the subversion system. The system was initially developed at the University of Hildesheim, but has been released as Open Source [2]. Currently, it is still under development and has been taken up by organizations like UBS, GDV (association of German insurances), and many others.

SVNControl is a remote administration tool with graphical user interface for the version management system Subversion. SVNControl supports the administration on repository level (e.g. to create, rename or delete repositories consistently), user or group level, access permission level and on scripting level, i.e. to take control over several hooks controlling valid check-ins etc.

Basic administrative functions are relevant to all users and should therefore be treated as commonalities in a product line. However, some more advanced features like scheduling of permissions as well as scripting and hooks are candidates for a special distribution for advanced administrators. Based on discussions with users, also the entire user management is in question in some environments, because often in organizations, users and user attributes like groups are centrally administered, e.g. by LDAP or ActiveDirectory and therefore, this functionality should not be available.

While currently the product is built without variability, we decided it makes a good case study, as a need for variability can be clearly identified and the code is well known to the second author.

4.3 Modeling Variability

Following the brief introduction of SVNControl in the section above, we will now discuss how we represented the variability using our prototypical production environment in terms of the decision model approach as described in [19]. Due to space limitations, we will restrict ourselves to the variabilities for the scheduling and the hook functionality.

The definition of a decision consists of:

- A unique name used to reference the decision.
- The relevancy specifies the circumstances under which the definition is meaningful.
- A textual description of the decision.
- A range to define or restrict the values that the decision can take. The cardinality defines how many values the decision (seen as a set) may have.
- Constraints among values of the various decision variables
- Binding times: Define a range of points in time, which describe when the decision can be bound to a concrete value.

As mentioned above, we will discuss two decisions in this case study. They refer to the capability of the resulting product to administrate

- permissions using a scheduler.
- the hook scripting mechanism.

Figure 4 depicts the decision modeling view of our production environment showing those decisions for SVNControl. The definitions of the decisions can be maintained in the domain decision table in the upper left part. In particular, the user can specify a value range (e.g. boolean values) and an individual binding time range for each decision (in the lower part). In this case study we will neither consider the relevance nor the constraints of the decisions.

When the domain decision model is specified, the information on the decisions must be transformed into source code related information. Therefore, depending on the binding time range of the individual decisions, the code generator of our prototypical production environment will produce a set of constants for each avail-

able binding time. We will now discuss how the relation between the decisions and the variant artifacts can be realized for Java as target language.

In Java, compile time decisions can simply be represented as constant values to be evaluated as expressions in alternatives (if-statements), because, according to the Java Language Specification [8], the compiler will evaluate the constants during compile time and inline or exclude the source code influenced by the alternative. Consequently, the if-statement in Java in combination with constants acts like a preprocessor statement in other languages. Of course, it would be clearer to have an explicit preprocessing step, but this is the way Java is defined. This confusion of preprocessing-IF and runtime-IF can be considered a shortcoming of the Java-language definition.

For example, if the decision is made at compile time that the scheduling functionality should be available, the code generator will produce a constant class containing a constant named according to the identification of the decision and initialized with the given default value as follows:

```
/**
 * Configuration constant for the
 * decision "Can user or group …?".
 */
public static final boolean
    OPT_SCHEDULES = false;
```

The production code itself may now contain appropriate alternatives depending on the value of the compile time decision, e.g. in our case study code to display the related GUI elements in the case that the decision value is `true`.

So far, this was standard implementation of a compile time variability. However, the interesting part is that the production environment can handle also startup and runtime binding for the same decisions.

For these decisions more information must be taken into account in order to construct an appropriate decision-making mechanism at startup time or runtime. Taking the domain decision table as input, the model-based generator will produce object constants (enum values) that will carry additional information to be provided to the runtime decision-making mechanism. Even if constants are generated, the related concrete decision values may change during runtime of the program, e.g. using a dialog which initializes itself according to a set of these enum constants. The following source code fragment depicts one of the produced enum constants showing also some additional information from the decision model like the description and the value range:

```
/**
 * Configuration constant for the
 * decision "Can user or group …?".
 */
OPT_SCHEDULES("Can user or group…?"
    ,BooleanValueRange.
    BOOLEAN_RANGE, …),
```

Only two more steps are needed to make startup or runtime decisions work: The decision-making dialog must be called at an appropriate point of time in the production code of SVNControl and SVNControl itself must be able to react when a certain decision is made.

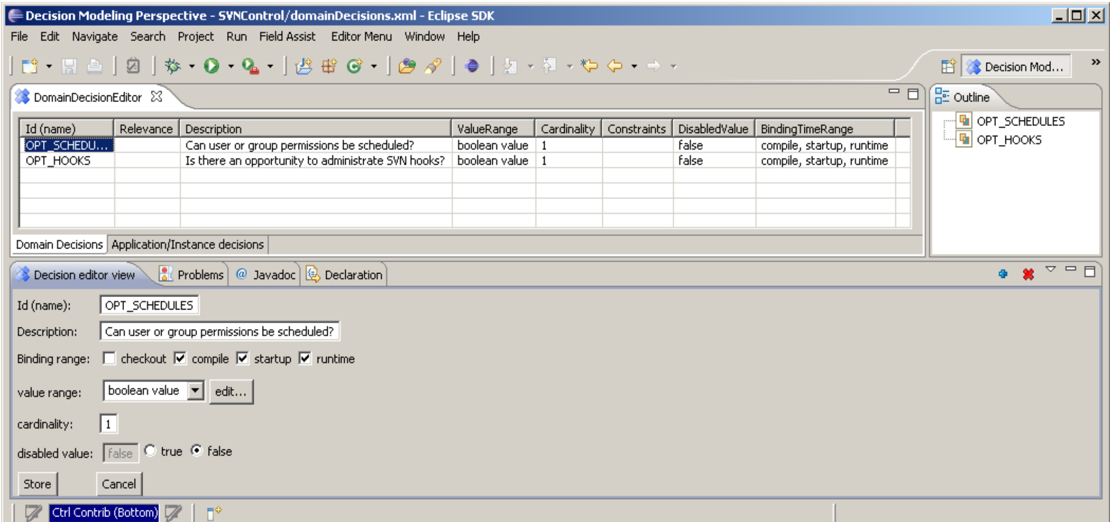


Figure 4 The decision modeling view of our production environment.

To address both issues, we resort to using AspectJ pointcuts to introduce startup calling code and to insert the necessary code fragments from runtime variability into the code. Other approaches (e.g., preprocessing) could have been used as well for this task, however, we use AspectJ as it provides rather clear and well-known mechanisms. In addition, we used a legacy system as a basis for our case study. Thus, it was a key requirement to use technologies that enable the introduction of variability with as little restructuring of architecture and code as possible.

The examples shown in this case study are given in the old style of AspectJ – AspectJ 5 facilitates Java annotations so that no proprietary keywords are necessary anymore. From a strict viewpoint, using the old style of AspectJ implies the use of a metaprogramming approach that is outside the target language. We will ignore this minor issue here, because the example could easily be refactored to new style and the well-known notation simplifies reading this paper.

Using AspectJ, the Java compiler as well as the runtime environment of the final product will be enhanced by code weaving facilities as indicated in Figure 2. The following example shows the pointcut related to the startup time decision for permission schedules in SVNControl. `StartupConfiguration` is assumed to be the enum class produced by the code generator for startup time decisions. Method parameters are not shown to keep the example simple.

```
aspect Startup_Schedules {

 pointcut myClass():
  within(MainWindow);

 pointcut myMethod(): myClass() &&
  execution(void MainWindow.
  initializeDynamicElements());

 before(): myMethod() {
  if (StartupConfiguration.
   OPT_SCHEDULES.getBooleanValue()) {
    // activate the scheduler UI
  }
}
```

The code artifacts to be injected for runtime decisions look similar. A runtime related aspect defines the call to the decision-making mechanism, e.g. as an action of a special menu item. To notify SVNControl about changes of the runtime decision value, the pointcut may register an observer [7] in the runtime core of our production environment. The concrete application may then react appropriately when a value is changed, i.e. in the case of SVNControl by enabling or disabling GUI parts related to the decision.

Due to the architecture of SVNControl, all variabilities sketched in this paper can be realized in a similar way as presented in this section.

By providing information on all binding times of each decision specified in the domain decision table and by separating the binding time related code into several aspects, we gain the flexibility to relate the realizing artifacts to decisions at development time and to postpone the decision on the concrete binding times until product derivation time and finally to smoothly switch among the available binding opportunities while product derivation time. In the next section, the mechanisms related to product derivation will be discussed.

4.3 Deriving the Products

Based on the results of the variability modeling and the domain engineering, i.e. the domain decision table, the initially generated constant sets for the supported binding times and the (implemented) pointcuts, now our production environment can be used to derive concrete products by instantiating the decisions and to build the individual products.

Using the product derivation view shown in Figure 5, the product engineer can now determine the concrete (initial) value, the binding time of all decisions previously specified in the domain decision table and the related code artifacts. Then, by executing the model-based code generator, the constant sets in the production code are (re)generated and the concrete values are stored in the production code of SVNControl. In particular, this step is important for the compile time decisions, because it determines the concrete values of the constants. Thereby, additional build information for the following build steps is gathered in order to give the current decision values control over the binary packaging process and, therefore, to influence which classes or pointcuts will be present in the final product. The next build step removes existing binaries from previous builds and calls the compiler with respect to the relevant pointcuts. Finally, based on the generated packaging information, only the binaries related to the selected decisions will be assembled together into executable Java archives.

4.4 Results

We have applied our prototypical production environment to introduce and realize compile-time, startup-time and runtime decisions in the context of the SVNControl application. Based on the decision model maintained by the domain and application modeling view of the Eclipse plugin, the generated constant classes, the runtime part and the build support, it was easy to realize the discussed decisions and therefore the intended variability. In particular, with little overhead also binding times postponed to application decisions can be realized easily.

The code produced by the model-based code generator is easy to read and fits to usual source code conventions. Beside tests whether the intended binding time for the decisions is realized and functional, we were also interested, whether the proper binary parts appear in the packaged result and whether the binaries related to disabled binding times disappear. Therefore, all tests were carried out after rebuilding and repackaging SVNControl. The intended functionality was fully functional and only the binary parts selected by the binding times in concrete decisions appeared in the packaged application, i.e. unintended binary fragments were completely absent. Thus, the production process that has been set actually achieved its underlying goals.

5. Summary and Outlook

In this paper, we argued for the importance of meta-variability. This concept describes the variation of variation attributes (as opposed to the mere variation of product characteristics). Meta-Variability, especially in its form of binding time variability is actually relevant, but has so far not been dealt with.

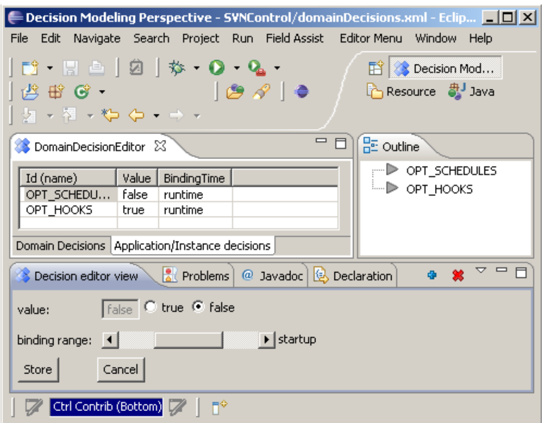


Figure 5 The product derivation view of our production environment.

We showed that by setting up an adequate production environment for variant code, we are able to deal with binding time variability very easily and in a canonical manner. We presented a case study of a system that is currently in daily use, introduced the variabilities artificially in order to have a "clean" test-bed.

The prototypical production environment showed that systematic support of binding time variability is possible.

In the future, we will study also other forms of meta-variability (e.g., the variability of constraints) and will aim to enhance our production environment along these lines. Furthermore, specific editors are planned to also support arbitrary artifacts.

We will further study possibilities of integrating our model-based generation with our forms of model-based code generation, in order to arrive at a seamless integration with target-language independent model-based development.

References

[1] Project homepage ANT, 2007. Online available at: http://ant.apache.org/.

[2] Project homepage SVNControl, 2007. Online available at: http://svncontrol.tirgirs.org/.

[3] M. Antkiewicz and K. Czarnecki. FeaturePlugin: Feature modeling plug-in for Eclipse. In *OOPSLA'04 Eclipse Technology eXchange (ETX) Workshop*, 2004.

[4] 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.

[5] P. Costanza, R. Hirschfeld, and W. De Meuter. Efficient layer activation for switching context-dependent behavior. In D. Lightfoot and C. Szyperski, editors, *JMLC 2006*, volume 4228 of *Lecture Notes in Computer Science*, Berlin / Heidelberg, 2006. SpringerVerlag Inc.

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

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, Massachusetts, 2000.

[8] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, 3. edition*. Addison-Wesley, 2005.

[9] 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 ESD-90-TR-222, Software Engineering Institute Carnegie Mellon University, 1990.

[10] Kyo C. Kang, Jaejoon Lee, and Patrick Donohoe. Feature-Oriented Product Line Engineering. *IEEE Software*, 19(4):58–65, 2002.

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference*

on Object-Oriented Programming, volume 1241 of *Lecture Notes in Computer Science*. SpringerVerlag Inc., 1997.

[12] F. van der Linden, K. Schmid, and E. Rommes. *Software Product Lines in Action - The Best Industrial Practice in Product Line Engineering*. Springer, 2007. http://www.splbook.net/.

[13] M. Mezini and K. Ostermann. Variability management with feature-oriented programming and aspects. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 127–136, New York, NY 10036, USA, 2004. ACM Press.

[14] D. Muthig. *A Light-weight Approach Facilitating an Evolutionary Transition Towards Software Product Lines*. PhD thesis, Fraunhofer-Institut für Experimentelles Software Engineering IESE, Kaiserslautern, University of Kaiserslautern, Germany, 2002.

[15] D. Muthig and T. Patzke. Generic implementation of product line components. In *Proceedings of the Net.ObjectDays (NODE'02), Erfurt, Germany*, October 2002.

[16] I. Pashov, M. Riebisch, and I. Philippow. Supporting Architectural Restructuring by Analyzing Feature Models. In *Proceedings 8th European Conf. On Software Maintenance and Reengineering, Tampere, Finland, March 24-26, 2004*, pages 25–33, 2004.

[17] Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. SpringerVerlag Inc., August 2005.

[18] K. Schmid, U. Becker-Kornstaedt, P. Knauber, and F. Bernauer. Introducing a software modeling concept in a medium-sized company. In *International Conference on Software Engineering (ICSE'22)*, New York, NY 10036, USA, 2000. ACM Press.

[19] K. Schmid and I. John. A Customizable Approach To Full-Life Cycle Variability Management. *Science of Computer Programming*, 53(3):259–284, 2004.

[20] Software Productivity Consortium Services Corporation, Technical Report SPC-92019-CMC. *Reuse-Driven Software Processes Guidebook, Version 02.00.03*, November 1993.

Value-Based Elicitation of Product Line Variability: An Experience Report

Rick Rabiser Deepak Dhungana Paul Grünbacher Benedikt Burgstaller
Christian Doppler Laboratory
for Automated Software Engineering
Johannes Kepler University Linz, Austria
{rabiser, dhungana, gruenbacher, burgstaller}@ase.jku.at

Abstract

Understanding and modeling the variability of an existing system is a highly critical and challenging task when adopting a product line approach. Only little guidance is available for identifying the variable elements in a complex system and for choosing the appropriate level of granularity for modeling. Also, product line engineers have to find a balance between the technically feasible variability and the externally visible variability reflecting the business perspective of an organization. In this paper we describe experiences in developing and applying a value-based process for eliciting product line variability which aims at integrating the technical and business perspectives in product line engineering. We developed the process in a series of workshops carried out with our industry partner Siemens VAI, the world's leading company in plant building for the iron, steel, and aluminum industries.

1. Introduction and Motivation

Numerous variability modeling approaches and tools are available in software product line engineering (SPLE), e.g., [1, 2, 13, 17, 18]. While these approaches provide good support for managing and formally describing variability, the elicitation of the variability of existing systems still remains a challenging task when adopting a product line approach. Explicit support for variability elicitation is thus needed to acquire knowledge about variability from both technical and business stakeholders.

In our ongoing research cooperation with Siemens VAI we have been developing a decision-oriented approach for SPLE [6, 15]. When creating initial variability models using our modeling language and tools [7, 14] we noticed a lack of elicitation techniques for identifying and understanding the variability of existing systems, capturing the tacit knowledge of different stakeholders, and choosing the right level of granularity for modeling variability.

Today's highly customizable, component-based software architectures offer an extremely high degree of technical variability. However, not all technically possible variants of a system are also relevant for customers. A key challenge of SPLE thus lies in simultaneously understanding both the technically feasible variability and the externally visible variability [12]. Product line scoping approaches, e.g., [16], address some of these issues. However, little guidance is available for finding the right balance between what could be modeled and what should be modeled. Finding the right level of granularity is challenging as both technical and business requirements need to be met.

Linking business and technology issues has received increased attention in software engineering. For example, the field of value-based software engineering (VBSE) [3] aims to overcome the traditional value-neutral approach in software engineering that treats all artifacts as equally important. Value-based variability modeling means to consider the business value and the associated risks of variability. Furthermore, VBSE suggests that variability management must not be seen as a pure modeling problem. Extracting tacit variability knowledge from diverse heterogeneous stakeholders is a collaborative process [8, 10, 11] that relies on involving software engineers that have been developing the reusable assets as well as people marketing and selling these assets need. Collaborative methods in software engineering emphasize stakeholder involvement. For instance, the EasyWinWin approach [4] has demonstrated the use of collaborative techniques to elicit stakeholder value propositions in requirements engineering. The field of collaboration engineering (CE) [5] provides further insights into general patterns of group collaboration that are also useful to define variability elicitation processes.

In this paper we describe a process for eliciting product line variability that aims at integrating three research areas: (i) SPLE with a focus on variability modeling and management, (ii) VBSE and in particular the question: how much is enough in variability modeling?, as well as (iii) CE with patterns of collaboration that enable different people working together to produce mutually satisfactory results. The process emerged in course of several variability modeling workshops we conducted with our industry partner. The paper is structured as follows: In Section 2 we describe how we iteratively developed the process and discuss lessons learned. In Section 3 we present the resulting process model. Section 4 rounds out the paper with a conclusion and an outlook on future work.

2. Defining the Process

We conducted a series of workshops with engineers and project managers of Siemens VAI to elicit the variability of a complex software system supporting continuous casting in steel plants [9]. The goals of these workshops were to understand the variability of different parts of the system and to define a repeatable process for eliciting variability which can be used by product line engineers in their daily practice. We started with a tentative process and tested it in an initial workshop. Based on experiences and feedback from participants we iteratively adapted and enhanced the process in further workshops. In total, three 3-hour workshops were conducted to capture the most *relevant* variability of the six largest and most complex subsystems of the software system. *Relevance* in this context means that the variability addresses a development risk (high loss if a certain decision is not taken or taken delayed during derivation) and an important business aspect (directly creating customer value in application engineering). More specifically the workshops aimed at the following goals (*italics* denote refinements of the initial goals based on experiences).

(G1) Finding the most important differences between products previously developed.

(G2) Analyzing these differences to develop a shared understanding of the system's variability and variability management in general.

(G3) Documenting the rationale and importance (value, risk) of the identified variability *together with known consequences for engineering and development.*

(G4) Developing a shared understanding of the impact of the identified variability on engineering. This includes for instance how and why the identified variability is implemented in the system.

(G5) Defining the variability in understandable terms (e.g., in the form of questions to be answered during product derivation).

(G6) Prioritizing variability for application engineering and product derivation to find the most essential aspects for later modeling.

2.1. Workshop Activities

The first workshop involved two groups with experience in two large and important subsystems of Siemens VAI's software system. Each group consisted of three engineers that had been involved in the development of the subsystem. The workshop was organized to collaboratively develop one flipchart per subsystem (Figure 1), with yellow cards describing the variability, blue cards describing the rationale of the variability, and red cards describing the variation points in the form of questions representing decisions to be taken during product derivation. A moderator facilitated the process. One scribe took care of arranging the materials on the flipcharts. Another scribe took notes about observations and lessons learned in the process.

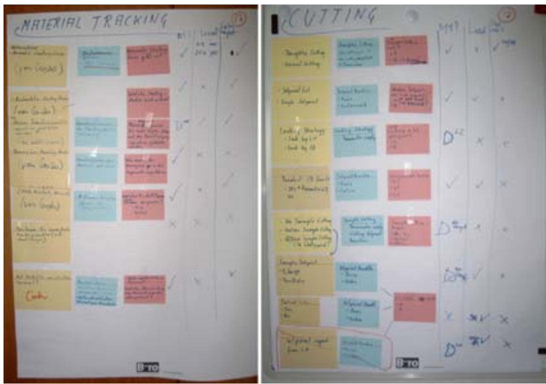


Figure 1: Participants use cards to capture the variability for selected subsystems.

Firstly, as stated in goal G1, participants collected the most important differences between previously developed products. Each difference was written on a separate yellow card. In the following moderated plenary discussion (G2) these differences were analyzed and rephrased or adapted where necessary to develop a shared understanding of variability and to improve clarity. When discussing the rationale for the collected variability (i.e., customer requirements or internal organizational decisions) participants requested documenting the consequences of this variability for development. We adjusted the tentative process as well as

goal G3 (cf. the parts in italics) as follows: Each group discussed the implementation of the variability in their subsystem and documented how it affects the product derivation process. In a moderated plenary discussion, the collected consequences were analyzed and adjusted, put on blue cards, and arranged with yellow cards describing the variability.

While conducting the two new activities, an additional activity based on a new goal (see G6) turned out to be crucial. While discussing the consequences, participants explored possible risks and the relevancy of variability. This confirmed the need of value-based elements in the process. In a moderated plenary discussion, the team therefore defined: (i) whether a decision on the identified variability must be taken early in product derivation (i.e., at the first milestone in application engineering) and (ii) whether the decision has a local impact within the subsystem and/or system-wide impact affecting the entire system. A table arranged beside the cards on the flipchart, with the columns M1 (important for first milestone), local (significant local influence on subsystem), and system (significant system-wide influence) was used to capture the results of this discussion (cf. Figure 1).

Finally, variation points were elicited in the form of questions representing decisions to be taken during product derivation. These were put on red cards and arranged with existing cards (Figure 1). In additional iterations variation points were reprioritized, dropped if considered unimportant or rephrased if necessary to increase clarity. Also, the relevancy table was adjusted in some cases.

The first workshop took 3 hours. Based on the adjusted tentative process two more workshops were conducted with the same moderator and scribes but different developers and architects of Siemens VAI. During these two additional workshops another adjustment to the process was made. It turned out to be insufficient to deal with variation points in one subsystem only. Participants found it important to also elicit variability of other subsystems that influences local variability. Selected variability in other subsystems was thus also captured but marked as external (by putting the source subsystem on the yellow cards in brackets, cf. Figure 1).

2.2. Lessons Learned

Time Boxing. Precisely defined time boxes for each process activity turned out to be very useful. There is a constant danger that activities take longer than anticipated (often caused by fruitless discussions). Moderators have to guide participants towards mutually accepted agreements. This is however not always possible in the workshop. Points not agreed upon can be recorded and dealt with later. Participants also appreciate smoothly run workshops as the time they could spend in workshops was typically limited.

Prioritization. Moderators should not ask participants to focus on the most important variability only as this might limit the creative process in the initial steps. However, moderators have to ensure to focus on variability with the highest importance during subsequent moderated plenary discussions.

Feedback. It is also important to use the elicited information to create initial variability models shortly after the workshop to provide quick feedback. Workshop participants and especially senior management need these concrete models to justify the effort spent and to validate the work results.

Facilitation. It was useful that both a moderator and a scribe conducted the process. While the moderator guides the participants and the discussions, the scribe is responsible for protocols and documentation.

Levels of Variability. Due to the involvement of different stakeholders, a large palette of variability is elicited ranging from technical details to marketing considerations. Figure 2 shows a simple model of decision layers we found useful to guide post-workshop modeling activities.

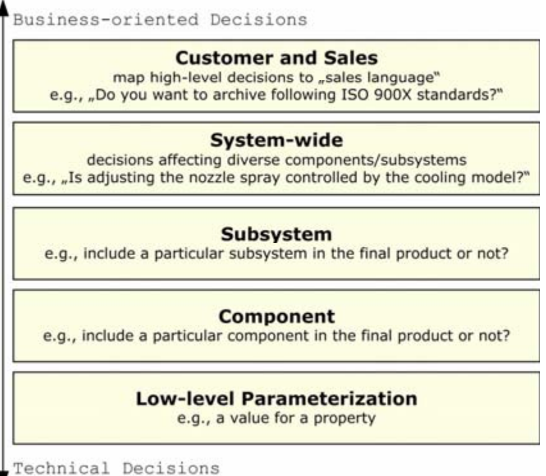


Figure 2: Layers of variability.

Complement results with variability recovery tools. Not all types of variability can be elicited in collaborative workshops. It is beneficial to complement workshop results with results from variability recovery tools such as parsers analyzing existing architecture models and configuration files.

3. Model of the Collaborative Process

The feedback and experiences from the workshops allowed us to define a repeatable process for variability elicitation. The process is *value-based* as is relies on stakeholder involvement in the variability modeling process and on consequent assessment of the identified variability with respect to relevancy, i.e., both risk impact and business value of variability. The process is *collaborative* as it implements the general patterns of group collaboration known from CE [5]: (i) *generate* (a group moves from having fewer to having more concepts with which to work; i.e., subgroups elicit differences), (ii) *reduce* (the group moves from having many to focusing on a few concepts deemed worthy of more attention; e.g., discussing collected differences in a moderated plenary discussion), and (iii) *organize* (a group derives shared understanding of the relationships among concepts; e.g., discussing importance and impact of variability).

The process consists of the following activities also depicted in Figure 3:

Explain goals and agenda. This can be seen as a "warm-up" step. The moderator explains the goals of the workshop and the agenda. Participants report on the key functionality of subsystems they are responsible for to provide a starting point for further discussion.

Assign participants. The moderator confirms assignment of participants to subgroups focusing on selected subsystems based on their knowledge and background.

Describe significant variability of subsystem. Each subgroup discusses the most significant variability regarding a particular subsystem by analyzing the last few projects they have been involved in. Participants also consider the variability of other subsystems influencing the variability of their own subsystem. Participants write one statement about a variability one a yellow variability card.

Discuss identified variability. In a moderated plenary discussion the variability elicited by each group is discussed one by one, rephrased where necessary depending on participants' comments, and posted on a flipchart (one per subsystem) by the scribe.

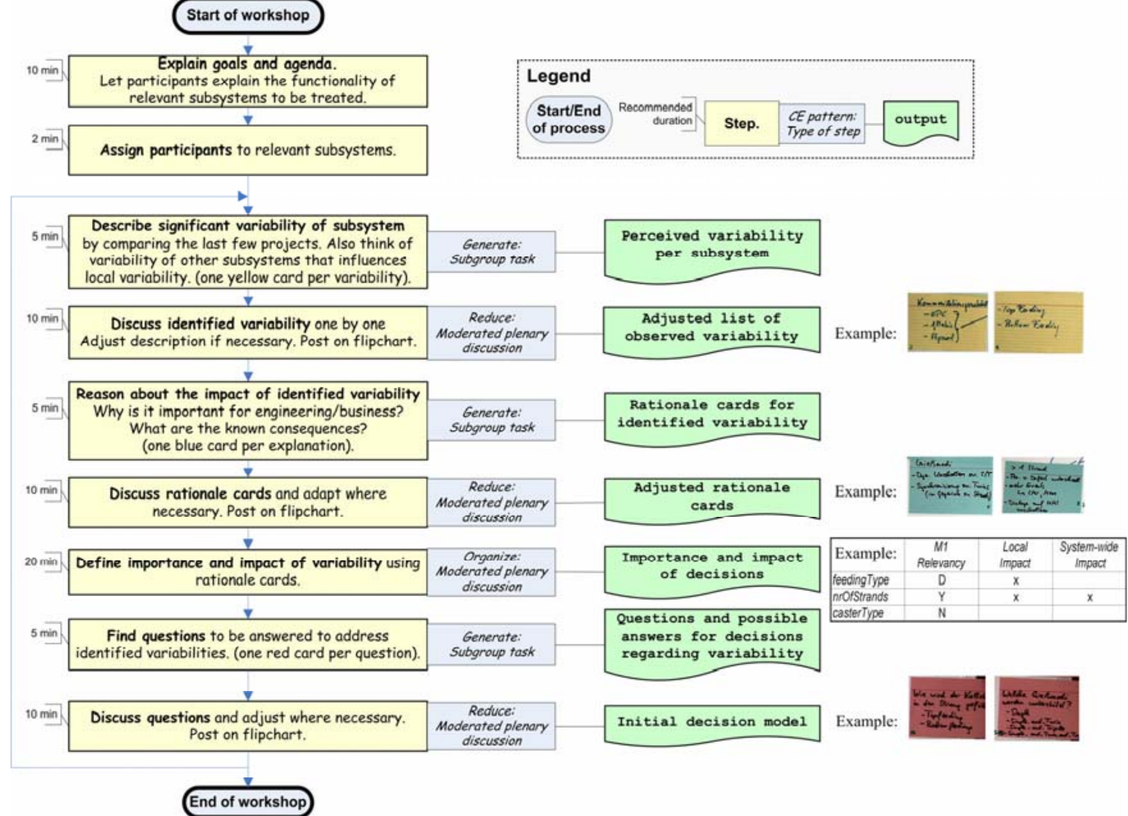


Figure 3: A value-based, collaborative process for eliciting product line variability.

Reason about the impact of identified variability. Each subgroup discusses the impact of the identified variability on engineering and/or business. The moderator asks questions such as: Why is this variability important for engineering? Why is this variability important for business and application engineering? What are the possible consequences of not taking the variability into account? Participants write each impact statement on separate blue rationale cards.

Discuss rationale cards. In a moderated plenary discussion captured impacts are discussed one by one, adapted where necessary, and posted on the respective flipchart besides the variability they belong to.

Define importance and impact of variability. Importance and impact of variability are assessed in a moderated plenary discussion using the rationale cards. Different categories (e.g., based on development phases or scope of impacts) are used to analyze the value of elicited variability. The scribe captures the following information in a table arranged at the right of each flipchart: "importance in early project phases (i.e., for milestone 1)", "local impact within the subsystem", and "system-wide impact beyond the subsystem". Variability is important for early project phases if not handling it would lead to major problems in terms of development effort, cost, or possible failures.

Find questions. Based on the variability cards, the related rationale cards, and the information regarding importance and impact, each group suggests questions that might be asked to stakeholders to address the identified variability in product derivation. Questions have to be found at least for that variability marked as important early in a project and/or having a significant system-wide impact. Participants put down each question on separate red question cards.

Discuss questions. Guided by the moderator all participants discuss each question one-by-one and rephrase it where necessary. The scribe puts them on the flipchart for the subsystem they belong to.

The output of the process typically is one flipchart per discussed subsystem containing the following information (cf. Figure 1):

Variability cards describe the differences that occurred in the last few projects. Variability from other subsystems that influenced the local variability of the subsystem is also described.

Rationale cards denote why the variability is important for engineering and/or business.

Question cards represent the variation points to be modeled in form of questions. The possible answers to these questions are described on the variability cards. Answering questions means choosing certain variants.

The *importance and impact table* describes the importance of variability for early phases of a project (i.e., for milestone 1) and whether it has local (subsystem) and/or system-wide impact.

This information can be used to create an initial variability model. As we follow a decision-oriented approach [6] the variability models can be easily created based on the question cards providing the question and name of the decision, the variability cards providing the possible answers, and the rationale cards allowing to model meta-information for decisions. The importance and impact table informs the modeler about the priorities.

The variability elicited by the process is a good starting point for variability modeling. However, we stress that it needs to be complemented with variability modeling activities on the more technical level. For example, we use automated tools to find potential variability in existing architectural models and system configuration files.

We have also developed an electronic process guide that captures the process in a more formal manner. The process guide was created using the Process Composer tool (see Figure 4) provided as part of the Eclipse Process Framework[1]. The different activities are modeled as tasks and the outcomes (i.e., the various cards) as work products. Disciplines group similar tasks together (e.g., moderated plenary discussions). The process guide is available as a hypertext allowing users to use the process from various perspectives.

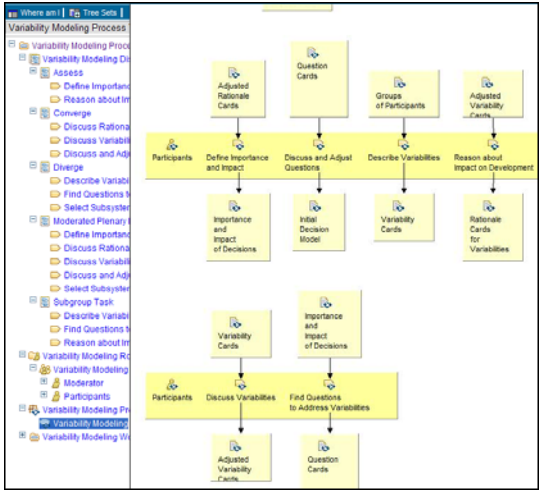


Figure 4: Process Guide.

[1] www.eclipse.org/epf/

4. Conclusions and Future Work

We presented a collaborative process for eliciting variability during product line adoption that incorporates value-based principles and involves people with an intimate knowledge about a subsystem's variability (developers, software architects, etc.). The process aims to capture the most relevant variability, i.e., variability that addresses development risks and important business aspects in application engineering.

We successfully applied the approach in several workshops conducted with technical stakeholders such as software architects, developers, and technical project managers of our industry partner Siemens VAI. We learned that such a process is highly valuable to find out what variability should be modeled at which level of granularity – an aspect that most existing approaches do not address.

We believe that collaborative approaches nicely complement more technical approaches to variability modeling. Collaborative processes have been defined in diverse domains based on the same general patterns of group collaboration we also adopted in our process [4, 11]. The process was developed in the context of our decision-oriented SPLE approach in mind. We believe, however, that it is general enough to be useful for other variability modeling approaches too, especially because of the use of well-known and proven collaboration engineering patterns.

Collaborative technologies such as Group Support Systems[2] can be applied to support the collection, structuring, and assessment of ideas in the process. The successful use of such technologies has been demonstrated in related areas such as requirements negotiation [4] or product line scoping [11].

We plan to conduct further workshops with Siemens VAI and other companies in the future to enhance and validate our process. We also plan to run workshops with sales people and marketing staff in order to also elicit variability based on business objectives and marketing decisions.

Acknowledgements

This work has been conducted in cooperation with Siemens VAI and supported by the Christian Doppler Forschungsgesellschaft, Austria. We would like to express our sincere gratitude to the staff of Siemens VAI for their support and for sharing valuable insights.

[2] http://www.groupsystems.com

References

[1] 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.

[2] 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 *Lecture Notes in Computer Science: Software Product-Family Engineering, 5th International Workshop, PFE 2003*, vol. LNCS 3014, F. van der Linden, Ed.: Springer Berlin / Heidelberg, 2003, pp. 66-80.

[3] S. Biffl, A. Aurum, B. Boehm, H. Erdogmus, and P. Grünbacher, *Value-Based Software Engineering*: Springer, 2005.

[4] B. W. Boehm, P. Grünbacher, and R. O. Briggs, "Developing groupware for requirements negotiation: lessons learned," *IEEE Software*, vol. 18(3), pp. 46-55, 2001.

[5] R. O. Briggs, G. J. de Vreede, and J. F. Nunamaker Jr., "Collaboration Engineering with ThinkLets to Pursue Sustained Success with Group Support Systems," *Journal of Management Information Systems*, vol. 19(4), pp. 31-64, 2003.

[6] D. Dhungana, R. Rabiser, and P. Grünbacher, "Decision-Oriented Modeling of Product Line Architectures,"Proc. of the *Sixth Working IEEE/IFIP Conference on Software Architecture*, Mumbai, India, IEEE Computer Society, 2007.

[7] D. Dhungana, P. Grünbacher, and R. Rabiser, "DecisionKing: A Flexible and Extensible Tool for Integrated Variability Modeling," in *First International Workshop on Variability Modelling of Software-intensive Systems - Proceedings*, K. Pohl, P. Heymans, K.-C. Kang, and A. Metzger, Eds. Limerick, Ireland: Lero - Technical Report 2007-01, 2007, pp. 119-128.

[8] D. Dhungana, R. Rabiser, P. Grünbacher, H. Prähofer, C. Federspiel, and K. Lehner, "Architectural Knowledge in Product Line Engineering: An Industrial Case Study,"Proc. of the *32nd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, Cavtat/Dubrovnik, Croatia, IEEE Computer Society, 2006.

[9] C. Federspiel, J. Bogner, N. Hübner, R. Leitner, W. Oberaigner, K. König, and L. Lindenberger, "Next Generation Level2 Systems for Continuous Casting,"Proc. of the *5th European Continuous Casting Conference (ECCC)*, Nice, France, IOM Communications Ltd, 2005.

[10] T. Käkölä and J. C. Duenas, *Software Product Lines - Research Issues in Engineering and Management*: Springer, 2006.

[11] M. A. Noor, R. Rabiser, and P. Grünbacher, "Agile product line planning: A collaborative approach and a case study," *The Journal of Systems and Software (to appear)*, 2007, (doi:10.1016/j.jss.2007.10.028).

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

[13] pure systems GmbH, "Variant Management with pure::variants, Technical Whitepaper," http://www.pure-

systems.com/fileadmin/downloads/pv-whitepaper-en-04.pdf, 2006.
[14] R. Rabiser, P. Grünbacher, and D. Dhungana, "Supporting Product Derivation by Adapting and Augmenting Variability Models,"Proc. of the *11th International Software Product Line Conference (SPLC 2007)*, Kyoto, Japan, IEEE CS, 2007.
[15] R. Rabiser, D. Dhungana, P. Grünbacher, K. Lehner, and C. Federspiel, "Product Configuration Support for Non-technicians: Customer-Centered Software Product-Line Engineering," *IEEE Intelligent Systems*, vol. 22(1), pp. 85-87, 2007.
[16] K. Schmid, "Planning Software Reuse – A Disciplined Scoping Approach for Software Product Lines," PhD Theses in Experimental Software Engineering, Fraunhofer IRB, 2003.
[17] 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*, vol. 53(3), pp. 259-284, 2004.
[18] M. Sinnema, S. Deelstra, J. Nijhuis, and J. Bosch, "COVAMOF: A Framework for Modeling Variability in Software Product Families," in *Lecture Notes in Computer Science: Third Software Product Line Conference (SPLC 2004)*, R. Nord, Ed.: Springer Berlin / Heidelberg, 2004, pp. 197-213.

Tracing between Features and Use Cases: A Model-Driven Approach

Mauricio Alférez[1] Uirá Kulesza[1] Ana Moreira[1] João Araújo[1] Vasco Amaral[1]

[1]*CITI/Dept. Informática, FCT, Universidade Nova de Lisboa*
2829-516 Caparica, Portugal
{mauricio.alferez, uira, amm, ja, vasco.amaral}@di.fct.unl.pt

Abstract

Use cases and features applied together could form the basis for a systematic method to identify and model SPL requirements. This paper presents a model-driven approach which addresses the tracing between features and use cases. This adopts a simple and flexible metamodel integration strategy to support the tracing between variability and requirements models. It also defines a set of activities in domain engineering to model, specify and trace SPL requirements and features. These activities are illustrated using a home automation system product line.

1. Introduction

Software product lines have emerged as a feasible and relevant software development paradigm that allows to the companies to perform important improvements in time to market, cost, productivity and quality [1-3]. This is achieved by enabling the strategic management of common and variable features of a system family. A system family is defined as a set of programs that shares common functionalities and maintain specific functionalities that vary according to specific family members. A Software Product Line (SPL) can be seen as a system family that addresses a specific market segment [1].

Several SPL development approaches have been proposed [1-5]. Most of them include activities of identification of common and variable features of the SPL by means of domain analysis activities. A feature can be seen as a system property or functionality that is relevant to some stakeholders and is used to capture commonalities or discriminate among products in SPLs [4]. The SPL features are generally represented in domain analysis using feature models [6], however, other requirements models, such as, use cases, scenarios and state machines, could be also used to better describe the SPL requirements. In the subsequent SPL development stages, the feature models, as well as the complementary requirements models are used along all the process as a reference to guide the SPL development.

The majority of the SPL approaches offer processes to elaborate feature and additional requirements models, however, most of them do not address traceability between these models explicitly. Some authors [3, 7-9] have presented some directions on how to manage the traceability between use cases and features. Nevertheless, they do not offer an easy to evolve and to implement strategy based on current and available model-driven tools and techniques. In addition, most of them do not show explicitly how to use the traceability information between feature and requirements models to generate important traceability views or specific models according to different SPL configurations.

This paper proposes a base metamodelling strategy and some domain engineering activities that allow to the developer to trace between variability and requirements models. This approach aims at being extensible and adaptable to any variability and requirements engineering modeling technique with a well-defined metamodel. In this paper, we have focused on exemplifying it with feature and use case models as the variability and SPL requirements models, respectively.

Our approach is supported by model-driven tools and techniques which use the information provided by the trace links to automatically derive other useful models such as different traceability views of the requirements artifacts. With this work we aim at to establish a first stepping stone to support the future incorporation of more domain and application engineering activities.

This paper starts with an overview of our approach in Section 2 and follows by illustrating the approach main activities using a home automation system case study, in Section 3. Section 4 discusses lessons learned, Section 5 shows related work and, finally, Section 6 concludes the paper and presents some future work.

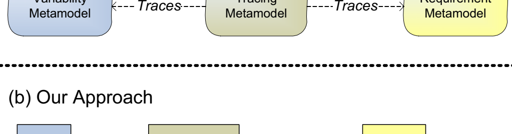
2. A Model-driven tracing approach

To model, specify and trace SPL requirements, we followed a model-driven approach wherein the process is supported by models, metamodels and bindings between them. The adopted strategy and the approach main activities are described next.

2.1. Approach strategy overview

Figure 1(a) shows an overview of the strategy we have adopted in our approach. In this strategy, a variability model is used to represent the common and variable SPL features, one or more requirements models are used to detail the complete specification of the SPL requirements, and a tracing metamodel is used to link abstractions between the variability and the requirements models to enable the navigation between them using MDD techniques and tools.

Figure 1(b) presents a general schema of the metamodel organization when use cases and feature models are used to accomplish the strategy shown in Figure 1(a). The tracing between use cases and features is supported by the definition of a traceability metamodel. It allows linking relevant abstractions of the use case model (e.g., use cases and actors, summarized in Figure 1(a) as "*Use Case Element*") and feature models, to integrate requirements and variability artifacts.



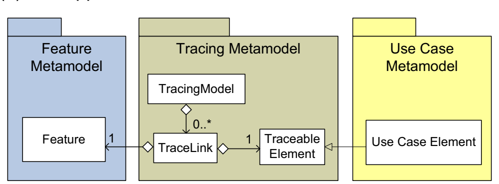


Figure 1. Our approach overview

2.2. Approach main activities

Our approach is organized in a set of activities from the domain and application engineering perspectives. In this paper, we focus only on some of the domain analysis activities in the domain engineering perspective. Figure 2 shows how the activities are organized and the artifacts produced during their execution[1].

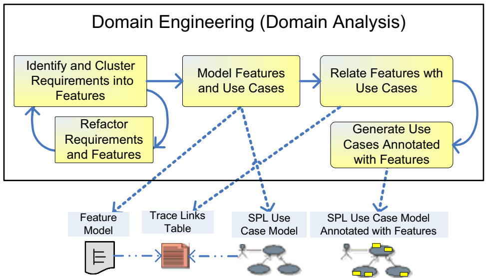


Figure 2. Main activities and artifacts

At the domain analysis level, our approach models and specifies the SPL requirements and generates tracing views of the relationships between the artifacts. This is achieved by performing the following activities:

(i) Identify and cluster requirements into features. The SPL requirements can be elicited using traditional requirements engineering techniques. During this activity, the requirements are also organized in clusters according to the specific SPL features they are related to.

(ii) Refactor requirements and features. The SPL requirements could result to be linked to more than one feature during the requirements clustering activity. We propose to refactor such requirements to try to assure that each one of them is related to only one common or variable feature. As a consequence of the requirements refactoring, the features must be also refactored to accommodate the new requirements clusters.

(iii) Model features and use cases. This activity structures and represents the SPL requirements and the variability using use case and feature models. Use case models specify the functional requirements and feature models specify the SPL features and variability information.

(iv) Relate features with use cases. The relationships between features and use cases are explicitly marked in a table, as a first step towards supporting traceability between them.

1 Although these activities are organized sequentially, they are typically executed iteratively.

(v) Generate use cases annotated with features. In this activity, the relationships between features and use cases, as well as the SPL use case and features models are used by a MDD tool developed for our approach to automatically generate specific use case models annotated with features [10]. In the annotated model, each use case is shown with the respective features related to it. Therefore, it is also possible to obtain the set of use cases related to a specific feature. This allows to the domain analysis engineers and architects to reason about how each use case is related to the SPL features and to analyze the impact of changing specific features in SPL requirements.

3. Applying the approach to an example

To illustrate the activities described in the previous section, we have chosen a home automation system, called Smart Home (see also [3]). Smart homes have a wide variety of electronic and electrical devices which include lights, thermostats, blinds and fire detection sensors, security devices such as cameras, glass break and motion detection sensors, white goods such as washing machines, communication devices such as phones and entertainment devices such as televisions.

The Smart Home system is designed to coordinate the behavior of the devices to fulfill complex tasks automatically. It also enables the inhabitants to visualize and control the status of the devices from a common user interface.

This system is a SPL case study that is being developed in the context of the European AMPLE project [11]. Due to its complexity, we will focus only on a subset of the security module.

3.1. Identify and cluster requirements into features

Requirements identification can be accomplished by inspecting existing documents that describe the problem domain (i.e., existing catalogues [12]), stakeholders interview transcripts or by using mining techniques [13, 14]. Other approaches such as [7] and [3] already address this activity in detail.

During requirements identification we obtained a subset of the requirements *(Ri)* of the Smart Home security module. By inspecting these requirements they were clustered into features *(Fi)* as shown in the following tabular descriptions (Table 1).

Table 1. Smart Home security module requirements and clusters

F1. Room Surveillance
R1. The system shall provide room surveillance.
F2. Indoor Camera Surveillance and Indoor Motion Detection
R2. Room surveillance shall be accomplished by indoor camera surveillance or indoor motion detection.
R3. Indoor motion detection and camera surveillance shall be configured and activated to be used.
R4. The system shall activate indoor camera surveillance when activates indoor security.
F3. Admittance Control
R5. The system shall be able to identify users.
R6. The system shall automatically open the front door when it has identified an authorized user.
R7. The system shall automatically close the door after 2 minutes.
R8. The system will be used by the inhabitants and the house owner that may or may not be also an inhabitant.
R9. The home owner shall be able to configure all the security services.
F4. Identify User by Biometrical Analysis, Smart Card or PIN
R10. The system shall be able to identify users by means of biometrical analysis, smart card or PIN.
F5. Intrusion Detection
R11. The inhabitant shall be able to activate indoor and outdoor security through configuring security management.
F6. Glass Break Detection
R12. Glass break detection shall be configured and activated to be used.
R13. The system shall activate as minimum glass break detection as an intrusion detection mechanism.
F7. Outdoor Motion Detection and Camera Surveillance
R14. Outdoor security shall be accomplished by motion detection or camera surveillance.
R15. Outdoor motion detection and camera surveillance shall be configured and activated to be used.

3.2. Refactor requirements

During the identification of the SPL requirements and their clustering into features, we refactor, whenever possible, the requirements to be related to only one feature. In addition, the features must be also refactored to accommodate new requirements clusters. Refactoring is important to facilitate the definition of trace links between requirements and features. It also contributes to a better modularization of the SPL requirements by improving the separation of the variable parts of each requirement [15].

Table 2 shows the refactorizations of some of the requirements shown in Table 1. For example, requirements *R2* and *R3* were refactored into *R2A* and *R3A* to address *Indoor Camera Surveillance,* and *R2B* and *R3B* to address *Indoor Motion Detection* separately. *R2* and *R3* refactoring motivated to refactor the feature *F2* which was split into feature *F2A*, to group the parts of the *R2* and *R3* requirements addressing *Indoor Camera Surveillance (i.e., R2A and R3A)*; and feature *F2B*, to group the parts of the *R2*

and *R3* requirements addressing *Indoor Motion Detection* (i.e., *R2B* and *R3B*).

Table 2. Refactoring of some of the Smart Home security module requirements and features

F2A. Indoor Camera Surveillance
R2A. Room surveillance shall be accomplished by camera surveillance.
R3A. Indoor camera surveillance shall be configured and activated to be used.
R4. The system shall activate indoor camera surveillance when activates indoor security.
F2B. Indoor Motion Detection
R2B. Room surveillance shall be accomplished by indoor motion detection.
R3B. Indoor motion detection shall be configured and activated to be used.
F4A. Identify User by Biometrical Analysis
R10A. The system shall be able to identify users by means of biometrical analysis.
F4B. Identify User by Smart Card
R10B. The system shall be able to identify users by means of smart cards.
F4C. Identify User by PIN
R10C. The system shall be able to identify users by means of a PIN.
F7A. Outdoor Camera Surveillance
R14A. Outdoor surveillance shall be accomplished by camera surveillance
R15A. Outdoor camera surveillance shall be configured and activated to be used.
F7B. Outdoor Motion Detection
R14B. Outdoor surveillance shall be accomplished by outdoor motion detection.
R15B. Outdoor motion detection shall be configured and activated to be used.

3.3. Model use cases and features

It is possible for textual requirements to express variability by using a certain set of keywords or phrases. However, documenting requirements variabilities in that manner could lead to ambiguities [3]. Variability at the requirements level must be documented systematically and unambiguously to support traceability between different kinds of artifacts. To achieve this goal, we model and specify SPL requirements and variabilities in models, such as use case and feature models.

After refactoring requirements and features (Section 3.2), we build the feature model. Figure 2 shows the feature model for the Smart Home security module. This model has three main features: *Room Surveillance*, *Admittance Control* and *Intrusion Detection*. *Room Surveillance* is an optional feature that includes *Indoor Camera Surveillance* and, optionally, *Indoor Motion Detection*. The inhabitant can be admitted to enter the house after passing either a *Biometrical Analysis*, *Smart Card*, or entering a *PIN*. In case of selecting intrusion detection, the *Glass Break Detection* is included and optionally, motion detection sensors and/or cameras for outdoor security.

We can derive the SPL use cases from the requirements and features identified previously. Use case modeling is used to better structure the SPL requirements and add more semantics to the features [16]. These, together with feature modeling are used in the SPL development process to guide and help the developers' activities.

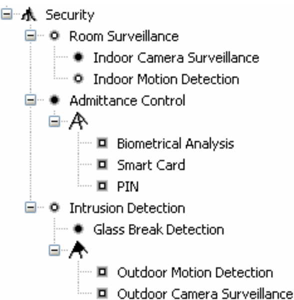


Figure 2. Feature model for the Smart Home security module

Figure 3 shows the use case model of our case study. This diagram shows that *Activate Indoor Security* includes *Activate Camera Surveillance* and can eventually extend its behavior with *Activate Motion Detection*. Similarly, *Activate Outdoor Security* includes to *Activate Glass Break Detection* and can eventually extend its behavior with *Activate Motion Detection*. To open the front door, the inhabitant must be identified and this can be done by using a smart card, a PIN or a more sophisticated process such as biometrical analysis. Finally, the *House Owner* actor is the person in charge to configure the security options.

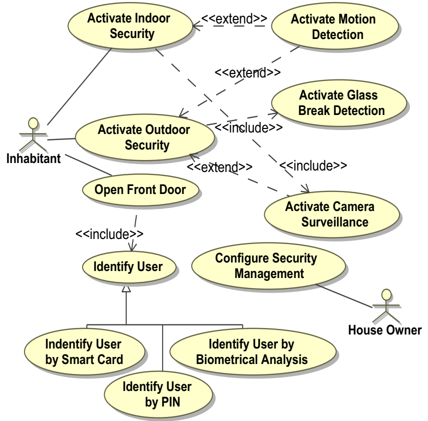


Figure 3. Use case model of the Smart Home security module

The first identified SPL features and use cases can be refined and incremented to consider new variabilities or products that need to be included in the family. Both use case and feature models must be updated when new features are considered or existing ones need to be modified or removed.

3.4. Relate features to use cases

The relationships between features and use cases in the Smart Home security module are specified in the Table 3. By inspecting the requirements and features in Sections 3.1 and 3.2, we related, for example, the *Open Front Door* use case with *Admittance Control*, refined into *Biometrical Analysis*, *Smart Card*, and *PIN* features because to open the front door, the system requires *Admittance Control.*

Table 3. Relationships between use cases and features

Features \ Use Cases	Identify User	Identify User by Biometrical Analysis	Identify User by Smart Card	Identify User by PIN	Open Front Door	Activate Indoor Security	Activate Outdoor Security	Activate Motion Detection	Activate Glass Break Detection	Activate Camera Surveillance	Configure Security Management
Room Surveillance						x		x		x	x
Indoor Camera Surveillance						x				x	x
Indoor Motion Detection						x		x			x
Admittance Control	x				x						x
Biometrical Analysis		x			x						x
Smart Card			x		x						x
PIN				x	x						x
Intrusion Detection							x				x
Glass break detection							x		x		x
Outdoor Motion Detection							x	x			x
Outdoor Camera Surveillance							x			x	x

The information provided by this kind relationships table is used as the basis to support forward and backward traceability between features and use cases; and the reasoning about the impact of feature interactions in the SPL requirements expressed by means of the use cases models. In this paper, we focus on the description of the traceability functionalities. The information about feature interactions offered by our approach will be useful during the elaboration and design of product line architectures to allow an adequate modularization and implementation of their respective features.

3.5. Generate use cases annotated with features

The relationships established in the previous activity allow the generation of special use case models annotated with features.

Different kinds of traceability views can be implemented to represent features and use cases. These views allow the domain analysis engineers and architects to reason about the domain analysis artifacts interdependencies. Currently, the traceability views that our approach generates in this activity are: (i) A tree structure that shows the list of use cases with the related features and optionally, the list of features with the related use cases; and (ii) a use case model annotated with the respective related features.

Figures 4 and 5 are examples of the first type of traceability view between the features and use cases of our case study. The expanded branch in Figure 4 shows the features related to the use case *Open Front Door*.

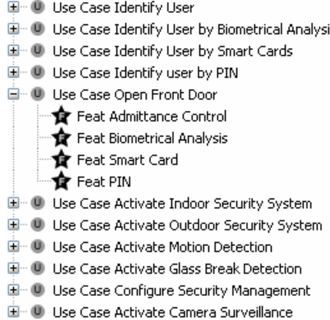


Figure 4. Features related to use cases

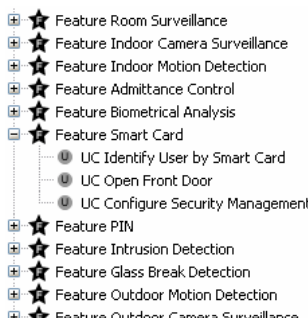


Figure 5. Use cases related to features

Similar to the Figure 4, Figure 5 depicts the use cases related to a specific feature from our case study. It shows that the feature *Smart Card* is related to the use cases: *Identify User by Smart Card*, *Open Front Door* and *Configure Security Management*.

4. Discussions and lessons learned

Some of the lessons learned during the execution of this work are discussed next.

Benefits of implementing this approach. Traceability between the variability and requirements models has been addressed using a traceability metamodel which integrates and unifies the feature and use cases metamodels (section 2.1). This base unifying strategy has some benefits like flexibility, simplicity, maintainability and extensibility. *Flexibility,* because the strategy can be applied to any requirements notation or technique with a well-defined metamodel. *Simplicity,* because the integration between the metamodels of the feature and requirements models is easy to understand and to implement. *Maintainability,* because each one of the main concerns in the metamodel, i.e., variability, requirements and traceability, can be modified and evolved relatively in isolation, causing few side-effects in the other metamodels and the MDD tools that use them. Finally, it is *extensible* because new elements in the tracing metamodel could be added with relative ease, e.g, abstractions that allow recording the rational employed to establish the relationships.

Tracing between features and use cases. Currently, our approach relates the use case behavior to specific SPL features. However, there are cases where only a "portion" of the use case is related with a specific feature. In those cases, the following strategies can be adopted: (i) refactor the use case to "extract" the variable part to an extension use case; or (ii) represent more fine grained relationships between features and use cases. The first strategy is compatible with our approach and it does not require any change on the approach models and tool. On the other hand, the adoption of strategy (ii) can be addressed by specifying each use case by means of activity diagrams and by allowing their customization using composition rules. Both strategies are being investigated to improve our approach.

Non-functional requirements modeling. Currently, our approach does not offer explicit support for specifying and modeling non-functional requirements (NFRs). We are investigating two different ways to incorporate the modeling of NFRs: (i) to represent the NFRs directly in the feature model, thus making possible the creation of relationships between NFRs (modeled as features) and use cases in our traceability table; and (ii) to adopt additional NFRs modeling notations, such as goal models [12], and to define the tracing between the NFRs and the SPL features. Although we intend to explore and compare both alternatives, we have already identified that the adoption of the first strategy brings the benefit to allow to reason about NFRs interdependence as a problem of feature interaction [17].

5. Related work

Some approaches have addressed the modeling of SPL requirements using feature models and UML (e.g., use cases models). For example, Czarnecki et al [16] and Bragança et al [18] use graphical elements in their models, such as, presence conditions or notes, to indicate variability. Similarly, Gomaa [9] requires the use of stereotypes to indicate common or variable parts in the UML models. As a result, all these mechanisms scatter and pollute variability information over the UML models which difficult their traceability and evolution.

On the other hand, Gomaa [9], Pohl [3] and Griss et al [7] describe their processes that include traceability activities, but do not provide any specific tool support for modeling, tracing and generate SPL requirements. Other authors like Eriksson et al [8] employ existing commercial requirements tools to represent the artifacts. However, they do not show how existing model-driven development technologies can be adopted to promote the seamless tracing between the different SPL requirements models used.

We believe that the feature model should be used as the higher level view of the product family. Variability must be only expressed in the feature models to avoid polluting other models with variability information (usually expressed in notes, presence conditions or stereotypes). Finally, we recognize that it is also fundamental to define how existing model-driven development technologies can be used to allow the composition and tracing between all the SPL requirements models. In this paper, we set the base of an approach that addresses all these needs. Additionally, we show how model-driven development techniques can be used to process feature and requirements models to support traceability and also to derive other useful models.

6. Conclusions and future work

In this paper, we presented a model-driven approach to model, specify and trace SPL features and

requirements. We adopted a simple metamodel integration strategy to allow the tracing between features and use cases. We also illustrated some of the domain analysis activities in the domain engineering perspective, using part of the security module of a home automation system SPL called Smart Home.

We are currently extending our approach and supporting tool [10] to address some other concerns, such as: (i) to support the modeling and tracing of NFRs in the context of SPLs; (ii) to offer interesting trace views to reason about feature and requirement interactions; (iii) to show how the scenario technique can be used as a complementary technique to describe the requirements; (iv) to deal with volatile requirements and support the modeling of activity diagrams and composition rules as proposed by the Volatile Concerns approach [15].

Finally, in the context of the AMPLE project, we are defining a more complete approach, which provides support to trace from features and requirements models to artifacts of latter software development stages, such as, architecture models and source code.

Acknowledgement. The authors are partially supported by European Commission Grant IST-33710: Aspect-Oriented, Model-Driven Product Line Engineering (AMPLE).

References

[1] P. Clements and L. M. Northrop, Software Product Lines: Practices and Patterns. Boston, MA, USA: Addison-Wesley, 2002.

[2] D. M. Weiss and C. T. R. Lai, Software Product-line Engineering: a Family-based Software Development Process. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

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

[4] K. Czarnecki and U. W. Eisenecker, Generative Programming: Methods, Tools, and Applications: ACM Press/Addison-Wesley Publishing Co., 2000.

[5] J. Greenfield and K. Short, Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. Indianapolis, IN, USA: Wiley, 2004.

[6] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study", Software Engineering Institute, Technical report, CMU/SEI-90-TR-021, 1990.

[7] M. L. Griss, J. Favaro, and M. d' Alessandro, "Integrating Feature Modeling with the RSEB", presented at 5th International Conference on Software Reuse, 1998.

[8] M. Eriksson, J. Börstler, and K. Borg, "The PLUSS Approach - Domain Modeling with Features, Use Cases and Use Case Realizations", in 9th International Conference on Software Product Lines, Rennes, France, Springer, 2005, pp. 33-44.

[9] H. Gomaa, Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures: Addison-Wesley, 2004.

[10] "AMPLE Project Research Group at FCT/UNL", http://ample.di.fct.unl.pt/.

[11] AMPLE, "Ample Project", http://www.ample-project.net/.

[12] L. Chung, B. Nixon, E. Yu, and J. Mylopoulos, Non-Functional Requirements in Software Engineering, 1 ed: Kluwer Academic Publishers, 1999.

[13] A. Sampaio, R. Chitchyan, A. Rashid, and P. Rayson, "EA-Miner: A Tool for Automating Aspect-Oriented Requirements Identification", in Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, Long Beach, CA, USA, ACM Press, 2005, pp. 352-355.

[14] I. John, J. Dörr, and K. Schmid, "User Documentation Based Product Line Modeling", Fraunhofer IESE, Technical report No. 004.04/E version 1.0, 2004.

[15] A. Moreira, J. Araújo, and J. Whittle, "Modeling Volatile Concerns as Aspects", presented at 18th Conference on Advanced Information Systems Engineering, Luxemburg, Luxemburg, 2006.

[16] K. Czarnecki and M. Antkiewicz, "Mapping Features to Models: A Template Approach Based on Superimposed Variants", presented at 4th International Conference on Generative Programming and Component Engineering, Tallinn, Estonia, 2005.

[17] M. Jackson and P. Zave, "Distributed Feature Composition: A Virtual Architecture for Telecommunications Services", IEEE Transactions on Software Engineering, vol. 24, pp. 831-847, 1998.

[18] A. Bragança and R. J. Machado, "Automating Mappings between Use Case Diagrams and Feature Models for Software Product Lines", in Proceedings of the 11th International Software Product Line Conference, Kyoto, Japan, IEEE Computer Society, pp. pp. 3-12.

Svamp — An Integrated Approach to Modeling Functional and Quality Variability

Mikko Raatikainen*, Eila Niemelä†, Varvana Myllärniemi*, Tomi Männistö*
*Helsinki University of Technology (TKK), †VTT Technical Research Centre of Finland
*{Mikko.Raatikainen,Varvana.Myllarniemi,Tomi.Mannisto}@tkk.fi, †Eila.Niemela@vtt.fi

Abstract

Software variability modeling is a complex task. To manage this complexity, we introduce an approach called Svamp. The main contribution of Svamp is to model concepts through ontologies and offer tool support for capturing functional and quality variability in software product family architectures. Variability description languages are defined by different ontologies that provide meta-models. For structural and functional descriptions, the concepts, properties, and rules are defined by Kumbang ontology. Quality Attribute ontology defines the domain knowledge of a specific quality attribute, while Quality Variability ontology provides the concepts and rules related to quality variation. The approach is exemplified by our integrated tool suite, provided as a plug-in for the Eclipse platform.

1. Introduction

Variability is the ability of software to be efficiently extended, changed, customized, or configured for use in a particular context [23]. Typically, variability is defined in software when software is developed for reuse. For example, variability is defined in the assets and the architecture of a software product family during the domain engineering phase. The developed variability is then taken advantage for differentiation of software. For example, defined variability in the assets of a software product family is used to derive the different products of a software product family. Software, which differs from other software by taking advantage of variability, is referred to as a variant.

Variability can become complex, since the number of potential variants grows exponentially when new variability is introduced. In addition, variability concerns and affects not only functionality but also quality attributes of software. Consequently, in order to ensure correctness of functionality of a variant and predict its quality properties even in the most complex circumstances, variability needs to be explicated such that it can be efficiently managed and even automated with tool support. Toward this end, an essential characteristic is clarity of underlying concepts for different software artifacts. Conceptual clarity is especially important when variability spans different software artifacts such as requirements, architectural elements, functionality, and quality. Such variability can even affect diverse concerns of stakeholders in an organization.

In this paper, we discuss an approach to capturing variability of a software product family called Software variability modeling practices (Svamp). For given functional and quality requirements, we outline concepts for modeling the structure of features and components that contribute to the functionality and quality attributes of the components. The modeling concepts have been defined rigorously as ontologies. The feasibility of the concepts is shown with an integrated tool suite. With the resulting model, derivation of system variants seems feasible such that the variants fulfill functional and quality requirements.

The rest of the paper is organized as follows. Section 2 provides background of the method. Section 3 describes the approach. In Section 4, the developed tool suite is introduced. In Section 5, we discuss experiences and future research. Section 6 draws conclusions.

2. Background

Software variability management has emerged recently, especially in the area of software product families that focus on enhancing development of a set of different variants within an organization [3]. A key issue in and a lesson learned about the success of software product families is that the products of a software product family follow the same fundamental structure, referred to as a software product family architecture [2]. Consequently, software product family architecture seems to be especially relevant from the point of view of variability, although other development artifacts are affected as well.

A software architecture describes the high-level structure

of a software system. Software architecture is an important means, for example, for performing different types of analysis and for achieving different quality attributes, and communication. Variability is added to software family architecture while still retaining other key aspects of software architecture. The state of the art and practice for managing software architecture is based on views and viewpoints. A view is "a representation of a whole system from the perspective of a related set of concerns" [8]. The guidelines for constructing and using a view are described in a viewpoint. The rationale for using different viewpoints is to take into account different stakeholder concerns, which are, in fact, a major intention of view-based approaches.

Several modeling approaches have been proposed to express variability. Feature models [11] are one of the first widely known approaches that take into account variability. A feature refers to user visible characteristics of a system. Recently, other modeling approaches peculiar to variability have emerged, such as ConIPF [7] and decision-oriented modeling [4]. These approaches introduce a modeling method with constructs for modeling software assets and variability within the assets.

In addition, different approaches to modeling variability in existing models of software assets have emerged. For example, orthogonal variability modeling [19] augments existing models with variability specific information. Covamof [22] augments existing models with a variability specific model and another model that captures dependencies of a variability model. The methods can therefore be used in conjunction with any software artifact such as requirements or detailed design, or with any architectural viewpoint.

The modeling concepts, however, focus typically on functionality or structure of software. Quality attribute variability, especially at the architectural level, seems still to be a research challenge [13].

3. Svamp modeling concepts

The Svamp approach is to model functional and quality variability at the architectural level. The approach adheres to state-of-the-practice in architecture description by applying different viewpoints. More specifically, a feature and structural viewpoint specifies the structure and functionality, and also variability within these. The structural viewpoint is also referred to informally as a component viewpoint. The elements in a structural viewpoint, that is, its components, are then augmented with quality attributes and, further, quality variability information. The architectural level was selected in the present approach since it seems to be especially significant for variability, as argued above.

Consequently, the approach uses several integrated models to model a software product family (Figure 1): a Kumbang model, consisting of structural and feature viewpoints

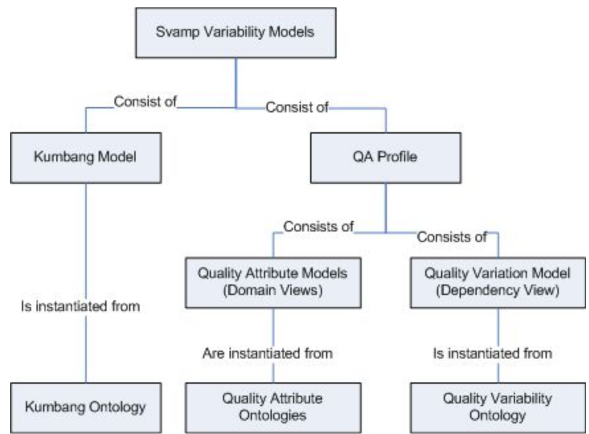


Figure 1. Svamp variability models and ontologies.

for functional and structural characteristics; a quality attribute profile, consisting of a quality attribute model for each quality attribute of the components in the structural viewpoint of the Kumbang model; and a quality variability model for expressing variability within these quality attributes. Each of these three models is defined in its own ontology; the corresponding ontology provides a meta-model for the modeling concepts.

Kumbang concepts form the basis for modeling since other models use the components defined in a Kumbang model; hence, the Kumbang model needs to be specified first. Roughly, Kumbang concepts synthesize existing feature modeling methods and structural modeling of architectural components, in particular Koala [25]. Kumbang adds explicit variability concepts into these methods and provides formal semantics for the concepts. In the following, we only briefly outline basic capabilities of Kumbang, whereas a comprehensive description can be found in [1].

The feature viewpoint is used for modeling feature types, which represent user visible functional characteristics of a system. Kumbang uses the term "type" to refer to an element in the variability model, while elements referring to a specific variant are, e.g., feature instances or simply features. Features can be composed such that other features are their subfeatures. Such a composition structure is specified within a feature type using subfeature definitions, which specify the cardinality and possible types of composed features. Further, feature types can inherit each other. Feature types can be characterized with attribute definitions, which represent name/value pairs. Finally, constraints can be used to specify more elaborate rules for selection of different feature instances; in a very simple case, by specifying that a certain feature requires another feature. Hence, Kumbang

feature modeling concepts synthesize many existing feature modeling methods, and can be used to capture typical variability constructs found in other feature modeling methods.

Structural viewpoint specifies component types. A component type represents a distinguishable architectural element with explicitly defined interfaces. The approach is ignorant as to whether a component actually refers to, e.g., a run-time or design-time element or a specific component technology, as far as the component adheres to this definition. Again, the term "component type" is used in the variability model similarly as in features. An interface type represents a set of operation signatures; these are attached to component types using interface definitions. Interface direction is provided or required and the interface can be optional. Components can be composed with each other. The construct used for specifying component composition is a part definition that specifies the cardinality of possible types of composed components. Similarly to feature types, component types can inherit each other, be characterized by attribute definitions, and specify constraints that restrict how instances in the structural viewpoint can be selected. Hence, the variants of structural viewpoint can differ in terms of composition of components, connections between the interfaces, and attribute values defined.

In order to integrate feature and structural viewpoint, implementation constraints can be used to specify relationships between them. In a very simple case, a specific feature may require a specific component. In general, the constraints can be bi-directional and impose many-to-many relations between viewpoints. Consequently, the implementation constraints can be as complex as can be specified with Kumbang constraint language [12].

To address quality attributes, the variability model needs to be then augmented with information on its quality characteristics. This is done by specifying the quality properties using the quality attribute model (QA model) and the quality variability model (QV model), defined separately from Kumbang (cf. Figure 1). The components of the structural viewpoint are supplemented with relevant quality profiles. Similarly to Kumbang, both QA model and QV model have been defined as ontology, the former as quality attribute (QA) ontologies and the latter as quality variability (QV) ontology.

Each QA ontology defines the technical dimension of the quality attribute. For example, the main concepts of the security QA ontology are security assets, attributes, threats, solutions, and metrics (Figure 2) [20], whereas the reliability QA ontology defines processes, methods, models, and metrics [26]. That is, QA ontologies are quality attribute specific, and, hence, the concepts in each ontology are different. QA ontologies are orthogonal and managed separately because different expertise is required for defining different QA ontologies. Furthermore, the concepts defined

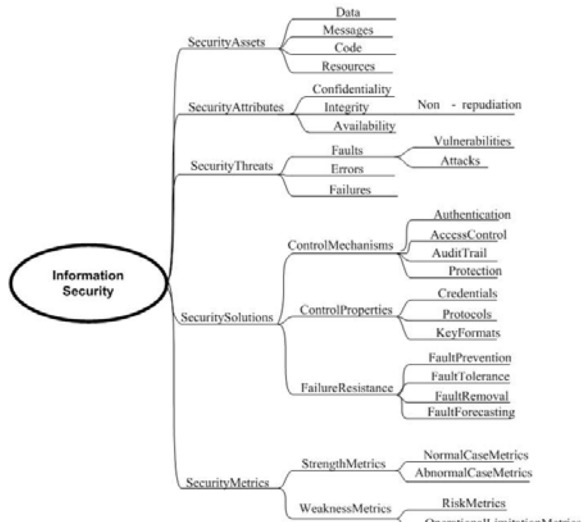


Figure 2. The security QA ontology [12].

in a QA ontology depend on the dissected entity: in defining the security QA ontology, the focus was on information security of service centric systems, while in the reliability QA ontology, the focus was on reliability-aware architecting. Thus, the scope of the reliability QA ontology is larger; therefore, more concepts have been defined.

The QA metrics concept (Figure 3) consists of metrics classes, e.g., strength metrics and weakness metrics. Concepts of QA metrics are common for all quality attributes, whereas only part of the metrics classes and actual metrics in the metrics classes can be shared by different QA ontologies and the others are quality attribute specific. Each metric has the following properties: description; purpose; target, i.e., where the metric can be used; applicability, i.e., when the metric can be used; a set of formulas; range value for the measurement; and the best value of the measurement unit. A rule set constrains the formulas and the used measurement unit by defining the set of targets of measurement, the set of value ranges for the measurement unit, and the time when the metric is valid.

The QV model is defined by four concepts: importance, scope, binding time, and dependency map. The importance of the QA is defined by three distinct property values, i.e., high, medium and low. The importance property is required for making decisions on QA variation. Rules related to the importance property define whether QA variation can take place, for example, QA of high importance cannot be changed at run-time or it can be lowered to the medium level only; in what circumstances QA variation is allowed, for example, QA of low importance can be removed while making

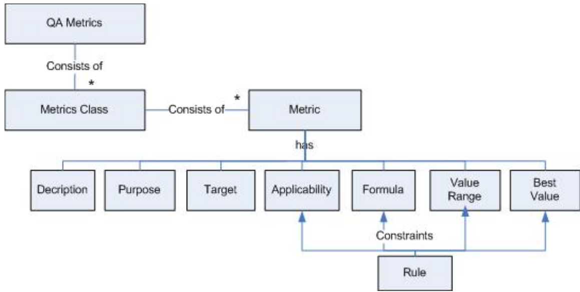


Figure 3. Concepts related to metrics of the QA ontology.

tradeoffs; and how quality attribute variation is to be carried out, for example, QA has to be fixed in product derivation. Some of the rules can be generic, but more often they are software product family specific.

The scope defines four granularity levels for QA variation. That is, scope determines where quality attribute variation can take place by defining a set of boundary types, possible values being family, product, service, or component. One of the values has to be selected. Scope selection restricts the types of appropriate metrics and measuring techniques. For example, at the family and product levels, only those metrics intended for system-level use (cf. Target in Figure 3) can be used. Thus, there are relations between the QA ontologies and the QV model that are considered while defining QA profiles by the QPE tool (see section 4.2).

Binding time defines when quality attribute variation can take place; quality attribute can be changed at design-time, in assembly, in start-up, or at run-time. The binding time is needed for making design decisions and required adaptations and tradeoffs, i.e., QA variation, between quality attributes. Design time tradeoffs are made by determining the optimal architecture with the help of quality evaluation methods and supporting tools, e.g., estimating reliability by the RAP method [9]. Run-time adaptation is made by specific algorithms implemented as part of middleware services. In [18], an example of run-time performance adaptation is given.

The dependency map describes relations between variable quality attributes. This information is required for making tradeoffs between quality attributes. So far, methods exist for making tradeoffs at design-time but no generic solution for making run-time tradeoffs. The QV model is defined in more detail in [16].

4. Tool support

The Svamp approach is supported with a tool suite developed as plug-ins on the Eclipse Platform [5]. Kumbang Modeler is used to model the structural and feature viewpoint whereas Quality Profile Editor (QPE) is used to model quality properties.

4.1. Kumbang Modeler

Kumbang Modeler [14] is a tool that can be used for creating the Kumbang model, that is, to model functional and structural variability in a software product family architecture from feature and structural points of view. The user can specify product family features, architectural elements, and relations between them using constraints. Kumbang Modeler hides the complexity of concrete syntax behind a graphical user interface (Figure 4) and guides the user in the modeling task.

Kumbang Modeler checks the model for syntactic correctness. Further, it checks that at least one valid product configuration can be derived from the model. That is, it checks that all required interfaces can be connected to corresponding interfaces, all constraints can be satisfied, and no cyclic loops exist in inheritance or part structures. This checking is implemented using an efficient smodels inference engine [21], a general-purpose inference tool based on the stable model semantics of logic programs.

After the structural and functional modeling is completed, the user of the tool can augment the model with quality profiles. For this purpose, the tool suite transforms the relevant information of the Kumbang model into the UML2 model specified by Eclipse UML2 meta-model, which is the format understood by the QPE plug-in. The process of using the QPE tool is described in the following.

4.2. Quality Profile Editor

The Quality Profile Editor (QPE) tool [6] takes QA ontologies as input. These ontologies are defined by the quality engineers by using an ontology definition tool, e.g. Protégé. In addition, the software family architect responsible for modeling also needs a list of quality requirements. The user interface of the QPE tool helps in instantiation of QA ontologies. QA ontologies are imported in OWL (Web Ontology Language) files [17] for the QPE tool.

The QPE tool produces a QA profile that instantiates the related QA and QV ontologies and, hence, contains the defined quality properties with metrics, quality variation rules, and dependencies on other quality properties in the same QA profile or in other QA profiles. In QA profiles, the QA properties are defined as UML stereotypes. UML defines profiles as a lightweight mechanism to extend the UML meta-model for adapting the language with domain specific constructs. These extensions are defined by stereotypes that can also contain properties and tag definitions used to set values to property attributes.

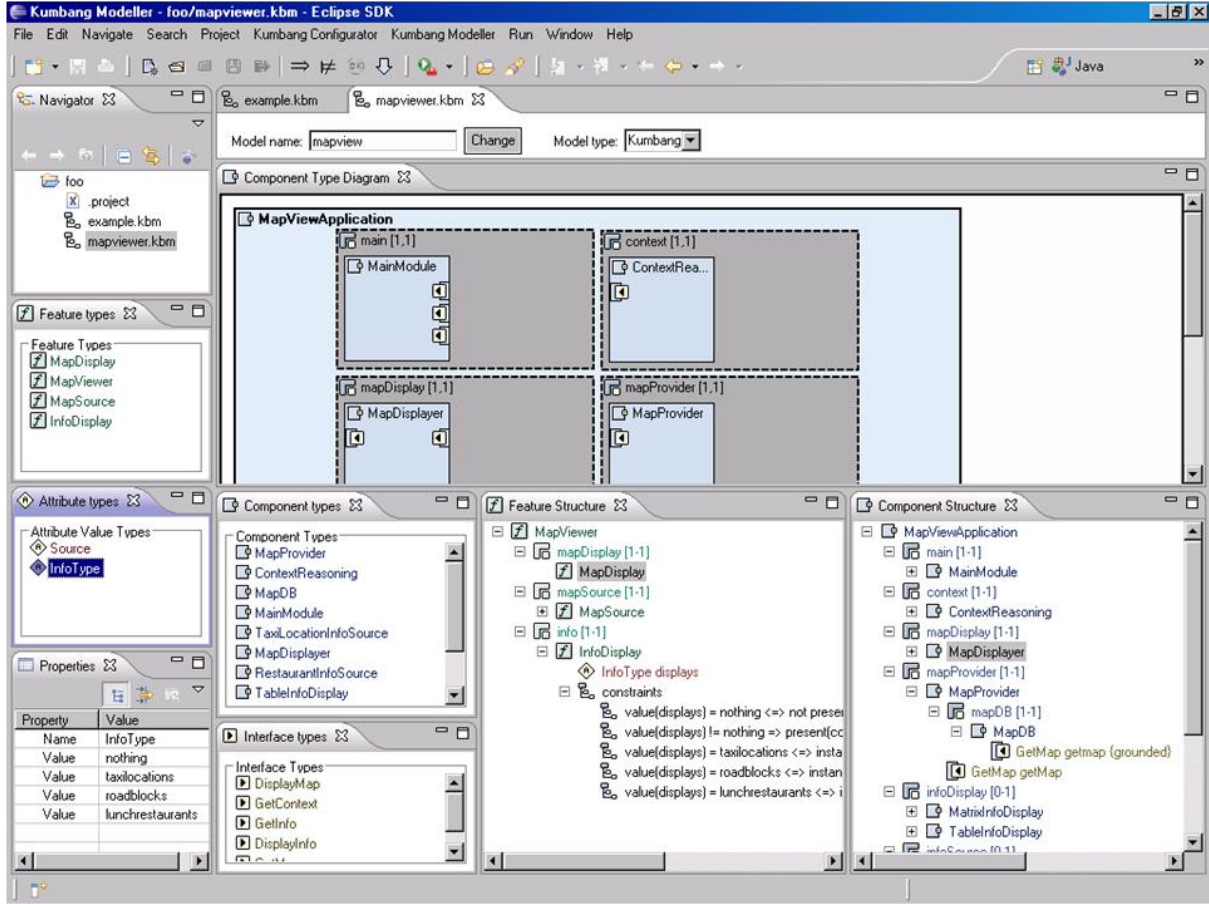


Figure 4. Kumbang Modeler graphical user interface[14].

Figure 5 depicts the user interface of the QPE tool. The left side is used to define quality properties that the family architecture has to meet. On the right side, the architect can select a quality property and bind an appropriate QA metric from one of the QA ontologies to it. Finally, dependencies on other quality properties are linked. For example in Figure 5, Req2 from Demo profile and R3 from the Reliability profile are linked to the Rel1 property.

QA profiles are stored as separate files because of their evolution management; new QA properties can be added and existing ones removed without affecting other QA profiles. However, the software family architect is responsible for checking dependencies between QA properties (inside one QA profile or between the properties in different QA profiles), because the QPE does not check dependencies automatically while the QA profiles are updated.

The stereotypes in the QA profiles are used for mapping the QA properties to the structural elements of the family model (Figure 6). Thus, quality properties as UML2 stereotypes facilitate the viewpoint based approach by enabling a separate focus on components, features, and quality attributes. The only concept the family architect can select while mapping QA properties to the architecture models is Binding time. The reason is that the architecture design is the earliest possible phase when a decision about timing of QA variation can be made. Mapping of QA properties to structural elements can be made with any Eclipse UML2 compatible plug-in; in the case of Svamp it was TOPCASED [24].

The QA property information incorporated into the models is used while evaluating the software product family architecture. Evaluation is made using appropriate evaluation tools, i.e., the evaluation tools are QA specific. The RAP tool [10] supports reliability and availability prediction from the models of software product family. Thus, to evaluate the satisfaction of reliability aspects, UML2 mod-

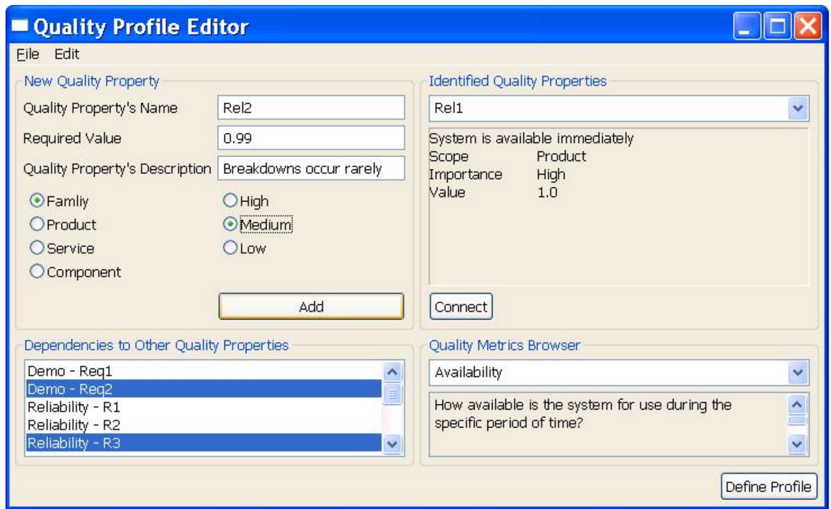


Figure 5. Defining quality properties with the QPE tool.

els produced by the TOPCASED tool are imported in the RAP tool and QA property information used in reliability and availability prediction.

5. Discussion and Further Work

The Svamp approach has been applied to example cases carried out in a laboratory. So far, the following observations have been made:

Functional variability modeling supports functional and structural views, but variability also occurs, e.g., in behavior and deployment views. Despite being feasible for modeling even dynamic concepts, Kumbang concepts per se are not convenient for modeling complex behavior; hence, extensions are needed.

Quality variability modeling supports security and reliability modeling in regard to metrics. More exploratory work is required for facilitating quality-aware architecting, such as performance ontology and quality-driven adaptation of software product families, i.e., tradeoffs made at run-time. In addition, other execution qualities need to be considered together with reliability and security. Further, feasibility of modeling different quality attributes and analyzing them design time needs to be studied in more depth.

As a result of the common tooling platform, tools are independent modules that can be integrated with other tools that conform to the Eclipse Platform and UML2. However, the tool suite needs further improvement, especially in regard to interoperability and automatic transformation of different models. Nevertheless, our experiences with Eclipse as a common platform are encouraging.

We have currently provided concepts and a tool suite for modeling variability. However, in order to take full advantage of variability modeling, a derivation tool and quality evaluation tools using the models are needed. Kumbang Configurator [15] can be used to automate product derivation by checking completeness, consequences, and consistency. Kumbang Configurator supports derivation based on Kumbang models, but currently does not take into account quality attributes. On the one hand, it seems feasible to extend Kumbang Configurator to support quality attributes during derivation; however, this requires further research. On the other hand, the RAP [10] tool supports reliability and availability prediction, but model transformation between Kumbang Configurator and the RAP tool has not been studied.

6. Conclusion

This paper introduced a new approach to modeling variability of software product families by combining functional and quality attribute variability modeling. The concepts have been defined as multiple ontologies with different purposes: Kumbang ontology defines concepts for functional variability, Quality Attribute ontologies define concepts related to specific quality attributes, and Quality Variability ontology defines the meta-model for quality variation. The use of ontology orientation has enabled the development of automated tool support, constructed on the commonly used tooling platform Eclipse. The approach has been tested for feasibility with a simple example. However, more research is needed, especially for more complex systems and derivation support.

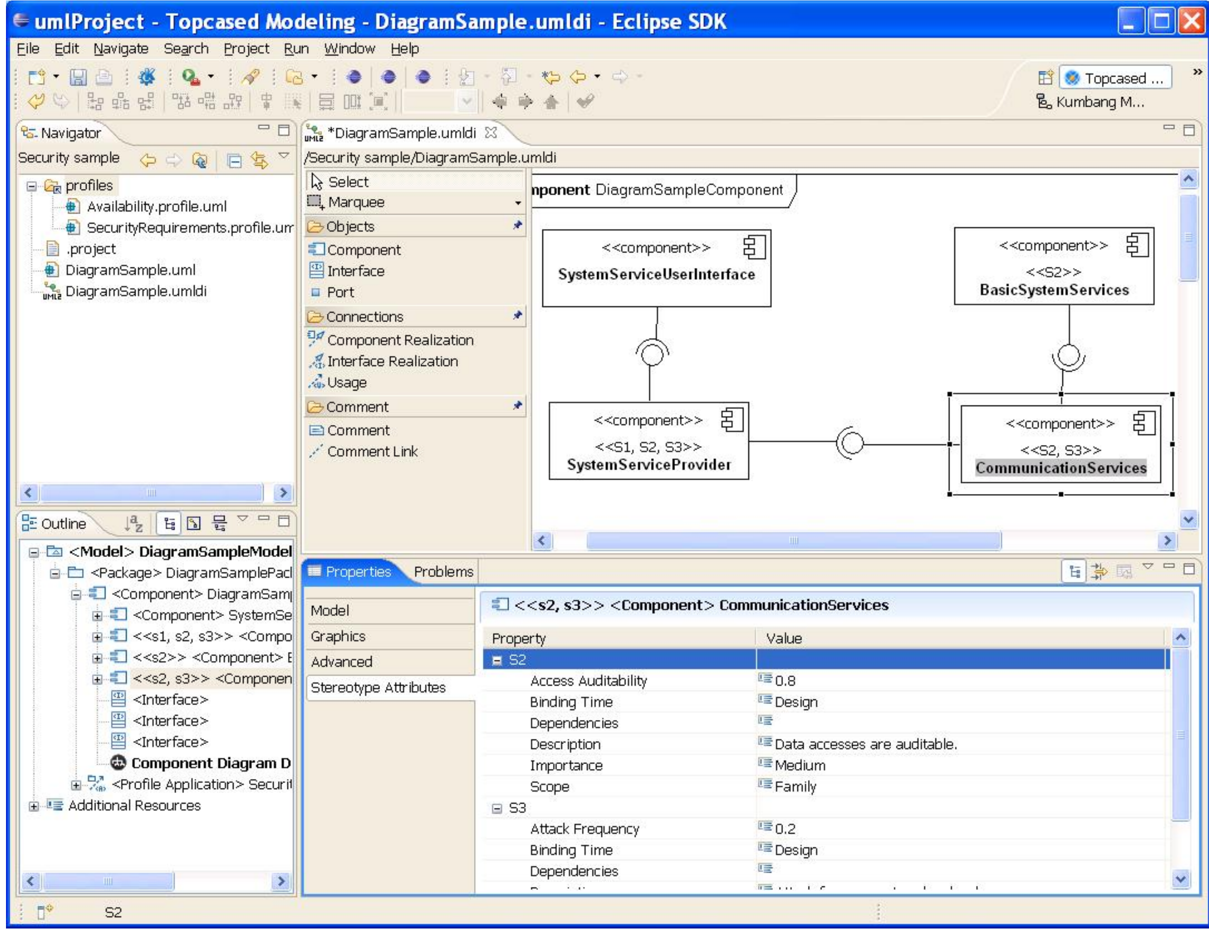


Figure 6. Mapping quality properties to the architectural elements of the structural view.

Acknowledgments

We thank our colleagues Timo Asikainen, Antti Evesti, Hanna Koivu, Pekka Savolainen, and Jiehan Zhou, who participated in the Svamp project by making valuable contributions to specific parts of the presented work. This work was funded by the Finnish Funding Agency for Technology and Innovation (Tekes) and by VTT.

References

[1] T. Asikainen, T. Männistö, and T. Soininen. Kumbang: A domain ontology for modelling variability in software product families. *Advanced Engineering Informatics*, 21(1), 2007.

[2] J. Bosch. *Design and Use of Software Architecture*. Addison-Wesley, 2000.

[3] P. Clements and L. M. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[4] D. Dhungana, R. Rabiser, and P. Grunbacher. Decision-oriented modeling of product line architectures. In *WICSA '07: Proceedings of the Sixth Working IEEE/IFIP Conference on Software Architecture*, 2007.

[5] Eclipse platform. http://www.eclipse.org/, 2008. Visited January 2008.

[6] A. Evesti. Quality-oriented software architecture development. Technical report, VTT Publications, 2007.

[7] L. Hotz, K. Wolter, T. Krebs, S. Deelstra, M. Sinnema, J. Nijhuis, and J. MacGregor. *Configuration in Industrial Product Families*. IOS Press, 2006.

[8] IEEE. *IEEE Std 1471-2000 IEEE Recommended Practice for Architectural Description of Software-Intensive Systems -Description*, 2000.

[9] A. Immonen. A method for predicting reliability and availability at the architectural level. In T. Käkölä and J. Duenas, editors, *Software Product-Lines - Research Issues in Engineering and Management*. 2006.

[10] A. Immonen and A. Niskanen. A tool for reliability and availability prediction. In *EUROMICRO '05: Proceedings of the 31st EUROMICRO Conference on Software Engineering and Advanced Applications*, 2005.

[11] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, SEI, 1990.

[12] KumbangTools. http://www.soberit.hut.fi/KumbangTools/, 2007.

[13] V. Myllärniemi, T. Männistö, and M. Raatikainen. Quality attribute variability within a software product family architecture. In *Short paper in Conference on the Quality of Software Architectures (QoSA)*, 2006.

[14] V. Myllärniemi, M. Raatikainen, and T. Männistö. Kumbang tools. In *Software Product Line Conference*, 2007.

[15] V. Myllärniemi, M. Raatikainen, and T. Männistö. KumbangSec: An approach for modelling functional and security variability in software architectures. In *1st Workshop on Variability Modelling of Software-intensive Systems*, 2007.

[16] E. Niemelä, A. Evesti, and P. Savolainen. Modeling quality attribute variability. In *ENASE 2008*, Submitted.

[17] OWL. http://www.w3.org/TR/owlfeatures/, 2008. Visited January 2008.

[18] D. Pakkala, J. Perälä, and E. Niemelä. A component model for adaptive middleware services and applications. In *EUROMICRO '07: Proceedings of the 33rd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO 2007)*, 2007.

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

[20] P. Savolainen, E. Niemelä, and R. Savola. A taxonomy of information security for service-centric systems. In *EUROMICRO '07: Proceedings of the 33rd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO 2007)*, 2007.

[21] P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2), 2002.

[22] M. Sinnema, S. Deelstra, J. Nijhuis, and J. Bosch. Covamof: A framework for modeling variability in software product families. In *Proceedings of Software Product Line Conference (SPLC)*, pages 197–213, 2004.

[23] M. Svahnberg, J. van Gurp, and J. Bosch. A taxonomy of variability realization techniques. *Software — Practice and Experience*, 35, 2005.

[24] TOPCASED. http://topcasedmm.gforge.enseeiht.fr/website, 2008. Visited January 2008.

[25] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala component model for consumer electronics software. *Computer*, 33(3):78–85, Mar. 2000.

[26] J. Zhou and E. Niemelä. Ontology-based software reliability modeling. In *Software and Services Variability Management - Concepts, Models and Tools*, 2007.

How complex is my Product Line? The case for Variation Point Metrics

Roberto E. Lopez-Herrejon
Computing Laboratory
University of Oxford
rlopez@comlab.ox.ac.uk

Salvador Trujillo
IKERLAN Research Centre
STrujillo@ikerlan.es

Abstract

Software Product Lines aim at capturing the variability and commonality of a family of related programs that share a common set of assets. Variation points capture variability on the artifacts that constitute a product line. Depending on the feature configuration, the variation points are bound according to instantiation logic or mechanism to realize an actual program variant. We argue that this crucial role played by variation points makes them prime subjects for the development of metrics that could provide insights into qualitative and quantitative properties of product lines. In this paper, we show how a basic structural complexity metric can be adapted to variation points and apply it to a case study. We believe that further research on variation point metrics and corresponding tool support can significantly contribute to the current understanding of variability management specially for software intensive systems.

1. Introduction

Software Product Lines (SPL) aim at capturing the variability and commonality of a family of related programs that share a common set of assets [6, 8, 19]. *Variation points* capture variability on the artifacts that constitute a product line and are broadly defined as places in the design or implementation where variation can occur [12]. In this paper, we use a more concise definition provided by Pohl's et al. that defines a variation point as the representation of a variability subject within domain artifacts enriched by contextual information [19]. The context mentioned in this definition refers to the instantiation logic or mechanism to realize an actual artifact variant.

Extensive research has shown how measuring properties like complexity, understandability, maintainability, reusability, etc. can greatly improve, influence and guide software development practices [9, 14, 15]. Several metrics for product lines have been proposed [1, 2, 5, 10, 21, 24], but despite this effort the field remains largely unexplored. Furthermore, the need of SPL metrics has been highlighted as one of the crucial items in the research agenda of the area [11].

Because of their importance for expressing variability, we argue that variation points are prime subjects for the development of metrics that could provide insights into qualitative and quantitative properties of product lines. To support this claim, we show how a basic structural complexity metric can be adapted to variation points and apply it to a case study. We believe that further research on variation point metrics and corresponding tool support can have a significant impact on the current understanding of variability management.

2. Cyclomatic Complexity

Software quality is typically evaluated using metrics, which are measurements of software attributes or properties. Extensive research has produced metrics to assess, with different degrees of success, several of these properties [9, 14, 15].

Structural complexity metrics aim at providing insights into and quantify the relations and interactions of the components or modules that constitute a software system [14, 15]. A basic metric is McCabe's *cyclomatic complexity* that measures the number of linearly-independent paths through a program module [16]. To define paths, a module is represented as a strongly connected graph where the nodes represent program statements and the edges indicate control flow. The formula that captures cyclomatic complexity is:

$$V(G) = e - n + 2 \tag{1}$$

where `V(G)` is the cyclomatic complexity, `e` the number of edges and `n` the number of nodes.

Alternatively, `V(G)` can be computed as the number of binary decisions plus one as follows:

$$V(G) = bd + 1 \tag{2}$$

where `bd` is the number of binary decisions. In the case

```
switch ( c1) {
  case A :  s0; break;
  case B :  if (c2) s1; break;
  case C :  if (c3) s2 else s3;
}
```

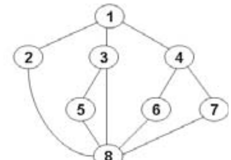


Figure 1. Cyclomatic Complexity Example.

where n-way decisions exists (like Java's switch statement) they are modeled as n-1 binary decisions.

An important property to note of this metric is that it is additive, thus the cyclomatic complexity of a set of modules is the summation of the value of each module. Let us consider the example in Figure 1. It shows a piece of Java code and its corresponding graph representation. Node 1 represents the `switch` statement with its corresponding cases represented by nodes 2, 3, and 4 . Consider case C. It contains an `if-else` statement that splits the control flow in two, represented by nodes 6 and 7. Finally, node 8 represents the next statement after `switch`. Thus the cyclomatic complexity for this example is V(G) = 11 - 8 + 2 = 5.

Based on empirical studies, McCabe and others have proposed threshold values for different complexity ranges [15, 16, 22]. For example, a module with value lower than 10 is considered as simple, whereas one with value greater than 50 is considered as extremely complex.

3. Variation Point Cyclomatic Complexity

It is a common practice in many variability implementation techniques to intermingle common and variable code within a single artifact. In such cases the variable code reifies the variation points and embeds the logic required to instantiate the different variants. This logic is described in a variability language specific to an implementation technique. This type of language is commonly built using similar constructs present in high level languages. For example, consider the following code written in X*ML-based Variant Configuration Language (XVCL)* [13][1]:

```
<select option="SHIFT_DATA">
    <option value="EXPLICIT">
        return circularShifts.elementAt(lineNumber).toString();
    </option>
    <option value="IMPLICIT">
        Pairs pair = (Pairs) circularShifts.elementAt(lineNumber);
        String originalLine = lineStorage.getLine(pair.GetIndex());
        String circularShiftedline =
          originalLine.substring(pair.GetOffset()) + " " +
originalLine.substring(0, pair.GetOffset());
        return circularShiftedline;
    </option>
 </select>
```

This code describes the logic that realizes a variation point by selecting between two options, denoted as `option` tags, that produce two different sets of Java statements contained within each `option` tag. Furthermore, the XML tags `select` and `option` behave like `switch` and `case` Java statements; namely, its execution chooses an option depending on the value of a variable, `SHIFT_DATA` in our case. Thus we argue that a metric defined for standard programming languages can be applied to variability languages provided that the latter support the constructs and abstractions required by the metric. In our example, because XVCL supports standard control flow constructs it is possible to create a graph based on the XVCL tags from which cyclomatic complexity can be computed. In next section we describe this process, apply the metric to a case study, and analyze the results.

[1]Example taken from file x_CircularShift.

4. KWIC Product Line

We applied our metric to the *KeyWord In Context (KWIC)* product line case study of Zhang and Jarzabek [23]. This product line is based on the KWIC index systems proposed by Parnas to study different criteria for modular software decomposition [18]. These systems accept ordered set of lines, each formed with an ordered set of words which in turn consists of ordered sets of characters. The lines can be shifted so that the first word of a line is removed and appended to its line. The output is a list of all circular shifts in alphabetical order. For the KWIC product line, Zhang and Jarzabek consider the following variants [23]:

- A list of noise words that can be removed when shifted.
- Input method that can either be from a file or console.
- Output method that can be also from a file or console.
- Case sensitive or insensitive.
- Shift processing by line or by all lines.
- Shift data that can be store explicitly (sets of strings) or implicitly (pairs of index and offset).

These variants yield a total of 10 different system instances. For their implementation 13 *x-frames*, XVCL artifacts, were used[2]. We manually computed our complexity metric for those x-frames by considering each `select-option` tag construct as an n-way binary decision and each `while` construct as a binary decision. We found that `x_AlphabeticShifts` which coordinates both sort algorithms and `ErrorHandling` are the most complex modules with a value of five. They are followed by `x_Input` and `FileIO` with a value of four. Figure 2 summarizes the results.

All the values we obtained fall within the range of simple as considered by McCabe and others. However, when the product line is viewed as a whole, the complexity value is thirty two[3] which falls in the range of complex. This prod-

[2]We obtained the source code from [13].

[3]Computed using the additive property of the metric.

Module Name	Complexity
BubbleSort	2
ErrorHandling	5
FileIO	4
InsertionSort	2
MemoryIndicator	1
Pairs	1
x_AlphabeticShifts	2
x_CircularShift	5
x_Input	4
x_LineStorage	1
x_MasterControl	2
x_Output	2
kwic_spc	1

Figure 2. Complexity Values.

uct line was intentionally designed for illustration purposes with reduced complexity in mind. What our metric reveals is a way to quantify this complexity.

5. The Case for Variation Point Metrics

Variation points are undoubtedly a cornerstone of variability management. Despite this prominent role, current product line metrics measure variability at other different levels: lines of code, asset development, service utilization, etc. We argue that by shifting the focus to variation points it could be possible to leverage some of the rich and extensive work on software metrics for the analysis of variability. Rather than a comprehensive result, our example of cyclomatic complexity applied to variation points is just a single, yet encouraging, step towards that goal. As such, it leaves many open questions and venues for further research.

Some open issues are identifying for product lines the threshold values, their corresponding complexity value ranges, and any possible dependencies on the implementation techniques. Also of importance is studying any possible limitations of cyclomatic complexity (due to its additive nature) to compare product lines. To satisfactorily address these issues, multiple and real case studies must be analyzed.

Variability is present not only at the source code level, but it is also manifested in multiple artifacts throughout the product line development life cycle. Thus a comprehensive metrics suite should also include non source code variability.

An aim of this paper is to foster the discussion on product line metrics, specially those centered around variation points. We believe such discussion may ultimately lead to better and general theories and tools to assess product families and their development technologies.

6. Related Work

Several pieces of work address SPL metrics. Here we summarize those more closely related to our approach.

Chang et al. propose three metrics to evaluate product line architectures: architectural requirement conformance, conflict freedom[4] and tailorability [5]. These metrics however are defined in terms of architectural drivers.

Van der Hoek et. al propose two component-level metrics, Provided Service Utilization and Required Service Utilization [21]. Contrary to our work, both metrics are based on the notion of service that the authors define as any publicly accessible resource present in an Architecture Description Language [17]. They applied these metrics to three case studies and observed their positive impact in detecting and analyzing SPL structural problems.

Zubrow and Chastek [24] sketch measures for SPL management in terms of costs, schedule, asset development, etc. They also stress the crucial importance of metrics and the need of further research on this area. Along the same lines, Kang argues that it is important to develop metrics on key indicators such as cost of production, project completion time, quality, productivity, reuse, etc [11].

Closer to our work, Her et al. [10] present metrics for evaluating reusability of core assets. One of their metrics, tailorability, is computed by counting variation points and analyzing their validity (absence of unexpected side effects) after binding. However, to the best of our knowledge, this metric does not take into account the complexity of the logic that binds the variation point as our work does.

The COMAVOF framework explicitly represents variation points and traces their dependencies throughout the product line development cycle [20]. We believe such view could be exploited for the development of metrics addressing variability evolution, a central issue of variability management [4]. A work that addresses product line evolution is proposed by Ajila and Dumitrescu where they measure evolution in terms of Lines of Code not at the variation point level [1].

Aldekoa et al. [2] adapts the Maintainability Index to the feature level and apply it to a simple case study. This metric is computed using an averaged cyclomatic complexity, however it is based on the generated code not at the variation point level.

7. Conclusions and Future Work

Only a few studies have used variation points to elaborate metrics for the analysis of product line properties despite

[4] Original name is Free of Conflicts.

their recognized importance of variation points as first-class entities for expressing and managing variability. In this paper we adapted cyclomatic complexity, a basic complexity metric, to the space of variation points and applied it to a simple case study. We believe the results obtained provide a glimpse of the potential that variation point metrics could provide for understanding product line complexity and evolution, comparing variability techniques and their applicability to actual projects, and developing product line economics.

We plan to apply this metric to larger and real case studies for which tool support must be developed. This could provide insights into the complexity threshold values for product lines as well as suggest guidelines for product line refactoring.

An open question to address is how variation point complexity relates to generated code complexity. Another issue is how incremental[5] and decremental[6] variability paradigms cope with large number of variation points and the implications for understandability, scalability, etc.

References

[1] S. Ajila and R. T. Dumitrescu. Experimental use of code delta, code churn, and rate of change to understand software product line evolution. *Journal of Systems and Software*, 80(1):74–91, 2007.

[2] G. Aldekoa, S. Trujillo, G. Sagardui, and O. Díaz. Experience measuring maintainability in software product lines. In *JISBD*, pages 173–182, 2006.

[3] D. Batory. AHEAD Tool Suite. http://www.cs.utexas.edu/users/schwartz/ATS.html.

[4] J. Bosch, G. Florijn, D. Greefhorst, J. Kuusela, J. H. Obbink, and K. Pohl. Variability issues in software product lines. In *PFE*, pages 13–21, 2001.

[5] S. H. Chang, H. J. La, and S. D. Kim. Key issues and metrics for evaluating product line architectures. In *SEKE*, pages 212–219, 2006.

[6] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[7] K. Czarnecki and M. Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In *GPCE*, 2005.

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

[9] N. E. Fenton and S. L. . Pleeger. *Software Metrics. A Rigurous and Practical Approach*. Addison Wesley, 2 edition, 2003.

[10] J. S. Her, J. H. Kim, S. H. Oh, S. Y. Rhew, and S. D. Kim. A framework for evaluating reusability of core asset in product line engineering. *Information & Software Technology*, 49(7):740–760, 2007.

[5] Also known as compositional. An example is Feature Oriented Programming [3].

[6] An example is Czarnecki's template based superimposed variants [7].

[11] International Conference on Software Product Lines (SPLC). Research panel, 2006.

[12] I. Jacobson, M. Griss, and P. Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success*. Addison Wesley Professional, 1997.

[13] S. Jarzabek. XVCL Website. http://xvcl.comp.nus.edu.sg/.

[14] S. H. Kan. *Metrics and Modelins in Software Quality Engineering*. Addison Wesley, 2 edition, 2003.

[15] L. M. Laird and M. C. Carol. *Software Measurement and Estimation: A Practical Approach*. IEEE Computer Society, 2006.

[16] T. J. McCabe. A complexity measure. In *IEEE Transactions on Software Engineering*, volume 2, pages 308–320, 1976.

[17] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. Software Eng.*, 26(1):70–93, 2000.

[18] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[19] K. Pohl, G. Bockle, and F. J. van der Linden. Software product line engineering: Foundations, principles and techniques. In *Springer*, 2005.

[20] M. Sinnema, S. Deelstra, J. Nijhuis, and J. Bosch. Covamof: A framework for modeling variability in software product families. In *SPLC*, pages 197–213, 2004.

[21] A. van der Hoek, E. Dincel, and N. Medvidovic. Using service utilization metrics to assess the structure of product line architectures. In *IEEE METRICS*, pages 298–308, 2003.

[22] Various. Software Engineering Institute. Maintainability Index Technique. http://www.sei.cmu.edu/.

[23] H. Zhang and S. Jarzabek. An xvcl approach to handling variants: A kwic product line example. In *APSEC*, pages 116–125, 2003.

[24] D. Zubrow and G. Chastek. Measures for software product lines. Technical report, CMU/SEI-2003-TN-031, October 2003.

A Multiple Views Model for Variability Management in Software Product Lines

R. Bashroush, I. Spence, P. Kilpatrick, TJ. Brown, C. Gillan
School of Electronics, Electrical Engineering and Computer Science
Queen's University Belfast
{r.bashroush, i.spence, p.kilpatrick, tj.brown, c.gillan}@qub.ac.uk

Abstract

With current trends towards moving variability from hardware to software, and given the increasing desire to postpone design decisions as much as is economically feasible, managing the variability from requirements elicitation to implementation is becoming a primary business requirement in the product line process. Nowadays, a medium size software system may encompass hundreds if not thousands of variability points introducing a new level of complexity that current techniques struggle to manage. In this paper, we present a new approach to variability management by introducing a multiple views model (4VM) where each view caters for specific set of concerns that relate to a particular group of stakeholders.

1. Introduction

Within Software Product Lines, features play an important role in specifying the fixed and variable parts of the architectures of product families and configurable systems. In its simplest form, a feature is an aspect of a system, such as a behavior or an attribute, from the end user's point of view. Feature Modeling emerged from the work by KC Kang et al [1] on domain analysis techniques.

Managing variability within the feature model is a key step for the success of a product family. Variability management is about managing the commonalities and variabilities within a product line. Commonalities are structured lists of assumptions that are true for all product members. Variabilities are structured lists of assumptions about how product members differ.

A classic example of variability is found in mobile phone product lines where variabilities include: the screen size, number of keys, language, etc.

A Variation Point identifies a variability within the product line and its possible bindings by describing several variants. A variant is a possible way to realize or bind a variation point at a specified stage of the development process (design time, compilation time, run-time, etc.) [2].

Bachmann and Bass [3] proposed a classification for architectural variabilities (Functional, Data, Control, Technology, etc.) while Svahnberg and Bosch in [4] talked about different levels of variability (Product Line, Product, Component, etc.).

As variability is geared more towards software, and as more products are being included within a single product line, current complex systems tend to comprise a large number of variability points which makes traditional manual feature modeling techniques cumbersome and difficult to use. As a result, a number of variability management techniques have emerged.

Among those are FODA [1] and FORM [5] by KC Kang et al; FeatuRSEB [6] which combined aspects of the FODA method and the Reuse-Driven Software Engineering Business (RSEB) [7] method; and Bosch's modeling techniques [8]. Other commercial methodologies and tools include BigLever Software Gears [9] and Pure::Variants [10].

Although current techniques provided many useful facilities for managing variability, a number of limitations are still exhibited. The ability to encompass and present a large number of variability points along with their relationships in one view remains a challange. While some chose to use different presentation techniques (e.g. three dimensional space, special purpose output devices and panels, etc.) to try to alleviate this limitation, we approached the problem by dividing the feature model into a number of views,

where each view caters for a specific set of concerns and relates to a particular group of stakeholders.

In the following, we begin in section 2 by discussing the scope and concerns covered by our model. Section 3 then introduces the Four Views Model (4VM) and gives details of each of the views. Finally, we draw conclusions in section 4.

2. The 4VM Scope

In this section, we discuss some variability management requirements and concerns which we have identified through experience and collaboration with other research and industrial partners. These requirements are in the form of information and relationships that should be captured about features in a feature model. The Four Views Model (4VM) is built around these concerns. More concerns can be added to the list in the future to accommodate special application domain or enterprise requirements (e.g. feature evolution, etc.).

2.1. Feature dependency

Within real-life systems, features in a model affect each other in a number of ways. Some features cannot be supported unless other feature(s) are supported in a product (mutually dependent); other features cannot be supported in the same product at the same time (mutually exclusive).

For example, consider an automobile product family where: engine size (e.g. 1.1L, 2L, etc.), gearbox (e.g. Auto, Manual – gears:4,5,6 etc.), and chassis type (sport, saloon, estate, etc.) are among the features of the product family. The number of gears in a gearbox is dependent on the engine size; so an engine size 1.1L and a 5-gear gearbox may be mutually exclusive (cannot coexist in the same product). Similarly, chassis type is dependent on the engine size; an estate chassis may require at least an engine size of 1.8L (mutually dependent).

Dependencies can be quite difficult to model, especially those that relate to quality attributes. Hence, dependencies should not only be represented as first class citizens in any feature model, but also the technique used for capturing dependencies should allow for complex dependency representation.

2.2. Feature interaction

While the presence or absence of features within a feature model may affect the existence of other features (feature dependency), feature interaction is concerned with how different feature combinations affect the system architecture. Features are realized in an architecture using different components and configurations. Different feature combinations might lead to the inclusion of different architectural components and configurations.

For example, consider two optional features: FeatureA and FeatureB. Assume that, if FeatureA is supported by a product, it is realized in the architecture using Component1; similarly, if FeatureB is supported, it is realized in the architecture using Compnent2. Within a product that supports FeatureA, if supporting FeatureB means only the inclusion of Component2 in the product architecture, then these features are considered independent (do not interact). However, if supporting FeatureB (at the same time as FeatureA) means the inclusion of other components than Component1 and Component2 (and perhaps the exclusion of Component1 and/or Component2), then FeatureA and FeatureB are considered to be interacting features.

Predicting feature interaction in a system is a challenging task. Minimizing feature interaction is considered good practice as it reduces the architecture complexity when relating features to architectural structures. One way to minimize feature interaction is by restructuring the feature model and introducing new features to abstract those interactions (which we refer to as *feature abstraction* and is discussed in section 3).

2.3. Variability binding time

As discussed earlier, variation points are places in the design or implementation where variation occurs. Variability is due to unmade decisions that are left open as long as economically feasible. However, specifying the point in time when a variation point is to be bound to a specific variant is important.

A number of possible binding times have been identified and used in industry. Examples are:

- *Design time*: where the decision about a variability point is made at the design stage. Beyond that point (e.g. implementation stage, run time, etc.), this variation point is not visible. An example of a design time binding is to allow for linking features to the inclusion/exclusion of architectural components as well as the reconfiguration of the architecture. This is design time variability and binding.
- *Implementation time*: the variation point is not decided upon until implementation. For this binding time, variation points appear at the code level. A good example of implementation time

variability with C/C++ is the use of pre-processor directives. In the compiled version of the system (the executable), variability points introduced using pre-processor directives are invisible.

- *Link time*: this is when the variation point is not decided upon until linking time. An example of link time variability is MS Windows Dynamic Link Libraries (DLLs).
- *Load time*: the variation point is not decided upon until the load of the system. Load time variability can be introduced using a number of mechanisms such as configuration files.
- *Run time*: Depending on the application, this tends to be the most desirable binding time. This is when variation points are left open until the run time when the end user can make the decision on how to bind the variability. However, due to price (cost, effort, time to implement, etc.) and complexity (complexity of the system, size of code, etc.) this is not always a feasible option. There are numerous examples of run time variability where variation points are bound including, for example, using the application's "options" or "settings" menu.

2.4. Feature implementation time

In industry, software systems are usually built incrementally; there is rarely a software product that is built as a final release from the first edition. Products are usually enhanced and features added to them continuously over time. Planning for future releases of products, the features to be implemented in these products, and the timing, is a key step for the success and sustainability of a product line.

So, feature implementation time should also be captured within the feature model as it contributes to *product versioning*.

2.5. Cost/Benefit analysis

The effort needed and cost involved in realizing features as well as their foreseen benefit should be documented in the feature model. This provides valuable input to the overall project costing and the product versioning process.

Although in general it is not an easy task to specify the cost/effort and benefit involved in realizing a given feature, adequate estimates can be obtained using information gathered and experiences gained from previous similar projects.

2.6. Open/Closed sets of features

Within industrial projects, it is rarely the case that the architect is furnished with the system's comprehensive and complete set of features. Rather, features are continuously added (and modified) to the initial feature model over time - even after the system design process has commenced.

Designing a system around an open and changing set of features that can be modified anytime is a very challenging task. To overcome this problem, some industries differentiate between two types of features: *closed* and *open* features.

Closed sets of features are sets of features that cannot be changed or modified by the architect or the development team and serve as the core of the product or product line. Modifying such features requires the approval of a management appointed committee or a designated authority which would analyze the impact and feasibility of any requested modification to such features.

On the other hand, *open sets* of features are those that tend to change over time (for example due to technology advance or the addition of new features) and are less likely to affect the overall system when altered. Such features can be modified and changed by the project manager, architect, or the development team depending on the nature of the feature.

Such information should be clearly specified in the system feature model.

2.7. Negative features

Naturally, the development of feature models has typically focused on the features that are to be supported by a product or product line. Little attention has been paid to features that are not to be supported by a given product (or a range of products). Limiting the features supported by different products within a product line supports the development of product ranges, for example, varying from low-end products (that support a minimum number of features) to high-end ones (with most/all of the features enabled).

Negative features are features that are specified *not to be supported* by a given product(s). If such negative features are specified, the product (or product line) architecture should be designed in a way to prohibit the enabling of such features by end users of the product.

If such features are not identified and counted for at a very early stage in the design process, they could lead to different kinds of problems based on the nature of the product line.

In more critical application domains, overlooking negative features could have more adverse effects. For example, overlooked negative features had more serious consequences within a US Department of Defense (DoD) funded project that was aimed at developing a GPS (Global Positioning System) based product family. The products within the family varied from low precision civilian based products to high precision high-end military versions. However, end users buying the low end civilian products, with simple tweaking of the system, were able to get access to the services and precision available for the high-end military systems.

2.8. Alternative feature names

Variability management exists at the different stages of the development life-cycle, from requirements, to architecture design and implementation. Different teams (e.g. stakeholders, architects, developers, etc.) use their own mechanisms to manage variability and to express features. So, it is possible that the same feature could be referred to by different names within different teams. Hence, it is important to keep track of the features and their alternative names within the feature model.

2.9. Feature cardinality

It is always desirable to delay design decisions as much as is economically feasible (creating variation points). However, variation points come with a price (increased complexity of the system, performance degradation, increase in cost and marketing time, etc.). One potential solution to alleviate the effect of open variation points is by attaching a limited number of possible variants that could be bound to a given variation point. This is usually referred to as *feature cardinality*.

2.10. Multiple views

It is generally agreed that different stakeholders have interest in viewing different aspects (views) of the product line variability model. So, it is important for a variability management mechanism to be able to extract and present relevant information about the family model in dedicated views for different groups of stakeholders (users, system analysts, developers, etc.). This could considerably contribute to alleviating the graphical overload when showing all the information in one view (compared to multiple views). This forms the basis of the 4VM model and is discussed in more detail in the following section.

3. 4VM

In the previous section a number of issues which need to be captured within a feature model were identified and discussed. In this section, the Four Views Model for Variability Management (4VM) is introduced. The 4VM proposes a four view presentation of the feature model. The 4VM addresses all the issues and concepts identified in the previous section. The views adopted in the 4VM model are:

- *Business View*: where the information related to the project management, cost/benefit analysis, etc. is presented.
- *Hierarchical & Behavioral View*: where the way the different features are organized (usually presented in a tree structure) along with the behavior attached to each feature is presented.
- *Dependency & Interaction View*: where the dependency and interaction among features is presented.
- *Intermediate View*: where some design decisions are injected into the feature model to take it one step further towards the architecture domain in an attempt to bridge the gap between the feature model and the system architecture.

In the following section, each of these views is discussed in detail and example views are taken from the network emulator case study [11].

3.1. Business view

The Business View is aimed at the project business and management stakeholders. It acts as a portal for inputting and presenting information related to:

- Feature implementation time
- Feature Cost/Benefit analysis
- Open/Closed sets of features
- Negative features

These properties are usually specified and used by the project managers to carry out system-wide business analyses which support decision making such as when to introduce features within a product line; what features are feasible from a business perspective, etc. An example business view is shown in Figure 1 below.

In this example, a sample business view is displayed using a prototype tool for the network emulator case study [11]. A red circle indicates a mandatory feature while a green circle indicates an optional/alternative feature. A line across the circle (e.g. Effects, Packet Classifier, etc.) indicates a closed feature or feature set,

that is one that cannot be deleted or modified by the architects/developers.

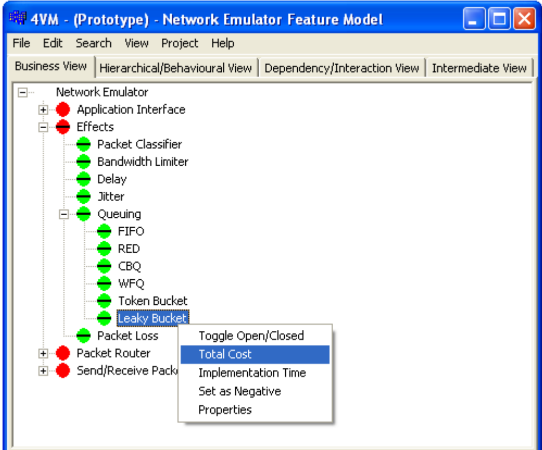


Figure 1. 4VM - Business view example

We could also see in the example above that the *Effects* feature (and sub-features) is marked as closed. This means that only a designated authority can modify this feature set (add new effects, modify existing properties, etc.). By right clicking over the feature, it is possible to change feature properties such as its cost, implementation time, etc. Also, the tool could allow for generation of project costing (based on the information contained within the feature model), feature introduction timeline (product versioning), etc.

3.2. Hierarchical & Behavioral view

The Hierarchical and Behavioral View is the view provided by most existing feature modeling techniques. In this view, information related to the structure of the feature model and the behavior of the features is captured. Among other potential users, this view is mainly targeted at architects and developers.

Within our group, work is in progress for developing CASE tool support for this view [12] where the Use Case Maps (UCM) notation [13] is being used to model feature behavior. Figure 2 below shows an example (taken from the network emulator case study) of what is typically presented within the Hierarchical and Behavioral view.

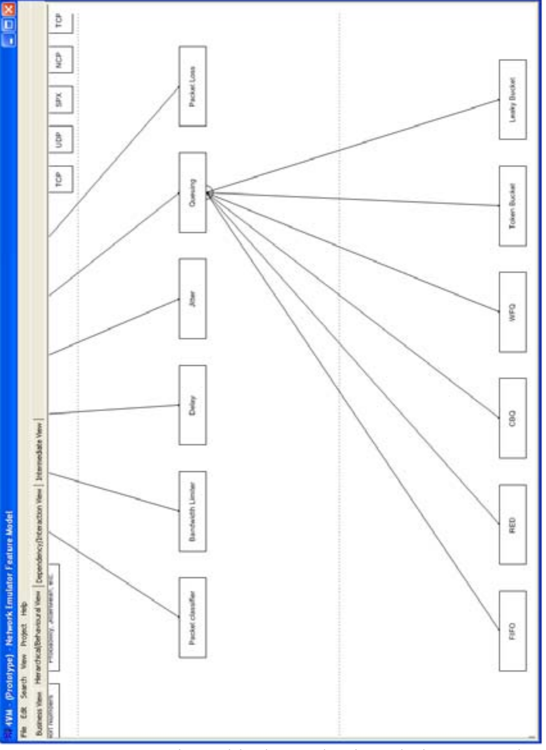


Figure 2. 4VM - Hierarchical & Behavioural view example

3.3. Dependency and Interaction view

Due to the size and complexity of feature dependency and interaction within real-life systems, a separate view is created within the 4VM to model these relationships. The Dependency and Interaction View is complementary to the Hierarchical and Behavioral View.

In this work, feature dependency and feature interaction are defined as follows:

- *Feature Dependency*: a feature-to-feature dependency where the inclusion of one or more features affects one or more features within the system.
- *Feature Interaction*: a feature-to-architecture dependency where the inclusion of one or more features affects the architecture structure (different component sets and/or configurations, etc.).

In this view, logic design is proposed to capture the dependency and interaction relationships. Once the relationships are modeled, standard logic algorithms can be used to simplify the models.

The feature dependency model takes as input the user selected feature set and verifies it against the

model pointing out any conflicts within the feature selection.

Once feature dependency is verified, the selected feature set is fed to the feature interaction model that outputs a new *mutually exclusive set of features* with new features introduced to *abstract feature interaction* which is a novel approach proposed to handle feature interaction.

Returning to the network emulator case study [11], consider the "requires" relationship that exists between Modifying/encoding IP packets and Sending/Receiving IP packets. For a system to support Modifying and encoding of IP packets, it should be able to receive (and send) such packets in the first place. Assume that a new feature is to be added to the system to introduce the support for *secure communication*. Although secure communication (using IPSec) will not affect the sending and receiving of packets at the network level, it would require a change to the coding (encryption is added to the process) and decoding (decryption is added to the process) of IP packets. Figure 3 below shows the dependency and interaction view for IP support in the network emulator case study.

In this example, the feature dependency model captures the dependency of *Modify/Ecnode IP* feature on *Send/Receive IP* feature. This is done using an AND gate. If *Send/Receive IP* feature is not selected, *Modify/Ecnode IP* feature cannot be selected. The mapping of textual relationship description into logic circuits can be relatively straightforward where "not" maps to inverters, "and" to AND gate, and "or" to OR gate. With more complex expressions and relationships, existing logic methods and algorithms can be used at a later stage to simplify the overall model.

In Figure 3, the first column to the left shows what options the architect has to choose from. An empty circle means an optional feature.

Once the architect makes his selection, the selection is validated against the dependency model and any conflict is reflected in the second column (the middle one). The architect could then go back and choose a different feature set to resolve the conflict.

Once a non-conflicting feature set is selected, it is then passed to the interaction model where interactions are resolved by introducing new abstract features. In the example above, the *Modify/Ecnode IPSec* feature was introduce to abstract the interaction between *Modify/Ecnode IP* feature and *Secure Comm* feature.

The advantage of resolving feature interaction at this stage is that it minimizes architecture complexity by making the relationship between the feature set and the architecture structure a one-to-many relationship rather than a many-to-many relationship. This is achieved by making the feature set a mutually independent set with the introduction of abstract features.

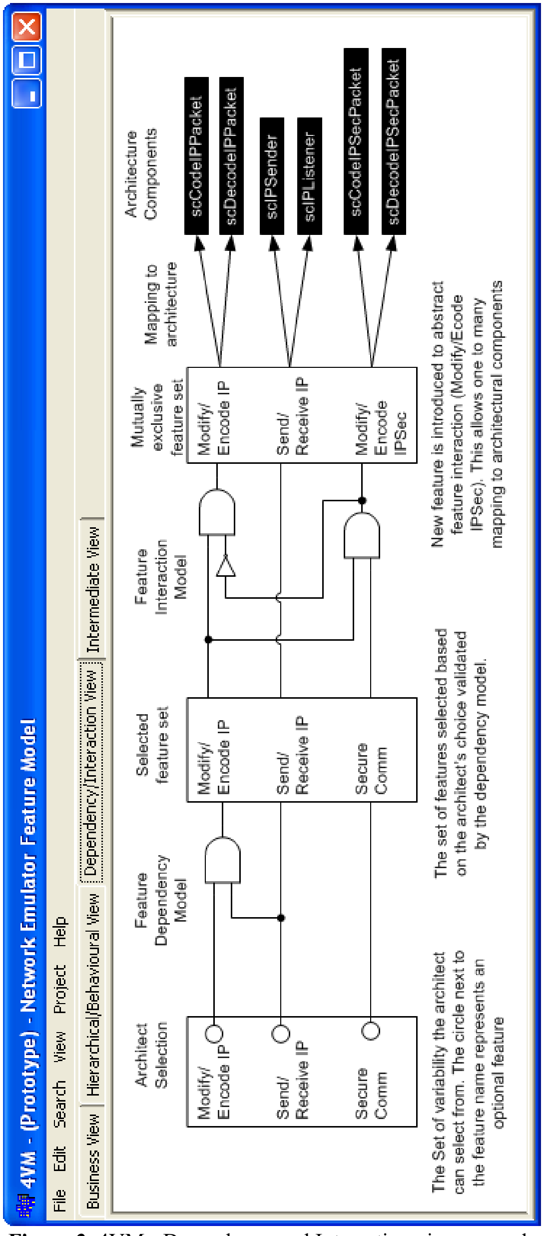


Figure 3. 4VM - Dependency and Interaction view example

The graphical notation used in this example is for demonstration purposes. Logic gates can be replaced with other shapes that are friendlier to non-hardware

architects. Also, textual logic expressions can be used instead of a graphical notation.

3.4. Intermediate view

Finally, the intermediate view has been introduced in an attempt to bridge the gap between feature modeling and the architecture design. This gap exists between the two domains due to the fact that the feature model is based on end-user and stakeholder concerns while the architecture structure is designed to accommodate technical concerns.

To bridge this gap, the intermediate view proposed attempts at injecting design decisions into the feature model to take it one step further towards the architecture domain. As such, it may be regarded as an intermediate stage between feature model and system architecture.

The structure of the intermediate view and the selection of the design decisions to be injected in the feature model to create the intermediate view depend heavily on the architecture design approach used. For example, in the network emulator case study [11], ADLARS [14] was used as the ADL for the architecture design and description. ADLARS partitions the space into three dimensions: Concurrency (captured within Tasks), Structure and Functionality (captured within Components) and Behavior (Captured by Interaction Themes). So, the feature model would be much easier to map to architecture structures if it shows what features are to be implemented concurrently and what features are mere functionality. By injecting such design decisions in the feature model, we end up with the intermediate view which is easier to follow at the architecture design process.

A small part (due to lack of space) of the intermediate view of the network emulator case study is shown in Figure 4 below.

Figure 4 shows three types of features:

- *Concurrency features*: which are features that require a separate thread of execution each, and map to different ADLARS tasks within the system architecture description.
- *Functionality features*: which are features that describe system functionality (usually as a part of a specific thread of execution) and map to ADLARS components and sub-components within the system architecture description.
- *External features*: these are features that are external to the system or product family (over which we have no control) and with which the system would need to interact. These are classified in three types:
    - *Platform*: related to the platform the system is running on (RTOS, Unix, Win32, etc.)
    - *Third party software*: e.g. TUNDrive, a piece of third party software that provides user applications with a virtual Ethernet network interface card over Unix based systems (the one used in the network emulator case study).
    - *Networking technologies*: e.g. TCP/IP, IPX, etc. in case our system needs to communicate over the network (which is the case for the network emulator).

Also, to better identify with ADLARS (where Tasks are composed of Components, etc.), the features within the intermediate view are related in three ways:

- *Composition*: which is represented by a bottom up arrow and means that a given feature is composed of the features below it. For example, the “Forward Packets” feature (Figure 4) is composed of two features, “Packet Receiver” and “Packet Sender”.
- *Realization*: which is represented by a top down arrow and means that a given feature is realized or deployed by the features below it, that is, the parent feature is a template feature implemented by one of the children features. For example, the “Interrupt Communication” feature could either be: “Read Packets”, “Write Packets” or “Forward Packets”.
- *Environment*: which relates the variability of a feature to an external feature (environment). For example, the “Packet Sender” feature is related to what network protocol is used (e.g. TCP/IP, IPX, etc.) which is an environment feature.

It is worth mentioning here that the intermediate view model developed and described in this section is designed to work best within an architecture process that starts with feature modeling and uses ADLARS for architecture design and description. For other design approaches and ADLs (e.g. ALI [15]), appropriate intermediate views can be developed accordingly.

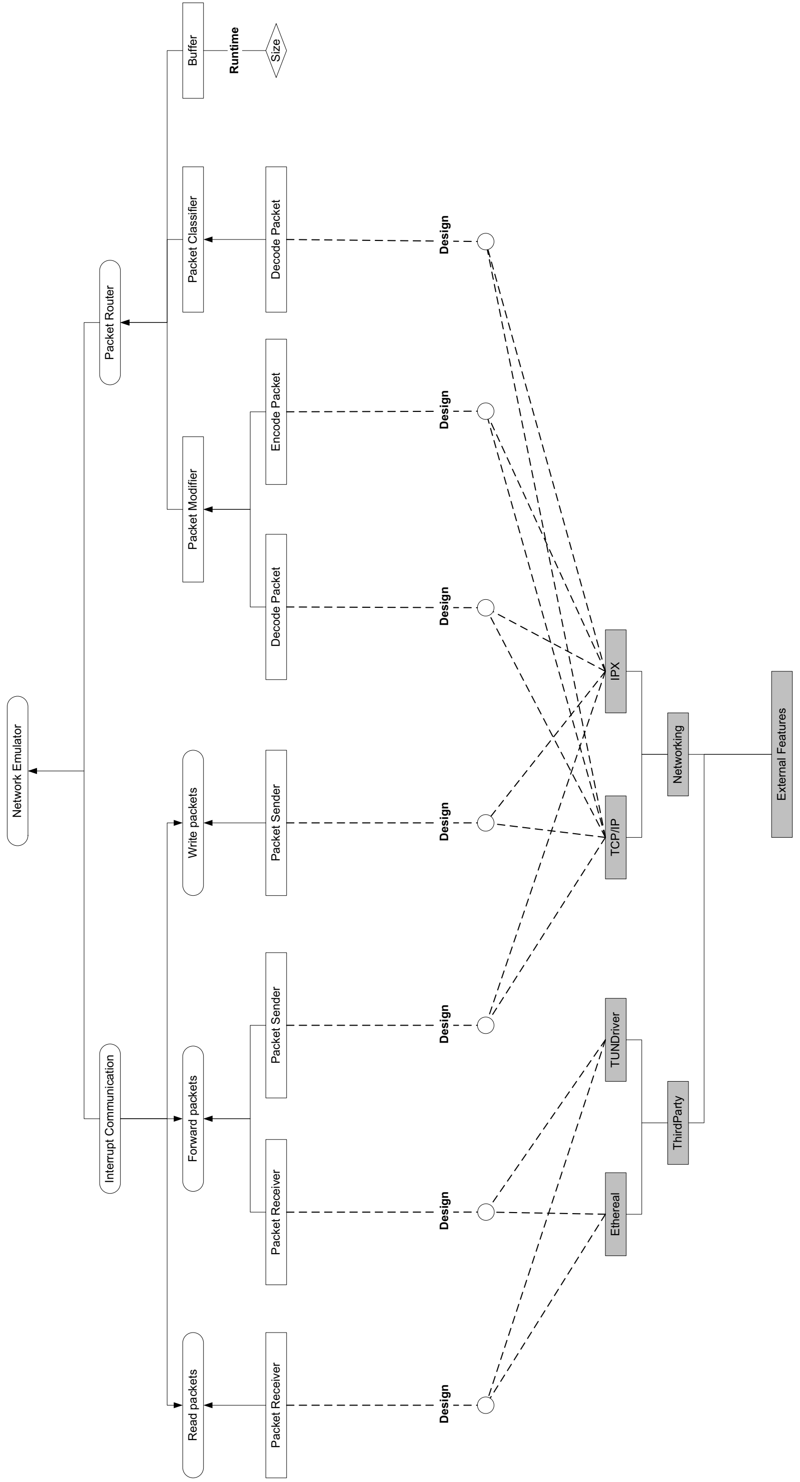


Figure 4. 4VM - Intermediate view example

4. Conclusion

In this paper, a number of feature modeling needs are identified and discussed. These needs are summarized below:

- Capturing complex feature dependency
- Capturing and resolving feature interaction
- Specifying variability binding time
- Specifying feature implementation time (product versioning)
- Capturing information related to the feature Cost/Benefit analysis
- Specifying Open and Closed sets of features
- Specifying Negative features
- Capturing alternative feature names
- Specifying feature cardinality
- Allowing for multiple views

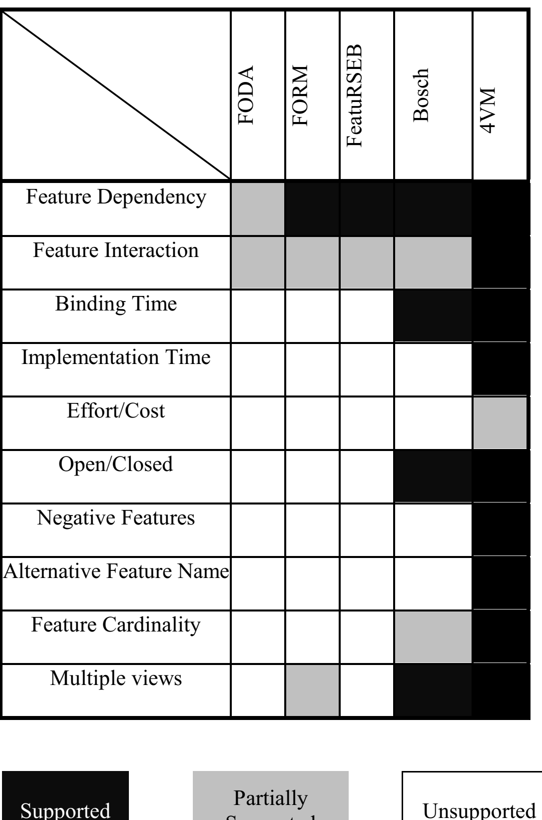
Then, a multiple-view model feature modeling technique is introduced. The Four View Model for Variability Management (4VM) technique proposes the distribution of the feature modeling information into four views where each view is be dedicated to a particular theme and stakeholders. These views are:

- *Business View*: where the information related to the project management, cost/benefit analysis, etc. is presented. This view is geared towards project managers as main users where then can specify feature costing, open and closed features, feature introduction time (product versioning), etc.
- *Hierarchical & Behavioural View*: where the way the different features are organized (usually presented in a tree structure) along with the behaviour attached to each feature is presented. This view is geared towards architects and captures the end user concerns. This is the view that is currently adopted by most feature modelling techniques.
- *Dependency & Interaction View*: where the dependency and interaction among features is presented. This view is geared more towards architects and provides a formal basis for capturing feature dependency using logic design. Also, feature interactions are modelled in the same way and resolved by the introduction of abstract features.
- *Intermediate View*: where some design decisions are injected into the feature model to take it one step further towards the architecture domain in attempt to bridge the gap between the feature model and the system architecture. This view is geared towards architects and provides a transition stage towards the architecture.

The next stage in this research is to take the prototype tool (shown in the figures) and try to develop a full featured CASE tool. The shape and structure of the graphical notation to be used in each of the views is also an open research question and industrial feedback will be an important factor in making such decisions.

Finally, the table below shows how the 4VM measures against the identified requirements discussed in this chapter compared to existing feature modeling techniques.

The 4VM supports all the identified needs. The only two restrictions in the current version are: first, 4VM provides a fixed number of views (four views) for the feature model rather than unrestricted configurable multiple-views; and second, 4VM does not allow for complicated cost/benefit analyses on the feature model. These two issues are to be addressed in the future versions of the 4VM model and its toolset.



	FODA	FORM	FeatuRSEB	Bosch	4VM
Feature Dependency	Partially Supported	Supported	Supported	Supported	Supported
Feature Interaction	Partially Supported	Partially Supported	Partially Supported	Partially Supported	Supported
Binding Time	Unsupported	Unsupported	Unsupported	Supported	Supported
Implementation Time	Unsupported	Unsupported	Unsupported	Unsupported	Supported
Effort/Cost	Unsupported	Unsupported	Unsupported	Unsupported	Partially Supported
Open/Closed	Unsupported	Unsupported	Unsupported	Supported	Supported
Negative Features	Unsupported	Unsupported	Unsupported	Unsupported	Supported
Alternative Feature Name	Unsupported	Unsupported	Unsupported	Unsupported	Supported
Feature Cardinality	Unsupported	Unsupported	Unsupported	Partially Supported	Supported
Multiple views	Unsupported	Partially Supported	Unsupported	Supported	Supported

Supported | Partially Supported | Unsupported

Table 1. Comparison between the 4VM and existing feature modeling techniques based on the needs discussed in this paper.

5. Acknowledgement

We would like to thank Felix Bachmann and the SPL group at the SEI/CMU for their valuable input to this work during its initial stages in 2004. Also, we would like to thank Jaap van der Heijden, Chritiene Aarts and Bas Engel at the Software Architecture department, Philips Research Labs, Eindhoven, for their input and feedback on this work.

6. References

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

[2] J. Bosch, G. Florijn, D. Greefhorst, J. Kuusela, H. Obbink, and K. Pohl, "Variability Issues in Software Product Lines." In proceedings of the 4th International Workshop on Product Family Engineering, Berlin, Germany, 2002. pp. 13-21.

[3] F. Bachmann and L. Bass, "Managing Variability in Software Architecture." In proceedings of the ACM SIGSOFT Symposium on Software Reusability, May 2001. pp. 126-132.

[4] M. Svahnberg and J. Bosch, "Issues Concerning Variability in Software Product Lines." In proceedings of the Third International Workshop on Software Architectures for Product Families, 2000. pp. 146-157.

[5] K. C. Kang, J. Lee, and P. Donohoe, "Feature-Oriented Product Line Engineering," *IEEE Software*, vol. 19, pp. 58-65, July/August 2002.

[6] M. Griss, J. Favaro, and M. d'Alessandro, "Integrating Feature Modeling with the RSEB." In proceedings of the Fifth International Conference on Software Reuse, Vancouver, BC, Canada, June 1998. pp. 76-85.

[7] I. Jacobson, M. Griss, and P. Jonsson, *Software Reuse - Architecture, Process and Organization for Business Success*. New York: ACM Press, 1997.

[8] J. v. Gurp, J. Bosch, and M. Svahnberg, "On the Notion of Variability in Software Product Lines." In proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA 2001), August 2001. pp. 45-54.

[9] "BigLever Software Gears," http://www.biglever.com/solution/product.html.

[10] "Pure-Systems Pure::Variants," http://www.pure-systems.com/Variant_Management.49.0.html.

[11] R. Bashroush, I. Spence, P. Kilpatrick, and T. J. Brown, "A Real-time Network Emulator: ADLARS Case Study." In proceedings of the 3rd Asia Pacific International Symposium on Information Technology, Istanbul, Turkey, January 2004. pp. 610-618.

[12] T. Brown, R. Gawley, R. Bashroush, I. Spence, P. Kilpatrick, and C. Gillan, "Weaving Behavior into Feature Models for Embedded System Families." In proceedings of the 10th International Software Product Line Conference SPLC 2006, Baltimore, Maryland, USA, August 2006. pp. 52-64.

[13] R. J. A. Buhr and R. S. Casselman, *Use Case Maps for object-oriented systems*: Prentice Hall, 1996.

[14] R. Bashroush, T. J. Brown, I. Spence, and P. Kilpatrick, "ADLARS: An Architecture Description Language for Software Product Lines." In proceedings of the 29th Annual IEEE/NASA Software Engineering Workshop, Greenbelt, Maryland, USA, April 2005. pp. 163 - 173.

[15] R. Bashroush, I. Spence, P. Kilpatrick, and T. Brown, "Towards More Flexible Architecture Description Languages for Industrial Applications," *Gruhn and F. Oquendo (Eds.): EWSA 2006, Lecture Notes in Computer Science, Volume (4344),*, pp. 212-219, September 2006.

Understanding Decision Models – Visualization and Complexity reduction of Software Variability

Thomas Forster, Dirk Muthig, Daniel Pech
Fraunhofer Institute for Experimental Software Engineering (IESE)
Fraunhofer-Platz 1, D-67663 Kaiserslautern, Germany
{forster, muthig, pech}@iese.fraunhofer.de

Abstract

With the increasing size and complexity of software systems also the amount of software variability grows. In this paper we present decision models as a means of dealing with software variability and views on decision models that are supposed to make the large amount of variability manageable. Also some mechanisms for supporting the process of decision modelling and resolving decision models are introduced. In a final experiment we evaluate the presented views and process support mechanisms.

Keywords: product lines, variability management, views, decision model, Decision Modeller, empirical study

1. Introduction

Variability has ever been an inherent property of software, which arises from the need of developing software artefacts for the deployment in different contexts. This requirement necessitates the development of highly adaptable software artefacts. The locations at which a software artefact can be extended or configured for a particular context are so-called variation points. They make up a software component's variability. This variability has to be managed somehow, nowadays more than ever as a result of the rising amount of variability. There are two major reasons for the increasing software variability. The first one is the development of software product lines. There design decisions are left open intentionally and are postponed to a binding time as late as possible in the software development process. The second reason is the relocation of functionality from mechanics and hardware to software. This phenomenon can particularly be observed in domains using embedded systems, as for instance the automotive or avionic domain. [1]

The huge amount of variability brings up new challenges according the management of those. In order to design the variability management more efficiently several requirements addressing qualities, such as usability, testability, scalability, traceability, must be considered. In this paper we will focus on scalability and traceability, which interleave with understandability and reduction of complexity of decision models:

- **Scalability** in our context means that it must be possible to handle rather large and complex product lines.
- **Traceability** means in our context that it must be possible to keep variabilities consistent and maintainable. Thus, it must be possible to establish dependencies between variation points that realize certain variability in software artefacts of different development phases.

The remainder of this paper is organized as the following. In the next section we introduce the concept of decision models. Section 3 then discusses different views on decision models. In section 4 a tool, that implements the suggested concepts, is presented. The tool is then validated in a small case study presented in section 5. Finally section 6 concludes this paper.

2. Decision Models

Because variability plays an important role in current-day software, we need a model to explicitly document and manage this variability.

The two main approaches that are applied for modelling variability are feature models and decision models. However, this paper's focus is on decision models, thus we give a brief introduction to those.

A decision model is defined as a model that "captures variability in a product line in terms of open decisions and possible resolutions. In a decision model instance, all decisions are resolved. As variabilities in

generic work products refer to these decisions, a decision model instance, also called resolution model, defines a specific instance of each generic work product and thus specifies a particular product line member". [3]

Basically a decision model is a table, whereby each row in the table represents a decision and each column a property of a decision. A decision has the following properties:

- ID: A unique identifier for the decision (usually an integer value)
- Question: A question which makes the decision more understandable when deriving a decision resolution model.
- Variation Point: The point in an asset which is affected by this decision.
- Resolution set: A set of answers to the decision's question.
- Effect: An effect for each possible answer to the decision's question (constraints).
- As necessary, further properties can be added.

The KobrA method sub-divides decisions into two types, simple decisions and complex decisions. A simple decision directly affects a product line asset and does not affect any other decisions. [4]

An example for a complex decision is decision 2 in Figure 1, which shows an exemplary decision model for a coffee machine. If the decision is resolved with coffee slot, decision 4 is resolved with no. This is due to the reason, that it makes no sense to install a container for coffee-beans, if there is no mill which can crush the beans.

Answering all questions and thus resolving all decisions leads to a resolution model. A resolution model (configuration) consists of all decisions and the answers to their questions. That is, a resolution model constitutes a concrete member of a product line.

3. Visualization mechanisms

Since human beings can assimilate complex coherences easier when those are visualized somehow we come up with a metamodel and graphical notion for modelling constraints among decisions. For modelling constraints we use logical expressions as already proposed by Schmid and John who use those to formulate constraints in their approach [5] or xADL which uses boolean expressions to model constraints in the form of boolean condition guards. [6]

Furthermore, we introduce several views on decision models which are used to reduce the complexity and to focus on particular aspects of a large model. In order to further structure the huge amount of decisions within a view, resulting from large and complex projects, we bring in the concept of layers.

ID	Question	Variation Point	Resolution	Effect
1	Can cups be warmed up?	Cup warmer	yes	A warming plate is attached
			no	No warming plate is attached
2	Does the machine have a crushing mill for coffee, a slot for putting in coffee powder or both?	Input	crushing mill	A crushing mill is attached
			coffee slot	A slot for coffee powder is installed Resolve decision 4: no
			both	A crushing mill and a slot for coffee powder is attached
3	Does the machine have a milk frother or a cappuccinatore?	Milk frother	milk frother	A milk frother is attached
			cappuccinatore	A cappuccinatore is attached
4	Does the machine have a container for coffee beans?	Bean container	yes	A container for coffee beans is attached Resolve decision 2 with: crushing mill or both
			no	No container for coffee beans is attached

Figure 1: Exemplary decision model

3.1. Constraint Modelling View

Metamodel

The metamodel for logical constraint expressions is depicted in Figure 2 and Figure 3. A logical constraint expression is a hierarchy of operators and operands. In more detail a logical constraint expression can consist of an implication or any other operator as top level node. An implication, as in predicate logic, is an operator always consisting of two operands, a premise and a conclusion. A conclusion constitutes facts, which become effective if the premise (assumption) is fulfilled. Premises and conclusions in turn consist of other operators.

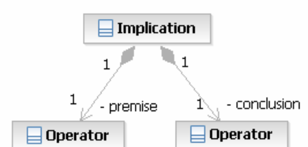


Figure 2: Metamodel for implications

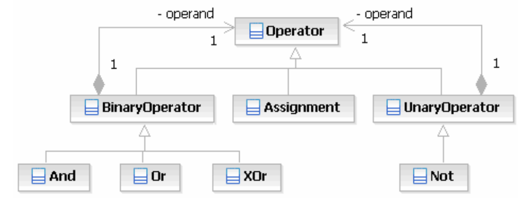


Figure 3: Metamodel for operators

Following syntactic rules must be adhered to when instantiating the metamodel:

- An implication always contains exactly one premise and one conclusion
- Binary operators must contain exactly two operands.
- Unary terms must contain exactly one operand.
- Assignments can only be contained as leafs.
- Elements already referenced in an implication's premise cannot be referenced in the implication's conclusion or the other way around.

Notation

As graphical notation for constraint expressions we choose a notation similar to that used in digital technology for logic gates. The graphical representations are shown in Figure 4. However, as digital technology does not contain something similar to an implication, a new icon for this has to be introduced. An implication will be represented by a circle containing an arrow. The representation of an assignment in a logical expression (which reflects a decision in the decision model) is a rounded rectangle as shown in Figure 5. The rectangle consists of two compartments, the head compartment which contains the decision's name and a second compartment containing the domain values the decision is to assume.

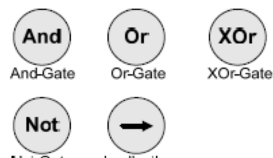


Figure 4: Logical gates

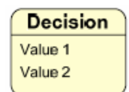


Figure 5: Decision

To create a constraint, the decisions and gates are then connected using arrows as in the constraint illustrated in Figure 6.

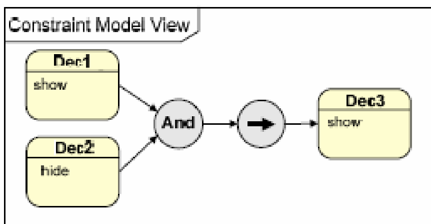


Figure 6: Exemplarily constraint

The meaning of the example constraint is that, "if decision *dec1* is resolved with the value *show* and decision *dec2* is resolved with the value *hide*, then decision *dec3* must be resolved with the value *show*". On the one hand the direction of arrows prescribes the direction of how to read the constraint and on the other hand it defines the relationship between the connected items, i.e. which element is the operand and which the operator. An arrow connecting two items referenced in a premise designates the source element as and operand and the target element as the operator. Arrows in a conclusion must be interpreted the other way around. This way of modelling constraints always results in a tree like structure. Thus, if an implication is contained in the expression, that implication builds the centre of the model to which sides respectively expands a tree of gates and decisions.

3.2. Dependency View

A further view that helps to keep track of large and complex decision models is the dependency view

[7][8]. It abstracts which decisions in a decision model are related to each other, but not how exactly. Consequently it can be seen of as an abstraction of all existing constraints.

Metamodel

The dependency view is based on the metamodel shown in Figure 7. The creation of constraints among decisions in a decision model results in a set of relations because a constraint usually references several decisions. Therefore, the relationship element captures only the knowledge of how two decisions (a source decision and a target decision) are related, i.e. which one has an effect on the other one, but not what exactly that effect is.

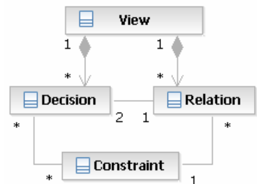


Figure 7: Metamodel for dependency view

Notation

One possibility of representing the dependency view is as a node- and edge-based view. Decisions are represented by a rounded rectangle that contains the name of the according decision.

Relations between decisions are shown as simple arrows that connect the related decisions and therefore visualize which decisions influence each other. The direction of the arrow indicates if a decision is influenced by another decision or if it influences another decision. The arrow's source decision is the affecting decision, whereas the target decision is the affected one. However, from what exactly the influence results and what preconditions must be met in order to fire can not be seen in this view. An example for a dependency view can be seen in Figure 8. It shows for example, that decision 1 affects two decisions, namely decision 2 and decision 3. This influence is the result of a constraint with the name c1, which is indicated by the label next to the arrow representing the according constraint. Another thing to see is that decision 6 is influenced by decision 3 and decision 4. The two influences have different reasons, firstly constraint c3 and secondly constraint c4.

A second way of representing the dependency view is text based. Therefore, a table consisting of five columns is used as shown in Table 1. The first column shows a decision's name, the second column shows by which other decisions it is influenced. Column three adds the number of decisions affecting the given decision. The fourth column then indicates, which other decisions are influenced by the given decision. Again this number is added in the fifth and last column of the table.

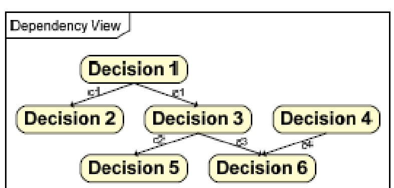


Figure 8: Graphical representation of the dependency view

Decision	Influ-encing Dec.	# Influ-ncing Dec.	Influ-enced Dec.	# Influ-enced Dec.
Decision 1		0		2
			Decision 2	
			Decision 3	
Decision 2		1		0
	Decision 1			
Decision 3		1		2
	Decision 1		Decision 5	
			Decision 6	
Decision 4		0		1
			Decision 6	
Decision 5		1		0
	Decision 3			
Decision 6		2		0
	Decision 3			
	Decision 4			

Table 1: Tabular representation of the dependency view

A last option of representing the dependency view is emphasizing the hierarchy of decisions influencing each other. This can be compared to a call hierarchy in a programming language like java for instance. Accordingly, for a decision model the view focuses on a single decision and either shows which decisions affect this decision or which other decisions are affected by this particular one. Figure 9 shows an example for such a hierarchy, in which the starting point is decision 1. It can be seen, that decision 1 affects decision 3 and that in turn influences decision 6. Decision 6 however does not affect any further decision, as it is the hierarchy's endpoint.

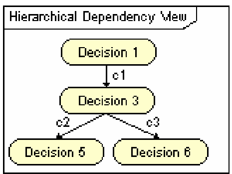


Figure 9: Hierarchical representation of the dependency view

3.3. Layered view

Each of the previously presented views can be enriched by using layers to further structure the contained elements.

An example for the application of layers could be the mapping to development phases (e.g. analysis, design, implementation, testing). Decisions in a layer then only affect decisions in a lower layer or product line assets of the according development phase.

Metamodel

The metamodel for layers is depicted in Figure 10. A layered view on a decision model consists of an arbitrary number of layers (configurable by the user). Those layers in turn consist of compartments that introduce a further way to structure decisions within a layer. Finally the compartments contain links to concrete decisions of the decision model.

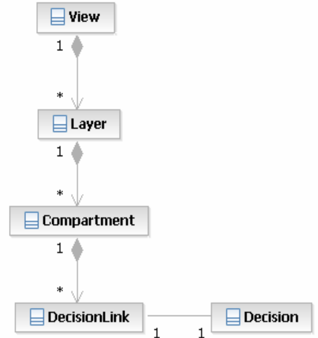


Figure 10: Metamodel for views and layers

Notation

The notation for a view is a box consisting of two sections. The head section contains the view's name. The second section consists of layers.

The notation for layers and compartments is exactly the same as the one for dependency views, namely boxes, containing the according name in a head compartment. They can be differentiated due to their containment hierarchy (a view consists of layers and a layer in turn of compartments).

A link to a decision is represented as a rounded rectangle containing the decision's name. The above described notations are illustrated in Figure 11.

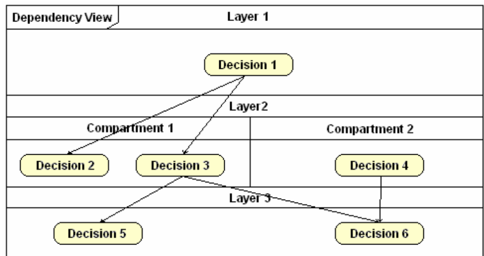


Figure 11: View and layer notation

3.4. Filtering

A useful concept to reduce a decision model's complexity is filtering. That is, the masking of certain representation elements based on some property [9][10]. Such a property for filtering could be the information of a decision to which stakeholder that decision is relevant. For example, decisions representing features (i.e. top most decisions) are usually of interest to people responsible for requirements and the customer, whereas, decisions representing variability on design or code level (i.e. lowest decisions) are only of interest to developers. A property for filtering could also be the information if decisions are simple decision or a complex decision. Applying such a filter to a view would then look like in Figure 12. The view on the left of the figure shows an unfiltered dependency view, after applying the filter for simple decisions only decision 5 and decision 6 are still visible, because they are the only simple decisions in the view (since they don't affect any other decisions).

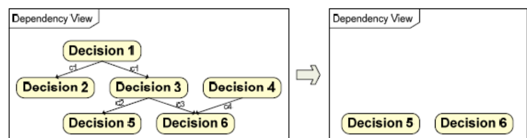


Figure 12: Filtered view

3.5. Resolution Processes

An issue in the process of product configuration, for decision models with a large number of decisions, is the selection of an appropriate starting point for the configuration and the order in which decisions are traversed and resolved. That especially applies to people who did not develop the decision model and thus have no knowledge about its structure. Therefore,

the domain engineer could create a process, which guides application engineers in resolving the decision model. That is, the process consists of a well defined order of how to resolve decisions. The process creation could be accomplished automatically by using particular strategies. Such a strategy could be that decisions at the top of a dependency hierarchy have to be resolved first, then the decisions deeper in the hierarchy. Another possible strategy, based on a layered view, could be to first resolve decisions in the top layer, then the ones in the next layer and so on. There might exist other strategies how to find a resolution process, but this is not elaborated on here, because it is out of the scope of this paper.

4. Tool support

As described in section 2 variability and variation points can be managed using decision models. In principle it is possible to handle decision models manually, for instance by using Excel sheets, which map variation points on artefact elements and decisions for instantiation of the model (see in Figure 13 for an example). The resolution of a decision can constrain other decisions (Decision 8 is resolved to “Yes” if Decision 1 is resolved to “No”). Constraints, consistency problems with large artefact models and their associated decision models as well as the instantiation process lead to the development of tool support in form of the Decision Modeller. The tool is described in more detail in [12] and basically served as a proof of concept for [13].

	Question	Resolution	Effect
1	Should the wiper stop after ignition?	Yes No	- Remove stereotype variant from FinishWiping - Resolve decision 8 to yes
..	…	…	…
8	A second wiper present?	Yes No	- Remove class …

Figure 13: Decision Table

The conceptual architecture of the Decision Modeller tool is shown in Figure 14. As the Decision Modeller is realized as a set of Eclipse plug-ins, the architecture primarily follows a Model-View Controller style. There are five conceptual components:

- **UI**: The UI component provides the graphical user interface elements necessary to work with the Decision Modeller tool. This includes a set of tree viewers for displaying the model as well as wizards to create Decisions, Constraints and projects.
- **ExternalTool**: The ExternalTool component represents external modelling tools (e.g., Rational Software Modeler, Visio) that can be extended by the Decision Modeller to manage variability and to resolve constraints and instantiate variants in these external tools. The Decision Modeller provides standard interfaces for its features and resolution functionality.

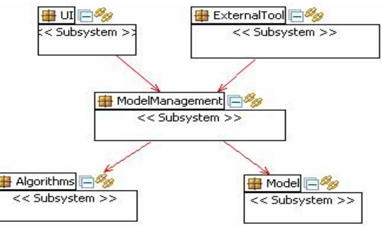


Figure 14: Decision Modeller – Conceptual Architecture

- **Modelmanagement**: The Modelmanagement controls the life cycle of the in-memory data models at runtime. Moreover, it is a façade to the components Algorithms and Model, which should not be accessed directly and not communicate with each other.
- **Model**: The Model component encapsulates the internal data model of the Decision Modeller.
- **Algorithms**: The Algorithms component realizes a reasoning engine for constraint evaluation.

For validation purposes we implemented the concepts listed in section 3 as a contribution in terms of a visualization component to the Decision Modeller.

5. Validation

In order to validate the concepts from section 3 we conducted an experiment based on the Goal-Question-Metric method (GQM) [11].

5.1. Goal

The goal of the experiment was to understand if the implemented concepts help to reduce complexity and improve scalability and traceability of decision models.

5.2. Question

To operationalise the goal following questions are defined:

- **Q1**: How long does it take the attendees to accomplish a certain task referred to modeling or understanding?
- **Q2**: How many faults do the attendees produce in the task?

5.3. Hypotheses

According the questions following null hypothesis can be formulated:

H_0 – There is no difference between tool support with and without visualization component, neither with respect to usability nor with respect to the reduction of complexity or improvement of scalability and traceability of decision models.

Due to the expected observations three further hypotheses can be formulated:

- *H_1 – The concepts implemented by the visualization component reduces the complexity of decision models.*
- *H_2 – The concepts implemented by the visualization component improve the scalability of decision models.*
- *H_3 – The concepts implemented by the visualization component improve the traceability of decision models.*

5.4. Metrics

To collect the data needed to answer questions Q1 and Q2 the following metrics were used:

- **Time** needed to accomplish a task.
- **Number of faults** made in the task. Due to time constraints it was not possible to work out an explicit definition for faults. Therefore, the decision what exactly stated a fault was made by the person who evaluated the results of the experiment.

With this data collected, conclusions about the effort spend on a solution and the efficiency of a solution can be drawn. The correlation between the two metrics and a solution's efficiency is depicted in Figure 15. In case a task is solved in a short time and with no or almost no faults, the task's solution was developed efficiently. If the task is accomplished with a few faults and a short time or the other way around, with almost no faults and in no short time, the solution has a medium degree of efficiency. If the number of faults or the needed time exceeds a particular boundary, the solution is inefficient.

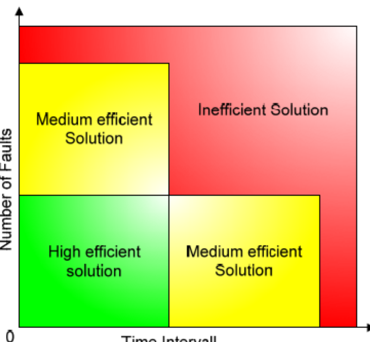


Figure 15: Correlation of faults and time to determine a solution's efficiency

5.5. Setup

The participants in the experiment were split into two groups, each consisting of 3 students of computer science. The first group had to accomplish a set of given tasks with a version of the Decision Modeller that did not have the additional visualization component. The second group had to process the same set of tasks, but with a version of the Decision Modeller that contained the visualization component. Both groups were given the same experiment description document and a short introduction to feature and decision modelling, so they had a common base of knowledge. Of course, the tool introduction was slightly different for both groups, as the tool setup was different. The first part of the document contains an initial questionnaire to determine the subjects' experiences in modelling and modelling tools. This was important, because a higher experience in this field would reduce the time needed to accomplish some tasks. The document's second part contained three tasks and their description.

- The first task was about modelling constraints for the decision model of a coffee machine. As input the subjects got a decision model, already containing all simple decisions, and the coffee machine's feature model, as well as its detailed description. This task addresses scalability, because the decisions had to be related to each other. For this purpose the correct decisions and according values had to be found in the model.
- The second task was to extract information from a given decision model. For this task the subjects did not have to model anything. This task also addresses scalability, because the necessary information had to be found in the model)
- The third and last task required the subjects to write down a possible resolution process for a

given decision model. Here traceability is addressed, because it was required to retrieve an understanding of how the decisions are related to each other

Before and after processing a task the subjects had to write down the actual time in order to determine the task's duration. For counting the number of faults, also the subjects' workspaces, containing their outputs produced in the experiment, were collected.

5.6. Analysis

Since at the point in time this paper was written, the experiment was ongoing work and thus the number of current participants was rather small. Consequently, the data analysed here do not underlie any statistical evidence, but reveal possible trends. Thus, the acceptation or rejection of hypothesis in the following is based on the observed trends.

The experiment's initial questionnaire showed that both groups were well suited for a comparison. That is due to the quite equal standard of knowledge of feature and decision modelling of all attendees. The questionnaire's result is shown in Figure 16. The numbers next to the different section are the numbers of attendees for those sections. Even though one half of all attendees exhibited a little more experience with decision modelling then the other half the groups still remain comparable as this divergence is balanced by the initial introduction. Furthermore, none of the attendees knew the Decision Modeller neither in its plain version nor in the extended version.

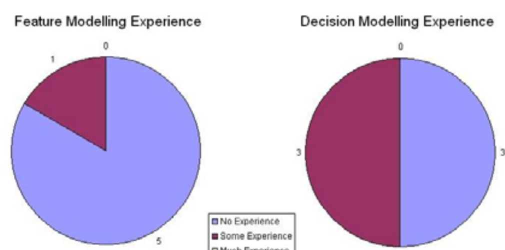


Figure 16: Modelling Experience

Figure 17 shows the average time needed by both groups to accomplish the different tasks. The tasks are shown on the x-axis, whereas, the average time needed by the attendees is shown on the y-axis. For each task the average of both groups is contrasted by two columns.

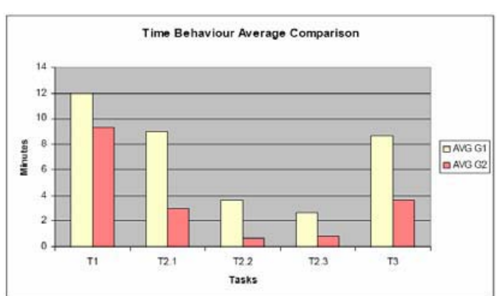


Figure 17: Comparison of the average time behaviour

Figure 18 shows how faulty a solution is. A 0 (on the y-axis) indicates a solution without any faults, whereas, a 3 indicates an insufficient solution with too many faults. Again, for each task the average of both groups is contrasted by two columns.

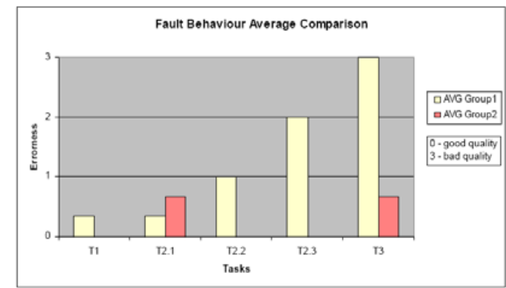


Figure 18: Comparison of fault behaviour

After each task the attendees were asked for their subjective sensation about the task's difficulty. The results are depicted in Figure 19. The y-axis is shows values between 1 and 4. 1 indicates a very easy sensation and a 4 a very hard sensation. Like in the charts before the results for each group are shown in separate columns.

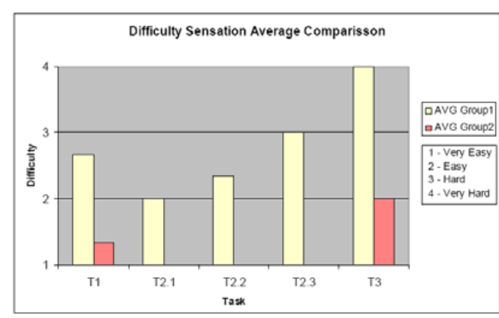


Figure 19: Subjective sensation of difficulty

Hypotheses H_1 & H_2

Since complexity and scalability are properties which influence each other they could not be dealt with separately. Consequently, hypotheses H_1 and H_2 were both addressed by the main task 1 and 2 and its sub-tasks. The tasks demanded the subjects to model constraints among decisions and to extract certain information from a rather small but medium complex decision model which consisted of 15 decisions and 13 constraints. The information to extract was, for instance, related to the model's integrity. As Figure 17 and Figure 18 show, group2 with the visualization support solved tasks 1 and 2.1 to 2.3 in a shorter time than group2 without the support. Moreover, in almost all tasks group2 made fewer errors than group1 and the task's post questionnaires yielded that group2 thought that the tasks were easier than group1 thought. Consequently also H_1 and H_2 could be confirmed.

Hypothesis H_3

Task 3 was supposed to evaluate hypothesis H_3. Therefore, the experiment's subjects had to create a possible process which defined in which order the decisions in the decision model should be resolved optimally. Optimally meant that as little decisions as possible had to be made in order to create a resolution model. Since the perfect solution for this task did not exist the subjects' solution was evaluated with respect to traceability. Thus, it would be optimal to start with decisions referring to requirements and then resolving the according sub-trees which result from constraints among the decisions. Figure 17, Figure 18 and Figure 19 show that no subject from group1 found approximately good solution. Moreover, it took them quite long to accomplish the task and they felt that the task was extraordinarily complicated. To group2, however, the task seemed to be easy. That was also reflected in their solutions. They accomplished the task in a short time and were able to create, not an optimal, but good solution. An optimal solution would have been traversing the process using as few steps as possible. As a result H_3 could be accepted.

6. Conclusions

In this paper we proposed several concepts and views on decision models which are supposed to support the management of large-scale decision models. Those were validated in an experiment which indicated that those helped to reduce the complexity and improves scalability and traceability of decision models. Due to time constraints the experiment could only be conducted with a small number of subjects. Consequently, the results can only be seen as evidences without statistical proof. However, the experiment is work in progress and will be executed with further subjects in order to consolidate the statements made here.

7. References

[1] J. Bosch: "Software variability management", Science of Computer Programming, vol. 53, pp. 255–258, Dezember 2004

[2] K. Kang, S. Cohen, J.Hess, W. Novak, and A. Peterson: "Feature-Oriented Domain Analysis (FODA) Feasibility Study", Tech. Report CMU/SEI-90-TR-21, Software Engineering Institute (SEI), November 1990

[3] J. Bayer, O. Flege, and C. Gacek: "Creating Product Line Architectures", Lecture Notes in Computer Science, Vol. 1951, Proceedings of the International Workshop on Software Architectures for Product Families, March 2000

[4] 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

[5] K. Schmid, and I. John: "A Customizable Approach To Full-Life Cycle Variability Management", IESE-Report, 001.04/E, Kaiserslautern, 2004

[6] D. Dhungana, R. Rabiser, and P. Grünbacher: "Decision-Oriented Modelling of Product Line Architectures", The Working IEEE/IFIP Conference on Software Architecture (WICSA'07), p. 22, 2007

[7] K. Berg, J. Bishop, and D. Muthig: "Tracing Software Product Line Variability – From Problem to Solution Space", Proceedings of the SAICSIT'05, pp. 182-191, 2005

[8] M. Sinnema, S. Deelstra, J. Nijhuis, and J. Bosch: "Managing Variability in Software Product Families", Proceedings of the 2nd Groningen Workshop on Software Variability Management (SVMG 2004), 2004

[9] M. Becker: „Anpassungsunterstützung in Software-Produktfamilien", Dissertation, Technische Universität Kaiserslautern, 2004

[10] M., Coriat, J. Jourdan, and F. Boisbourdin: "The SPLIT Method", Proceedings of the First Software Product Line Conference, pp. 147-166, Kluwer Academic Publishers, 2000

[11] V. Basili, G. Caldiera, and H.D. Rombach. "The Goal/Question/Metric Paradigm". Encyclopedia of Software Engineering (Ed.: John Marciniak), vol. 1, pp. 528-532. John Wiley & Sons, 1994

[12] T. Kruse: "Managing Decision Model Constraints in Product Line Engineering", Diploma Thesis, Technical University of Kaiserslautern, 2004

[13] D. Muthig: "A Light-weight Approach Facilitating an Evolutionary Transition Towards Software Product Lines" , PhD, Technical University of Kaiserslautern, 2002

Variability Management on Behavioral Models

Patrick Tessier, David Servat, Sébastien Gérard
CEA, LIST, Gif-sur-Yvette, F-91191, France
{Patrick.Tessier, David.Servat, Sebastien.Gerard}@cea.fr

Abstract

This paper deals with managing variability on behavioral models. Such models are generally more complex, less tractable by hand than the static, structural parts of a system description. This calls for specific support to check the consistency of variability expression model-wide: defining some elements as variable may impact several behavioral constructs, even elements that designers may not be aware of – e.g. events attached to triggers. Moreover, variable elements at the structural level usually imply variable behavioral constructs in ways that are not easily foreseeable: a seemingly perfectly valid variability scheme may lead to ill-formed behavioral models after derivation. This paper takes UML state machine diagrams as a case study and presents some technical solutions to maintain consistency between both levels: a propagation mechanism to deal with the impact of variability model-wide and a constructive method to check the well-formedness of state machines obtained by derivation.

1. Introduction

The system family paradigm was first proposed in [13, 15]. It has been the research topic of several research projects such as ESAPS, CAFÉ, Families[1]. Their focus was to foster reuse of models for a whole set of similar applications. An application is seen as a specific instance of an application family called an application domain. This is known as the "product line" approach [4]. A system family model relies on the design and composition of common and specific functionalities, called variable elements. Each specific member of a system family, called the product model, results from one derivation of one single system family model.

The purpose of our work is to apply the system family paradigm to the development of real-time systems. Our research team proposes extensions and design method guidelines to help engineers improve their experience of the UML in this particular domain. We participated in the recent advent of the MARTE profile, which is the standardized UML profile for real-time and embedded systems design and analysis [14]. For such systems, behavioral models are critical and complex whereas structural models remain fairly easy to understand and manage. Structural and behavioral parts of a system description usually present complex links that are hard to manage when one wants to depict commonalities and differences between models. Designing families of real-time systems is not an easy task and calls for specific support: 1) means of expressing variability; 2) consistency checks to help manage variability model-wide, both at structural and behavioral levels; 3) correct-by-construction derivation of product models from the system family model.

This paper presents some results of our ongoing work in this context within the limited scope of models where behavior is depicted with a set of state machines attached to structural elements. After a brief overview of related works, the paper is organized into three sections: the first one (section 3) presents the chosen conceptual model and UML profile used to express variability on models. A case study is introduced at the end of this section. Then we describe two specific mechanisms that help designers maintain consistency on their variability-enabled models: 1) a mechanism of variability propagation ensures that the impact of the addition of variability on any modeling elements is fully reflected throughout the entire model (section 4); 2) a mechanism to check the well-formedness of state machines obtained by derivation is described in section 5. Both of these mechanisms help monitor the derivation process of a product model from the system family model. In the course of this paper we show how this in turn helps assess the consistence and completeness of the system family model as a whole.

2. Related works and overall context

Means of expressing variability have been the subject of many research works. Those grounded in the UML language have usually pointed out the lacks of UML2 (see for instance [16]) and extensions have been proposed in terms of profile definitions. Roughly

[1] http://www.esi.es/Families/

speaking the works in this area have consisted in proposing adaptations of established product-line approaches in the UML domain. In [6, 7, 10] the FODA approach [9] was adapted to UML. In [18, 19], ideas from FODA are reused. Results from the Families project led to a conceptual model and a UML profile where several approaches were combined [2].

It is not the goal of this paper to discuss the expressiveness of the various approaches found in this area. They all rely on the distinction between common and variable elements - the latter being usually tagged by some given stereotype - and the addition of constrained relationships between such variable elements. Allowed constraints and choice of constraint language may vary. A rationale is usually provided for the choice of one element among a set of possibilities.

Then the overall process usually follows the same principles from one approach to the other. From a common system family model one intends to drive the generation of several product models, using a decision model to guide the process: the decision model is a graph or tree-based model that lists the various possibilities in terms of variability resolution. Following paths in this graph results in resolving sets of variable elements and progressing towards a model with less variability than before. Ultimately a model with no remaining variable elements is obtained: a product model. The list of choices made in the process characterizes the product in terms of functionalities, qualities, or the like.

Let us make some comments:

1. We may note that for now there is no global convergence towards a standardized UML profile to express variability. The MARTE profile does not fill the gap so far on this aspect.
2. A commonality of these approaches - which is the driver for the present contribution - is that the variability is expressed essentially in structural models. When some approaches describe variability in the behavior, then usually nothing ensures that derived product models are valid. For instance no checks are performed on the derived state machines or behavioral constructs. We think that a more rigorous approach is needed for real-time systems. Variability must be checked so that all possible derivation from a common family model shall produce an acceptable, meaningful product model, e.g. in our case, state machines obtained by derivation must be verified.
3. As said previously, the derivation process usually relies on a guide called decision model or feature model. This model is hand-made during the construction of the system family. This hand-made approach is not tractable when one addresses behavioral models. The presence or absence of structural elements and rationale thereof can usually be traced back to some intelligible needs from a designer point of view. Usually variability in the functional features or the platform choices is at stake. Yet understanding the implication of variability on the topology of transitions and triggers on state machine diagrams is far less intuitive. Even to check that a state machine is well or ill formed is not an easy task to perform manually.

Given this context and bearing in mind the overall complexity of the problems mentioned here, the following sections do not pretend to propose definite solutions but rather some operational mechanisms to help designers maintain consistency within a deliberately limited and constrained set of variability-enabled models. The main characteristic of our approach is to find ways to monitor the derivation process so that the overall consistency and completeness of the system family model can be assessed.

3. Modeling variability in UML

This section presents the meta-model that has been designed to express variability in a UML model. This work is grounded in the proposed conceptual variability model of the Families project [2]. It currently serves as the basis for the proposal of artifact-variability (i.e. design elements centered) of the ATESST[2] project [5] whose goal is to propose a refined version of the EAST-ADL language for automotive systems. The presentation is deliberately limited here to the core elements that enable to understand the UML profile constructs used later in the case study. An extensive description of the conceptual model and profile can be found in [17].

3.1. Meta-model to express variability

A System Family Model (SFM) factorizes several product models into one. It is therefore made up of common elements and variable elements. Model elements not tagged as variable elements are implicitly considered common to all products.

Two kinds of variable elements exist (see figure 1):

- *Variable element*: these are the variable elements that are explicitly introduced by the user.
- *PropagatedVariableElement*: these are elements that acquire the variability feature through their relationships to other possibly variable elements.

[2] http://www.atesst.org/

This is intended to be automatically assigned by a tool.

In our approach, variable elements are propagated from source variable elements along model and meta-model relationships so that impacts of the addition of variability are fully covered. The distinction between both types of variable elements helps to achieve traceability on what the propagation tool produced.

Specifying that a model element is variable is not sufficient to describe a system family model. In fact variable elements are generally not isolated, but constrain one another: presence or absence of one implies various choices on others, etc. One clearly needs to express such dependency constraints between variable elements.

To this end, several approaches propose mechanisms to add constraints between variations. In [3] a "requires" dependency is used to link two variable elements; in [19] OCL constraints are used to define more elaborated variable element dependencies.

In our approach, OCL constraints are also used. Yet because constraints in this formalism may be complex to write, we introduced the concept of a variation group, which features several predefined types of constraints among a set of variable elements: the constraints may be expressed via OCL or with some textual user-defined language.

A variation group features six predefined kinds of variability constraints:

- *Equivalence*: the listed variable elements work as a group, they will be either all present or absent in any product model.
- *Alternative*: only one of the listed variable elements will appear in any product model.
- *OneAmongSeveral*: several variable elements listed – at least one in any case - will be present in the product model.
- *Implication*: a variable element implies the existence of another variable element in the system model.
- *CustomizedCombination*: the constraint between listed variable elements is written directly by the designer. The constraint is a logical expression where operators are not, and, or, xor, implies.

A rationale is provided along with all *VariationGroups* in the *Motivation* property that describes the motivation.

These predefined constraint kinds cover most of the common needs, such as the «requires» or «excludes» relationships from the FODA methodology. However at times more complex constraints need to be expressed. For this, the *ComplexVariationGroup* can be used: it allows specifying constrained combinations of variation groups in a hierarchical manner.

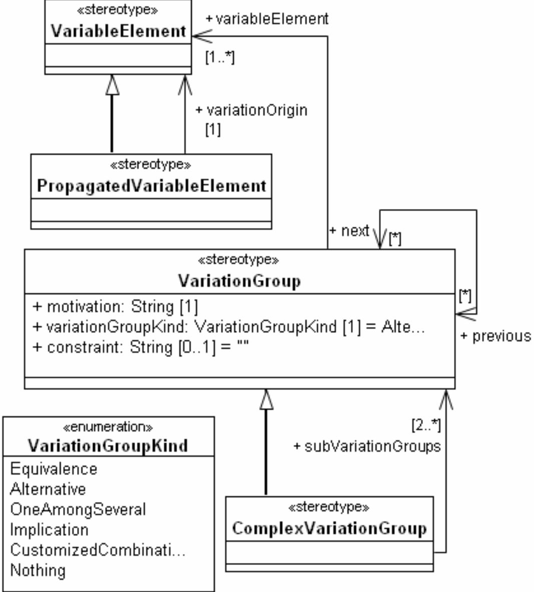


Fig.1: Meta-model for variability modeling.

3.2. A UML profile for variability modeling

This paper is focused on the impact of variability on state machine diagrams. Consequently, the presentation of the profile is limited to what is relevant for state machine diagrams and class diagrams.

A variable element is marked by the stereotype «*VariableElement*» (Table 1). This stereotype can be applied on a class, a property (an attribute of a Class), or an operation. To express variability in a state machine, the stereotype can be applied on a Transition or a Vertex (a generalized class for PseudoState and State in the UML). Other more elaborated elements of state machine diagrams, such as Entry/Exit points, Regions, ConnectionPointReferences or Ports of protocol StateMachines are not considered for the moment in this work.

Stereotype	BaseClass	Tags
«VariableElement»	Class	
	Property	
	Operation	
	Vertex	
	Transition	

Table 1: «*VariationElement*» stereotype

As said previously, variable elements resulting from propagation of variability along the model and meta model relationships are covered by the "*PropagatedVariationElement*" stereotype, which

owns a property called *VariationOrigin* that references the source variationelement from which it originates.

Stereotype	BaseClass	Tags
«PropagatedVariableElement»	Class	VariationOrigin
	Property	
	Operation	
	Vertex	
	Transition	

Table 2: «*PropagatedVariableElement*» stereotype

Constrained clusters of variable elements are supported by a class stereotyped as "*VariationGroup*" (table 4). The type of the embedded constraint is specified in the *variationGroupKind* property. A rationale for the cluster may be specified by the user with the *motivation* property. A variation group saves references to the clustered variation element via its property *variationElements*.

Stereotype	BaseClass	Tags
«VariationGroup»	Class	variationGroupKind motivation variableElements

Table 4: "*VariationGroup*" stereotype

Tag	Type	Multiplicity
variationGroupKind	VariationGroupKind	[1..1]
motivation	String	[1..1]
variableElements	VaraibleElement	[1.*]

Table 5: Properties of «*VariationGroup*»

The modeler first defines the elements that are variable by tagging them with the "*VariationElement*" stereotype. Then in order to introduce constraints between variable elements, he introduces classes stereotyped as "*VariationGroup*", fills in the references to the variable elements, provides a rationale and a constraint, for instance in OCL or textual language. The name of the *VariationGroup* classes does not matter, yet a good practice is to give names relevant to the type of the embedded constraint. A practical example is given in the next section.

3.3. Watch case study

We consider the case of a watch that offers various alarm modes: a sound signal, a visual signal or a combination of both. There are several ways to model such a system. We assume here that the watch system is modeled by a single *Watch* class (figure 2). It implements two interfaces. The *WatchControl* interface defines *start()* and *stop()* operations to trigger the watch system as a whole. The *AlarmControl* interface provides operations to trigger the alarm function – *startAlarm()* and *stopAlarm()* operations. The *Watch* class has associations to other elements, namely a *Display* used for the watch as a whole, a *DisplayAlarm* specific to the alarm function and a *Beeper* that provides for the sound mode. To deal with both alarm modes the *Watch* class features two internal operations, *startSoundAlarm()* and *startVisualAlarm()* to trigger the *start()* operation of either of the associated *Beeper* or *DisplayAlarm*. The execution sequence and respective delegation of calls are depicted in the state machine diagram of figure 3.

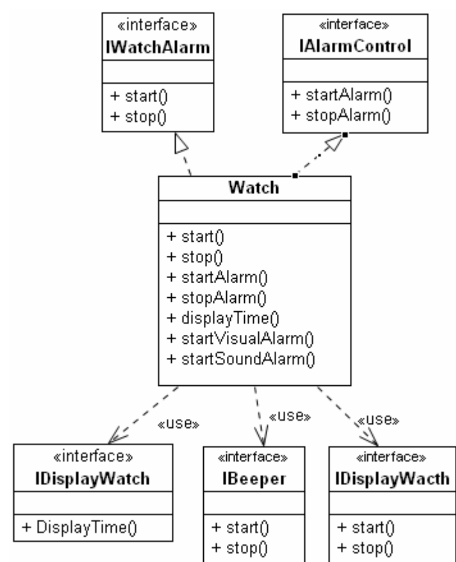


Fig.2: The complete watch

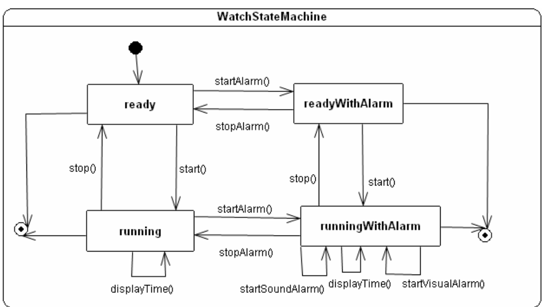


Fig.3: Behavior of the complete watch

Consider now that one wants to derive a system family out of this rather complete watch, to factorize other less advanced models – providing either one of the alarm modes or perhaps even featuring no alarm function at all. We assume that the overall system does not change: less advanced systems can be represented by downgraded *Watch* classes featuring fewer operations and updated execution behavior.

To do so one tags some of the modeling elements with the "*VariableElement*" stereotype (figure 4):

- *startAlarm()* and *stopAlarm()* should be tagged as variable, because these operations are specific to the alarm functionality,
- *startSoundAlarm()* and *startVisualAlarm()* should also be tagged as variable, because they are only used when the alarm function is enabled.

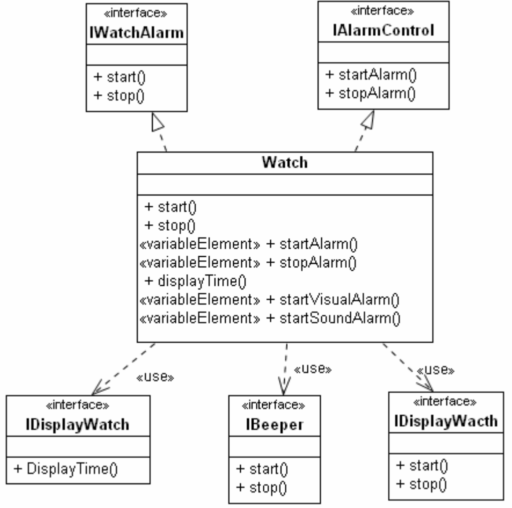


Fig. 4: Watch system family model

Some watches may feature only one of these operations (depending on what is the enabled alarm mode) or both in the case of the complete watch. To define such alternative configurations, one introduces two *VariationGroups* with specific constraints among a list of *VariationElements*. This is depicted on figure 5.

A variation group named *AlarmVariationGroup* is added to set that *startAlarm()* and *stopAlarm()* have a strong relation. These operations have to be all present or absent in the model.

A second variation group lists *startVisualAlarm()* and *startSoundAlarm()*. The relation between them is soft, all combinations are possible. Nevertheless, the variation group is used to explain why theses operations are variables.

The constraint specified on the *VariationGroup* can be written using OCL. The *AlarmVariationGroup* constraint would be defined as follows:

Context WatchModel **inv** AlarmVariationGroup:
(self.elementExist(WatchModel.Watch.startAlarm) **and**
self.elementExist(WatchModel.Watch.stopAlarm)
or
(**not** self.elementExist(WatchModel.Watch.startAlarm.exist) **and**
not self. elementExist(WatchModel.watch.stopAlarm))

where *elementExist*() is a predefined OCL function that returns true if the model element is present in the model and false otherwise.

This ends the presentation of the expression means. We end up with a model where variability has been set on the structural part of the model. In the following section we will see how variability is kept consistent in both behavioral and structural levels and how the derivation process can be checked. The same case study will be considered throughout the paper.

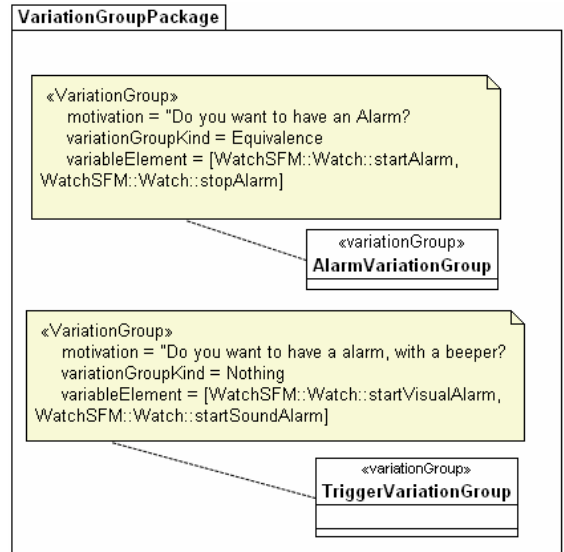


Fig. 5: Variation group definitions

4. Variability propagation

Ensure consistency in a system family model is crucial. A lack of consistency may result in ill-formed state machines at derivation time. When a transition is tagged as a variable element, it may be the case that one of the derived state machines features non-reachable states, i.e. with no incoming transition, even though the state itself was not tagged variable. Such states should not be generated.

To cope with that, our approach propagates variation across the system family model, along both model and meta-model relationships. A set of rules has been defined to address various situations. The following are examples of such rules, to cope with Triggers and CallEvents – elements most of the time unknown to users, who are only aware of the Transitions and do not have an in-depth view of the model repository. Rules are followed by their OCL expressions:

An operation specified as a variable element implies that all CallEvents referencing this operation are variable elements and all

Triggers that are associated to this operation via the CallEvents are also variable elements.

```
Context Trigger inv:
self.event.isKindOf(CallEvent) and
self.event.operation.isStereotyped('VariableElement')
implies
(self.event.isStereotyped('PropagatedVariableElement') and
self.isStereotyped('PropagatedVariableElement'))
```

If Trigger is a variable element then all associated Transition elements are variable elements.

```
Context Transition inv:
 self.trigger.isStereotyped('VariableElement') implies
 self.isStereotyped('PropagatedVariableElement')
```

If a state is a variable element then all its incoming and outgoing Transitions are variable elements.

```
  Context State inv:
self.isStereotyped('VariableElement') implies
self.incoming
  ->forAll(t,t.isStereotyped('PropagatedVariableElement')) and
self.outgoing
  ->forAll(t,t.isStereotyped('PropagatedVariableElement'))
```

Such a propagation process targets to faithfully reflect the potential impacts of variability on one model entity onto the rest of the model, so as to prevent errors coming from careless addition of variability. Yet on simple cases, it can even be more productive. In our case study, this simple propagation mechanism automatically provides an update of the state-machine diagram from figure 3 according to the variability information put on the structural diagram (figures 4 and 5). The resulting state machine is depicted on figure 6.

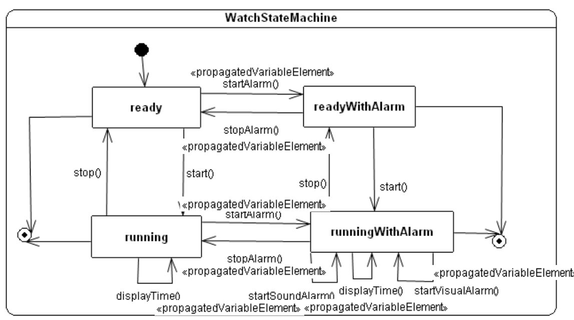


Fig. 6: State machine after propagation

This illustrates the interest of such a propagation mechanism, which can even produce behavioral diagrams out of a variability-enabled structural model.

Let us note here that the propagation mechanism does not consider the actual constraints embedded within *VariationGroups*. Rules are based on containment or cross-reference relationships, not on the semantics of the link between variable elements. Finally it is likely that the modeler shall introduce a range limit to this propagation mechanism, otherwise some rules may result in all model elements being tagged as variables.

At this stage one is assured that the *VariableElement* stereotype is applied consistently system-wide. The constraints embedded in the *VariationGroups* can be used to derive what is called a decision model, in which paths represent possible sequence of choices on the variable elements. The construction of the decision model out of the *VariationGroup* information can be found in [17].

It is however another problem to know whether all paths result in correct product models. If constraints have not been well designed, ill-formed models may be derived. Imagine that the incoming transitions of a state are all variable elements and there exists a decision where all of them are removed from the model, we may end up with a state with no incoming transition, hence an ill-formed state machine.

For example in our case study, the state machine depicted on figure 7 may be derived: it is ill-formed because both states *ReadyWithAlarm* and *RunningWithAlarm* cannot be reached.

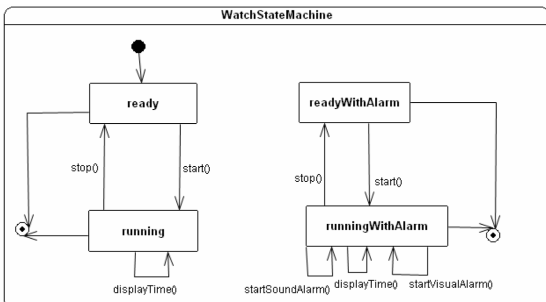


Fig. 7: Ill-formed state machine

This situation may occur even if such a state was tagged as variable by the propagation mechanism: this process is blind to the constraints embedded in *VariationGroups*. Such an interesting feature is for the moment beyond the capabilities of our approach. For the state to be removed automatically when no incoming transitions is left, one would have to alter the *VariationGroups* introduced so that they incorporate the state as part of their managed variable elements. It remains to be investigated whether such an enhanced propagation mechanism would suffice to induce

correct-by-construction derived models. We may have doubts considering the complexity of variability schemes usually imposed on structural models. Thus, one has probably to face the fact that the derivation process may result in ill-formed models. The following section presents means to cope with that, in the case of state machine diagrams.

5. Evaluation of derived state machines

As said previously, the derivation of ill-formed product models from a common system family model is very difficult to avoid. Thus the correctness of derived product model shall be evaluated. In our case, we have to assess whether all possible choices made during derivation result in valid state machines.

To do so, our approach constructs a function that captures the state machine topology. The configuration of its transitions and states are seen as the variables of this function. The evaluation of the function returns *null* if the state and transition configuration is invalid, else returns a compact representation of the valid state machine.

Based on this, we evaluate the various state machine configurations obtained from each path of the decision model. This helps us evaluate the overall consistency of the system family model. If some branches lead to ill-formed behavioral diagrams then there must be a design fault somewhere in the variability model.

Our study is conducted on state machine diagrams.

In this part, formal foundation in briefly explained and then the process is presented. The process is divided in three parts. First the transformation of a state-machine into regular expression is presented. Then the analysis of this expression and finally a process of derivation are described.

5.1. A Formal foundation for derivation

Our approach makes use of the following works on regular expressions, automata and Kleene algebra [8, 11], which are very well suited to analyze automata-based specification and study the impact of variations.

We construct a regular expression that represents a complete state machine, featuring variable and non-variable elements. We then evaluate this expression according to various valuations of its variable elements. The evaluation is made in the Kleene algebra. If the variable element – transition or state – is present, the valuation amounts to the identifier of the element – e.g. transition “t9”, name of state, etc. If the variable element is absent, it amounts to null (Ø in Kleene algebra). As a result, if the overall regular expression evaluates to *null*, the state-machine is ill-formed; otherwise the result represents the topology of the derived state-machine obtained for a given combination of variable elements.

The following paragraphs explain the calculus of the regular expression, its evaluation and analysis.

5.2. A regular expression for state machine

The regular expression should capture the topology of the state-machine. To do that, the UML state machine is first transformed into a regular automaton, defined as follows:

- its alphabet is chosen as the set of UML transition and state identifiers
- its recognized language is chosen as the sequences of transitions and states of the state machine.

Figure 8 shows the corresponding automaton of the complete state machine of our watch case study from figure 6. The regular expression is then computed out of this automaton using a classical algorithm from the literature [12]. The expression is not shown here because of its length.

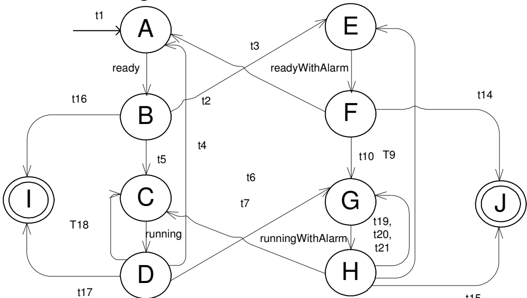


Fig. 8: Corresponding automaton

5.3. Evaluation of the regular expression

To identify variability impact, the regular expression is considered as a function whose variables are issued from UML variable elements. Each evaluation of this function represents the topology of the state machine after choices on variable elements have been performed.

Variables of the function are easily deduced from the UML variable elements. In our example the variable are t2, t3, t6, t7, etc. All combinations of values are not possible for each variable. Indeed, the set of possible value combinations have to be calculated by taking into account the constraints introduced in the *VariationGroups*. For example, transitions t2 and t6 always have the same value. The existence of t2 and t6 depends on the existence of the same variable element, the *startAlarm* trigger. If this trigger is not present in the product model, both transitions are removed.

The evaluation of the regular expression uses the Kleene algebra. More precisely, the following properties are used:

Let A be an alphabet, and let $L \in \mathbf{P}(A^*)$ (all languages based on A^*). Then the following properties hold:

$$\emptyset \mid L = L \mid \emptyset = L$$
$$\emptyset . L = L . \emptyset = \emptyset$$

For example, if t2, t3, t6, t7 are valuated to Ø (imagine a decision which gets rid of these transitions), the evaluation of the function is:

$$F(t2,t3,t6,t7 \rightarrow \emptyset)=(t1.((ready.t5.running.t4))^*.ready.(t16 \mid (t5.running.t17)))$$

We apply this evaluation for all possible combinations. When the function equals Ø for a particular combination, it means that no correct state machine can be derived, thus this particular decision branch is not valid. When the evaluation is not Ø, the function returns a regular expression that represents the topology of the derived state machine. In such cases, one can easily transform the regular expression into a UML state machine – the transformation used is bijective.

In our case study, five behavioral derivations are possible (see figure 9).

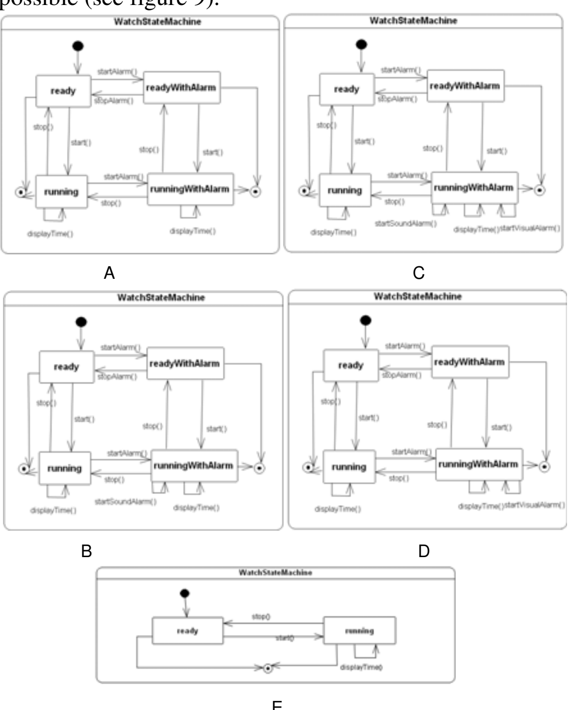


Fig. 9: Five valid state machines

Let us note that such an approach enables to suppress isolated states. As no valid sequence of state and transitions lead to such states, they cannot appear in non-null regular expressions.

5.4. Derivation analysis

After calculating all possible derivations of the state machine, a comparison with the structural derivation is performed. The number of structural derivations may be greater than the number of possible behavioral derivations due to the topology of the state-machine. This construction provides an evaluation of the consistency of both structural and behavioral parts of the model with respect to the derivation process.

First, all derivations of the associated classes are calculated. To do that, we only need to calculate all possible combinations of variable elements that respect the constraints defined by variation groups. Because the number of structural variations is not great in a class (operation and property), this step is reasonably easy to perform. In our example, eight possible classes can be derived (see figure 10).

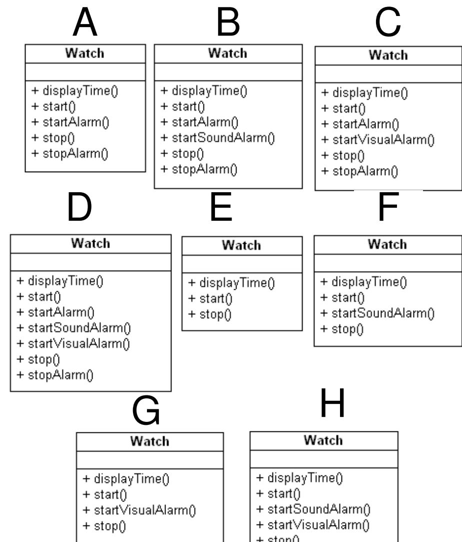


Fig. 10: Eight possible structural derivations

Recall that there is a one to one relation between the derived class and its state machine, a mismatch is clearly shown here: only *five* valid state machines are possible whereas *eight* classes can be obtained. This mismatch must come from a design failure in the structural variability.

If we take a closer look at the derived classes, we note that cases F, G and H feature either one of the specific alarm mode triggering operations, *startSoundAlarm()* or *startVisualAlarm()*, without the trigger *startAlarm().* This contradicts the behavioral parts where all state machine feature

startSoundAlarm() and *startVisualAlarm()* only when both *startAlarm()* and *stopAlarm()* are present.

To solve this, one has to add a constraint in the system family model, to indicate that both *startSoundAlarm()* and *startVisualAlarm()* exist only if the alarm functionality is chosen (see the variation group on figure 11).

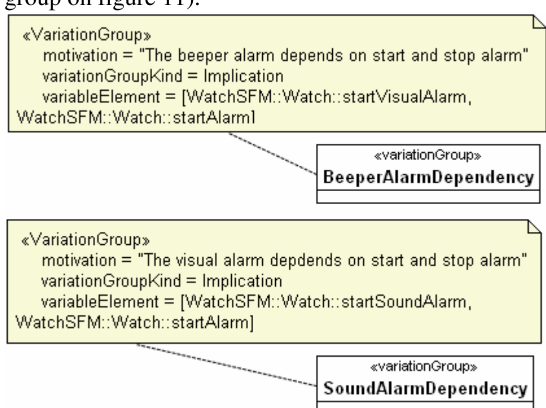


Fig. 11: Refined variation group specification

We may further analyze the result of the derivation process. We are left with five possible cases, A to E. Case E is the watch with no alarm, case D is the full watch (both sound and visual alarms), case C is the watch with visual alarm and case B is the watch with the sound alarm. There remains case A: it features a watch that has the generic alarm triggers *startAlarm()* and *stopAlarm()*, yet no specific alarm mode. This comes from an underspecification of our case study. In fact we assumed that whenever the alarm function is enabled, it would take either or both of the form sound or visual. Yet this was not fully accounted for in the *VariationGroups*. One should update the *TriggerVariationGroup* of figure 5 and change its kind to *OneAmongSeveral* to enforce that at least one specific alarm mode is chosen.

This shows how a carefully monitored derivation process can provide valuable information as to the consistency of the variability scheme introduced at both structural and behavioral levels. The example that we used in this paper, though very simple, shows that errors and underspecifications can easily appear when one has to deal with a variability-enabled model.

5.5 Overall derivation process

Figure 12 summarizes how the various mechanisms presented takes place into a monitored derivation process. The first step consists in calculating all possible regular expressions that may be engendered from the variable state machine. This step acts as a filter to eliminate those that do not respect the constraints expressed in the variation groups of the system family model (see 5.2 above). The second step consists in analyzing the coherence between structural and behavioral derivations. As shown in the case study, the analysis detects incoherencies (see 5.3). Based on this, one can more easily track down design errors or underspecifications and update the model. When the analysis sends no error message, we can be sure that the variability scheme is sound and that all possible derivations can be made.

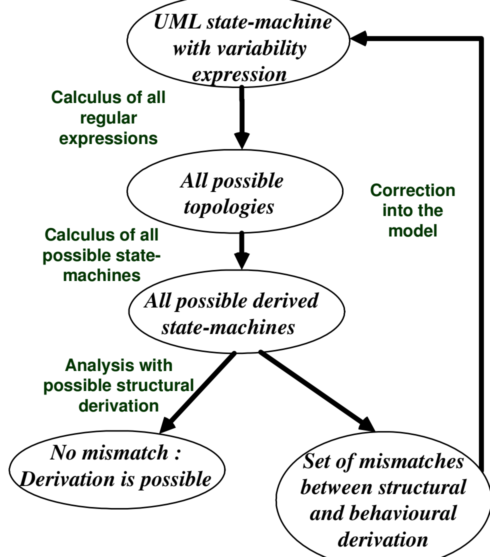


Fig. 12: Monitoring of derivation

6. Conclusion

The approach presented in this paper belongs to a wide range of research works that address the problem of introducing variability within UML models and provide means of managing its complexity. We have found yet that few of these works deal with behavioral models. This is our main concern here, as we are developing design methods for real time and embedded systems for which behavioral modeling is crucial.

Our approach aims at providing as much tool support as possible to variability modeling. In a previous paper [17] we have described the overall process and how decision model could be constructed

from the information structured in our *VariationGroups*. In this paper we described two additional mechanisms: 1) a propagation mechanism ensures that the impacts of variability is fully reflected system-wide based on a set of rules that can be tailored to user needs; 2) an evaluation mechanism enables to analyze what are the allowed derivation branches. Several aspects of these mechanisms can be enhanced: the propagation mechanism could take into account the semantics of the variability constraints instead of simply following model and meta model structural relationships. The evaluation of valid state machines should be extended to more complex state machines, which might reveal insufficiencies in the algebra and or method chosen. Finally the combinatory explosion of calculating all possible structural derivations shall be dealt with - one may think of using constraint-solver here.

Our main goal here was to show that a carefully monitored derivation process provides valuable information to assess the consistency and completeness of a system family model, even for as simple cases as the one presented in this paper. Our approach is operational and supported with various plugins for the Papyrus open-source UML modeler [1]: an assistant helps the design of system family model and supports the propagation mechanism. Another tool provides information about the system family model: number of possible derivations, generation of the decision model. The last tool is an assistant to help the designer to derive a given product model from the system family model.

7. References

[1] "Papyrus UML modeler." http://www.papyrusuml.org: CEA LIST.

[2] J. Bayer, S. Gérard, Ø. Haugen, J. Mansell, B. Møller-Pedersen, J. Oldevik, A. Solberg, P. Tessier, J.-P. Thibault, and T. Widen, "A Unified Conceptual Model For Product Family Variability Modelling," in Software Product Lines, ResearchIssues in Enigneering and Management, T. Käkölä and J. C. Dueñas, Eds. Berlin: Spinger-Verlag, 2006, pp. 195-241.

[3] M. Clauß, "Modeling variability with UML," presented at Net.ObjectDays, Erfurt, Germany, 2001.

[4] P. Clements and L. M. Northrop, Software Product Lines: Practices and Patterns. Boston: Addison Wesley, 2001.

[5] P. Cuenot, D. Chen, S. Gérard, H. Lönn, M.-O. Reiser, D. Servat, R. T. Kolagari, M. Törngren, and M. Weber, "Towards Improving Dependability of Automotive Systems by Using the EAST-ADL Architecture Description Language," in Architecting Dependable Systems IV, Springer, Ed. Berlin / Heidelberg, 2007, pp. 39-65.

[6] M. L. Griss, J. Favaro, and M. d'Alessandro, "Integrating Feature Modeling with the RSEB," presented at 5th Conference on Software Reuse, Victoria, B.C, 1998.

[7] J. v. Gurp and J. Bosch, "Managing Variability in Software Product Lines," presented at Landelijk Architectur Congres, Amsterdam, 2000.

[8] J. E. Hopcroft, R. Motwani, and J. D. Ullmann, Introduction to automata Theory, Language and Computation: Addison Wesley, 2001.

[9] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-Oriented Domain Analysis (FODA)," Carnegie Mellon University CMU/SEI-90-TR-21 ESD-90-TR-222, 1990.

[10] K. C. Kang, J. Lee, and P. Donohoe, "Feature-Oriented Product Line Engineering," in IEEE Software, vol. 19, 2002, pp. 58-65.

[11] S. C. Kleene, "Representation of events in nerve nets and finite automata," in Automata Studies, Annals of Mathematics Studies, vol. 34: Princeton University Press, 1956.

[12] M. V. Lawson, Finite Automata. Boca Raton, Florida: CRC Press LLC, 2004.

[13] M. D. McIlroy, "Mass-Produced Software Component," presented at NATO SCIENCE COMMITTEE, Garmisch, Germany, 1968.

[14] OMG, "A UML profile for MARTE," Object Management Group, http://www.omg.org realtime/07-05-01, 2007.

[15] D. L. Parnas, "Designing software for ease of extension and contraction," in IEEE Transactions on Software Engineering, vol. SE-5, 1979, pp. 128-137.

[16] P. Tessier, S. Gérard, F. Terrier, and J. M. Geib, "Variability Expression within the context of UML: Issues and Comparisons," in Advances in UML/XML based Evolution, Emerging and Innovative Technologies, H. Yang, Ed. De Montfort University, England: Idea Group Publishing, USA, 2005.

[17] P. Tessier, S. Gérard, F. Terrier, and J.-M. Geib, "Using variation propagation for Model-Driven Management of aSystem Family," presented at Sofware Product Line Conference (SPLC), Rennes, 2005.

[18] T. Ziadi, "Manipulation de Lignes de Produits en UML," in IFSIC. Rennes: Université de Rennes 1, 2004, pp. 185.

[19] T. Ziadi, L. Hélouët, and J.-M. Jézéquel, "Towards a UML Profile for Software Product Lines," presented at International Workshop on Product Family Engineering, Seana / Italy, 2003.

Statecharts and Variabilities

Nora Szasz, Pedro Vilanova*
ORT University
Cuareim 1451, 11100 Montevideo, Uruguay
{szasz,vilanova}@ort.edu.uy

Abstract

We present a formalism that allows to specify the behavior of product lines using UML statecharts. We use feature diagrams to describe the common and variant components of a product line, and define mappings that associate features with statecharts, describing the effect of the features on the products in which they are present. We define how to combine different statecharts that specify possible variants of a line. This definition provides a very simple way to obtain the specification of the behavior of any product of the line.

1 Introduction

The complexity of software systems has generated the need of founding developments on abstract models. Modeling allows, among other things, to verify the systems before their construction and to guide their development by means of techniques such as automatic code generation. Visual and graphical notations become more important everyday for communication between people, as well as for human-computer interaction. In particular, for model based software development, the Unified Modelling Language (UML) [10] provides graphical notations for modelling different aspects of systems and it has become the standard for both the academy and the industry. UML follows the object oriented paradigm and allows the description of static and dynamic aspects of software systems. It is a set of languages -mainly graphical notations-, supported by an important number of tools. UML provides several tools for behavior specification: actions, activities, statecharts, interactions and use cases. In particular, statecharts and interactions are specially meant for software design. Statecharts, originally introduced by Harel [9], are used to specify the behavior of class instances (intra-component behaviour). They are compact, expressive and allow describing from simple systems to complex reactive ones.

Software reusability has become a major challenge for the software industry. Reusable artifacts increase productivity by reducing development time. The development of products that vary in more or less peripheral aspects has given rise to the concept of software product lines. A software product line consists of a family of systems that share functionalities and satisfy, in general, the needs of a particular market segment [8, 4]. We use features diagrams [5, 7] to model variability in a family of products. Feature diagrams describe the common, optional and alternative functionalities of a product line, with a hierarchical structure. To obtain a specific product of a line, a configuration is determined by the set of chosen optional and alternative features.

In this work we aim at specifying the behavior of product lines using UML statecharts. In order to specify the behavior of all the products of the line, we determine how each functionality contributes to the behavior of the whole line by defining a function that assigns a statechart to each functionality of the family. The mapping complies with the hierarchical structure and the feature restrictions, i.e., the more features a product has, the richer is the statechart that models it. For this, we define an extension relation between statecharts, to represent when a statechart has a more complex structure than another one. We also define how to combine different extensions of the same statechart into a new coherent statechart. Finally, to describe the behavior of a particular product of the line (which corresponds to a particular configuration of the feature diagram), it is enough to combine the statecharts that implement all the features present in the product. This work constitutes the first step in the construction of a tool that allows designers to specify the behavior of product lines using UML statecharts.

The rest of the article is structured as follows: In section 2 we define feature diagrams and configurations, based on [5, 7]. In section 3 we present UML statecharts syntax, based on the work of [12]. Section 4 introduces the extension relation between statecharts and the combining func-

*This work was partially supported by the Uruguayan Technological Development Program: PDT Project 54-106 "Extensions of UML models for behavioral design of real-time systems and product lines".

tion mentioned above, along with some properties. Finally, in section 5 we define statecharts with variabilities as mappings from feature diagrams to statecharts. We conclude with a brief outline of possible further work.

2 Feature Diagrams

Feature diagrams are used to document features. A feature is a property of a system that directly affects end users, which can be either human or other systems. In the case of software product lines, the main goal of a feature diagram is to specify commonalities and differences amongst the products of the line. In this context, a feature is a distinctive characteristic of a product.

Czarnecki [5, 7] proposes three types of features, namely *mandatory*, *optional* and *alternative*. Additionally there is a *consists of* relation among features, meaning that a feature comprises one or more other features. We call the composed feature *parent* feature and its components *children* or *subfeatures* of the parent feature. Additionally, a set of constraints over features can be defined. A constraint is a proposition over the set of features.

In order to define a particular product of a line a feature diagram can be *configured*, by choosing which features are present in the product, complying with the following interdependency rules: mandatory features must always be present in a product if their parent feature is present, optional features may or may not be present in a product if their parent feature is present, and exactly one of the alternative subfeatures must be present in a product when their parent feature is present.

A feature diagram can naturally be represented as a tree, where the nodes represent the features and the arcs represent the *consists of* relation between them.

2.1 Definition of Feature Diagrams

Given $\mathcal{F}$ a set of feature names, we define a feature diagram as a 6-tuple $\Upsilon = \langle \mathsf{L},\mathsf{N},\mathsf{N_C},\mathsf{R_M},\mathsf{R_O},\mathsf{R_A}\rangle$, where: $\mathsf{L} \in \mathcal{F}$, is the product line name (the root of the tree[1]); $\mathsf{N} \subseteq \mathcal{F}$ is the set of features, ($\mathsf{L} \notin \mathsf{N}$); $\mathsf{N_C} \subseteq \mathcal{C}$ is the set of constraints over features, where $\mathcal{C}$ are the propositional calculus formulas with variables $f_i \in \mathcal{F}$ and connectives $\wedge$, $\vee$ and $\neg$; $\mathsf{R_M}, \mathsf{R_O} \subseteq \{\mathsf{L}\}\cup\mathsf{N} \times \mathsf{N}$, are the mandatory and optional *consists of* relations respectively; and $\mathsf{R_A} \subseteq \{\mathsf{L}\}\cup\mathsf{N} \times \mathcal{P}(\mathsf{N})$ is the alternative *consists of* relation. In addition, the union of the relations $\mathsf{R_M}$, $\mathsf{R_O}$ and $\mathsf{R_A}$ must constitute a tree with nodes in $\mathcal{F}$ and root L. We call $\mathsf{FD}_\mathcal{F}$ the set of feature diagrams with features in $\mathcal{F}$.

[1] Following [5], the root of the tree is not a feature but a concept, thus satisfying the condition that every feature has a parent. For the sake of homogeneity, in this work we will consider it to be a feature.

Basic functions on Feature Diagrams

First we define the projection functions for feature diagrams: M, O and $\mathsf{A}\colon \mathsf{FD}_\mathcal{F} \to \mathcal{F}$ are the set of mandatory, optional and alternative features, defined respectively as $\mathsf{M}(\langle \mathsf{L},\mathsf{N},\mathsf{N_C},\mathsf{R_M},\mathsf{R_O},\mathsf{R_A}\rangle) := \{f{\in}\mathcal{F} \mid \exists\langle f',f\rangle{\in}\mathsf{R_M}\}$, $\mathsf{O}(\langle \mathsf{L},\mathsf{N},\mathsf{N_C},\mathsf{R_M},\mathsf{R_O},\mathsf{R_A}\rangle) := \{f{\in}\mathcal{F} \mid \exists\langle f',f\rangle{\in}\mathsf{R_O}\}$, and $\mathsf{A}(\langle \mathsf{L},\mathsf{N},\mathsf{N_C},\mathsf{R_M},\mathsf{R_O},\mathsf{R_A}\rangle){:=}\{f{\in}\mathcal{F} \mid \exists\langle f',A\rangle{\in}\mathsf{R_A}.\ f{\in}A\}$.

We write the feature diagram argument as a subscript in the functions: For $\Upsilon \in \mathsf{FD}_\mathcal{F}$, we will write M_Υ, O_Υ and A_Υ. We will also use the projection functions L_Υ, N_Υ, $\mathsf{N_C}_\Upsilon$, $\mathsf{R_M}_\Upsilon$, $\mathsf{R_O}_\Upsilon$ and $\mathsf{R_A}_\Upsilon$ respectively to denote the components of a feature diagram $\Upsilon = \langle \mathsf{L},\mathsf{N},\mathsf{N_C},\mathsf{R_M},\mathsf{R_O},\mathsf{R_A}\rangle$.

The following functions and relations will be used later: $\mathsf{chld} \subseteq \mathsf{FD}_\mathcal{F}{\times}\mathcal{F}{\times}\mathcal{F}$ is the child relation: $\mathsf{chld}_\Upsilon(f',f)$ iff $\langle f,f'\rangle{\in}\mathsf{R_M}_\Upsilon{\cup}\mathsf{R_O}_\Upsilon \vee \exists A{\subseteq}\mathsf{N}_\Upsilon.\ (\langle f,A\rangle{\in}\mathsf{R_A}_\Upsilon{\wedge}f'{\in}A)$. The set of features of a given feature diagram is given by the function $\mathsf{fts}\colon \mathsf{FD}_\mathcal{F} \to \mathcal{P}(\mathcal{F})$, defined as $\mathsf{fts}_\Upsilon := \mathsf{M}_\Upsilon{\cup}\mathsf{O}_\Upsilon{\cup}\mathsf{A}_\Upsilon$. $\mathsf{subft} \subseteq \mathsf{FD}_\mathcal{F}{\times}\mathcal{F}{\times}\mathcal{F}$ is the transitive closure of the subfeature relation in a feature diagram: $\mathsf{subft}_\Upsilon(f',f)$ iff $\mathsf{chld}_\Upsilon(f',f) \ \vee\ \exists f''{\in}\mathsf{N}_\Upsilon.\ (\mathsf{chld}_\Upsilon(f',f'') \wedge \mathsf{subft}_\Upsilon(f'',f))$. The set of subfeatures of a given feature in a feature diagram is given by the function $\mathsf{subfts}\colon \mathsf{FD}_\mathcal{F}{\times}\mathcal{F} \to \mathcal{P}(\mathcal{F})$, defined as $\mathsf{subfts}_\Upsilon(f) := \{f'{\in}\mathsf{fts}_\Upsilon \mid \mathsf{subft}_\Upsilon(f',f)\}$. Finally, $\mathsf{Subfts}\colon \mathsf{FD}_\mathcal{F}{\times}\mathcal{P}(\mathcal{F}) \to \mathcal{P}(\mathcal{F})$, is the set of all the subfeatures of the members of a given set of features in a feature diagram, including the set itself: $\mathsf{Subfts}_\Upsilon(F) := F \cup \bigcup_{f\in F} \mathsf{subfts}_\Upsilon(f)$

2.2 Configurations

Feature diagrams describe the common and variant functionalities of products in a product line. In order to obtain specific products of a line defined by a feature diagram Υ, we define the possible configurations of Υ as the instances of the tree that are consistent with the relations amongst its features and the constraints of Υ. Configurations are represented as trees of features, where all the features are mandatory. Formally, given a set of feature names $\mathcal{F}$, a configuration is a 3-tuple $\Phi = \langle \mathsf{P},\mathsf{F},\mathsf{R}\rangle$, where: $\mathsf{P} \in \mathcal{F}$, is the product name (the root of the tree); $\mathsf{F} \subseteq \mathcal{F}$, is the set of features of the product ($\mathsf{P} \notin \mathsf{F}$); and $\mathsf{R} \subseteq \{\mathsf{P}\}{\cup}\mathsf{F}{\times}\mathsf{F}$ is the *consists of* relation. Additionally, configurations must be trees under the relation R, with root P. We call $\mathsf{CS}_\mathcal{F}$ the set of configurations in $\mathcal{F}$.

A configuration of a feature diagram is determined by the set of optional and alternative features that are selected for the product. We define the function $\mathsf{conf}\colon \mathsf{FD}_\mathcal{F} \times \mathcal{P}(\mathcal{F}) \hookrightarrow \mathsf{CS}_\mathcal{F}$, that builds a configuration from a feature diagram Υ and a set F of features. In fact, for a feature diagram $\Upsilon = \langle \mathsf{L},\mathsf{N},\mathsf{N_C},\mathsf{R_M},\mathsf{R_O},\mathsf{R_A}\rangle$ and $C \subseteq \mathcal{F}$, the function $\mathsf{conf}_\Upsilon(C)$ is determined by the elements of C that are optional and alternative features of $\mathcal{F}$.

So, in the definition of conf we do not take into account the features in C that are not in $\mathsf{O}_\Upsilon \cup \mathsf{A}_\Upsilon$. For the alternative features, exactly one child can be chosen from each parent. So $\mathsf{conf}_\Upsilon(C)$ is defined iff for every chosen feature exactly one of its alternative children is chosen in the result. Additionally, all formulas in $\mathsf{N_C}$ must be satisfied by the configuration[2].

Basically, the function conf "erases" all optional and alternative features that are not in C as well as the subtrees that have those features as roots: So, let $F' = \mathsf{Subfts}_\Upsilon(\mathsf{O}_\Upsilon \cup \mathsf{A}_\Upsilon \backslash C)$, $\mathsf{F} = \mathsf{M}_\Upsilon \cup \mathsf{O}_\Upsilon \cup \mathsf{A}_\Upsilon \backslash F'$, $\mathsf{R}' = (\mathsf{R}_{\mathsf{M}_\Upsilon} \cup \mathsf{R}_{\mathsf{O}_\Upsilon} \cup \{\langle f,f' \rangle \mid \exists A \subseteq \mathsf{N}.\ \langle f',A \rangle \in \mathsf{R}_{\mathsf{A}_\Upsilon} \wedge f \in A\})$, and $\mathsf{R} = \mathsf{R}' \cap (\mathsf{F} \cup \{P\} \times \mathsf{F})$. Besides, all the formulae in $\mathsf{N_C}$ must be satisfied for the features present in the configuration. So, if for all $\alpha \in \mathsf{N_C}$, $\mathsf{F} \cup \mathsf{F}^\neg \models \alpha$, then $\mathsf{conf}_\Upsilon(C) := \langle \mathsf{L},\mathsf{F},\mathsf{R} \rangle$, otherwise $\mathsf{conf}_\Upsilon(C)$ is undefined (where $\mathsf{F}^\neg = \{\neg f_i \mid f_i \in F'\}$). The set of all possible configurations of a feature diagram is given by the function $\mathsf{Confs}\colon \mathsf{FD}_\mathcal{F} \to \mathcal{P}(\mathsf{CS}_\mathcal{F})$, defined as $\mathsf{Confs}_\Upsilon := \bigcup_{C \subseteq \mathsf{fts}_\Upsilon} \{\mathsf{conf}_\Upsilon(C) \in \mathsf{CS}_\mathcal{F} \mid \mathsf{conf}_\Upsilon(C) \text{ not undefined}\}$.

2.3 Example

Let $\mathcal{F} = \{P, f_1, f_2, f_3, f_4, f_5, f_6\}$
$\Upsilon_1 = \langle P, \{f_1, f_2, f_3, f_4, f_5, f_6\}, \emptyset, \{\langle P,f_1 \rangle, \langle f_2,f_3 \rangle\},$
$\{\langle P,f_2 \rangle, \langle f_2,f_4 \rangle\}, \{\langle f_3, \{f_5, f_6\} \rangle\} \rangle \in \mathsf{FD}_\mathcal{F}$

In figure 1 we show an example of the notation. Graphically, optional features are marked with a black dot, and alternatives features with a line across alternative group. The possible configurations of Υ_1 are the following:
$\mathsf{conf}_{\Upsilon_1}(\emptyset) = \langle P, \{f_1\}, \{\langle P,f_1 \rangle\} \rangle$
$\mathsf{conf}_{\Upsilon_1}(\{f_2, f_5\}) = \langle P, \{f_1, f_2, f_3, f_5\}, \{\langle P,f_1 \rangle, \langle P,f_2 \rangle, \langle f_2,f_3 \rangle, \langle f_3,f_5 \rangle\} \rangle$
$\mathsf{conf}_{\Upsilon_1}(\{f_2, f_4, f_6\}) = \langle P, \{f_1, f_2, f_3, f_4, f_6\}, \{\langle P,f_1 \rangle, \langle P,f_2 \rangle, \langle f_2,f_3 \rangle, \langle f_2,f_4 \rangle, \langle f_3,f_6 \rangle\} \rangle$

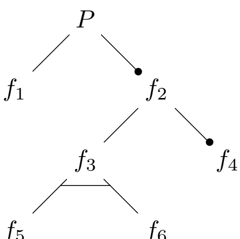


Figure 1. Feature Diagram Example

[2] We could have assigned a dependent type to the function conf to take into account the restrictions, but we keep it simpler by defining partial functions

3 Statecharts

UML statecharts constitute a notation to describe behavioral aspects of a system. They were first introduced by Harel [9] and incorporated to the different versions of UML with some variations. Statecharts are a generalization of state automata. Basically, they consist of *states* and *transitions* between them. The main feature of statecharts is that states can be refined, defining a state hierarchy. The decomposition of a state can be either *sequential* or *parallel*. In the first case, a state is decomposed into a new state automaton (OR state), while in the second case a state is decomposed in two or more automata that can execute concurrently (AND state). Transitions are directed arrows between states. A transition connects a *source state* to a *target state*, and inter-level transitions are allowed. Transitions are labeled by a trigger event, a sequence of actions and the type of history of the target state. There is a *history mechanism* that allows transitions to reenter a sequential state in the last active substate.

In this section we present the main concepts and definitions for statecharts, based on [12].

3.1 Domains

$\mathcal{S}$ is the set of state names, $\mathcal{T}$ is the set of transition names ($\mathcal{S} \cap \mathcal{T} = \emptyset$), A is the set of action names, $\mathsf{E} \subseteq \mathsf{A}$ is the set of events, $\mathsf{HT} = \{\mathsf{none}, \mathsf{shallow}, \mathsf{deep}\}$ are the history types, $\mathsf{T} = \mathcal{T} \times \mathcal{S} \times \mathcal{P}(\mathcal{S}) \times \mathsf{E} \times \mathsf{A}^* \times \mathcal{P}(\mathcal{S}) \times \mathcal{S} \times \mathsf{HT}$ is the type of the transitions.

3.2 States

The set SC of UML statecharts is inductively defined by the following rules, simultaneously with the functions $\mathsf{name}\colon \mathsf{SC} \to \mathcal{S}$, that is, the name of the statechart, $\mathsf{snames}\colon \mathsf{SC} \to \mathcal{P}(\mathcal{S})$, the set of names of the state components (substates) of the statechart, $\mathsf{tnames}\colon \mathsf{SC} \to \mathcal{P}(\mathcal{T})$, the set of names of the internal transitions between the substates of the statechart, and $\mathsf{stype}\colon \mathsf{SC} \to \{\mathsf{basic}, \mathsf{and}, \mathsf{or}\}$, the type of the statechart. We will also use in the definition the relation $\mathsf{disjnames}(s_i, s_j)$, that holds if s_i and s_j do not have common state names in their components. It is defined at the end of this section.

Basic Statecharts

$s = [\hat{s}, (en, ex)]$ is a basic statechart with name $\hat{s}$ and entry and exit actions en, ex respectively.

$$\frac{\hat{s} \in \mathcal{S} \quad en, ex \in \mathsf{A}^*}{[\hat{s}, (en, ex)] \in \mathsf{SC}} \ \text{Basic}$$

3

$\mathsf{name}([\hat{s},(en,ex)]) := \hat{s}$,
$\mathsf{snames}([\hat{s},(en,ex)]) := \{\hat{s}\}$,
$\mathsf{tnames}([\hat{s},(en,ex)]) := \emptyset$,
$\mathsf{stype}([\hat{s},(en,ex)]) := \mathsf{basic}$.

We will use the following notational convention: statecharts names will be of the form $\hat{s}$, $\hat{s_1}$, $\hat{s_2}, .., \hat{r}$, $\hat{r_1}$, $\hat{r_2}, \ldots$ and the variables $s, s_1, s_2, .., r, r_1, r_2, \ldots$ will denote statecharts.

And-Statecharts

$s = [\hat{s},(s_1,..,s_n),(en,ex)]$ is an and-statechart with name $\hat{s}$ and entry and exit actions en, ex respectively. The statecharts $s_1, .., s_n$ are called the parallel components (substates) of s[3].

$$\frac{\begin{array}{l} s_1, .., s_n \in \mathsf{SC} \\ \forall i{\neq}j.\ \mathsf{disjnames}(s_i,s_j) \\ \hat{s} \in \mathcal{S} \\ \{\hat{s}\} \cap \bigcup_{1\leq i\leq n} \mathsf{snames}(s_i) = \emptyset \\ en, ex \in \mathsf{A}^* \end{array}}{[\hat{s},(s_1,..,s_n),(en,ex)] \in \mathsf{SC}} \ \mathsf{And}$$

$\mathsf{name}([\hat{s},(s_1,..,s_n),(en,ex)]) := \hat{s}$,
$\mathsf{snames}([\hat{s},(s_1,..,s_n),(en,ex)]) :=$
$\{\hat{s}\} \cup \bigcup_{1\leq i\leq n} \mathsf{snames}(s_i)$,
$\mathsf{tnames}([\hat{s},(s_1,..,s_n),(en,ex)]) := \bigcup_{1\leq i\leq n} \mathsf{tnames}(s_i)$,
$\mathsf{stype}([\hat{s},(s_1,..,s_n),(en,ex)]) := \mathsf{and}$.

Or-Statecharts

$s = [\hat{s},(s_1,..,s_n),T,(en,ex)]$ is an or-statechart with name $\hat{s}$ and entry and exit actions en, ex respectively. The statecharts $s_1, .., s_n$ are the components of s and T is the set of transitions between the components of s[4].

$$\frac{\begin{array}{l} s_1, .., s_n \in \mathsf{SC} \\ \forall i{\neq}j.\ \mathsf{disjnames}(s_i,s_j) \\ \hat{s} \in \mathcal{S} \\ \{\hat{s}\} \cap \bigcup_{1\leq i\leq n} \mathsf{snames}(s_i) = \emptyset \\ T \subseteq \mathsf{T}\ (^*) \\ en, ex \in \mathsf{A}^* \end{array}}{[\hat{s},(s_1,..,s_n),T,(en,ex)] \in \mathsf{SC}} \ \mathsf{Or}$$

(*) For each $\langle\hat{t},s_s,S,e,\alpha,T_d,s_t,ht\rangle \in T$ it is required that: $\mathsf{name}(s_s) \in \bigcup_{1\leq i\leq n} \mathsf{name}(s_i)$, $S \in \mathsf{conf\text{-}all}(s_s)$[5] $\hat{t}$ is unique in T, $T_d \in \mathsf{conf\text{-}all}(s_t)$, and $\{\hat{t}\} \cap \bigcup_{1\leq i\leq n} \mathsf{tnames}(s_i) = \emptyset$

$\mathsf{name}([\hat{s},(s_1,..,s_n),T,(en,ex)]) := \hat{s}$,
$\mathsf{snames}([\hat{s},(s_1,..,s_n),T,(en,ex)]) :=$
$\{\hat{s}\} \cup \bigcup_{1\leq i\leq n} \mathsf{snames}(s_i)$,
$\mathsf{tnames}([\hat{s},(s_1,..,s_n),T,(en,ex)]) :=$
$\bigcup_{t\in T} \mathsf{name}(t) \cup \bigcup_{1\leq i\leq n} \mathsf{tnames}(s_i)$,
$\mathsf{stype}([\hat{s},(s_1,..,s_n),T,(en,ex)]) := \mathsf{or}$

We further define:
$\mathsf{SC\text{-}BASIC} = \{s \in \mathsf{SC} \mid \mathsf{stype}(s) = \mathsf{basic}\}$,
$\mathsf{SC\text{-}AND} = \{s \in \mathsf{SC} \mid \mathsf{stype}(s) = \mathsf{and}\}$,
$\mathsf{SC\text{-}OR} = \{s \in \mathsf{SC} \mid \mathsf{stype}(s) = \mathsf{or}\}$,
$\mathsf{disjnames}(s,s') :=$
$(\mathsf{snames}(s) \cap \mathsf{snames}(s')) \cup (\mathsf{tnames}(s) \cap \mathsf{tnames}(s')) = \emptyset$,
$\mathsf{disjnames}(s,T) := \forall t \in T.\ \mathsf{name}(t) \notin \mathsf{tnames}(s)$

3.3 Transitions

We will generally use the variables $t, t_1, t_2, ..$ to denote transitions, and $\hat{t}$, $\hat{t_1}$, $\hat{t_2}, ..$ to denote their names. As state and transition names are mutually disjoint and unique in a statechart, we can refer to a state or a transition univocally by its name.

Given $t = \langle\hat{t},s_s,S,e,\alpha,T,s_t,ht\rangle \in \mathsf{T}$, the following are defined: $\mathsf{name}(t) := \hat{t}$, is the name of the transition t, $\mathsf{sou}(t) := s_s$, $\mathsf{tar}(t) := s_t$, are the source and target states of t respectively, $\mathsf{souRes}(t) := S$, $S \in \mathsf{conf\text{-}all}(s_s)$, is the source restriction set, $\mathsf{ev}(t) := e$, is the triggering event of t, $\mathsf{act}(t) := \alpha$, is the action associated to t, $\mathsf{tarDet}(t) := T$, is the target determinator set, $\mathsf{historyType}(t) := ht$, is the history type of t (see [12] for more details). We say a transition t uses the history mechanism, if $\mathsf{historyType}(t) \in \{\mathsf{deep}, \mathsf{shallow}\}$.

4 Extensions

4.1 Extension relation

We define a relation $\succ$ between statecharts, such that $s_1 \succ s_2$ (read "s_2 extends s_1") if s_2 enriches states or transitions of s_1 with more complex structures. Basically, we can extend a statechart by either adding a parallel or sequential statechart to it, adding a new transition between two existing states, or adding actions in transitions or entry and exit actions. For this last extension we define the relation $\sqsubseteq$ between sequences of actions as the ordinary subsequence relation between elements of A^*.

[3] For our purposes, it is the same to define the substates of s as a set $\{s_1, .., s_n\}$. We follow [12] and define it as a sequence.

[4] In [12] or-statecharts have also an active state. This component is placed for semantical purposes, and it is not essential in the context of this work, so we do not consider it here. Adding it to the definitions that follow is straightforward.

[5] The function conf-all is defined to handle inter-level transitions just like normal transitions on the level of the uppermost states that the inter-level transition exits and enters. S is a complete configuration of the source state, i.e., a set containing a "path" from the uppermost level state to the source of t. The same applies to the target determinator set T_d (see [12] for more details).

The extensions can be performed zero or more times to a statechart or to any of its components, thus yielding the definition of the partial order $\succ$ given in figure 2. In the definition, we assume that the well-formedness conditions of the definitions given in section 3.2 hold whenever we build a statechart.

4.2 Intersection

Given a statechart, it can be refined in several ways. The question is whether there is a way to combine different extensions into an integral new statechart. Formally, given $r_1, r_2 \in \mathsf{SC}$ such that $\exists s{\in}\mathsf{SC}.\ s \succ r_1 \wedge s \succ r_2$ we want to define a new statechart $r_1 {\cap} r_2$ such that $r_1 \succ r_1{\cap}r_2$ and $r_2 \succ r_1{\cap}r_2$. Moreover, we want to do this with the minimum amount of refining steps as possible, i.e., $\forall r_3{\in}\mathsf{SC}.\ (r_1 \succ r_3 \wedge r_2 \succ r_3) \Rightarrow r_1{\cap}r_2 \succ r_3$

We will not always be able to define the intersection of two statecharts, even if they are both extensions of the same statechart (consider, for instance, s to be a basic statechart which is extended parallelly on the one hand and sequentially on the other). We therefore need to handle inconsistent statecharts. For the sake of completeness, we will add a new statechart to the syntax defined in section 3.2: $\perp$ will stand for the bottom element of the set SC, as well as for any statechart having it as a component. Regarding the $\succ$ relation, $\perp$ is a extension of any statechart. Formally, we have that $\forall s{\in}\mathsf{SC}.\ s \succ \perp$

The definition of $\cap$ is simple but quite extensive, since we must consider all the pairs r_1, r_2 such that r_1 and r_2 can be extensions of the same statechart s, according to the definition given in figure 2. It basically consists of carrying out both extensions on the original statechart, whenever this is possible. The reader can just look at the first cases, and will be able to deduce the rest of them.

Definition. Let $r_1, r_2 \in \mathsf{SC}$ such that $\exists s{\in}\mathsf{SC}.\ s \succ r_1 \wedge s \succ r_2$. We define $r_1{\cap}r_2$ by induction on $s \succ r_1$:

ext-and1

If $s = [\hat{s},(en,ex)], r_1 = [\hat{s},(s'),(en,ex)]$, then we have the following cases:

- $r_2 = [\hat{s},(s''),(en,ex)]$

$$\Rightarrow r_1{\cap}r_2 := \begin{cases} [\hat{s},(s',s''),(en,ex)] & \text{if } \mathsf{disjnames}(s',s'') \\ \perp & \text{otherwise} \end{cases}$$

- $r_2 = [\hat{s},(s''),\emptyset,(en,ex)]$
$\Rightarrow r_1{\cap}r_2 := \perp$
- $r_2 = [\hat{s},(en',ex)]$, with $en \sqsubseteq en'$
$\Rightarrow r_1{\cap}r_2 := [\hat{s},(s'),(en',ex)]$
- $r_2 = [\hat{s},(en,ex')]$, with $ex \sqsubseteq ex'$
$\Rightarrow r_1{\cap}r_2 := [\hat{s},(s'),(en,ex')]$

ext-and2

If $s{=}[\hat{s},(s_1,..,s_n),(en,ex)]$, $r_1{=}[\hat{s},(s_1,..,s_n,s'),(en,ex)]$, then we have the following cases:

- $r_2 = [\hat{s},(s_1,..,s_n,s''),(en,ex)] \Rightarrow r_1{\cap}r_2 :=$
$$\begin{cases} [\hat{s},(s_1,..,s_n,s',s''),(en,ex)] & \text{if } \mathsf{disjnames}(s',s'') \\ \perp & \text{otherwise} \end{cases}$$
- $r_2 = [\hat{s},(s_1,..,s_i',..,s_n),(en,ex)]$,
with $s_i \succ s_i' \Rightarrow r_1{\cap}r_2 :=$
$$\begin{cases} [\hat{s},(s_1,..,s_i',..,s_n,s'),(en,ex)] & \text{if } \mathsf{disjnames}(s_i',s') \\ \perp & \text{otherwise} \end{cases}$$
- $r_2 = [\hat{s},(en',ex)]$, with $en \sqsubseteq en'$
$\Rightarrow r_1{\cap}r_2 := [\hat{s},(s_1,..,s_n,s'),(en',ex)]$
- $r_2 = [\hat{s},(s_1,..,s_n),(en,ex')]$, with $ex \sqsubseteq ex'$
$\Rightarrow r_1{\cap}r_2 := [\hat{s},(s_1,..,s_n,s'),(en,ex')]$

inside-and

If $s = [\hat{s},(s_1,..,s_n),(en,ex)]$,
$r_1 = [\hat{s},(s_1,..,s_i',..,s_n),(en,ex)]$ with $s_i \succ s_i'$, then we have the following cases:

- $r_2 = [\hat{s},(s_1,..,s_n,s'),(en,ex)] \Rightarrow r_1{\cap}r_2 :=$
$$\begin{cases} [\hat{s},(s_1,..,s_i',..,s_n,s'),(en,ex)] & \text{if } \mathsf{disjnames}(s_i',s') \\ \perp & \text{otherwise} \end{cases}$$
- $r_2 = [\hat{s},(s_1,..,s_j',..,s_n),(en,ex)] \Rightarrow r_1{\cap}r_2 :=$
$$\begin{cases} [\hat{s},(s_1,..,s_i'{\cap}s_j',..,s_n),(en,ex)] & \text{if } i=j \\ [\hat{s},(s_1,..,s_i',..,s_j',..,s_n),(en,ex)] & \text{if } \mathsf{disjnames}(s_i',s_j') \\ \perp & \text{otherwise} \end{cases}$$
- $r_2 = [\hat{s},(s_1,..,s_n),(en',ex)]$, with $en \sqsubseteq en'$
$\Rightarrow r_1{\cap}r_2 := [\hat{s},(s_1,..,s_i',..,s_n),(en',ex)]$
- $r_2 = [\hat{s},(s_1,..,s_n),(en,ex')]$, with $ex \sqsubseteq ex'$
$\Rightarrow r_1{\cap}r_2 := [\hat{s},(s_1,..,s_i',..,s_n),(en,ex')]$

ext-or1

If $s{=}[\hat{s},(en,ex)], r_1{=}[\hat{s},(s'),\emptyset,(en,ex)]$, then we have the following cases:

- $r_2 = [\hat{s},(s''),(en,ex)] \Rightarrow r_1{\cap}r_2 := \perp$
- $r_2 = [\hat{s},(s''),\emptyset,(en,ex)] \Rightarrow r_1{\cap}r_2 :=$
$$\begin{cases} [\hat{s},(s',s''),\emptyset,(en,ex)] & \text{if } \mathsf{disjnames}(s',s'') \\ \perp & \text{otherwise} \end{cases}$$
- $r_2 = [\hat{s},(s'),\emptyset,(en',ex)]$, with $en \sqsubseteq en'$
$\Rightarrow r_1{\cap}r_2 := [\hat{s},(s'),\emptyset,(en',ex)]$
- $r_2 = [\hat{s},(s'),\emptyset,(en,ex')]$, with $ex \sqsubseteq ex'$
$\Rightarrow r_1{\cap}r_2 := [\hat{s},(s'),\emptyset,(en,ex')]$

ext-or2

If $s{=}[\hat{s},(s_1,..,s_n),T,(en,ex)]$,
$r_1{=}[\hat{s},(s_1,..,s_n,s'),T,(en,ex)]$, then we have the following cases:

- $r_2 = [\hat{s},(s_1,..,s_n,s''),T,(en,ex)] \Rightarrow r_1{\cap}r_2 :=$
$$\begin{cases} [\hat{s},(s_1,..,s_n,s',s''),T,(en,ex)] & \text{if } \mathsf{disjnames}(s',s'') \\ \perp & \text{otherwise} \end{cases}$$
- $r_2 = [\hat{s},(s_1,..,s_i',..,s_n),T,(en,ex)]$,
with $s_i \succ s_i' \Rightarrow r_1{\cap}r_2 :=$
$$\begin{cases} [\hat{s},(s_1,..,s_i',..,s_n,s'),T,(en,ex)] & \text{if } \mathsf{disjnames}(s_i',s') \\ \perp & \text{otherwise} \end{cases}$$

Adding a new parallel component

$$\frac{[\hat{s},(en,ex)] \in \text{SC-BASIC} \quad s' \in \text{SC}}{[\hat{s},(en,ex)] \succ [\hat{s},(s'),(en,ex)]} \text{ ext-and1} \qquad \frac{[\hat{s},(s_1,..,s_n),(en,ex)] \in \text{SC-AND} \quad s' \in \text{SC}}{[\hat{s},(s_1,..,s_n),(en,ex)] \succ [\hat{s},(s_1,..,s_n,s'),(en,ex)]} \text{ ext-and2}$$

$$\frac{[\hat{s},(s_1,..,s_i,..,s_n),(en,ex)] \in \text{SC-AND} \quad s_i \succ s'_i}{[\hat{s},(s_1,..,s_i,..,s_n),(en,ex)] \succ [\hat{s},(s_1,..,s'_i,..,s_n),(en,ex)]} \text{ inside-and}$$

Adding a new sequential component

$$\frac{[\hat{s},(en,ex)] \in \text{SC-BASIC} \quad s' \in \text{SC}}{[\hat{s},(en,ex)] \succ [\hat{s},(s'),\emptyset,(en,ex)]} \text{ ext-or1} \qquad \frac{[\hat{s},(s_1,..,s_n),T,(en,ex)] \in \text{SC-OR} \quad s' \in \text{SC}}{[\hat{s},(s_1,..,s_n),T,(en,ex)] \succ [\hat{s},(s_1,..,s_n,s'),T,(en,ex)]} \text{ ext-or2}$$

$$\frac{[\hat{s},(s_1,..,s_i,..,s_n),T,(en,ex)] \in \text{SC-OR} \quad s_i \succ s'_i}{[\hat{s},(s_1,..,s_i,..,s_n),T,(en,ex)] \succ [\hat{s},(s_1,..,s'_i,..,s_n),T,(en,ex)]} \text{ inside-or}$$

Adding a new transition

$$\frac{[\hat{s},(s_1,..,s_n),T,(en,ex)] \in \text{SC-OR} \quad t \in \text{T}}{[\hat{s},(s_1,..,s_n),T,(en,ex)] \succ [\hat{s},(s_1,..,s_n),T \cup \{t\},(en,ex)]} \text{ add-trans}$$

Adding actions

$$\frac{[\hat{s},(s_1,..,s_n),T,(en,ex)] \in \text{SC-OR} \quad \alpha \sqsubseteq \alpha' \quad t = \langle \hat{t},s_s,S,e,\alpha,T,s_t,ht \rangle \in T \quad t' \in \text{T}}{[\hat{s},(s_1,..,s_n),T,(en,ex)] \succ [\hat{s},(s_1,..,s_n),(T \backslash \{t\}) \cup \{t'\},(en,ex)]} \text{ ext-act-trans}$$

where $t' = \langle \hat{t},s_s,S,e,\alpha',T,s_t,ht \rangle$

$$\frac{[\hat{s},(en,ex)] \in \text{SC-BASIC} \quad en \sqsubseteq en'}{[\hat{s},(en,ex)] \succ [\hat{s},(en',ex)]} \text{ ext-act-en1} \qquad \frac{[\hat{s},(s_1,..,s_n),(en,ex)] \in \text{SC-AND} \quad en \sqsubseteq en'}{[\hat{s},(s_1,..,s_n),(en,ex)] \succ [\hat{s},(s_1,..,s_n),(en',ex)]} \text{ ext-act-en2}$$

$$\frac{[\hat{s},(s_1,..,s_n),T,(en,ex)] \in \text{SC-OR} \quad en \sqsubseteq en'}{[\hat{s},(s_1,..,s_n),T,(en,ex)] \succ [\hat{s},(s_1,..,s_n),T,(en',ex)]} \text{ ext-act-en3}$$

$$\frac{[\hat{s},(en,ex)] \in \text{SC-BASIC} \quad ex \sqsubseteq ex'}{[\hat{s},(en,ex)] \succ [\hat{s},(en,ex')]} \text{ ext-act-ex1} \qquad \frac{[\hat{s},(s_1,..,s_n),(en,ex)] \in \text{SC-AND} \quad ex \sqsubseteq ex'}{[\hat{s},(s_1,..,s_n),(en,ex)] \succ [\hat{s},(s_1,..,s_n),(en,ex')]} \text{ ext-act-ex2}$$

$$\frac{[\hat{s},(s_1,..,s_n),T,(en,ex)] \in \text{SC-OR} \quad ex \sqsubseteq ex'}{[\hat{s},(s_1,..,s_n),T,(en,ex)] \succ [\hat{s},(s_1,..,s_n),T,(en,ex')]} \text{ ext-act-ex3}$$

Reflexivity and transitivity

$$\frac{s \in \text{SC}}{s \succ s} \text{ reflexivity} \qquad \frac{s \succ s' \quad s' \succ s''}{s \succ s''} \text{ transitivity}$$

Figure 2. Extension Relation $\succ$

6

• $r_2 = [\hat{s}, (s_1,..,s_n), T', (en, ex)]$,
with $T' = T \cup \{t\} \Rightarrow r_1 \cap r_2 :=$
$\begin{cases} [\hat{s}, (s_1,..,s_n, s'), T', (en, ex)] & \text{if disjnames}(s',\{t\}) \\ \bot & \text{otherwise} \end{cases}$
• $r_2 = [\hat{s}, (s_1,..,s_n), T', (en, ex)]$, with
$t=\langle\hat{t},s_s,S,e,\alpha,T,s_t,ht\rangle\in T$, $t'=\langle\hat{t},s_s,S,e,\alpha',T,s_t,ht\rangle$, $\alpha \sqsubseteq \alpha'$, $T' = (T\setminus\{t\})\cup\{t'\}$
$\Rightarrow r_1 \cap r_2 := [\hat{s}, (s_1,..,s_n, s'), T', (en, ex)]$
• $r_2 = [\hat{s}, (s_1,..,s_n), T, (en', ex)]$, with $en \sqsubseteq en'$
$\Rightarrow r_1 \cap r_2 := [\hat{s}, (s_1,..,s_n, s'), T, (en', ex)]$
• $r_2 = [\hat{s}, (s_1,..,s_n), T, (en, ex')]$, with $ex \sqsubseteq ex'$
$\Rightarrow r_1 \cap r_2 := [\hat{s}, (s_1,..,s_n, s'), T, (en, ex')]$

inside-or
If $s = [\hat{s},(s_1,..,s_n),T,(en,ex)]$,
$r_1 = [\hat{s}, (s_1,.., s'_i,.., s_n), T, (en, ex)]$ with $s_i \succ s'_i$, then we have the following cases:
• $r_2 = [\hat{s}, (s_1,..,s_n, s'), T, (en, ex)] \Rightarrow r_1 \cap r_2 :=$
$\begin{cases} [\hat{s}, (s_1,.., s'_i,.., s_n, s'), T, (en, ex)] & \text{if disjnames}(s'_i,s') \\ \bot & \text{otherwise} \end{cases}$
• $r_2 = [\hat{s}, (s_1,.., s'_j,.., s_n), T, (en, ex)]$, with
$s_j \succ s'_j \Rightarrow r_1 \cap r_2 :=$
$\begin{cases} [\hat{s}, (s_1,.., s'_i \cap s'_j,.., s_n), T, (en, ex)] \\ \quad \text{if } i = j \\ [\hat{s}, (s_1,.., s'_i,.., s'_j,.., s_n), T, (en, ex)] \\ \quad \text{if disjnames}(s'_i,s'_j) \\ \bot \\ \quad \text{otherwise} \end{cases}$
• $r_2 = [\hat{s}, (s_1,..,s_n), T', (en, ex)]$, with $T' = T \cup \{t\}$
$\Rightarrow r_1 \cap r_2 := [\hat{s}, (s_1,.., s'_i,.., s_n), T', (en, ex)]$
• $r_2 = [\hat{s}, (s_1,..,s_n), T', (en, ex)]$, with
$t=\langle\hat{t},s_s,S,e,\alpha,T,s_t,ht\rangle\in T$, $t'=\langle\hat{t},s_s,S,e,\alpha',T,s_t,ht\rangle$, $\alpha\sqsubseteq\alpha'$, $T'=(T\setminus\{t\})\cup\{t'\}$
$\Rightarrow r_1 \cap r_2 := [\hat{s}, (s_1,.., s'_i,.., s_n), T', (en, ex)]$
• $r_2 = [\hat{s}, (s_1,..,s_n), T, (en', ex)]$, with $en \sqsubseteq en'$
$\Rightarrow r_1 \cap r_2 := [\hat{s}, (s_1,.., s'_i,.., s_n), T, (en', ex)]$
• $r_2 = [\hat{s}, (s_1,..,s_n), (en, ex')]$, with $ex \sqsubseteq ex'$
$\Rightarrow r_1 \cap r_2 := [\hat{s}, (s_1,.., s'_i,.., s_n), T, (en, ex')]$

ext-trans
If $s = [\hat{s},(s_1,..,s_n),T,(en,ex)]$,
$r_1 = [\hat{s}, (s_1,..,s_n), T', (en, ex)]$ with $T' = T \cup \{t\}$, then we have the following cases:
• $r_2 = [\hat{s}, (s_1,..,s_n, s'), T, (en, ex)]$
$\Rightarrow r_1 \cap r_2 := \begin{cases} [\hat{s}, (s_1,..,s_n, s'), T', (en, ex)] \\ \quad \text{if disjnames}(s',\{t\}) \\ \bot \\ \quad \text{otherwise} \end{cases}$
• $r_2 = [\hat{s}, (s_1,.., s'_i,.., s_n), T, (en, ex)]$,
with $s_i \succ s'_i \Rightarrow r_1 \cap r_2 :=$
$\begin{cases} [\hat{s}, (s_1,.., s'_i,.., s_n), T', (en, ex)] & \text{if disjnames}(s'_i,\{t\}) \\ \bot & \text{otherwise} \end{cases}$
• $r_2 = [\hat{s}, (s_1,..,s_n), T'', (en, ex)]$,
with $T'' = T \cup \{t'\} \Rightarrow r_1 \cap r_2 :=$
$\begin{cases} [\hat{s}, (s_1,..,s_n), T' \cup \{t'\}, (en, ex)] & \text{if name}(t) \neq \text{name}(t') \\ \bot & \text{otherwise} \end{cases}$
• $r_2 = [\hat{s}, (s_1,..,s_n), T'', (en, ex)]$, with
$t'=\langle\hat{t}',s_s,S,e,\alpha,T,s_t,ht\rangle\in T$, $t''=\langle\hat{t}',s_s,S,e,\alpha',T,s_t,ht\rangle$, $\alpha\sqsubseteq\alpha'$, $T''=(T\setminus\{t'\})\cup\{t''\}$
$\Rightarrow r_1 \cap r_2 := [\hat{s}, (s_1,..,s_n), T'' \cup \{t\}, (en, ex)]$
• $r_2 = [\hat{s}, (s_1,..,s_n), T, (en', ex)]$, with $en \sqsubseteq en'$
$\Rightarrow r_1 \cap r_2 := [\hat{s}, (s_1,..,s_n), T', (en', ex)]$
• $r_2 = [\hat{s}, (s_1,..,s_n), T, (en, ex')]$, with $ex \sqsubseteq ex'$
$\Rightarrow r_1 \cap r_2 := [\hat{s}, (s_1,..,s_n), T', (en, ex')]$

ext-act-trans
If $s=[\hat{s}, (s_1,..,s_n), T, (en, ex)]$,
$r_1=[\hat{s}, (s_1,..,s_n), T', (en, ex)]$ with
$T'=(T\setminus\{t\})\cup\{t'\}$, $t=\langle\hat{t},s_s,S,e,\alpha,T,s_t,ht\rangle\in T$,
$t'=\langle\hat{t},s_s,S,e,\alpha',T,s_t,ht\rangle$, $\alpha\sqsubseteq\alpha'$, then we have the following cases:
• $r_2 = [\hat{s}, (s_1,..,s_n, s'), T, (en, ex)]$
$\Rightarrow r_1 \cap r_2 := [\hat{s}, (s_1,..,s_n, s'), T', (en, ex)]$
• $r_2 = [\hat{s}, (s_1,.., s'_i,.., s_n), T, (en, ex)]$, with $s_i \succ s'_i$
$\Rightarrow r_1 \cap r_2 := [\hat{s}, (s_1,.., s'_i,.., s_n), T', (en, ex)]$
• $r_2 = [\hat{s}, (s_1,..,s_n), T'', (en, ex)]$, with $T'' = T \cup \{t''\}$
$\Rightarrow r_1 \cap r_2 := [\hat{s}, (s_1,..,s_n, s'), T' \cup \{t''\}, (en, ex)]$
• $r_2 = [\hat{s}, (s_1,..,s_n), T'', (en, ex)]$, with
$t''=\langle\hat{t},s_s,S,e,\alpha,T,s_t,ht\rangle\in T$, $t'''=\langle\hat{t},s_s,S,e,\alpha',T,s_t,ht\rangle$, $\alpha\sqsubseteq\alpha'$, $T'' = (T\setminus\{t''\})\cup\{t'''\} \Rightarrow r_1 \cap r_2 :=$
$\begin{cases} [\hat{s}, (s_1,..,s_n, s'), (T'\setminus\{t''\})\cup\{t'''\}, (en, ex)] \\ \quad \text{if name}(t) \neq \text{name}(t'') \\ \bot \\ \quad \text{otherwise} \end{cases}$
• $r_2 = [\hat{s}, (s_1,..,s_n), T, (en', ex)]$, with $en \sqsubseteq en'$
$\Rightarrow r_1 \cap r_2 := [\hat{s}, (s_1,..,s_n), T', (en', ex)]$
• $r_2 = [\hat{s}, (s_1,..,s_n), T, (en, ex')]$, with $ex \sqsubseteq ex'$
$\Rightarrow r_1 \cap r_2 := [\hat{s}, (s_1,..,s_n), T', (en, ex')]$

The remaining action cases (ext-act-en1, ext-act-en2, ext-act-en3, ext-act-ex1, ext-act-ex2, ext-act-ex3) are analogous to the last ones, so we omit them for space reasons.

reflexivity
If $s = r_1$, then $\forall r_2 \in \mathsf{SC}.\ s \succ r_2$, we define $r_1 \cap r_2 := r_2$.

transitivity
If $s \succ r' \succ r_1$, then $\forall r_2 \in \mathsf{SC}.\ s \succ r_2$, we define $r_1 \cap r_2 := (r' \cap r_2) \cap r_1$

4.3 Properties of Intersection

Proposition 1
For all $s, r_1, r_2 \in SC$ such that $s \succ r_1$ and $s \succ r_2$, $r_1 \succ r_1 \cap r_2$ and $r_2 \succ r_1 \cap r_2$

Proof: The properties hold directly in all cases of the definition of $\cap$ by construction, since we always define $r_1 \cap r_2$ as an extension of both s_1 and s_2. The only non trivial case is the transitivity one: if $s \succ r' \succ r_1$ and $s \succ r_2$ then $r_1 \cap r_2$ is defined as $(r' \cap r_2) \cap r_1$. This is well defined, since from $s \succ r'$ and $s \succ r_2$, $r' \cap r_2$ is defined by induction hypothesis, and $r' \succ r' \cap r_2$, $r_2 \succ r' \cap r_2$. Then, again by induction hypothesis, from $r' \succ r_1$, we have that $(r' \cap r_2) \cap r_1$ is defined and $r_1 \succ (r' \cap r_2) \cap r_1$, $r_2 \succ (r' \cap r_2) \cap r_1$. □

Proposition 2
For all $s, r_1, r_2, r_3 \in$ SC such that $s \succ r_1$ and $s \succ r_2$, if $(r_1 \succ r_3 \wedge r_2 \succ r_3)$, then $r_1 \cap r_2 \succ r_3$

Proof: The property holds directly in all cases of the definition of $\cap$, since we always perform the least possible extension to both statecharts. The only non trivial case is the transitivity one: if $s \succ r' \succ r_1$ and $s \succ r_2$, then $r_1 \cap r_2 := (r' \cap r_2) \cap r_1$. Let $r_1 \succ r_3$ and $r_2 \succ r_3$. Then, by $r' \succ r_1$, we have that $r' \succ r_3$, and hence $r' \cap r_2 \succ r_3$ by $r_2 \succ r_3$ and induction hypothesis . Finally, from $r_1 \succ r_3$ and induction hypothesis, we have that $(r' \cap r_2) \cap r_1 \succ r_3$. □

Proposition 3 (Commutativity of $\cap$)
For all $s, r_1, r_2 \in$ SC such that $s \succ r_1$ and $s \succ r_2$, $r_1 \cap r_2 = r_2 \cap r_1$, up to the order of the substates.

Proof: We analyze cases where $\cap$ is defined, is different form $\perp$ and is not commutative by definition: In **ext-and1**, **ext-or1** and **ext-or2** $r_1 \cap r_2$ and $r_2 \cap r_1$ differ only in the order of substates. In **inside-and** and **inside-or** commutativity holds by induction hypothesis. Finally, in **ext-trans** and **ext-act-trans** commutativity trivially holds. □

Proposition 4 (Associativity of $\cap$)
For all $s, r_1, r_2, r_3 \in$ SC such that $s \succ r_1$, $s \succ r_2$ and $s \succ r_3$, $(r_1 \cap r_2) \cap r_3 = r_1 \cap (r_2 \cap r_3)$

The proof is tedious but straightforward, we skip it here for reasons of space.

Computation of the Intersection

Given $r_1, r_2 \in$ SC, both extensions of some $s \in$ SC, the method for computing $r_1 \cap r_2$ is to perform single steps of extensions. The transitivity case for the definition of $\cap$ and the properties stated above ensure confluence, i.e., no matter the order in which the extensions are done, we will obtain a unique "diamond" with s at the top and $r_1 \cap r_2$ at the bottom. In figure 3 we show the process for two statecharts that are extensions of $[\hat{s_1}, ([\hat{s_{11}}]), \emptyset, (\varepsilon, \varepsilon)]$: $r_1 = [\hat{s_1}, ([\hat{s_{11}}], [\hat{s_{12}}]), \{\langle \hat{t_1}, s_{11}, \emptyset, e, \varepsilon, \emptyset, s_{12}, \text{none}\rangle\}, (\varepsilon, \varepsilon)]$ and $r_2 = [\hat{s_1}, ([\hat{s_{11}}], [\hat{s_{13}}]), \{\langle \hat{t_2}, s_{11}, \emptyset, e, \varepsilon, \emptyset, s_{13}, \text{none}\rangle\}, (\varepsilon, \varepsilon)]$.

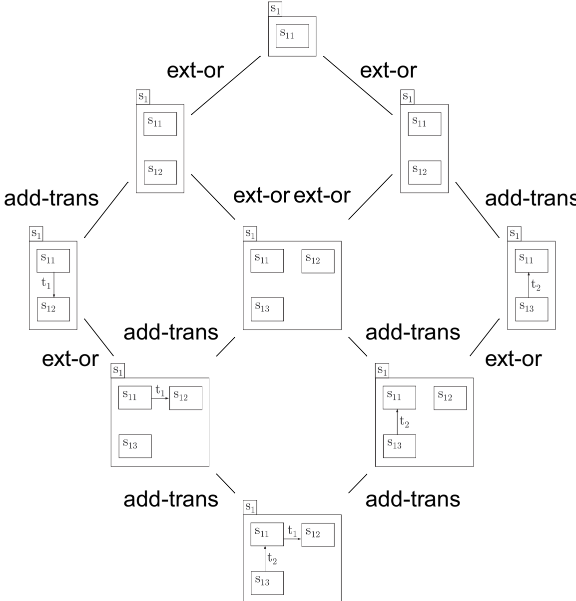


Figure 3. Computation of the Intersection

5 Statecharts with Variabilities

In section 2 we introduced feature diagrams as the means to represent the common and variant functionalities of the products of a software product line. Basically, they consist of sets of features organized under certain hierarchy represented as relations. Given a feature diagram Υ, each set of chosen features determines a particular configuration C, which represents the features present in a particular product of the line. Statecharts are used to describe the behavior of a system. In order to define the behavior of a whole product line, we must describe the effect that each feature has on the products in which it is present. For this, we introduce the set SC* of statecharts with variabilities:

Given a feature diagram $\Upsilon = \langle \mathsf{L}, \mathsf{N}, \mathsf{N_C}, \mathsf{R_M}, \mathsf{R_O}, \mathsf{R_A} \rangle \in \mathsf{FD}$, a SC* for Υ is a function $\Psi: \mathsf{N} \cup \{\mathsf{L}\} \to \mathsf{SC}$ that associates each feature of Υ with a statechart. In order to guarantee that the hierarchy of features represented by the relations $\mathsf{R_M}$, $\mathsf{R_O}$ and $\mathsf{R_A}$ is reflected by the statecharts that implement the features, we further require that: $\forall \langle f, f' \rangle \in \mathsf{chld}_\Upsilon.\ \Psi(f) \succ \Psi(f')$

With this restriction, observe that the image of the set of features fts_Υ under Ψ (i.e., $\Psi(\mathsf{fts}_\Upsilon) \subseteq \mathcal{P}(\mathsf{SC})$) has the same tree structure as the feature diagram Υ, where the parent-child relation between statecharts is the extension relation $\succ$. Then, given a configuration $C = \langle \mathsf{P}, \mathsf{F}, \mathsf{R} \rangle$ of Υ, in order to obtain the statechart that implements all the features present in F, we must just take the intersection of all the statecharts in the image of F under Ψ (i.e., $\Psi(\mathsf{F})$).

By the definition of configuration and observation above, if $\langle f, f' \rangle \in \mathsf{chld}_\Upsilon$, then $\Psi(f) \succ \Psi(f')$ and then (by the definitions given in 4.2), $\Psi(f) \cap \Psi(f') = \Psi(f')$. So, instead of calculating the intersection of all the statecharts in $\Psi(\mathsf{F})$, it is enough to consider the intersection of the leaves of the tree (i.e., of those features in F such that there is no $\langle f, f' \rangle$ in R).

Taking the example of section 2.3, and assuming there exist statecharts $s, s_1, ..s_6$, a possible SC* Ψ for Υ_1 could be defined as $\Psi(P)=s$, $\Psi(f_i)=s_i$ (i=1, ..6), and the following relations must hold between the statecharts: $s \succ s_1, s_2$, $s_2 \succ s_3, s_4$, and $s_3 \succ s_5, s_6$ (see figure 4).

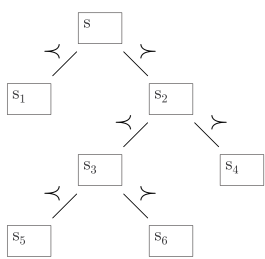


Figure 4. SC* Example

Then, for the configuration C_1=$\mathsf{conf}_{\Upsilon_1}(\{f_2, f_5\})$, we have that $\Psi(C_1)$ is the set $\{s, s_1, s_2, s_3, s_5\}$, with the structure shown in figure 5:

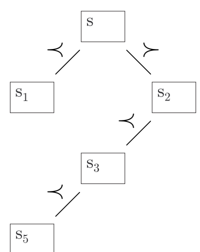


Figure 5. Configuration Example

Then, the statechart that specifies the product corresponding to the configuration C_1 is just $s_1 \cap s_5$

Finally, we introduce a last notion. Recall that the intersection of two statecharts may be undefined. We do not think this is a problem, since some configurations may produce inconsistent behavior of the system, and it is the designer's responsibility to deal with that fact. To consider this possibility, we define the following concept: Given $\Upsilon = \langle \mathsf{L,N,N_C,R_M,R_O,R_A} \rangle \in \mathsf{FD}$, Ψ an SC* for Υ and a configuration $C \in \mathsf{Confs}_\Upsilon$, we say that Ψ *covers* C iff $\bigcap_{f \in \mathsf{fts}_C} \Psi(f) \neq \perp$. We further define: Given $\Upsilon = \langle \mathsf{L,N,N_C,R_M,R_O,R_A} \rangle \in \mathsf{FD}$ and Ψ an SC* for Υ, we say that Ψ *covers* Υ if Ψ covers each possible configuration C of Υ: $\forall C \in \mathsf{Confs}_\Upsilon . \bigcap_{f \in \mathsf{fts}_C} \Psi(f) \neq \perp$

6 Conclusions and further work

We presented the basic ingredients for specifying the behavior of product lines using UML statecharts. We used feature diagrams to represent the common and variant functionalities of a family of products. For this, we presented a formal syntax of feature diagrams and configurations, based on Czarnecki's work [5, 7]. Based on von der Beeck's [12] UML statecharts abstract syntax, we defined an order relation for statecharts to represent when a statechart has a more complex structure than another one. We also defined how to combine different extensions of the same statechart into an integral new statechart. With these notions, given the description of a product line as a feature diagram, we defined a SC* for the line as a function that associates each feature of the feature diagram with a statechart. In this way, we can describe the effect that each feature has on the products in which it is present. The mapping must comply with the hierarchical structure and the feature restrictions, i.e., the more features a product has, the richer the statechart that models it must be. This definition provides a very simple way to obtain the specification of the behavior of any configuration of the product line as the combination of the statecharts that implement all the features present in the product.

This work constitutes the first step in the construction of a tool that allows designers to specify the behavior of product lines using UML statecharts. Such a tool follows a stepwise refinement approach to software design, allowing the user to start form a simple specification of the kernel features of the product line, and to progressively add new features to the line, specifying how each functionality contributes to the behavior of the whole line. The formalism has been tested with some non trivial examples, and it has shown easy to use, since it favours the incremental approach to the specification of the product line. We are working on a statechart editor to be incorporated to existing implementations, such as [1]. This tool will implement the concepts introduced in this paper.

Although there exist several extensions of UML models for specification of variability (among other works, [3, 8, 13]), we have not found any formal specification of variabilities for statecharts. In [2], Batory et al propose a class inheritance approach which progressively refines a base class. They use a composition based approach along with different dimensions to cope with system complexity.

Antkiewicz and Czarnecki present a template-based method to map feature models into UML models [6]. They start with a model which includes all the features and then they restrict that model. We have explored a similar approach in [11], but we found the incremental design more natural. It would be interesting to check whether Antkiewicz and Czarneckis approach can be matched to statecharts.

Finally, concerning formal semantics, von der Beeck in [12] defines a semantics for statecharts in terms of a labeled transition system and Kripke structures, which are very suitable for representing that the output of one transition can be used as the input of another one. We believe that our extension relation $\succ$ represents also semantic refinement with respect to this semantics. The complete statement of this fact and the corresponding proofs is work in progress.

Acknowledgements

We wish to thank María Victoria Cengarle for the initial ideas on which this work is based, and for her thoughtful comments on early versions of this article.

References

[1] M. Antkiewicz and K. Czarnecki. Featureplugin: feature modeling plug-in for eclipse. In *eclipse '04: Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange*, pages 67–72, New York, NY, USA, 2004. ACM.

[2] D. Batory, J. Liu, and J. N. Sarvela. Refinements and multidimensional separation of concerns. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 48–57, New York, NY, USA, 2003. ACM Press.

[3] M. V. Cengarle, P. Graubmann, and S. Wagner. Semantics of uml 2.0 interactions with variabilities. In *International Workshop on Formal Aspects of Component Software (FACS05)*, 2005.

[4] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison Wesley, 2002.

[5] K. Czarnecki. *Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models*. PhD thesis, Technical University of Ilmenau, 1998.

[6] K. Czarnecki and M. Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In *GPCE*, pages 422–437, 2005.

[7] K. Czarnecki, S. Helsen, and U. Eisenecker. Formalizing cardinality-based feature models and their staged configuration. *University of Waterloo*, 2004.

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

[9] D. Harel. Statecharts: A visual formalism for complex systems. *North-Holland*, 1987.

[10] OMG. Unified modeling language specification version 2.0. Technical report, Object Management Group, 2004.

[11] N. Szasz, C. Luna, and A. González. Hacia una formalización de líneas de productos mediante máquinas de estados con variabilidades. In *XXXIII Conferencia Latinoamericana de Informática, CLEI 2007*.

[12] M. von der Beeck. A structured operational semantics for uml-statecharts. *Springer*, 2002.

[13] T. Ziadi, L. Helouet, and J.-M. Jezequel. Behaviors generation from product lines requirements. In *UML2004 Workshop on Software Architecture Description and UML*, 2004.

Reflective Component-based Technologies to Support Dynamic Variability*

Nelly Bencomo, Gordon Blair, Carlos Flores, Pete Sawyer

Computing department, InfoLab21, Lancaster University, LA1 4WA, United Kingdom
email: {nelly, gordon, floresco, sawyer} @comp.lancs.ac.uk

Abstract

In this paper we propose an approach to support dynamic or runtime variability in systems that must adapt dynamically to changing runtime context. The approach is founded on reflective component-based technologies to support the dynamic variability at the architectural level. Adaptive behaviour is encoded in reconfiguration policies that are consulted at run-time when changes in the underlying environment are detected. Specifically, the reconfiguration policies dictate the component-based architecture to be used in actively changing contexts. However, the increasing number of variants and their interdependency relationships add to the complexity of variability management. Therefore, the paper also proposes a notation and associated models to address the management of dynamic variability. We describe our experience with applying this approach through a case study; the support and management of dynamic variability for service discovery protocols. ***Keywords**: dynamic variability, architectural reconfiguration, orthogonal variability models.*

1 Introduction

It is becoming common that systems should be able to adapt dynamically to changing contexts at runtime. Such systems exhibit degrees of variability that depend on runtime fluctuations in their contexts. We call this kind of variability *dynamic variability* or runtime variability. Although dynamic variability has been addressed by long-established concepts in the field of system families [16, 17, 34], we advocate that this work is insufficient to meet the needs of contemporary, dynamically adaptive distributed systems. In the case of systems that adapt dynamically, approaches to support variability cannot be just component specializations or conditions on variables [34]. Decisions should involve more powerful mechanisms that allow dynamic reorganization of the architecture. In other words, these mechanisms should be able to manage whole sets of components, their connections and associated semantics. In this paper we introduce an approach based on component frameworks [35] and reflection [23] as a mechanism to support the realization of dynamic variability of adaptive systems. Using this approach, unanticipated variability and interdependency relationships between variable software artefacts (such as components, connections, and components configurations) and context and environment conditions can grow to such a level that rigorous support for variability management is needed. Therefore, this paper also discusses notation and models to manage dynamic variability. The application of the proposed approach consider using a case study.

In the reminder of the paper we discuss dynamic variability in adaptive systems and the need for its management (Section 2). We then introduce the fundamental concepts of our approach (Section 3) and present a case study (Section 4). A discussion about the contributions of our research and related work are presented (Section 5), and finally, some remarks and future work are given (Section 6).

*This work has been supported in part by the EPSRC project EP/C010345/1 The Divergent Grid

2 Dynamic Variability

2.1 Overview

One of the reasons for software variability is to delay a design decision [34]. Instead of deciding on what system to develop in advance, a set of components and a common system family (reference architecture) are specified and implemented during a process called *Domain Engineering* [12]. Later on, during *Application Engineering*, specific systems are developed to satisfy the requirements and reusing the components and architecture. Variability is expressed in the form of *variation points*. A variation point denotes a particular location in a software-based system where decisions are made to express the selected *variant* [34]. Eventually, one of the variants should be chosen to be achieved or implemented. The time when it is done is called *binding time*.

Traditionally, decisions have been deferred to architecture design, implementation, compilation, linking, and deployment [1, 7, 12, 22, 26, 34]. Currently the aim is to postpone these decisions to even later points in time to allow dynamic variability at runtime. This raises several research challenges, such as the management of variabilities in dynamically adaptive systems, which are discussed in the next section.

2.2 Dynamic Variability in Adaptive Systems

A dynamically adaptive system operates in environments that impose changing contexts and requirements. The challenge that it causes comes from the need to support unanticipated adaptation or customization of the systems [32] according the needs of the fluctuating environment. The unanticipated conditions are related to:

(i) Environment or context variability: commonly the evolution of the environment cannot be predicted at design time; therefore the total range of contexts and requirements may be unknown at design time.

(ii) Structural variability: it covers the variety of the components and the variety of their configuration. This is consequence of the variability explained above. In order to satisfy the set of requirements for the new context, the system may add new components or arrange the current structural configuration (reconfiguration). Hence, the solutions cannot be restricted to a set of known-in-advance configurations and components.

The system should be prepared to deal with these two dimensions of variability described above. Adaptive systems must be prepared to identify a new context unknown at design time. Under the conditions of the new contexts, the system must be prepared to discover and include new components to meet new requirements or simply to improve the current state of the system when new components become available [32] and according some quality of service (QoS) properties. Moreover, solutions to manage the latter structural variability cannot be just the traditional component replacements and/or specializations, but decisions should involve more powerful mechanisms able to manage whole sets of components, their connections and semantics (configurations). A balance between the support for unanticipated adaptive capabilities and the guarantee of the correct structural composition and state of the system is another important challenge that must be taken into account.

The classification proposed in [36] distinguishes two different types of dynamic adaptation: *dynamic behaviour adaptation* and *dynamic reconfiguration*.

In dynamic behaviour adaptation, systems recognize new environmental conditions not envisioned during development. In this systems, control and order is emergent rather than predetermined [13, 37]. This kind of adaptation is proposed by researchers using mechanisms based on genetic algorithms or neural networks. Research on this kind of adaptation is still at an early stage to propose sound solutions for complex adaptive systems.

Dynamic reconfiguration requires that all feasible variants of behaviour can be somehow predefined before execution. During execution, the current state of the system and its environment and context is evaluated and the most appropriate behaviour variant is selected; i.e. the system is dynamically reconfigured using the most appropriate variant (configuration). Dynamic reconfiguration can be realized using two approaches: *software-based configuration* and *hardware-based configuration*. The latter is omitted in this research as the authors are concerned only with software-based reconfiguration.

Software-based reconfiguration can be achieved using two approaches: *pre-determined reconfiguration* and *online-determined reconfiguration*. *Pre-determined reconfiguration* is based on a set of configurations with known impact defined before the deployment of the application. In this case, the system only supports the configurations that are hard coded and fixed in advance by the developers. Therefore the system is only reconfigured (i.e. the system adapts) when in a predefined and hardcoded configuration. The implementation of a new configuration requires the system to be reinitiated. This case is very restrictive. The last case, *online-determined reconfiguration*, is a solution in-between pre-determined reconfiguration and dynamic behaviour adaptation. With online-determined reconfiguration, the system has a mechanism to identify the possible configurations at runtime. Online-determined reconfiguration supports dynamic extension of the application by adding, changing and removing artefacts (e.g. components and connections) at runtime. This approach "*requires a complex reconfiguration framework*" [36].

3 Achieving Dynamic Variability: our approach

To address the challenges posed by dynamic variability explained above, we propose the use of component frameworks and reflection as a flexible mechanism for supporting runtime variability. At Lancaster University, we have gained experience developing adaptive systems and middleware platforms using *component frameworks* and *reflective technologies* [8–10]. Component frameworks are collections of components that address a specific area of concern and accept "plug-in" components that add or extend behaviour [3, 9]. Reflective capabilities support introspection to observe and reason about the state of the system to make decisions on architectural reconfigurations. Adaptive behavior is defined by sets of *reconfiguration policies*. These

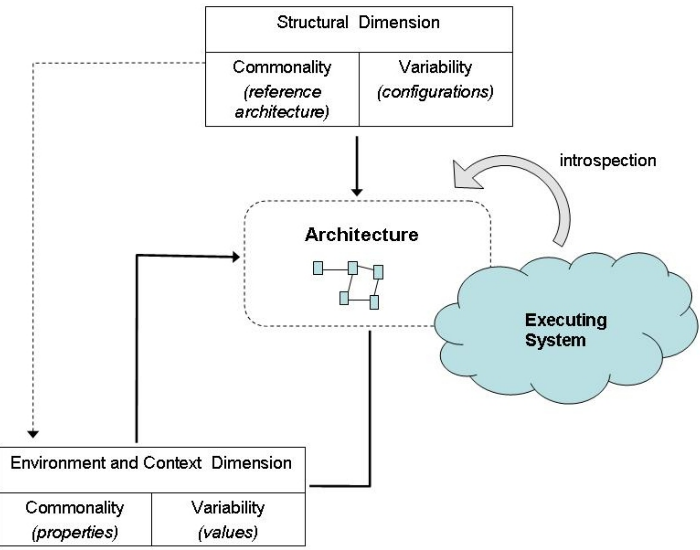


Figure 1. Dynamic Variability Dimensions

policies are of the form *on-event-do-actions* and *actions* are architectural changes using the component frameworks. A *context engine* receives relevant *environmental events* that are employed to identify the reconfiguration policy to be used. Crucially, component frameworks offer the medium to provide structural variability. Reflective capabilities offer the potential to reason about the possible variation points and their variants during execution. The proposed solution use the online-determined reconfiguration category of dynamic adaptation explained above.

Dynamic Variability: Dimensions

The approach deal with the two dimensions of dynamic variability identified, see Figure 1. The architecture defined by the component framework (reference architecture) basically describes the structural commonalities. Different configurations or *structural variants* will exist that follow the well-defined constraints imposed by the component frameworks. Policies describing the contexts and requirements will drive the evolution and execution (using reconfigurations). Essentially, the policy mechanism will set the basis for dealing with the *environment and context variability* identified above. The approach separates the application-specific functionality from the adaptation concerns, thereby reducing complexity [27].

Dynamic Variability: Models

The increasing number of variants and their relationships can make the management of variability a challenge. We propose a model-driven approach to manage the two dimensions of dynamic variability. A model of the structural variability specifies the architecture of the system that will evolve over time during the execution. A model of the environment and context variability specifies the conditions and events that will trigger changes in the architecture. Crucially, these models can be constructed using the domain-specific language-based tool called Genie [2, 4]. From the models designed using Genie, generation of different software artefacts including component source code, component framework configurations, and the reconfiguration policies [31] can be performed. The next section illustrates the case study introducing the fundamental concepts of the proposed approach.

4 Case Study: Dynamic Service Discovery

This section introduces an example of how our approach supports runtime variability for adaptive systems. Firstly, we present the motivation for dynamic service discovery and discuss the domain problem. We follow with the description of how commonalities and variants are identified and finally the variability model and its notation and application are described. The case study is in the context of mobile computing environments applications which need to

dynamically discover services from a wide range of options. A service discovery application proposes a good example of kind of systems that need support for runtime variability.

4.1 The Need to Dynamically Discover Services

Nowadays mobile computing is pervasively taking over traditional computing [14, 19, 25]. Mobile devices are characterized by sudden and unexpected changes in execution context. Applications running on these devices need to dynamically adapt according to the changing contexts. If devices (PDA, mobile, or laptop) are capable of detecting changes in the current environment, then they can also notify the user about new available services according to predefined preferences (e.g. comparison prices and categorized sales in a supermarket or printing services in an internet cafe). Service discovery protocols (SDP) were conceived to simplify the discovery and use of network resources such as printers, video cameras, directories, and mail servers, with minimum user intervention. Many different approaches to tackle different challenges related to heterogeneity of technology have led to a variety of proposed designs for SDPs [24]. Consequently it is not possible to completely foresee at design time which protocols will be used to advertise services in a given context or environment. The next section presents a solution to overcome the challenges posed by heterogenous service discovery protocols.

4.2 Family of Service Discovery Protocols for Adaptive Systems

Flores et all [15] present a configurable and reconfigurable middleware solution for the dynamic discovery of services advertised using heterogenous protocols in diverse environments. The solution takes into consideration a set of common core architectural elements that individual discovery protocols follow. Using the final architecture, individual discovery platforms can be implemented and dynamically plugged-in to the discovery middleware. Hence, different SDP personalities can be used to discover services advertised by heterogeneous platforms. This middleware solution has been evaluated with the development of four existing ad-hoc service discovery protocols: *ALLIA*, *GSD*, *SSD*, and *SLP* (i.e. 4 personalities). The offered solution enhances configurability and re-configurability and minimizes resource usage through reusable assets such as components and patternsof interaction [15].

Service Discovery Agents

A service discovery interaction platform uses three kinds of agents to advertise and discover services:

-**User Agent (*UA*)** to discover services on behalf of clients,

-**Service Agent (*SA*)** to advertise services, and,

-**Directory Agent (*DA*)** to support a service directory where *SA*s register their services and *UA*s send their service requests. A *DA* stores temporal service advertisements, matches requested services against advertisements, and replies to requesting clients when a positive match is found.

The agents identified above can be seen as *roles* that individual protocols assume. Depending on the required functionality, participating nodes using a given protocol personality might be required to support 1,2, or the 3 roles at any time.

Service Discovery Family Architecture

The architecture is shown in Figure 2. The six components of the architecture are detailed below:

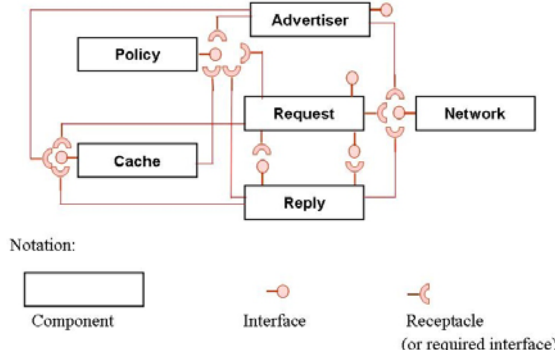


Figure 2. The Service Discovery Family Architecture

***-Advertiser Component*:** this component is utilized by *SA*s to advertise its services and by *DA*s to process incoming service advertisements storing them in cache. This component also deals with protocol messages related with the maintenance of a directory overlay network.

***-Request Component*:** this component is utilized by *UA*'s to generate service requestes. It is also employed by *DA*s to process incoming service requests, match them against local services previously stored in a cache. It can also forward request messages in both roles.

***-Reply Component*:** this component is used by both *UA*s and *DA*s to generate service replies when a positive match request-service occurs or to notify applications from a received replied request respectively.

***-Cache Component*:** common tasks performed by this component are the management of temporary data, storage

of received service advertisements, description of local services and location of neighbouring directories.

***-Policy Component*:** this component stores and deals with user preferences, application needs and/or inclusive context requirements.

***-Network Component*:** this component allows components connected to it to transmit and receive messages utilizing different routing schemes.

4.3 Commonalities and Variabilities

The common architecture explained above dictates the rules to be followed by the possible variants (i.e. configurations). In our specific case, any SDP personality in any environment and under any context needs the network component to interface with networks services or clients, and policies and cache components are always required since they interact with either discovery role. Therefore, the *Network*, *Cache* and *Policies* components will always be present in any valid configuration. The other three components and their bindings will be part of the configuration or not, depending on the roles the protocol might perform (i.e. *SA*, *UA*, or *DA*). Hence, roles (agents) directly define the structural variants.

Figures 3(a) and 3(b) show how the architecture can be configured to support either a *UA* or *SA* role by restricting the number of components to only those required to provide the determined functionality. By using a complete framework configuration, a *DA* can also be supported and the configuration to be used is shown in Figure 3(c). Hence, by configuring individual protocols according to the role (i.e. *UA*, *SA* or *DA*), the number of resources required by a multi-personality middleware service discovery can be significantly reduced to improve the footprint and potentially the performance of the system [15]. Notably, with the multiprotocol middleware platform, heterogeneous discovery platforms can be implemented with a common component architecture. This simplifies the configuration process since the component types and connection bindings remain the same for any protocol implementation. Thus, because of the common configuration pattern, the execution of a simple single component replacement algorithm is enough to re-configure the architecture. Similar common algorithms are required to perform a coarse-grained re-configuration when loading a new discovery personality or when changing the role in a given personality is required. Fine-grained and coarse-grained changes can be made in the framework to support context changes in the environment. Individual protocols can be changed in a fine-grained manner to, for instance, replace the network component with a new one to support a different routing scheme.

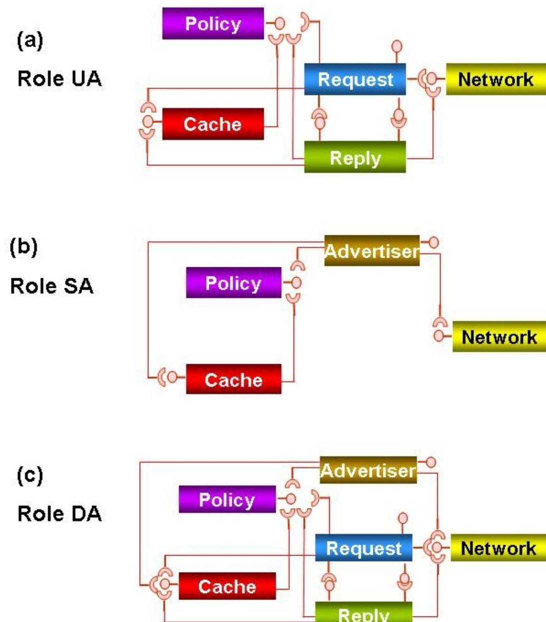


Figure 3. Configurations for the Different Variants

4.4 Variability Model

The approach presented above notably enhances reconfigurability. However, the increasing number of variants and their relationships make it crucial the structured management of variability. This section shows the notation and models we propose to address variability management.

4.4.1 Modelling Structural Variants

The model is based on the orthogonal variability modeling approach proposed in [28]. An orthogonal variability model defines the variability of a system family, i.e the variability information is in a separate model in the form of variation points (VPs) and variants. It associates the VPs and variants defined with other software development models such as design models or component models. An orthogonal variability modelling approach offers many advantages like (i) it promotes a good separation of concerns as the orthogonal models provide a cross-sectional view of the variability across other development artefacts (using the relationship 'artefact dependency'), (ii) in our specific approach, it proposes the basis for the management of traceability between the runtime variability models, the implementation models and the requirement models [31].

Figure 4 shows the model for the role variants explained

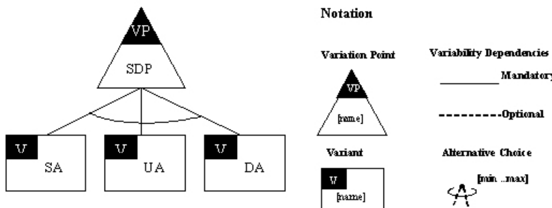


Figure 4. Variability Model

above. Essentially, different configurations associated with roles correspond to variants. The variability diagram describes the variation point "SDP" (Service Discovery Protocol) with three (structural) variants *SA*, *UA*, and *DA*. These variation points and variants associated correspond to the management of the *Structural Variability* described in 2.2

Figure 5 shows the use of the relationship 'artefact dependency', each structural variant is associated with the corresponding configuration. The configurations are represented using either XML or binary files to describe or perform their topologies. The specified variation points are specialized by a runtime selection between alternative component configurations. These configurations are designed using the domain-specific language (DSL) tool Genie to automatically generate the corresponding XML file or hard code as explained in [2].

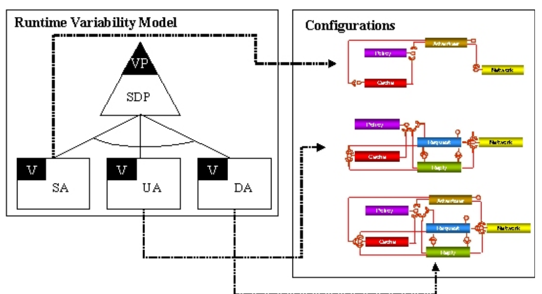


Figure 5. Variants SA, UA, and DA and their configurations

Other VPs and variants are also specified; for example the VP "Personality" defines the personality variants, i.e. *ALLIA*, *GSD*, *SSD*, *SLP* or any other specified in the future (see Figure 6).

4.4.2 Modelling Reconfigurations

An interesting aspect of adaptive systems like SDPs is the need to dynamically reconfigure the system from one variant to another when the context has changed. Examples of opportunities for reconfigurations (i) changes of the underlying network protocols if the operation changes to an ad-hoc domain, (ii) the use of a different role strategy if the system size increases (for scalability reasons), (iii) the use of a different role strategy to save energy (this example is explained below). This is quite distinct from traditional system families where, once a member (product) of the family is created, it does not change significantly during the lifetime of the software product. Using our approach a member of the family may be "transformed" into another one to adapt the system to meet the new requirements and suit a new context. To do this, the system should monitor specific aspects of the runtime environment and react to given changes while keeping a valid state. The system should be able to decide what kind of reconfiguration has to be performed if any. To model this behaviour it is necessary to define what adaptation means in terms of configurations.

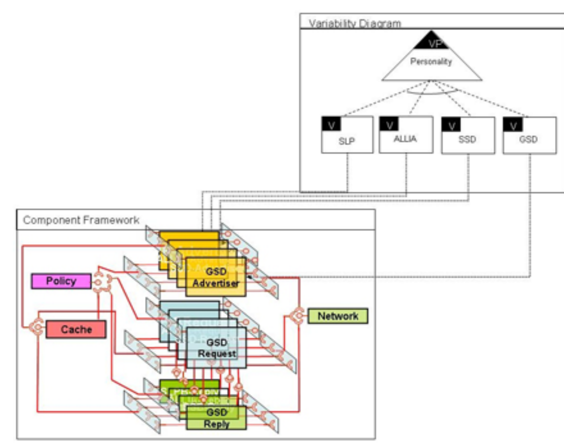


Figure 6. Variants Personalities

An *adaptation* is defined as the process of having the system going from a given configuration *Ci* to another configuration *Cj* given the conditions of the context *Tk*. The possible adaptations will be captured by the variability model.

The variability model of the case study is extended to show how an adaptation from one role (agent) to another is performed, i.e. it shows the context and requirements and the reconfiguration involved. Essentially, this is modeled using *transition diagrams*. A screenshot of the Genie model that specifies the transition diagram designed for service discovery protocols is at the bottom of Figure 7. An adaptation policy is associated with the relationship (arc) between the configuration for the variant *UA* (*Ci*) and the configuration for *DA* (*Cj*) for a given context *Tk* specified by the policy. The number of transitions (arcs) and adaptation policies to be inserted in the transition diagram will depend on how adaptable the system should be or is conceived. The transition diagrams and the policies associated correspond

to the management of the *Environment and Context Variability* described in Section 2.2.

Figure 8 shows an overview of the two dimensions of dynamic variability for the case of Service Discovery Protocols.

The following examples illustrate reconfiguration opportunities identified for the case of the protocol personality *SSD*.

Example 1: Nodes operating *SSD* protocols might run periodically consensus algorithms to reelect the *DA* nodes in charge of giving directory services to other nodes. Therefore, if a node *UA* has been chosen as a *DA*, this node should be reconfigured to match its new role. A pseudo code of the reconfiguration policy that guides the adaptation of the example is as follows:

```
if ( Elected-DA ) then
    reconfigure(UA,DA)
end
```

The reconfiguration policy (expressed in XML) associated with this reconfiguration is shown in Figure 7. Actually, the XML files of policies can be generated using the tool Genie.

Example 2: If a node *DA* has low battery and it was originally a node with the role *SA*, the node should be reconfigured to its original *SA* configuration. The same could happen if after the consensus algorithms to reelect the *DA* nodes another node is elected. The policy is as follows:

```
if(!Elected-DA ||(Low-Battery && RSA))then
    reconfigure(DA,UA)
end
```

The case study we used has necessarily been a simple one for reasons of space. In this sense, the transition diagram of the case study explained above just covers aspects associated with service discovery concerns using the multi-protocol component framework. However, aspects associated with networking issues can also be considered. In this case, two component frameworks would be associated with each structural variant in the transition diagram: the *Service Discovery* and the *Network* component frameworks, and the triggers specified in the arcs of the transition diagrams should also include properties and conditions associated with networking issues. Examples of situations that would be taken into account are (i) in a mobile application it is possible that a new network would come within range using a different technology, and, therefore, it may be necessary to reconfigure the *Network* component framework to satisfy the new network, or (ii) the battery life is in threat and, therefore, a *BlueTooth*-based networking might be preferred instead of a relatively power-hungry *WiFi*-based connection.

Other component frameworks that can be used in the specification of the structural variants of the transition diagrams are the *Spanning Tree* component framework (for the description of the topology of nodes in a network), the *Interaction* component framework (to choose between the different interaction types, e.g. publish-subscribe, group communication, peer-to-peer, data sharing and others), the *Security* and *Resource Management* frameworks. The inclusion of these component frameworks in the definition of the structural variants depends on the nature and concerns of the application to be developed.

Our approach has also been applied in the case of a real-world scenario: a wireless sensor flood forecasting application deployed on the River Ribble in the north west of England [21]. This case study includes adaptation concerns associated with the reconfiguration of the topology of the sensor networks and networking concerns using the *Spanning Tree* and *Network* component frameworks respectively. Some preliminary results are shown in [5, 18, 31].

5 Discussion: Contributions and Related Work

This section discusses the novel contributions of our research and contrasts the proposed approach with related work. Current approaches of system families (or product lines) base their support for variability on the configuration knowledge which is expressed explicitly when synthesizing a product (variant). This is enough in situations when the configuration is done statically. Traditionally, variability is meant to be solved at a predelivery moment [20]. In our case, the problem domain is in the field of customization of systems at runtime that noticeably takes place postdelivery. In this new field and as explained above not all the structural elements (such as components, configurations) or requirements are known at design time. It is not flexible enough to offer a fix set of variation points (hot spots) where different versions of components are replaced. With our approach sets of configurations are replaced in answer to context changes following the reconfiguration policies. Further more, new reconfiguration policies can be added at runtime changing dynamically the behaviour of the system. These policies are explicitly modelled in our approach what potentially improves the traceability during the software development process.

The proposed approach focuses on the structured managements of variation points that are bound at runtime. The dependencies between structural variability (architectural elements) and environment and context variability are made explicit. In a nutshell, the approach focuses on some of the (runtime) variability issues stated in [11, 16], such as no first-class representation of the concept of variability points, implicit dependencies, inflexible binding mechanisms, high

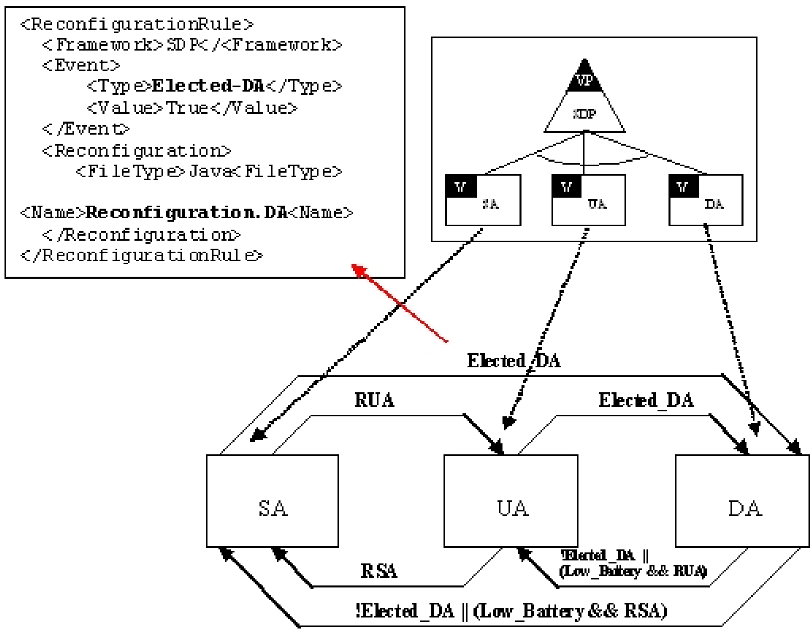


Figure 7. Reconfiguration Graph for the Variants SA, UA, and DA

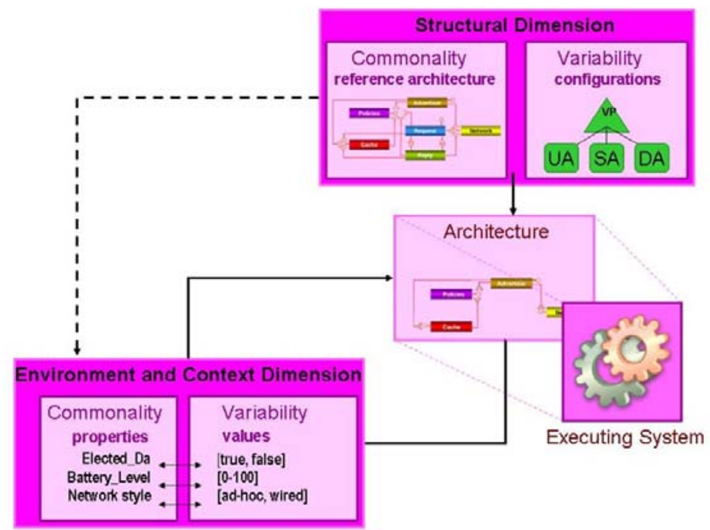


Figure 8. Dynamic Variability in the Service Discovery Protocols

resource costs, predictability, and addition of variants.

Many mechanisms for runtime variability management have been proposed. They are mainly focused on exchange of runtime entities, parametrization, inheritance for specialization, and preprocessor directives [16, 17, 29, 34]. Our approach proposes the management of whole sets of components, their connections and semantics (i.e. a more coarse grained approach). However, our approach is still

complementary to the finer grain styles cited above or in [38]. For example in each configuration traditional fine grain management of variability can be used to describe specific component replacements or specializations. Research work on MADAM [14] shares some of the principles of our approach as component frameworks to support variability. They also take into account the benefits of coarse-grained variability mechanisms. However our approach is more general as their focus is only on mobile computing applications. A similar research is found in [32]. They introduce the concept of composable components which is similar to our component frameworks. They apply recursive composition according to external requirements using ADLs what can be to some extent equivalent to our reconfiguration policies. However, they do not offer reflection capabilities, i.e their systems cannot reason about the current state or configuration of the system. Reflection offers support to determine where the points for variation are, what are the possible set of variations, or the state of the system at any point in time. However, using reflection has some drawbacks as the effect on performance and integrity issues. When developing reflective systems a trade-off between flexibility and performance has to be studied and a rigorous system development has to be performed.

In [18, 30, 31] we explain how the policy mechanisms contribute to providing a clear trace from user requirements to adaptation requirements [6] and their implementations. In this sense, the research related to requirements-driven composition in [32] is similar to our research.

6 Conclusions and Future Work

In this paper, we address how to manage effectively and in a structured way variation points that have to be bound at runtime. We focus on the development of systems that adapt to fluctuating environments following the established principles of systems families. The central elements of our approach are the concepts of component frameworks and reflection. The architecture offered by the component frameworks defines the invariant crucial for the reuse of common assets. The modeling approach proposed focusing on the explicit identification and documentation of the dynamic variability of the family.

A strong point of our approach is that it proposes a solution for the problem of unanticipated configurations and decisions that depend on the runtime context. It allows the specification of new reconfiguration policies, to discover and use new components and to vary the structural configuration (reconfiguration) of the system. Component frameworks and its specification, as presented in this paper, proposes the necessary flexibility, while offering formality to get the expected behaviour.

Substantial research remains to be done. For example, a concern is the combinatorial explosion related to the number of reconfiguration paths in the transition diagrams (i.e. the number of policy-based reconfigurations). However in the case study the number of reconfiguration paths is manageable, it might not be the case for other domains. We think that the combination of the specificity of *on-event-do-action* policies and higher-level policies that focus on general properties of the system can mitigate the problem.

Another concern is tool support for modeling variability and its integration into our Genie toolkit [33]. We have already some partial results shown in [5]. Tool support will help the scalability of the approach.

Acknowledgments We thank Paul Grace for his discussions on the above material.

References

[1] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355 – 398, 1992.

[2] N. Bencomo and G. Blair. Genie: a domain-specific modeling tool for the generation of adaptive and reflective middleware families. In *6th OOPSLA Workshop on Domain-Specific Modeling*, Portland, 2006.

[3] N. Bencomo, G. Blair, G. Coulson, and T. Batista. Towards a metamodelling approach to configurable middleware, 2005.

[4] N. Bencomo, P. Grace, and G. Blair. Models, runtime reflective mechanisms and family-based systems to support adaptation. In *Workshop on MOdel Driven Development for Middleware (MODDM)*, 2006.

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

[6] D. Berry, B. Cheng, and P. J. Zhang. The four levels of requirements engineering for and in dynamic adaptive systems. In *11th International Workshop on Requirements Engineering: Foundation for Software Quality (REFSQ'05)*, Porto, Portugal, 2005.

[7] D. Beuche, H. Papajewski, and W. Schröder-Preikschat. Variability management with feature models. *Science of Computer Programming. Special issue: Software variability management*, 53(3):333–352, 2004.

[8] G. Blair, G. Coulson, A. Andersen, L. Blair, M. Clarke, F. Costa, H. Duran-Limon, T. Fitzpatrick, L. Johnston, R. Moreira, N. Parlavantzas, and K. Saikoski. The design and implementation of open orb 2. *IEEE Distributed Systems Online*, 2(6), 2001.

[9] G. Blair, G. Coulson, and P. Grace. Research directions in reflective middleware: the lancaster experience. In *3rd Workshop on Reflective and Adaptive Middleware*, pages 262–267, 2004.

[10] G. Blair, G. Coulson, J. Ueyama, K. Lee, and A. Joolia. Opencom v2: A component model for building systems soft-

ware. In *IASTED Software Engineering and Applications*, USA, 2004.

[11] J. Bosch, G. Florijn, D. Greefhorst, J. Kuusela, H. Obink, and K. Pohl. Variability issues in software product lines. In *4th International Workshop Software Product Family Engineering*, Bilbao, Spain, 2001.

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

[13] K. Dooley. Complex adaptive systems : A nominal definition. 1997.

[14] J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjorven. Using architecture models for runtime adaptability. *Software IEEE*, 23(2):62–70, 2006.

[15] C. A. Flores-Cortés, G. S. Blair, and P. Grace. An adaptive middleware to overcome service discovery heterogeneity in mobile ad-hoc environments. *IEEE Distributed Systems Online*, 2007.

[16] M. Goedicke, C. Köllmann, and U. Zdun. Designing runtime variation points in product line architectures: three cases. *Science of Computer Programming Special Issue: Software variability management*, 53(3):353 – 380, 2004.

[17] M. Goedicke, K. Pohl, and U. Zdun. Domain-specific runtime variability in product line architectures. In *8th International Conference on Object-Oriented. Information Systems*, pages 384 – 396, 2002.

[18] H. J. Goldsby, P. Sawyer, N. Bencomo, D. Hughes, and B. H. Cheng. Goal-based modeling of dynamically adaptive system requirements. In *15th Annual IEEE International Conference on the Engineering of Computer Based Systems (ECBS)*, 2008.

[19] P. Grace, G. Blair, and S. Samuel. A reflective framework for discovery and interaction in heterogeneous mobile environments. *ACM SIGMOBILE Mobile Computing and Communications Review*, 9(1):2–14, 2005.

[20] J. V. Gurp, J. Bosch, and M. Svahnberg. On the notion of variability in software product lines. In *Working IEEE/IFIP Conference on Software Architecture (WISCA'01)*, page 45, 2001.

[21] D. Hughes, P. Greenwood, G. Coulson, G. Blair, F. Pappenberger, P. Smith, and K. Beven. Gridstix:: Supporting flood prediction using embedded hardware and next generation grid middleware. In *4th International Workshop on Mobile Distributed Computing (MDC'06)*, Niagara Falls, USA, 2006.

[22] J. Lee and D. Muthig. Feature-oriented variability management in product line engineering. *Communications of the ACM*, 49(12), 2006.

[23] P. Maes. *Computional reflection*. PhD thesis, Vrije Universiteit, 1987.

[24] R. Marin-Perianu, P. Hartel, and H. Scholten. A classification of service discovery protocols. Technical Report TR-CTIT-05-25, University of Twente, 2005.

[25] C. Mascolo, L. Capra, and E. Wolfgang. Mobile computing middleware. *LNCS 2597*, pages 20–58, 2002.

[26] R. v. Ommering. *Building Product Populations with Software Components. PhD Thesis*. PhD thesis, Rijksuniversiteits Groningen, 2004.

[27] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems and Their Applications*, 14(3):54–62, 1999.

[28] K. Pohl, G. Böckle, and F. v. d. Linden. *Software Product Line Engineering- Foundations, Principles, and Techniques*. Springer, 2005.

[29] E. Posnak and G. Lavender. An adaptive framework for developing multimedia. *Communications ACM*, 40(10):43–47, 1997.

[30] P. Sawyer, N. Bencomo, P. Grace, and G. Blair. Handling multiple levels of requirements for middleware-supported adaptive systems. Technical Report COMP 001-2007, Lancaster University, 2007.

[31] P. Sawyer, N. Bencomo, P. Hughes, Danny andl Grace, H. J. Goldsby, and B. H. C. Cheng. Visualizing the analysis of dynamically adaptive systems using i* and dsls. In *REV'07: Second International Workshop on Requirements Engineering Visualization*, Delhi, India, 2007.

[32] I. Sora, V. Cretu, P. Verbaeten, and Y. Berbers. Managing variability of self-customizable systems through composable components. *Software Process: Improvement and Practice*, 10(1):77–95, 2005.

[33] SSE. Varmod-prime tool-environment, university of duisburg-esse. http://www.sse.uni-essen.de/wms/en/index.php?go=256, 2006.

[34] M. Svahnberg, J. v. Gurp, and J. Bosch. A taxonomy of variability realization techniques. *Software: Practice and Experience*, 35(8):705 – 754, 2005.

[35] C. Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley / ACM Press, 2002.

[36] M. Trapp. *Modeling the Adaptation Behavior of Adaptive Embedded Systems. PhD Thesis*. PhD thesis, University of Kaiserslautern, 2005.

[37] M. Waldrop. *Complexity: The Emerging Science at the Edge of Chaos*. New York: Simon and Schuster, 1992.

[38] J. Zhang and B. H. Cheng. Model-based development of dynamically adaptive software. In *International Conference on Software Engineering (ICSE'06)*, China, 2006.

Towards Visualisation and Analysis of Runtime Variability in Execution Time of Business Information Systems based on Product Lines*

Ildefonso Montero, Joaquín Peña, Antonio Ruiz-Cortés
Departamento de Lenguajes y Sistemas Informáticos
Av. Reina Mercedes s/n, 41012 Seville (Spain)
University of Seville
{monteroperez, joaquinp, aruiz}@us.es

Abstract

There is a set of techniques that build Business Information Systems (BIS) deploying business processes of the company directly on a process engine. Business processes of companies are continuously changing in order to adapt to changes in the environment. This kind of variability appears at runtime, when a business subprocess is enabled or disabled. To the best of our knowledge, there exists only one approach able to represent properly runtime variability of BIS using Software Product Lines (SPL), namely, Product Evolution Model (PEM). This approach manages the variability by means of a SPL where each product represents a possible evolution of the system. However, although this approach is quite valuable, it does not provide process engineers with the proper support for improving the processes by visualising and analysing execution-time (non-design) properties taking advantage of the benefits provided by the use of SPL.

In this paper, we present our first steps towards solving this problem. The contribution of this paper is twofold: on the one hand, we provide a visualisation dashboard for execution-traces based on the use of UML 2.0 timing diagrams, that uses the PEM approach; on the other hand, we provide a conceptual framework that shows a roadmap of the future research needed for analysing execution-time properties of this kind of systems. Thus, due the use of SPL, our approach opens the possibility for evaluating specific conditions and properties of a business process that current approaches do not cover.

*This work has been partially supported by the European Commission (FEDER) and Spanish Government under CICYT project Web-Factories (TIN2006-00472) and under a scholarship from the Education and Universities Spanish Government Secretariat given to the author Ildefonso Montero.

1 Introduction

The development of *Business Information Systems* (BIS) is focused on providing techniques and mechanisms for designing software systems based on the business processes of the companies. One of the implementation approaches is based on deploying business processes, defined graphically, on process engines that execute the specification.

Variability in average-size business processes is high enough for motivating the use of tailored mechanisms to be managed. For that purpose, there exists only one approach devoted to manage business processes variability using *Software Product Lines* (SPL)[11]. In this approach, A. Schnieders *et al.* explore the idea of applying *Software Product Lines* (SPL) for managing runtime variability of an unique BIS in an approach called *Process Family Engineering* (PFE) [11]. In PFE, each *product* represents an *evolution* of the system (at runtime). In this context, term *product* is defined as a set of features that are enabled/running at a certain moment, and the term *evolution* is defined as a transition from one product to another. See Section 2 for a detailed definition of these concepts.

However, although PFE may be the solution to manage the evolution of the business process of a company, the proposed models, namely feature models (equivalence between *feature* and *business process* is defined in Section 2), are not expressive enough for documenting this evolution. The main problem is that this kind of models are devoted to model static variability, and not runtime variability [9]. See Section 3.2 for details on this problem.

The *Product Evolution Model* (PEM) [8], which is shown in Section 6, complements PFE for representing runtime variability in BIS properly. For that purpose, it integrates PFE with several proposals for modeling runtime variability in SPL, namely [5][6][7] (see Section 6 for a discussion of these approaches). This approach is oriented to provide a set of artifacts able to represent properly runtime

variability at design time and trigger events that drives these changes. PEMs are defined in two layers: (i) an abstract formal description of business evolutions, presented in Section 3.1 and (ii) a proposal for representing it based on a state-based notation where each state represents a product and each evolution between two or more states is represented by means of an inclusion or exclusion of features, presented in Section 3.2. PEM uses the *Business Process Model Notation* (BPMN) [3] for representing this, but the proposal is open to other notations. The main benefits of this approach are that it provides sufficient expressiveness for representing runtime variability in BIS, and events or conditions that fire business evolutions can be represented.

Although this approach presents a valuable solution for representing runtime variability in BIS at design time, process engineers need to visualise and analyse properties of the business at execution-time, for example: how long each product is active or which is the percentage of benefits obtained in each product at a certain moment. This kind of evaluation is studied in the *Business Process Mining* field [4][13][10]. However there not exists any approach in this field that manages variability using SPLs.

The main motivation of this paper is that, to the best of our knowledge, there does not exist any approach, that takes advantage of the extra information than a SPL approach provides. This extra information allows process engineers to visualise the execution of the process showing how the business evolves between several configurations. See Section 6 for a discussion on the approaches that inspires us and their deficiencies for the BIS field. This kind of visualisation is valuable since it allows process engineers to focus on higher level properties of the business, such as which product is more profitable, than those proposed in the Business Process Mining field.

The contribution of this paper is twofold: on the one hand, we provide a visualisation dashboard of execution traces based on the use of UML 2.0 timing diagrams and its integration with current approaches, which is detailed on Section 4. On the other hand, we have studied the problems related with the analysis of properties of execution traces providing a conceptual framework that shows a future research, which is shown in Section 5. Thus, our ideas open the possibility to reason about products, evolutions, triggers, etc., which, to the best of our knowledge, is not present in current approaches based on SPL, neither in the process mining field.

2 Adaptation of the SPL terminology to the context of the paper

In this paper, we use concepts of the SPL field with a slightly different meaning. These changes in the meaning occur because we applied these concepts to assets that are processes and not executable software pieces. In the following paragraph, we clarify the meaning of these concepts in our context:

Features* and *Business Processes: the *Product Evolution Model* (PEM) approach [8] establishes an equivalence between *business processes* and *features* as follows: (i) a *feature* represents a *subprocess* (part of a complete process that starts and ends), and (ii) *child features nodes* of a feature model, also considered *variants* in [9], corresponds with *concrete processes* (processes that do not present abstract or complex activities). In this paper, we also use a direct correlation between a *feature* and a *business subprocess*. Hereafter, we use the term *feature* to refer to both terms.

Predictable Evolutions* and *Products: Businesses evolve to adapt to environmental changes. This is done by including or excluding features or modifying existing ones. As shown previously, we use a SPL to manage these changes. Thus, we define a *product* as the set of features (subprocesses) that are enabled/running at a certain moment. In addition, we define the term *evolution* to denote the changes or transitions from one *product* to another. Note that we only take into account the *evolutions* that can be predicted at design time, called *predictable evolutions* (hereafter, *evolutions* for shortening).

Design-time*, *runtime* and *execution-time: It is called *design-time* as an interval of time in which we build the business process model and represent its variability, including *runtime* variability. *Runtime* is defined as an interval of time which starts when the business processes modeled are deployed on process engines. Thus, this term can be also named as *deployment time* or *configuration time*. Finally, *execution-time* is defined as the interval of time which starts when the business processes deployed are executed in the process engine. Thus, *runtime* variability is modeled at *design-time*, by means of PEM, and visualised and analysed at *runtime*. This analysis is based on the observation of *runtime* properties at *execution-time*. This observation is performed analysing the traces produced by the system.

Predictable* and *Unpredictable Triggers: *Triggers* act as stimulus of an evolution from a product to another. An *Unpredictable Trigger* is defined as something happening in the environment that fires an evolution that cannot be predicted at design time. A *Predictable Trigger* is defined as a condition that can be defined at design time that fires an evolution. See Section 4 for an example of *predictable* and *unpredictable triggers*.

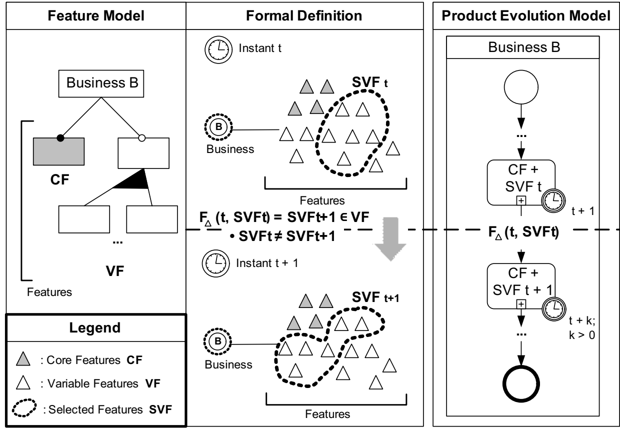


Figure 1.a. Rigorous Description **Figure 1.b. Graphical Notation**

Figure 1. Product Evolution Model approach defining an evolution of a business by the $F_\triangle$ function in t and $t + 1$.

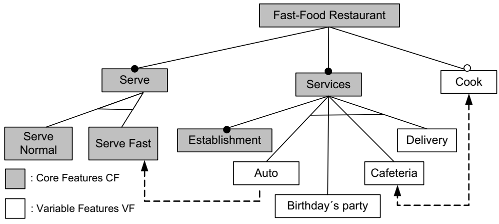


Figure 2. Case Study: Fast Food Restaurant

3 Representing Runtime Variability of BIS using the Product Evolution Model Approach

Product Evolution Model (PEM) is focused on providing a sufficiently expressive design-time model for representing runtime business properties. PEM provides in [8] an abstract rigorous description and a proposal for representing it by means of an extension of BPMN using stereotypes, including a case study. We show that description in the following sections.

3.1 PEM Rigorous Description

Let B be a business. Each business can be defined as a set of processes (denoted with P). Thus, B can be defined as follows:

$$B = \{P_1, P_2, ..., P_k\}; k > 0$$

Let *CF* be the set of common features, and let *VF* be the set of variable features, thus B is defined formally as a tuple containing all the CF and a subset of VF denoted as SVF:

$$B = (CF, SVF \in VF)$$

As shown before, in PFE, each set of features enabled at a certain moment represents a product. Thus, we can say that the CF of a B are always enabled at runtime, but the set of features in VF is not fixed at runtime.

Thus, we can set up a product line that takes into account this runtime variability. For formalizing these concepts we should redefine each business B as:

$$B = (CF, SVF \in VF, F_\triangle :$$
$$: t, \{Feature \times ... \times Feature\} \mapsto$$
$$\mapsto \{Feature \times ... \times Feature\})$$

where $F_\triangle$ is a function that given an instant t transforms the set of SVF_t into the new set of variable features of the following time instant t+1, that is to say SVF_{t+1}, formally:

$$F_\triangle(t, SVF_t) = SVF_{t+1} \in VF$$
$$\bullet SVF_t \neq SVF_{t+1}$$

Figure 1.a sketches a graphical representation of $F_\triangle$, where it is represented the transformation of SVF_t into SVF_{t+1}. In an instant t there exists a specific set of SVF_t for business B that evolves in instant $t + 1$ to a different set SVF_{t+1}.

3.2 PEM Graphical Notation

As shown previously, a business that evolves can be represented by $B = (CF, SVF \in VF, F_\triangle)$, where the evolution is defined by the $F_\triangle$ function in t.

In PFE, feature models are used to represent which features are variable and which are not. From this, the set of common (CF) and variable (VF) features can be obtained [1]. Thus, CF and VF can be represented by means of a feature model.

However, the feature model cannot establish the order of activation of features at runtime. This order is represented using $F_\triangle$, but as feature models are not devoted for representing runtime variability [5], they cannot be used for representing the variable t needed in the $F_\triangle$ function. For solving this problem, the *Product Evolution Model* (PEM) approach proposes a graphical notation that covers t and $F_\triangle$. This model is defined by means of a BPMN state machine where each state represents a product and each evolution between two or more states, is represented by means of a transition that is an application of the $F_\triangle$ function. In Figure 1.b, we show an evolution of a business from time t to

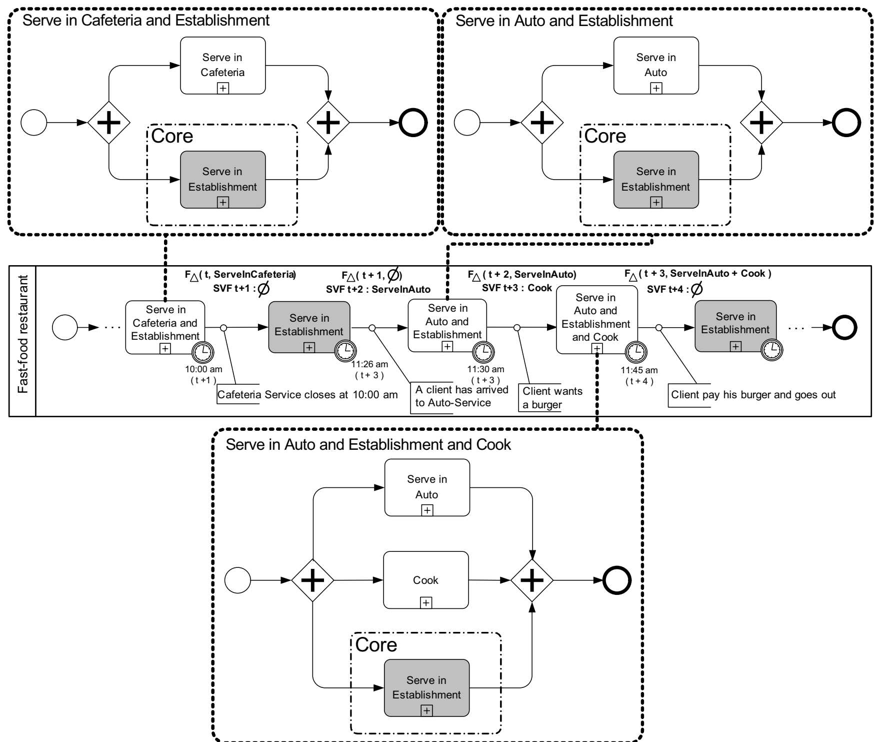


Figure 3. Fast-food restaurant Product Evolution Model BPMN Compositions

time $t+1$ by means of applying the $F_\triangle$ using a PEM model. As shown in the figure, there exists two different products. The first product is composed by the set of features CF and SVF_t. $F_\triangle$ in t fires an evolution at $t+1$ which implies the creation of the second product. This product is also composed by CF, since it never changes, and SVF_{t+1} which is different than previous SVF_t.

In order to illustrate PEM and the rest of the paper, we use a case study of a fast-food restaurant. Figure 2 depicts a simplified set of features pertaining to a fast-food restaurant: *Serve Normal*: which is defined as the normal activities for serving products in the restaurant, *Serve Fast*: which is defined as the activities needed for serving products in the restaurant when there exists a higher demand, and *Serve in Establishment*: which is defined as the activities for serving products performed only into the establishment. These features are CF and the rest are VF. In Figure 3 we present the PEM of this case study. Each state contains a BPMN state chart that represents how all the features are performed. It defines the evolution of the business at runtime showing that in every runtime instant t there exists a different SVF selected. For example, on a time instant t the restaurant opens its cafeteria service. In this moment, there exists two different processes running in parallel: *Serve in Cafeteria* and CF (*Serve in Establishment Normal/Fast*). When the restaurant closes its cafeteria service on time instant $t+1$,e.g. 10:00 am, the $F_\triangle$ function is applied and an evolution is performed to another state, that represent a different product, composed only by CF. After that, the restaurant opens its Auto-Service, because a client has arrived with his car at $t+2$. When this client orders a burger, the *Cook* subprocess is enabled, what happens in time instant $t+3$. When the burger is served, the system evolves to time instant $t+4$.

4 Visualisation of Runtime Variability in BIS

Process engineers need support for improving the processes by means of visualising and analysing the execution-time traces of business evolutions. For that purpose we provide a single view that illustrates all the transition from one product to another in certain moment. We use UML *Timing Diagrams* to represent this information. Timing diagrams are one of the new artifacts added to UML 2.0 which are used when the goal of the diagram is to reason about time. We call this view the *Business Dashboard*.

UML provides two different representations of timing diagrams: (i) *State* or (ii) *General value*. Both representations contain events and constraints that represent stimuli for an evolution. In Figure 6, we have included an example of each

Rigorous Description of PEM	UML 2.0 Timing Diagrams
Product	State
F∆	Transition / Stimulus
Trigger X	Predictable Trigger {X}
	Unpredictable Trigger X

Figure 4. Rigorous Description of PEM and Timing Diagram Correspondence

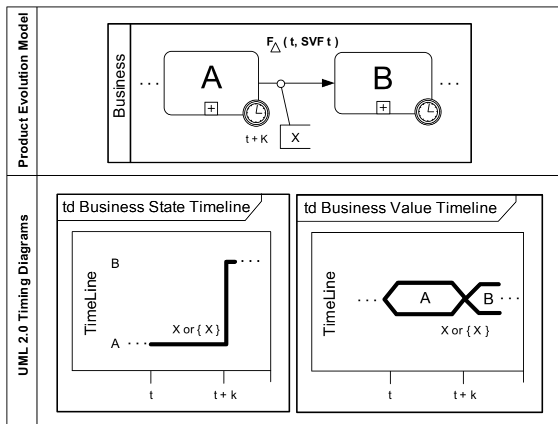


Figure 5. Obtaining Timing diagrams from Product Evolution Model

view. As shown in figure, the representation called *State* focuses on showing every evolution, while the representation called *General value*, focuses on each product instead of an implicit representation of an evolution. Given the characteristics of each view, the second representation, *General value*, is more adequate for software product lines where the number of products is high, while the first, *State*, is more adequate for software product lines where the number of products is low since evolutions are shown graphically.

Using the rigorous description defined previously in Section 3.1, we provide the correspondence between the information managed in PEM and timing diagrams. Figures 5 and 4 show the equivalence between a PEM and a timing diagram. As shown, each product modeled, using PEM, obtained from the application of the $F_\triangle$ function is equivalent to a state in a timing diagram. Notice that each $F_\triangle$ is performed in a time instant $t + k; k \geq 0$ when a trigger *X* holds. Notice that in timing diagrams, *X* denotes an unpredictable trigger, and $\{X\}$ a predictable trigger. See Figures 4 and 5 for an example of both kind of triggers. In PEM there is no difference between unpredictable and predictable triggers, since unpredictable only appears at execution-time and PEM is a design-time model.

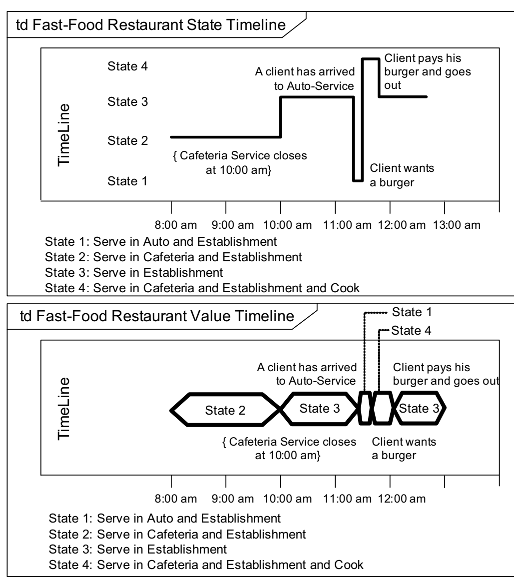


Figure 6. Visualising fast-food restaurant evolutions by means of UML 2.0 Timing diagrams

Figure 6 shows the timing diagrams of an execution trace of our case study. Each product is denoted by a state number. As shown in Figures 3 and 6, there exist four evolutions: (i) from product denoted by State 2 to another denoted by State 3 ($F_\triangle$ in t), which represents the predictable trigger: *Cafeteria Service closes at 10:00 am*. This implies an exclusion of *Serve in Cafeteria* feature from our product; (ii) from State 3 to State 1 products ($F_\triangle$ in $t+1$) that is performed when a client arrives to Auto-Service. This unpredictable trigger fires the second evolution that implies that feature *Serve in Auto* must be added or enabled in the new product; (iii) from State 1 to State 4 products ($F_\triangle$ in $t+2$) when a client wants a burger, that implies that feature *Cook* must be added in the new product; and finally (iv) from State 4 to State 3 ($F_\triangle$ in $t+3$) when client pays his burger and goes out.

In order to validate our approach, we have developed an automated transformation from a PEM execution trace to a timing diagram, concretely to *State* representation, using `gnuplot`[1], a command-driven interactive function and data plotting software. In Appendix we present an screenshot of the timing diagram of our case-study obtained using this transformation.

[1] http://www.gnuplot.info/

5 Roadmap for Research on Analysis

As shown previously, once runtime variability is visualised by means of timing diagrams, process engineers need to evaluate execution-time properties of the business. There are many basic analysis questions that can be performed, for example:

- Find constraints and events that fire a subprocess and calculate its relative frequency, i.e: *How many times a client arrives in Auto-Service?*
- Calculate relative frequency of the activation of a subprocess, i.e: *How many times Serve in Cafeteria subprocess is executed?*
- Analyse processes bottlenecks, i.e: *Which is the activity with the lowest level of performance?*

These kinds of questions are usually supported by current software tools for business process management and by the Process Mining approach [4][13][10]. They are focused only on analysing single/isolated subprocesses. However, given that PEM and PFE are based on SPL, there exist other analysis questions that may be supported providing higher level views for analysing the features, as for example:

- Analyse for each product: cost, risk and benefits.i.e: *Which is the percentage of benefits of product "State 1"?*
- Compare the performance of a certain feature when running in different products (dependencies with other features, events and/or constraints may affect the performance). i.e: *earning rate of product defined by state 1 is less than earning rate of product defined by state 2 on Fridays when it is executed in parallel with the Serve in Auto-Service feature*.

For arranging this research problem we propose two artifacts: (i) a metamodel for arranging and determining the needed information for supporting the analysis questions presented previously, which includes business process management support for current analysis questions, and (ii) a conceptual framework for future research on analysis which specifies how future research lines are related and may be conducted.

5.1 Analysis Metamodel

In this section we show the metamodel for arranging and determining needed information for supporting analysis questions presented previously. Figure 7 shows the metamodel that contains the following elements:

- **Business Process Management package**: it provides business process definition and represents the support for basic analysis questions provided by current tools for business process management.
- **Analysis Metamodel package**:
  - **Business Configurations**: states in timing diagram are considered business configurations represented by the *Business Configuration* metaclass. Each configuration contains a set of business processes which are modeled by means of the *Business Process* metaclass. It can be specialized to the *Core Business Process* or *Variable Business Process* metaclasses, previously denoted as CF and VF in the PEM definition.
  - **Predictable and Unpredictable Triggers**: these elements drive the evolutions of business configuration. They are modeled by the *Predictable*, and *Unpredictable* metaclasses.
  - **Financial Information**: Each business configuration has an associated cost, represented by the *Financial Information* metaclass, where we may add additional information about it; i.e: "*Serve in Establishment* process has an associated human resources cost of two employees" statement can be modeled by an association between the *Business Process* and *Financial Information* metaclasses instances, which attributes of second metaclass *type*, *value* and *unit* are initialised to "*human resource*", "2", and "*employees*" values respectively.
- **Dependency Metamodel package**: Business processes has associated a set of dependencies between them which are modeled by means of the *Dependency* metaclass. As shown in figure 7, the metaclasses in the *Dependency* package are based on Botterweck *et al.*'s metamodel for supporting feature configurations by interactive visualisation [2].

5.2 Conceptual Framework for Research on Analysis

For materializing these analysis operations we propose a conceptual framework for research on analysis based on filtering and analysing evolutions to perform queries using the information on the metamodel presented previously. Figure 8 shows it using a stereotyped association, *«uses»*, between the framework and our analysis metamodel. The framework also takes into account a representation for a Product Evolution Model and timing diagrams.

We have divided the elements included in the framework into those that can be implemented using our current results,

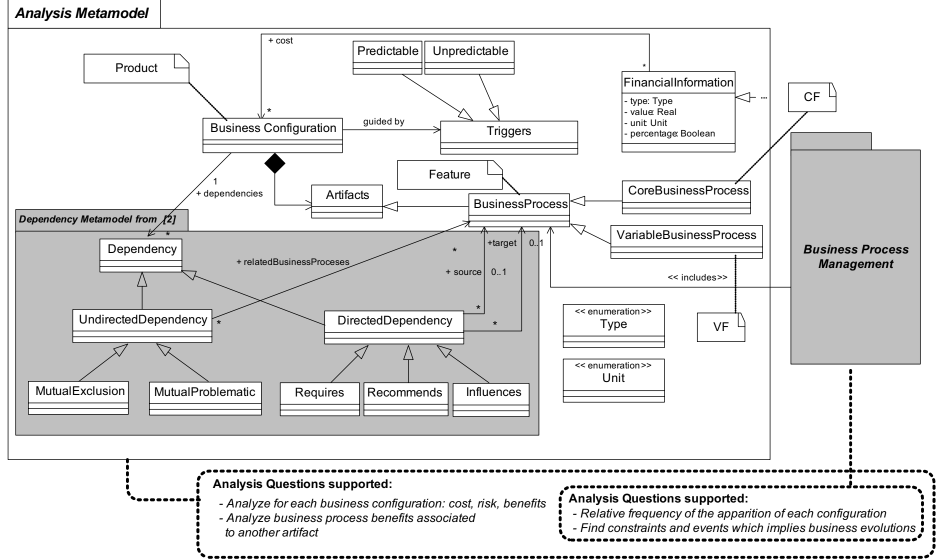


Figure 7. Proposed metamodel for analysing runtime variability in BIS (partially based on [2] metamodel)

labeled as *'implementation'*, and those that require for some research effort, labeled as *'research'*:

- **For visualising variability**:
  - **Artifact Factory** *(Implementation)*: Process engineers need to visualise evolutions. Evolutions are represented by means of *Timing Diagram* components. The component called *Artifact Factory* allows process engineers to generate timing diagrams from a business process modeled by means of Product Evolution Model using the information shown in the metamodel presented in Figure 7. PEM is represented by BPMN Product Evolution Model component. Notice that for obtaining PEM we need the core features which are obtained using FAMA [1]
- **For analysing variability**:
  - **Filter** *(Research)*: Process engineers can be interested in performing analysis questions about only one part of the timing diagram. For that purpose, the *Filter* component must provide query operations on BPMN Product Evolution Model and *Timing Diagrams*. The definition of these operations could be based on formalisms, such as Constraint Satisfaction Problem (CSP), Temporal Logic, Petri nets, etc. This is one of the possible future research lines, as shown in Figure 8
  - **Analyser** *(Research)*: As shown previously, process engineers need to perform analysis questions in order to improve their company. For that purpose the *Analyser* component should perform all possible analysis questions or operations from artifacts represented by *Timing Diagram* and BPMN Product Evolution Model components (the information shown in the metamodel). These operations can be grouped as basic operations, i.e:*obtain business process dependencies* and complex operations obtained by means of basic operations combinations, i.e: *obtain financial information about all possible business processes dependencies*. This represents another future research line.

6 Related Work

The *Business Process Mining* field, or *Process Mining* for short, is focused on extracting information about processes using execution traces [4][13][10]. For that purpose, these approaches provides some visualisation artifacts and frameworks for the automated analysis. Although this field is the most realted with the topic of this paper, there not exist any approach that support BISs based on product lines. Thus, these approaches cannot address the analysis questions provided by our approach and cannot represent the information on evolutions provided by our approach.

As shown in Section 3.2, feature models (FM) are one

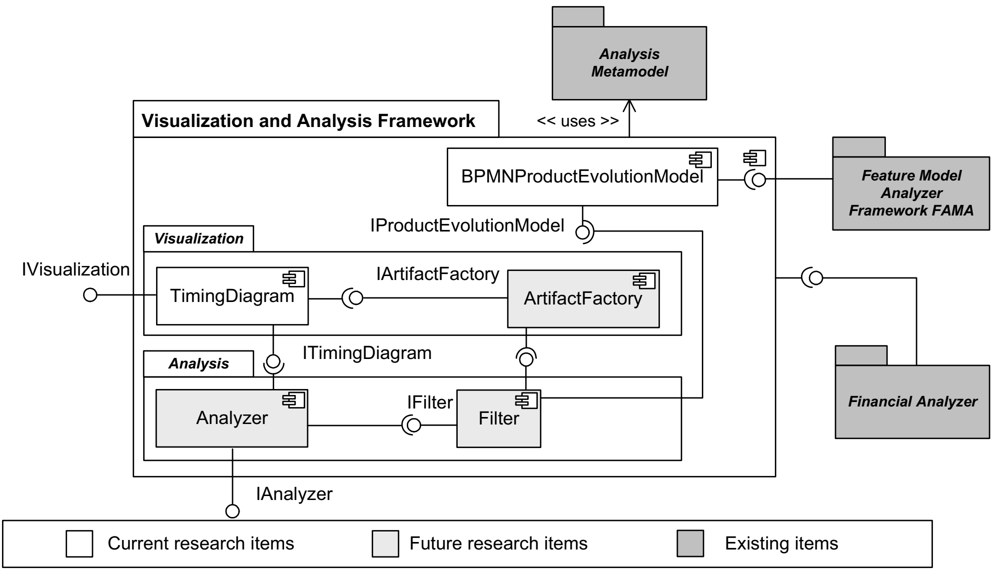


Figure 8. Runtime Variability Visualisation and Analysis Framework

of the most used artifacts for modeling variability. Unfortunately, as shown in Section 3.2, FM are devoted to design variability, and not for runtime variability [5]. There exists three approaches, to the best of our knowledge, that describes how to represent runtime variability in SPL.

First, J. Bosch *et al.* [7] introduce an extension of FM for representing runtime variability. Bosch's notation is slightly different from FODA's or FORM's notation. They introduce a new kind of feature for representing features that vary at runtime, called *external feature*, represented by means of a dashed rectangle. Figure 9 depicts an example of a feature model using this notation that represents the plugin support provided by the *Firefox* web browser. It represents that there exists one feature called *Website Debugger*, that can be enabled/disabled at runtime. As can be observed, the trigger events or conditions that fire this variability can not be represented with this approach, i.e: *plugin Website Debugger is enabled at runtime only in websites with domain* US.ES.

Sinnema *et al.* [12] propose a framework for modeling variability in SPL, called COVAMOF [2], which proposes a language for describing variation points named *COVAMOF Variability View Language* (CVVL) that takes into account enabling/disabling time. It is similar to the previous approach for representing runtime variability using in CVVL the tag *bindingtime*. The CVVL code for *Firefox* web browser example is the following:

```
<variationpoint id=Plugin>
   ...
   <variants>
       ...
    <variant id=Website Debugger>
       ...
       <bindingtime>runtime</bindingtime>
    </variant>
   </variants>
   ...
</variationpoint>
```

[2] www.covamof.com

H. Gomaa *et al.* [6][5] propose a set of models for representing runtime variability based on evolutionary reconfigurable software architectures. The different versions of an evolutionary system are considered a software product line, where each version of the system is a product and the reconfiguration is defined by a state machine that, for each component, represents the steps that have to be performed to evolve from a normal operation state to an inactive state. Once inactive, the component can be removed and replaced with a different version. Figure 10 depicts trigger events in the state machine. It represents how an optional feature named *Beeper* from a *Microwave System* feature model is enabled or disabled at runtime.

For runtime variability management in BIS, that is the focus of this paper, we have discussed in Sections 1 and the following proposals: *Process Family Engineering* (PFE) [11] and *Product Evolution Model* (PEM) [8] as a complement of PFE for representing a design model of runtime variability in BIS properly. However, none of these approaches provide any visualisation or analysis artifact for execution-time traces.

Given this state of art, to the best of our knowledge, there does not exist any approach for visualising and analysing runtime variability in execution-time of BIS using SPL techniques. This situation motivates us to propose a future

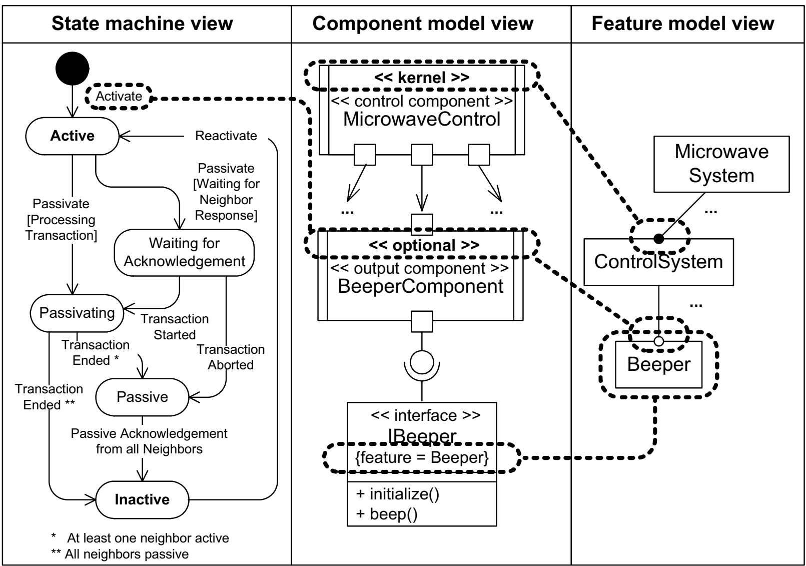


Figure 10. Gomaa approach (Figure taken from [6])

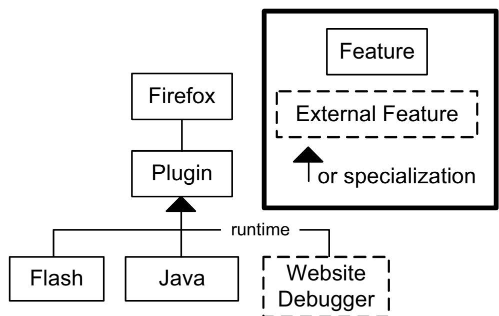


Figure 9. J. Bosch approach

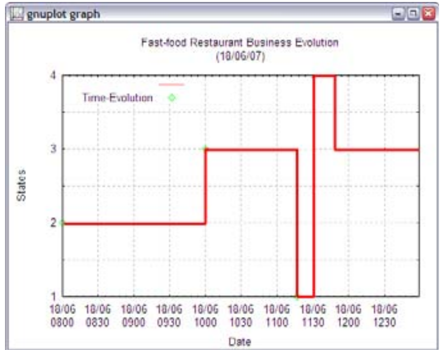


Figure 11. UML 2.0 Timing diagram obtained by gnuplot

research roadmap agenda and an approach for visualisation.

7 Conclusions and Future Research Roadmap

The main motivation of this paper is to provide to process engineers a first step toward an automatised visualisation and analysis of runtime variability in BIS based on SPL. For that purpose, we have explored the feasibility of using PEM for visualising and analysing runtime variability. As a result of our work we have proposed: (i) integration between PEM and a visualisation model based on UML 2.0 timing diagrams; (ii) a metamodel for arranging the information needed for analysing runtime variability in BIS; and (iii) a roadmap for research on analysing that can be used as a research agenda for this topic.

We think that this field is quite interesting and future research should be conducted. Thus the main research lines that could be derived from our framework are the following:

- Visualisation: to perform alternative techniques of proposed in this paper such as 3D representation, circle graphs, etc.
- Analysis: to explore possible basic and complex operations, obtained by means of basic operations combinations, for runtime business evolution execution-traces in order to perform queries, filters and analysis. As proposed in Section 5.2, the definition of these operations may be done using several formalism, such

as a *Constraint Satisfaction Problem* (CSP), Temporal Logic, Petri nets, etc.

In addition, due to our work is highly related to the *Process Mining* field, a survey of the techniques used in this field may help to clarify the first steps to be performed in the context of the future research lines identified

8 Acknowledgments

The authors would like to thank the reviewers of the Second International Workshop on Variability Modelling of Software-intensive Systems for their useful comments. We would like also to thank Patrick Heymans and David Benavides, whose comments and suggestions improved the presentation substantially.

References

[1] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés. FAMA: Tooling a framework for the automated analysis of feature models. In *Proceeding of the First International Workshop on Variability Modelling of Software-intensive Systems (VAMOS)*, 2007.

[2] G. Botterweck, D. Nestor, C. Cawley, and S. Thiel. Towards supporting feature configuration by interactive visualization. In *VISPLE'07: Proceedings of the 1st International Workshop on Visualization in Software Product Line Engineering - collated with SPLC 2007*.

[3] BPMI. Business process modeling notation BPMN version 1.0 - may 3, 2004. *OMG*.

[4] A. K. A. de Medeiros, C. Pedrinaci, W. M. P. van der Aalst, J. Domingue, M. Song, A. Rozinat, B. Norton, and L. Cabral. An outlook on semantic business process mining and monitoring. In R. Meersman, Z. Tari, and P. Herrero, editors, *OTM Workshops (2)*, volume 4806 of *Lecture Notes in Computer Science*, pages 1244–1255. Springer, 2007.

[5] H. Gomaa. Feature dependent coordination and adaptation of component-based software architectures. In *WCAT '07: Proceedings of the 4th Workshop on Coordination and Adaptation Techniques for Software Entities*, 2007.

[6] H. Gomaa and M. Hussein. Model-based software design and adaptation. In *ICSEW '07: Proceedings of the 29th International Conference on Software Engineering Workshops*, 2007.

[7] J. V. Gurp, J. Bosch, and M. Svahnberg. On the notion of variability in software product lines. In *WICSA '01: Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, 2001.

[8] I. Montero, J. Peña, and A. Ruiz-Cortés. Representing Runtime Variability in Business-Driven Development systems. In *Proceedings of the Seventh International Conference on Composition-Based Software Systems (ICCBSS08)*, 2008.

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

[10] A. Rozinat, A. A. de Medeiros, C. Günther, A. Weijters, and W. van der Aalst. The need for a process mining evaluation framework in research and practice. In *Proceedings of the Third International Workshop on Business Process Intelligence. (pp. 73-78). Brisbane, Australia: Queensland University of Technology.(2007).*

[11] A. Schnieders and F. Puhlmann. Variability mechanisms in e-business process families. In *Proceedings of BIS '06: Business Information Systems*, 2006.

[12] M. Sinnema, S. Deelstra, J. Nijhuis, and J. Bosch. COVAMOF: A Framework for Modeling Variability in Software Product Families. In *Proceedings of the Third Software Product Line Conference (SPLC04)*, San Diego, CA, 2004.

[13] W. M. P. van der Aalst, H. A. Reijers, A. J. M. M. Weijters, B. F. van Dongen, A. K. A. de Medeiros, M. Song, and H. M. W. Verbeek. Business process mining: An industrial application. *Inf. Syst.*, 32(5):713–732, 2007.

9 Appendix: gnuplot Experiment

In order to provide an experiment of automated transition from PEM to timing diagrams for visualising runtime business evolution execution-trace, we have deployed our case study PEM modeled by BPMN to a business process execution engine and it has been translated to WS-BPEL. We have developed two basic web services for representing choreography interaction between business process actors and we have executed it obtaining a runtime execution trace that has been stored in a file denoted as "fast-food-restaurant.dat". The following gnuplot script takes this file as input for plotting the timing diagram shown in Figure 11.

```
1  #*********************************************
2  # fast-food-restaurant.dem
3  # Author:
4  #   Ildefonso Montero Pérez - monteroperez@us.es
5  #   Dpto. Lenguajes y Sistemas Informáticos
6  #   Av. Reina Mercedes s/n, 41012 Seville (Spain)
7  #   University of Seville
8  # Description:
9  #   A gnuplot script to represent an UML 2.0
10 #   timing diagram of Fast-food restaurant
11 #   Product Evolution Model
12 #*********************************************
13 set title "Fast-food Restaurant Business
14           Evolution\n(18/06/07)"
15 set style data steps
16 set xlabel "Date"
17 set timefmt "%d/%m/%y\t%H%M"
18 set xdata time
19 set xrange ["18/06/07\t0800":"18/06/07\t1259"]
20 set ylabel "States"
21 set format x "%d/%m\n%H%M"
22 set grid
23 set key left
24 plot 'fast-food-restaurant.dat' using 1:3 t ' ', \
25      'fast-food-restaurant.dat' using 1:3 t
26      ' Time-Evolution' with points
27 pause -1 "Hit return to continue"
28 reset
```

Previously published ICB - Research Reports

2007

No 21 (September 2007)
Eicker, Stefan; Annett Nagel; Peter M. Schuler: "Flexibilität im Geschäftsprozessmanagement-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