International Workshop on Requirements Engineering for Product Lines

Article · January 2002

CITATIONS
3

READS
836

2 authors, including:

Klaus Schmid
Universität Hildesheim
262 PUBLICATIONS **5,325** CITATIONS

SEE PROFILE

Technical Report: ALR-2002-033

Proceedings

International Workshop on Requirements Engineering for Product Lines

September 9, 2002
Essen, Germany

Co-located with the IEEE Joint International Requirements Engineering Conference (RE02)

Birgit Geppert, Klaus Schmid (Eds.)

September 2002

This Avaya Labs Research Technical Report is available online at:
http://www.research.avayalabs.com/techreport.html

ALR-2002-033
ISBN 0-9724277-0-8

Preface

Product-line engineering has become more relevant to the requirements engineering community, and it continues to increase in importance. In the last IEEE International Symposium on Requirements Engineering (held in Toronto in 2001) one session and one tutorial were dedicated to this topic. This year marks the first time that the formerly bi-annual IEEE International Symposium on Requirements Engineering and the IEEE International Conference on Requirements Engineering are unified to the IEEE International Requirements Engineering Conference (RE). It is also the first time that workshops are offered and therefore also the first time that product-line engineering can be emphasized in this manner. We want to thank the organizers of RE for giving us the possibility to hold this workshop and hope that it will trigger a strong exchange between the requirements engineering and product-line engineering communities. Special thanks go to Klaus Pohl from the University of Essen, Germany, for chairing the RE workshops.

We want to thank the authors whose submissions made this workshop possible. Our gratitude is also extended to the program committee members, Günter W. Böckle, Jan Bosch, Paul Clements, Krzysztof Czarnecki, Stuart Faulk, Mats Heimdahl, Peter Knauber, Robyn R. Lutz, Klaus Pohl, Juha Savolainen, and David M. Weiss, for helping to organize this workshop and for their dedication in preparing thorough reviews of the submissions. We are also indebted to our keynote speaker, John MacGregor from Robert Bosch GmbH, Germany, for accepting the invitation and for his contribution to the workshop.

The topics covered by the accepted papers range from scenario and use-case modeling, over scoping and verification, to tool support. Among them we had a strong focus on applying use-case modeling techniques to product-line requirements. Use cases are an important tool in communicating the requirements of a system and are frequently used in today's software projects. They are one example of extending existing technology for the product-line field.

The intent of this workshop is both to enable discussion among the presented topics and to set directions for future work. We hope for a constructive dialogue between the product-line engineering and requirements engineering experts and look forward to future development in the area.

August 2002 Birgit Geppert, Klaus Schmid

Organization

REPL02 is co-located with the Joint IEEE International Conference on Requirements Engineering 2002 and held in Essen, Germany, September 9.

Workshop Chairs:

- Birgit Geppert
  Avaya Labs, Software Technology Research
  Basking Ridge, NJ, USA
  bgeppert@research.avayalabs.com
  +1 908 696 5116

- Klaus Schmid
  Fraunhofer IESE, Software Product Lines
  Kaiserslautern, Germany
  Klaus.Schmid@iese.fhg.de
  + 49 6301 707 158

Program Committee:

- Günter W. Böckle, Siemens AG, Germany
- Jan Bosch, University of Groningen, The Netherlands
- Paul Clements, Software Engineering Institute, USA
- Krzysztof Czarnecki, Daimler Chrysler AG, Germany
- Stuart Faulk, University of Oregon, USA
- Birgit Geppert, Avaya Labs, USA
- Mats Heimdahl, University of Minnesota, USA
- Peter Knauber, Fachhochschule (Univ.of Applied Sciences) Mannheim, Germany
- Robyn R. Lutz, Iowa State University, USA
- Klaus Pohl, University of Essen, Germany
- Juha Savolainen, Nokia Research Center, Finland
- Klaus Schmid, Fraunhofer IESE, Germany
- David M. Weiss, Avaya Labs, USA

Further Paper Reviewers:

- Eelke Folmer, University of Groningen, The Netherlands
- Jilles van Gurp, University of Groningen, The Netherlands
- Michel Jaring, University of Groningen, The Netherlands

Table of Contents

Requirements Engineering for Product Lines
- An Overview -

Birgit Geppert
Avaya Labs
Software Technology Research
Basking Ridge, NJ, USA
bgeppert@research.avayalabs.com
http://www.research.avayalabs.com/user/bgeppert

Klaus Schmid
Fraunhofer IESE
Software Product Lines
Kaiserslautern, Germany
schmid@iese.fhg.de
http://www.iese.fhg.de/Staff/schmid/

Product Lines: Software Engineering Becomes Efficient

For years the predominant view in software engineering was that of single-system development. We developed software for one system at a time and cared little about what other systems we are going to develop next. Reuse in this context was restricted to mere copy-and-adapt approaches. We analyzed what could be reused from previous projects (requirements specifications, code, design, etc.) and adapted these pieces to fit into our current system. This approach is opportunistic in nature. We reused components that we did not design to be reused in the current context and this led to inefficiency in software development, duplicated code, and multiple maintenance threads. This could have been avoided if the similarities in our systems would have been taken into account from the very beginning. Over time (and especially with the rise of component-based software engineering), another view entered the research landscape: let's build components once and for all, i.e., components that are reusable in arbitrary situations and without much adaptation effort. This was a very ambitious vision and as it turned out it was beyond the state of art of engineering. The demise of the IBM San Francisco framework and the Ariane 5 disaster are two examples for this [1].

Between these two extremes, software developers detected a promising middle ground: targeting a certain range of functionality for a limited set of products, the product line. While software engineering research neglected this pragmatic concept for years, it did take ground in practice. Many companies organized their software development in a way that was driven by this view. The ideas have been consolidated and (to some extent) also formalized as the concept of product-line development, and are now an established topic in several research organizations, including the Software Engineering Institute [2], the Fraunhofer Institute [3], Nokia, and Avaya Labs Research [4]. The result is reflected in several publications [5][6][7] and is also a hot topic in several meeting and discussion forums [8][9][10] where researchers and practitioners exchange their ideas and experience in this field. Reported results are indeed encouraging. Some companies report an order-of-magnitude improvement in effort, quality, and time-to-market when applying product line technology.

Requirements engineering for product lines: Is it really different?

The movement towards product-line development has some key implications for requirements engineering. First of all, there is the shift from a single-system to a family of systems. This implies that we need to model the requirements of multiple systems and that we need to represent the family and the differences among the individual family members in an explicit manner.

Another shortcoming in most requirements engineering research is a strong focus on a customer-centric viewpoint. A shift to a more market-oriented point of view is necessary. While some authors have already recognized the importance of this viewpoint (e.g., [11]), not much attention has been shown in the requirements engineering community in general. In the product-line context, however, we always need to take the market into account. Each particular system is sold to a single customer, but the product line as a whole targets a market.

A third difference lies in the way we can change the requirements for a product-line member. Having a product requirement that is slightly different from what is supported by the product-line infrastructure can result in much higher costs for the product. Negotiating with a customer to change slightly his or her requirements so that they are in accordance with the product-line infrastructure can make the product an order-of-magnitude cheaper, because suddenly the infrastructure can be used to its full potential. A requirements engineer who negotiates requirements must therefore not only be aware of the product requirements, but also of the constraints imposed by the product-line infrastructure. This is a challenge that has so far not been sufficiently accepted in requirements engineering research.

As these points exemplify, requirements engineering for product lines poses several challenges that are not present in traditional requirements engineering. In addition to these points the role of requirements engineering suddenly shifts. In particular, we now have two types of requirements engineering: one for the family as a whole and one for the individual system that should be built in the context of the product line. Often a whole business unit is concerned with the product line, and therefore requirements engineering for the family takes on a more strategic role for the company. Consequently, when we define the product line and particularly its scope we do indirectly define the economics of a major part (if not all) of the company.

This shift in importance can also influence the relative positioning of the roles of people in the organization. The task of the person defining the requirements for a product line and the one developing the market strategy for a business unit are virtually indistinguishable.

If we consider the individual parts of requirements engineering, we can easily identify additional impacts of the shift to product-line development:

Requirements Management: in the context of product-line development, we need to address the needs of multiple projects that may have overlapping, but non-identical, requirements and schedules. Consequently, the need to address the competing views of the projects comes into play.

Requirements Modeling: the dimension of variability is added to the "standard" requirements modeling problem. Also the interdependencies among requirements must be managed. This requires additions like decision modeling [12] or domain-specific languages (DSLs) [13]. Despite these differences, techniques like use-case modeling can also be applied to product-line modeling as several of the workshop papers discuss. Most research on product-line modeling actually focuses on UML. Today's industrial product lines, however, are built with other modeling techniques, in particular with text based requirements documents.

Requirements Process: as discussed above, in product-line development we have two life cycles, one for the family and one for the family members, and consequently two types of requirements engineering activities. We therefore also have two different requirements engineering processes for which product-line specific challenges need to be addressed.

Requirements Traceability: requirements traceability is always hard, and in a product line context it is even harder. First of all, the requirements we want to trace need not exist for each product. On the other hand, a certain implementation (or design element) may be linked with many different products and may be related to several different contexts, depending on the product.

Tool Support: this is an important topic. So far, however, this is also one of the major shortcomings of the product line field. There are currently no professional tools that may adequately support product line development, but some research prototypes exist. This may pave the way for future requirements engineering tool support.

Product-Line Requirements - And what about the rest?

Requirements engineering is only one aspect of product-line engineering. Product-line engineering can be truly successful only if we are able to map the requirements to a product-line

infrastructure that not only realizes the product-line commonalities, but also supports the predicted variabilities of the product-line members. A product-line architecture is not only designed for one single system, but for a family of systems. It is therefore concerned with instantiating different variations for different family members. Information hiding, domain-specific languages, or build-time parameters are examples of mechanisms that support such variations.

Another key success factor for product-line engineering is an appropriate organizational structure of the development organization. Identifying and transitioning to the right organizational structure is influenced by many factors, such as the existing organizational structure, the strategic positioning of the product line in the market, or the size of the organization. The organizational structure also needs to match the software architecture of the product line so that tasks and responsibilities can be assigned appropriately to the development units. This is actually a requirement on both, the organizational structure as well as the software architecture. Consequently, both need to co-evolve – and they need to do it in accordance with the evolving requirements.

Summary

This workshop deals with requirements engineering for product lines and emphasizes the difference from conventional requirements engineering. Product-line engineering shifts our focus from developing single systems to developing system families. The problems of eliciting, analyzing, and managing product-line requirements go beyond conventional requirements engineering. Among the problems are capturing requirements that are common to all members of the product family and managing expected variations.

While methodical aspects for eliciting and analyzing product-family requirements are highly relevant and in the focus of today's product-line engineering events, we should not neglect other aspects such as verification or tool support. These topics are essential to make product-line engineering more successful in practice and we look forward to future work on that.

References

[1] J. L. Lions, *ARIANE 5 Flight 501 Failure Report by the Inquiry Board*, http://java.sun.com/people/jag/Ariane5.html, August 2002

[2] Software Engineering Institute (SEI) at Carnegie Mellon, Homepage can be found at : http://www.sei.cmu.edu/plp/plp_init.html, August 2002

[3] Fraunhofer Institute for Experimental Software Engineering (IESE), Homepage can be found at: http://www.iese.fraunhofer.de/Business_Areas/Product_Line_Development/, August 2002

[4] Avaya Labs Research, Homepage can be found at: http://www.research.avayalabs.com/department/softtech/, August 2002

[5] Paul Clements, Linda Northrop, *Software Product Lines, Practices and Patterns*, Addison-Wesley, 2001

[6] Juha Kuusela, Juha Savolainen, *Requirements engineering for product families*, Proceedings of the International Conference on Software Engineering (ICSE), 2000

[7] David M. Weiss and C. T. Robert Lai, *Software Product Line Engineering*, Addison-Wesley, 1999

[8] The bi-annual International Software Product Line Conference (SPLC), information can be found at http://www.sei.cmu.edu/, August 2002

[9] CAFÉ project: From Concept to Application in System-Family Engineering, Homepage can be found at: http://www.extra.research.philips.com/euprojects/cafe/, August 2002

[10] International Workshop on Product Line Engineering The Early Steps: Planning, Modeling, and Managing (PLEES), Homepage can be found at: http://www.plees.info/, August 2002
[11] C. Potts and K. Takahashi and A. Antón, *Inquiry-Based Requirements Analysis*, IEEE Software, Vol. 11, No. 2, pp. 21-32, 1994
[12] Joachim Bayer, Dirk Muthig, and Tanya Widen, *Customizable Domain Analysis*, Proceedings of the Generative and Component-based Software Engineering Conference (GCSE), Germany, 1999 (Springer LNCS 1799)
[13] Juha-Pekka Tolvanen, *Domänenspezifische Modellierungssprachen für Produkt-familien (Domain-Specific Modeling Languages for Product Families)*, ObjectSpektrum, August/September, 2001

Requirements Engineering in Industrial Product Lines

John MacGregor
Robert Bosch GmbH
Frankfurt, Germany

1 Abstract

Requirements management is one of the fundamental aspects of process maturity and figures in the early stages of most assessment schemes [Paulk et al. 1995] [SPICE 1998]. In the industrial context, requirements engineering is therefore a "given". The product development process, whatever it may be, must follow the principles of requirements engineering. This paper examines how the principles of requirements engineering are guaranteed in the product line approach.

We examine how product line product development supports the goals of requirements engineering, especially with respect to the need that they share for traceability between artefacts of different software development phases. Furthermore, we examine feature models in detail, thereby showing how they can be employed to better ensure the realisation of customer requirements and to better react to changes in those requirements.

This paper first defines the industrial context and then examines the factors contributing to the importance of these two disciplines; also their synergy. Requirements engineering is then defined to give a framework for the discussion that follows. Next, the product line approach is examined in detail with attention being paid to the role of requirements and traceability. Lastly, the congruence of product line to the aims of requirements management is discussed and future directions are indicated.

2 Introduction

2.1 Industrial Context

This paper addresses "industrial" software development, as opposed, perhaps, to commercial, academic or scientific software development. That is, in the narrow sense, the development of software to be sold as a product or part of a product. The term "industrial" is used here in a broader, perhaps more traditional sense of development within industry where software is part of systems that are subsequently manufactured. Note that the "software industry" would be specifically excluded from this perhaps eclectic definition.

Rather than defining its scope by excluding parts of industry, the approach of the paper is to address the part of industry that exhibit the following characteristics:

- **Systems, with software** – the products are integrated packets of hardware and software. In the case of embedded systems, the algorithms for controlling the physics of the system must sometimes also be developed along with the hardware and software.
- **Large number of variants** – Companies can develop large numbers (hundreds, even thousands) of similar products, each adapted to meet special customer or legal requirements, to conform to national preferences or to international standards.
- **Hardware Resource Constrained -** The products are built on hardware platforms. In a volume manufacturing context the unit cost of the hardware platform is significantly greater than amortised development costs. This makes the optimal use of the hardware a significant consideration in the product design.

The product line approach is especially appropriate, when not unavoidable in these organisations. Several factors speak for reusing knowledge and elements of the development process, especially.

- The large number of common parts whose development can be minimised; - the converse to having a large number of variants
- The magnified impact of software defects that are released to manufacture
- Customers' requirements for assurance with respect to the development process.

It is always desirable to be able to develop the product predictably with consistently high quality as is the goal of process maturity. In industry, with so many products being developed, the development process is repeated so often that ensuring that it is repeated consistently and efficiently becomes increasingly critical. Process quality improvement programmes, such as ISO 9000, BOOTSTRAP, CMM or SPICE all focus on process standardisation. With standard processes comes the need for standardised products and it follows therefore that the product line approach and requirements engineering are closely allied in the industrial context.

2.2 Requirements Engineering

The goals of requirements engineering are to ensure that possibly changing customer requirements are documented to the satisfaction of all stakeholders and that those requirements are entirely and appropriately reflected in the products being developed. It is a holistic approach that addresses the entire spectrum of product development.

Requirements engineering recognises the critical role played by requirements in the development process. Requirements are the first link in the chain connecting the customer with the product he desires. Should this link break, there is no guarantee that the product fulfils the expectations of the customer, with commensurate impact on its marketability. Similarly the linkage between the product qualities, such as performance, reliability, security or safety, and their requirements must be verified in the whole development process through to coding and testing. Otherwise, the consequences in terms of liability, or even to life and limb, can be enormous.

Requirements engineering is composed of **requirements management,** the process to establish and maintain a complete, consistent and unambiguous specification of the product to be developed and **requirements analysis**; the process to define the new product based on the requirements.

Initial product requirements can be gathered from a number of sources: contracts, contract addenda as faxes or e-mails, transcripts of telephone calls between sales and the customer or minutes of requirements elicitation interviews. As such, they vary in degree of completeness and specificity. Moreover, these requirements are in the customer's language and therefore do not necessarily specify the technical aspects of the product unambiguously. Requirements in their raw form are therefore a poor basis for product development. Requirements analysis is the process of amassing these requirements and transforming them into a technically feasible product definition that can be used as the basis for development.

Requirements management is concerned with accommodating changes in the requirements that inevitably occur while the product is being developed. First, it ensures that changes are accepted in an orderly manner, that the effects of the changes are known in advance and that the requirements remain consistent, complete and unambiguous. Second, requirements management ensures that these changes are propagated and integrated in the subsequent phases of the development, test and release processes.

2.3 Product Line Approach

The representation of product line is still in flux, but for expository purposes, the original "Sixpack" representation as illustrated in Figure 1 [Foreman 1996] [SEI STR 1997] [ESAPS 1999] is fully adequate. Refer to the [SEI_Framework] for an more detailed, up-to-date description.

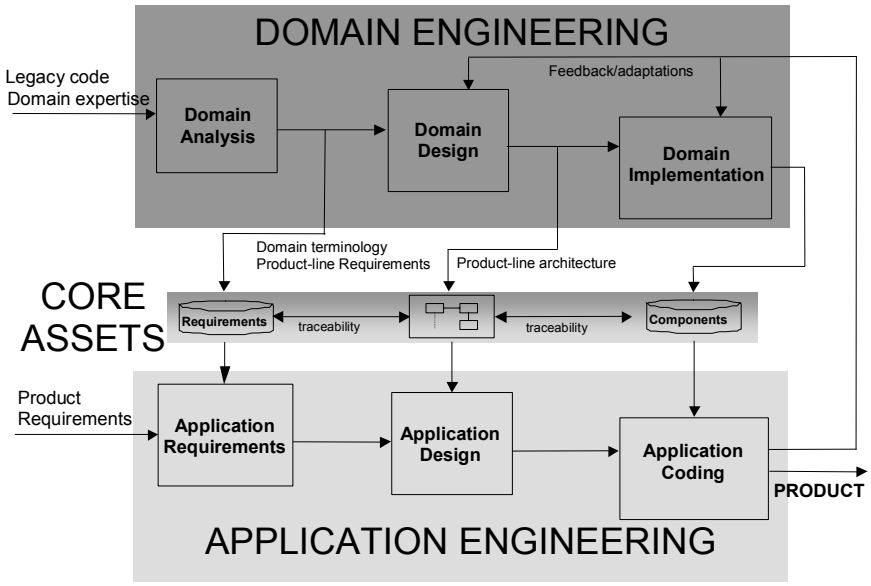


Figure 1: The "Sixpack" Model.

The product line approach is an approach to architecture-based reuse. Its basic assumption is that there are substantial commonalities among the product line products. Given a structuring or architecture that identifies the common parts and characterises what they have in common, software development artefacts, (such as specifications, test suites or software components) specific to a particular product can be generalised to cover the corresponding commonality. These generic artefacts, called **core assets**, can then be used directly in product development or adapted as appropriate instead of developing them from scratch. Thus they form in effect a platform from which the individual products can be developed.

The product line process is therefore composed of two asynchronous processes: the **domain engineering** process which creates core assets (development for reuse [Karlsson 1995]) and the **application engineering** process which uses core assets to create products (development with reuse).

A product line has a scope, its domain, which defines the set of products that can be developed. In the first stages of domain engineering**,** this scope is defined and the products to be included in the product line are identified or defined if they do not already exist. In the next step, the commonalities between their features or requirements as well the variabilities in the features / requirements are analysed to produce the platform specification. The platform can then be developed, using conventional means. This is not to say that the further process is trivial, merely that the problem with domain engineering lies in defining what to build rather than how to build it.

The most efficient way to build new products is to build them entirely from existing components, perhaps only changing parameterised values. New product development may involve requirements that cannot be covered by existing assets which dictates the modification of existing components or the development of new ones. Regardless whether parameterisation, modification or new development is ultimately employed, efficiency dictates that the most suitable asset be identified directly from the requirements. That is, the product line structure should support **traceability** from specific requirements (or sets of requirements) to the assets that can used to realise these requirements. Should components have to be adapted or developed, the product line structure should identify the code, design documents etc. for the (generic, as it does not exist) component that would best meet the requirements. For completeness sake, it should also be mentioned that when components must be adapted or developed, that the product line structure should nonetheless identify the components that can be (re)used without modification. Refer to [MacGregor 2001] for a more detailed explanation.

The structure of the platform reflects a strategy about how the components should be assembled. The domain engineering process identifies points in the architecture where different components can be used: **variation points**. In assembling a product, the requirements for the product must be transformed into the classification scheme dictated by the architecture. The appropriate components can then be identified in a

process sometimes called **binding** the components to the variation points or otherwise known as **configuring** or **deriving** the product. **Requirements traceability** to components is thus a core element of the produce line approach.

3 Requirements Representation in Product Lines

3.1 Aggregation of Requirements

Product line differentiates between commonalities among products and variabilities between them, where the variabilities present the challenge. Expressing the variabilities in terms of individual requirements would neither offer a significant reduction in the amount of analysis to identify variability nor offer a level of abstraction to identify similarities among requirements.

The possible mechanisms for aggregating the requirements both at the domain engineering and the application levels are:

- *Functionality:* which describes the system's capabilities are described in terms of the functionality provided by the components of the system
- *Use cases:* which describe the dynamic interaction between the user and the system in terms of stimuli and responses. ([Jacobson et al. 1997])
- *Features:* which describe the system in terms distinctive product characteristics; services provided to the user or customer for example. Note that features can be functions, but must not necessarily be functions.

In domain engineering, all three are valid approaches and their use depends on the needs of the modeller or user of the model. They are also not mutually exclusive. A functional breakdown, although intuitive, does not fully describe a system architecture. Features are especially useful for communicating with end-users or customers and use-cases are especially useful for formulating test suites, both for development testing and for customer acceptance testing.

In application engineering, feature models are favoured for their intuitiveness for users and their focus on variability. Their support for traceability is also advantageous for requirements engineering.

3.2 Features

Features are groups of requirements that give a product distinctive characteristics or qualities. The modelling representation of a feature generally consists of its name and a generic. The representation can additionally consist of a number of attributes with associated ranges of validity which then allows automatic consistency checking (See Outlook).

The Feature Oriented Domain Analysis (FODA) [Kang 1990] method is the earliest example of the application of feature modelling in product lines, albeit only in domain engineering. Subsequent enhancements and refinements include FeatuRSEB [Griss et al. 1998], FODAcom [Vici 1998] and FORM [Kang 1998]. Refer to [Hein et al 2000] for a survey of the feature modelling approaches and an assessment of their suitability for the configuration of products in application engineering.

The features of products are aggregated as (**compositional**) hierarchies of features and sub-features and are represented in feature trees. At any level, there may be any number of alternative features (**taxonomies**) that must (mandatory) or may (optional) be chosen (**cardinality**). Moreover, features have **dependencies**: the selection of one feature may rule out (mutual exclusion) or assume (requires, includes) the inclusion of another feature.

This aggregation of features represents all variability within the scope of the domain and is thus the domain feature model, or **feature model**. The term should not be confused with the modelling representation of a feature.

Figure 2 illustrates typical feature tree relationships and notation. In this example, an automobile must have a motor and a transmission while air conditioning is optional. The transmission must be either manual or automatic while the air conditioning can be climate control or manual. An 8 cylinder motor must have an automatic transmission. A 4 cylinder motor cannot have an automatic transmission.

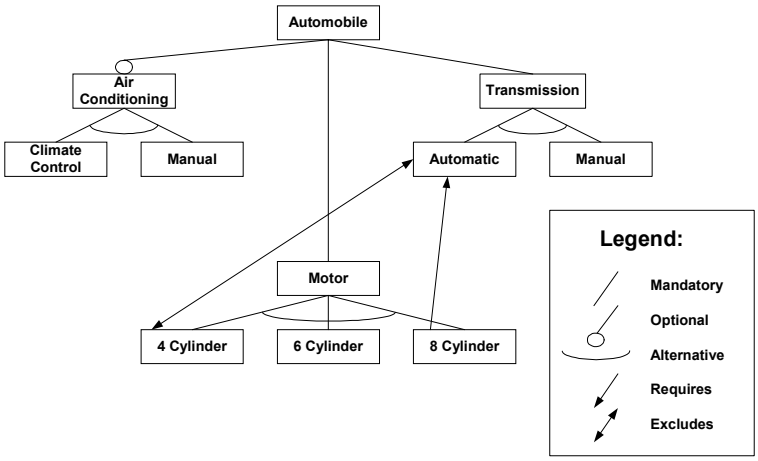


Figure 2. Example Feature Model

The application engineering product derivation process consists of traversing the feature tree in an orderly manner and selecting the optional features. The result is a product description containing all the features in the product (a **feature configuration**). In product line terminology, a product is fully specified when all of its variation points are bound; that is the product specification is complete when all features have been selected. Note that everything in the product that does not contribute to a feature has also been selected implicitly as it is part of a commonality.

The product line methodology is relatively new and concepts about how, in detail, the application engineering process should unfold remain relatively vague. Most approaches agree that the feature configuration is at least a formal requirements specification from which development can proceed. They also agree that when all development is complete, the system components must be configured (brought together and parameter values set). Between these two ends, there are a number of approaches. FORM [Kang 1998] extends FODA to application engineering. The Helsinki Institute of Technology [Tiihonen et al 1998] has its own approach and Generative Programming [Czarnecki & Eisenecker 1999] [Czarnecki & Eisenecker 2000]offers a method for directly generating the components. See the Outlook, below for our approach to bridging this gap.

4 Discussion

As mentioned above, compared to conventional system development processes, domain engineering has the additional aspects of defining the set of products to be built, amassing the union of their individual requirements, classifying their variability and producing specifications for the individual reusable assets. The techniques used in this requirements analysis are substantially conventional and the assets themselves can be realised using conventional development methods.

Note that with the product line approach, the platform is built once and reused many times. The rest of the paper concentrates therefore on requirements management in the application engineering process.

Note also that this discussion may give the impression that the benefits of using feature models can only be accrued using feature models, but in reality use-case or functional models can provide substantially the same benefits so long as they also guarantee complete and consistent results

Feature models can and should be formulated using user-level concepts although they can be formulated on the technical realisation level as well. Features are therefore the preferred method for expressing variation, especially in application engineering. In general using features avoids overwhelming users with irrelevant considerations as only the variable parts of the platform must be selected. In fact, when the inclusion and exclusion dependencies can be evaluated dynamically during the selection process further irrelevant, or inconsistent, decisions can be avoided in that the selection of a feature can include or exclude other features automatically.

4.1 Requirements Management Aspects

From a requirements management standpoint using feature models guarantees a consistent and complete requirements specification. The level of ambiguity is reduced to the level inherent in the feature model; which is both formal and formulated using concepts familiar to users or customers. (There is still a potential for ambiguity, as different users may interpret these concepts differently.) With an appropriate feature model, the feasibility of the requirements can be assessed as well.

When requirements change, the point of change can be quickly pinpointed using the feature model and its impact on the development process assessed. Because of the nature of the feature model, it is even conceivable that the impact of changes can be made transparent to users or customers (as long as company secrets remain intact, for example).

The product line approach is predicated upon the traceability of requirements, from the point of a specification, right through to code and testing. The product line approach can therefore be viewed as a means to achieving the consistency goals that requirements engineering sets. The issue of how traceability is to be implemented remains largely open (see Outlook below), however

4.2 Requirements Analysis Aspects

From a requirements analysis standpoint using feature models replaces the last parts of the analysis as the feature configuration is the definition of a new product, exactly as in requirements analysis. The feature model provides a consistent and complete common reference for all requirements. The formalisation of the composition rules combined with standards for product feature reduces the leeway for interpretation and the corresponding possibilities for errors or inconsistencies.

Moreover, using feature models reduces effort, shortens processing time and provides better quality analysis. The feature configuration process, in effect, reuses the feature architecture and the general benefits of reuse (as ascribed to product line): reduced effort, shorter processing time and better quality will therefore all accrue to the requirements analysis process as well.

5 Conclusions

The product line approach does not replace requirements engineering nor reduce its importance in any way. In fact, it can be viewed as a rigourous application of requirements engineering concepts which tries to eliminate superfluous sources of effort and defects.

The product line approach promises to allow the development of products with reduced effort and time to market while also increasing product quality. Applying it will bring commensurate benefits to the requirements analysis process and will optimise the requirements management process by increasing transparency and eliminating the maintenance of superfluous intermediate products.

Feature models, as defined in FODA, FORM, etc. and applied to application engineering, provide a useful structure for requirements elicitation and can be used to generate a consistent, complete and unambiguous requirements specification. In the future, it should be possible to use feature models to assess the feasibility of the requirements and to automatically select and configure the software components corresponding to the requirements, insofar as the requirements were foreseen in the product line.

6 Outlook

The author is manager of the EU project ConIPF (**Con**figuration in **I**ndustrial **P**roduct **F**amilies). The goal of the project is to develop a product line methodology that allows the use of feature models to directly configure software components.

In the artificial intelligence discipline of structure-based configuration, product-structure based compositional concept hierarchies have been used successfully to configure conventional products (without software) for over a decade. The similarities between these concept hierarchies and feature models are striking. In contrast to product line feature tree theory however, structure-based configuration

additionally considers the tasks and strategies associated with configuration. It also uses formal methods to guarantee consistency and completeness of the configuration.

The challenge of the project is twofold: first, to develop a knowledge base to transform a feature configuration (the set of features selected for a particular products) to the associated hard- and software component configuration; and secondly, to analyse the characteristics of product-line assets and to define the characteristics so that they can be configured optimally.

7 References

[Czarnecki & Eisenecker 1999] K. Czarnecki, and U. Eisenecker: *"Synthesizing Objects"*. In: Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP'99), Springer, 1999.

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

[ESAPS 1999] *"ITEA-ESAPS Full Project Proposal; European ESAPS Consortium",* June 1999.

[Foreman 1996] Foreman, J. "*Product Line Based Software Development- Significant Results, Future Challenges*." Software Technology Conference, Salt Lake City, UT, April 23, 1996

[Griss et al. 1998] M. L. Griss, J. Favaro, M. d'Alessandro, *"Integrating feature modelling with the RSEB"*, Proceedings. Fifth International Conference on Software Reuse (Cat. No.98TB100203). IEEE Comput. Soc, Los Alamitos, CA, USA, 1998, xiii+388 pp. p.76-85.

[Griss 2000] M. L. Griss, *"Implementing Product line Features with Component Reuse"*, to appear in Proceedings of 6th International Conference on Software Reuse, Vienna, Austria, June 2000.

[Hein et al 2000] A. Hein, M. Schlick, R. Vinga-Martins: *"Applying Feature Models in Industrial Settings",* In: P. Donohoe (ed.): Software Product Lines – Experience and Research Directions. Proceedings of the First Software Product Line Conference (SPLC1), August 28-31, 2000, Denver, Colorado, USA. Kluwer Academic Publishers, 2000.

[Jacobson et al. 1997] I. Jacobson, M. Griss, P. Johnson, *"Software Reuse: Architecture, Process and Organization for Business Success"*, Addison-Wesley, 1997.

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

[Kang 1998] K.C. Kang, *"FORM: a feature-oriented reuse method with domain-specific architectures",* in Annals of Software Engineering, V5, pp. 354-355.

[Karlsson 1995] E-A. Karlsson, Editor, *"Software Reuse - a Holistic Approach"*, John Wiley & Sons, 1995.

[MacGregor 2001] MacGregor, J.: *"A Proposal for a Product Line Derivation Process"*; Software Product Lines: Economics, Architectures, and Implications, Peter Knauber, Giancarlo Succi (Editors), Proceedings of Workshop #3 at 23rd International Conference on Software Engineering (ICSE2001), Toronto, Ontario, Canada, May 13, Fraunhofer IESE Report No. 051.01/E, 2001

[Paulk et al. 1995] M. Paulk, C. Weber, B. Curtis, M. B. Chrissis: *"The Capability Maturity Model: Guidelines for Improving the Software Process"*; Addison-Wesley; ISBN:0-201-54664-7

[SEI_Framework] P. Clements, L. Northrop: *"A Framework for Software Product Line Practice Version 3.0"* (Online); Software Engineering Institute, Carnegie Mellon University URL: http://www.sei.cmu.edu/plp/framework.html, March 2002

[SEI STR 1997] *"SEI Software Technology Review"* (Online); Software Engineering Institute, Carnegie Mellon University, 1997. URL: [http://www.sei.cmu.edu/str/descriptions/]

[SPICE 1998] ISO/IEC/15504-2*:"Information Technology – Software Process Assessment Parts 1-9"*, ISO/IEC JTC1/SC7 N1603, 1998

[Tiihonen et al 1998] J. Tiihonen, T. Lehtonen, T. Soininen, A. Pulkkinen, R. Sulonen, and A. Riitahuhta: *"Modeling Configurable Product Families"*. 4th WDK Workshop on Product Structuring, October 22-23, 1998, Delft University of Technology, The Netherlands

[Vici 1998] A. Vici, N. Argentieri, A. Mansour, M. d'Alessandro, and J. Favaro, FODAcom: An Experience with Domain Analysis in the Italian Telecom Industry. Proceedings of the 5th International Conference on Requirements Engineering (ICRE'98). IEEE Computer Society Press. Victoria BC, Canada, June 2-5, 1998. ISBN 0-8186-8377-5.

Use Case Description of Requirements for Product Lines

A. Bertolino^, A. Fantechi*, S. Gnesi^, G. Lami^ , A. Maccari[+]

*Dip. di Sistemi e Informatica - Università di Firenze - Italy
^Istituto di Elaborazione della Informazione - C.N.R. - Area della Ricerca C.N.R. - Pisa - Italy
[+] Nokia Research Center, Software Architecture Group - Finland

Abstract

Capturing the variations characterizing the set of products belonging to a product line is a key issue for the requirements engineering of this development philosophy. This paper describes ways to extend the well-known Use Case formalism in order to make possible the representation of these variations, in the perspective to make them suitable for an automatic analysis.

1. Introduction

The development of industrial software systems may benefit from the adoption of a development cycle based on the so called *system-families* or *product lines* approach [2,5]. This approach aims at lowering production costs when the products share the overall architecture and conception, but differ with respect to particular characteristics. These variations represent the customisation of the product with respect to the other members of a family of systems. The production process is therefore organized in product lines with the aim of maximizing the commonalities of the product family and minimizing the cost of variations.

In the first stage of a software project, called usually *requirements elicitation*, the knowledge of the system under construction is acquired. Inside a product line, both problems of capturing requirements common to all members of the product family, on one side, and of specializing the general product family requirements into those ones of a single product, on the other side, have to be addressed.

To deal with these problems, a close look to the nature of the relations between family and product requirements should be given: in these relations the concepts of parameterisation, specialization and generalization can play a major role.

On the other hand, the problems are not easily formalized, due to the nature of requirements, which are often given in a natural language prose. Without sacrificing the immediateness of the natural language, at least a notation that adds structure to the requirements should be adopted.

Use Cases are a powerful tool to capture functional requirements for software systems. They allow for structuring the requirements documents according to user goals and provide a means to specify the interactions between a certain software system and its environment. In his book on how to write Use Cases [1], Alistair Cockburn presents an effective technique for specifying the interactions between a software system and its environment. The technique is based on natural language specifications (i.e., phrases in plain English language) for scenarios and extensions. This makes requirements documents easy to understand and communicate even to non-technical people.

The purpose of this paper is to present how to extend Use Cases in the direction of addressing Product Line requirements, including some specific constructs to deal with variability.

This paper is structured as follows: in section 2 we describe the principal characteristics of Use Cases, in section 3 we discuss the impact of variability on the Product Line requirements. In section 4, possible extensions to Use Cases to face variability are discussed; finally in section 5, conclusions and future research directions are presented.

2. Background: Use Cases

A Use Case describes the interaction (triggered by an external actor in order to achieve a goal) between a system and its environment. A Use Case defines a goal-oriented set of interactions between external actors and the system under consideration. The term *actor* is used to describe the person or system that has a goal against the system under discussion. A primary actor triggers the

system behaviour in order to achieve a certain goal. A secondary actor interacts with the system but does not trigger the Use Case.
A Use Case is completed successfully when that goal is satisfied. Use Case descriptions also include possible extensions to this sequence, e.g., alternative sequences that may also satisfy the goal, as well as sequences that may lead to failure in completing the service in case of exceptional behaviour, error handling, etc.. The system is treated as a "black box", thus, Use Cases capture *who* (actor) does *what* (interaction) with the system, for what *purpose* (goal), without dealing with system internals. A complete set of Use Cases specifies all the different ways to use the system, and therefore defines the whole required behaviour of the system. Generally, Use Case steps are written in an easy-to-understand, structured narrative using the vocabulary of the domain. A scenario is an instance of a Use Case, and represents a single path through the Use Case. Thus, there exists a scenario for the main flow through the Use Case, and as many other scenarios as the possible variations of flow through the Use Case (e.g., triggered by options, error conditions, security breaches, etc.). Scenarios may also be depicted in a graphical form using UML Sequence Diagrams.
Figure 1 shows the template of the Cockburn's Use Case taken from [1]. In this textual notation, the main flow is expressed, in the "Description" section, by an indexed sequence of natural language sentences, describing a sequence of actions of the system. Variations are expressed (in the "Extensions" section) as alternatives to the main flow, linked by their index to the point of the main flow from which they branch as a variation.

USE CASE #	< the name is the goal as a short active verb phrase>	
Goal in Context	<a longer statement of the goal in context if needed>	
Scope & Level	<what system is being considered black box under design> <one of: Summary, Primary Task, Sub-function>	
Preconditions	<what we expect is already the state of the world>	
Success End Condition	<the state of the world upon successful completion>	
Failed End Condition	<the state of the world if goal abandoned>	
Primary, Secondary Actors	<a role name or description for the primary actor>. <other systems relied upon to accomplish use case>	
Trigger	<the action upon the system that starts the use case>	
Description	**Step**	**Action**
	1	<put here the steps of the scenario from trigger to goal delivery, and any cleanup after>
	2	<...>
	3	
Extensions	**Step**	**Branching Action**
	1a	<condition causing branching> : <action or name of sub-use case>
Sub-Variations		**Branching Action**
	1	<list of variations>

Figure 1. Use Case template

3. Variability in PF Requirements

Following the System Family Engineering Process Reference Model defined in the CAFÉ project [6], and shown in Figure 2, product family development is characterized by two processes: Application engineering and Domain engineering. Domain engineering is the process aiming at developing the general concept of a family together with all the assets which are common to the products of the family, whereas Application engineering is intended the process aiming at designing a specific product. During Application engineering a customer specific application will be defined. However, differently from the usual single product development, the definition process of the customer specific application is not only influenced by the requirements of the customer but also by the capabilities of the product family.
This diagram shows that it is possible to move from the family level (by means of the system family engineering activity) to the product level and vice versa (by means of the system family reverse engineering activity).
Going upwards, applications are developed considering the capabilities of the product family specialising, extending and adding family requirements. Consequently, software product families

need more sophisticated requirement processing and requirements should deal with some variability notion.
In particular, product family requirements can be considered in general as composed of a constant and a variable part. The constant part includes all those requirements dealing with features or functionalities common to all the products belonging to the family and that, for this reason, do not need to be modified. The variable part represents those functionalities that can be changed to differentiate a product from another.

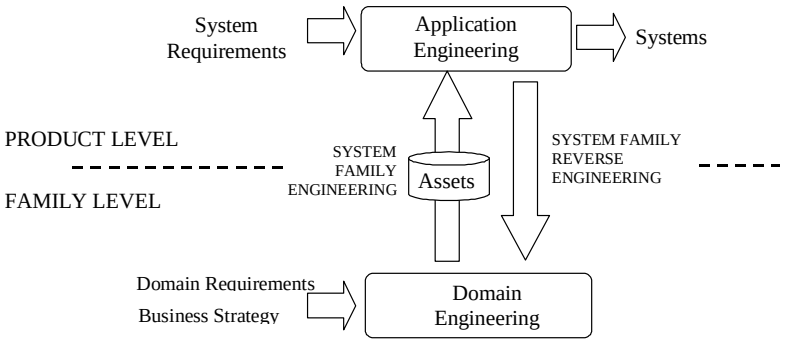


Figure 2. The CAFÉ-PRM reference framework

Following the above process we can see variability from two different perspectives: the first is the product perspective where each variability has to be considered as an aspect to be instantiated. Whereas, from the family perspective a variability is a goal to be reached by abstracting all the instances related to the existing products belonging to a product line.
It is possible to move down from the family level to the product level by an instantiation process and on the contrary from the product level up to the family level by an abstraction process. In these two different processes the main objects to pay attention on are variations.
We therefore aim at identifying needed extensions to express variability during requirements engineering.
Natural Language (NL) processing techniques, such as those proposed in [4] for evaluating requirement documents, can be fruitfully employed to this aim. Usually these techniques are used to detect and then remove ambiguity and vagueness in a requirements document because in traditional development process it is an undesirable side effect of the use of NL; in the context of Product Lines, on the contrary, we want to use them to identify automatically the necessary generic parts in order to specialise them into the specific product features (see the following examples):

Example 1

Req 34: *the system shall simulate different vehicles*

Different is a *vague* word, detected by the tools proposed in [4]
In a family description *different* could be considered as pointing at a variability to be specialized in the derived products:

Req A.34: *the system shall simulate cars*
Req B.34: *the system shall simulate trains*

Example 2

Req 217: the system shall be such that the mission can be pursued, possibly without performance degradation

Possibly is a word indicating *optionality*.
In a family description *possibly* could be considered as pointing at an optionality differentiating among products:

Req A-217: the system shall be such that the mission can be pursued, without performance degradation
Req B-217: the system shall be such that the mission can be pursued (performance degradation is admitted)

Notice that this is not the only interpretation of the optionality: the family requirement can be read also as imposing that any product will make its best effort to avoid performance degradation, if this is possible under adverse conditions. Hence, this kind of linguistic analysis can be useful to point out *potential* variability.

4. Extensions of Use Cases for Product Lines

When adopting Use Cases description of requirements for Product Families, variations can be addressed in Use Cases by adopting two complementary approaches:

1. Structuring the Use Case requirements as having two levels: the family level and the product level. In this way product-related Use Cases should be derived from the family-related Use Cases by an instantiation process;
2. Incorporation of the application level Use Cases into the domain level Use Case, in this way both the product and the family level requirements will stand at the same level into the same, all inclusive, Use Cases document.

4.1 Two levels Use Cases

The solution for capturing variability we discuss in this section is based on the inclusion of *tags* into the scenarios (both main scenario and extensions) that identify and specify variations. The tags can be of three kinds: Optional, Alternative, Parametric.

When a product is derived from the family, the variable parts must be instantiated in different ways according to the type of variability:

1. Alternative components: they express the possibility to instantiate the requirement by selecting an instance among a predefined set of possible choices, each of them depending on the occurrence of a condition;
2. Parametric components: their instantiation is connected to the actual value of a parameter in the requirements for the specific product;
3. Optional components: their instantiation can be done by selecting indifferently among a set of values which are optional features for a product instantiation.

The instantiation of these types of variability will lead to a set of different product-related instantiated Use Cases. Figure 3 expresses this situation as a UML class diagram.

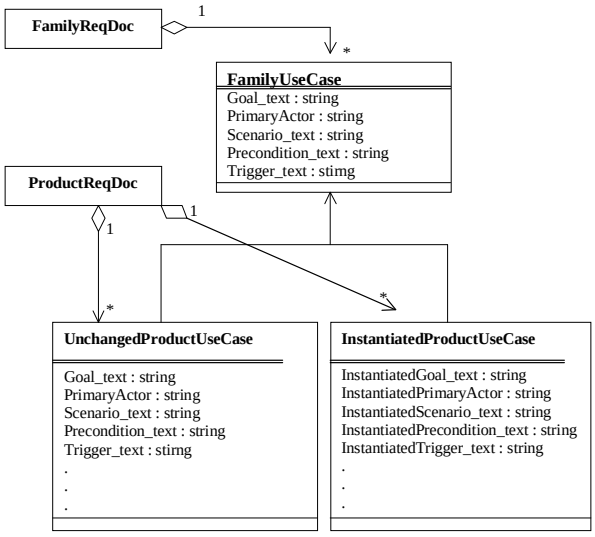


Figure 3. A UML model of two levels Use Cases Document

Example 3.

```
Primary Actor:  the {[V0] 33-serie} mobile phone (the system)
Goal: play a game on a {[V0] 33-serie} mobile phone and record score
Preconditions: the function GAME has been selected from the main MENU
Main Success Scenario:
- The system displays the list of the {[V1] available} games
- The user select a game
- The system displays the logo of the selected game
- The user selects the difficulty level by following the {[V2] appropriate}
   procedure and press YES
- The system starts the game and plays it until it goes over
- The user records the score achieved and {[V3] possibly} send the score to
   Club   Nokia via WAP
- The system displays the list of the {[V1] available} games
- The user presses NO

V0: alternative
      V0:  1. Nokia 3310 model
           2. Nokia 3330 model
V1: optional
       if V0=1 then game1 or game2
       else if V0=2 then game1 or game2 or game3
V2: parametric
 if V0=1 then procedure-A:   - press Select
                             - scroll to Options and press YES
                             - scroll to Difficulty Level and press YES
                             - select the desired difficulty level, press
YES
 else if V0=2 then procedure-B: - press Select
                                - scroll to Level and press YES
                                - select the desired difficulty level, press
YES
V3: parametric
 if V0=1 then function not available
 else if V0=2 then function available
```

4.2 One level Use Cases

The second way to manage variations in Use Cases is to include all the alternatives that can occur when a product will be considered in each Use Case. In this way the canonical structure of the Use Case should be modified in order to be suitable to include all the particular features that the whole product family set of instances can have. Figure 4 shows, again by a UML class diagram, that each Use Case includes all the possible alternatives due to the specialization of the generic Use Case.

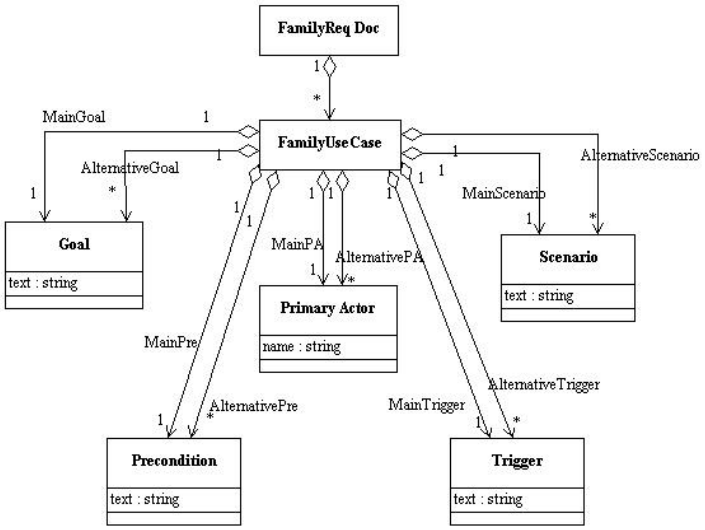


Figure 4. A UML model of one level Use Cases Document

Example 4.

```
Primary Actor: The 3310 – 3330 mobile phone
Goal: play a game on a 3310 – 3330 mobile phone and record score
Preconditions: the function GAME has been selected from the main MENU

Main Success Scenario:
-  3310 model: The system displays game1 or game2
   3330 model: The system displays game1 or game2 or game3
-  The user select a game
-  The system displays the logo of the selected game
-  The user selects the difficulty level by following the procedure below:
      3310 model:     press Select
                      scroll to Level and press YES
                      select the desired difficulty level and press YES
      3330 model:     press Select
                      scroll to Options and press YES
                      scroll to Difficulty Level and press YES
                      select the desired difficulty level and press YES
-  The system starts the game and plays it until it goes over
-  The user records the score achieved
-  3330 model: send the score to Club Nokia via WAP
-  3310 model: The system displays game1 or game2
   3330 model: The system displays game1 or game2 or game3
-  The user presses NO
```

4.3 Discussion

The extensions presented in the previous sections reflect different perspectives from which variability can be seen. The first approach considers the variations implicitly enclosed into the components of the Use Cases. The variations are then represented by tags that indicate those parts of the family requirements needing to be instantiated for a specific product in a product-related document. On the contrary, the other extension explicitly includes all the possible variations into a unique, all inclusive, document.

The tagged extension, because its dynamic structure, allows modifications to be faced more easily in terms of new functions or products in the product line. The flat, all inclusive approach is more static and, since it provides plenty of information at the same level.

By considering the nature of the two approaches, the flat representation of the use cases seems suitable for functionalities that are already mature and almost stable; while the tagged representation is more suitable when the functionalities related to the use case are still in the way to be precisely defined.

In reference to the CAFÉ-PRM reference framework, these considerations imply that the tagged extension is more useful during Application Engineering, in which variations are introduced on top of already existing products. During Domain Engineering the comprehensiveness of the flat extension allows to develop the general concept of a family.

The trade-offs between the two approaches have anyway to be better investigated. In fact, the two approaches may co-exist into the same Use Case document, where some parts may be expressed by using the tagged way and some other by using the flat way.

5. Conclusions and Future Works

In this paper we presented two possible ways to define extensions on the canonical Use Case structure in order to make them suitable to manage the variations of the product line requirements. The impact of these two approaches on the structure of the use cases and the consequent extensions to be made have been discussed in the paper.

The proposed Use Case extensions seem to be suitable to perform linguistic analysis to find ambiguity, inconsistency and incompleteness defects by means of automatic tools. Next work on this argument will deal on the application of existing linguistic techniques [3], [4] for evaluation and defects detection in use cases-based product line requirements. Furthermore, we intend to explore the respective pros and cons of the two approaches in practice.

6. References

[1] A. Cockburn. Writing Effective Use Cases, Addison-Wesley, 2000

[2] P. Clements, L.Northrop. Software Product Lines: Practice and Patterns, SEI Series in Software Engineering Addison Wesley, 2001.

[3] F.Fabbrini, M.Fusani, S.Gnesi, G.Lami. The Linguistic Approach to the Natural Language Requirements Quality: Benefits of the use of an Automatic Tool, 26th Annual IEEE Computer Society - NASA GSFC Software Engineering Workshop, Greenbelt, MA, November 27-29 2001.

[4] A.Fantechi, S.Gnesi, G.Lami, A.Maccari. Linguistic Techniques for Use Cases Analysis, Proceedings of the IEEE Joint International Requirements Engineering Conference - RE02. Essen, Germany, September 9 -13 2002.

[5] M. Jazayeri, A. Ran, F. van der Linden. Software Architecture for Product Families: Principles and Practice, Publishers: Addison-Wesley, Reading, Mass. and London, 1998.

[6] F. van der Linden Software Product Families in Europe: The ESAPS & Café Projects, IEEE Software July/August 2002

Modeling Variability by UML Use Case Diagrams

Thomas von der Maßen, Horst Lichter
{vdmass, lichter}@cs.rwth-aachen.de
Research Group Software Construction
RWTH Aachen

Abstract

Software Product Lines are characterized through common and variable parts. Modeling variability is one of the most important tasks during the analysis phase. In order to guarantee that domain experts and developers understand each other variability has to be modeled explicitly. In this paper we present and examine the different types of variability. We analyze and assess UML Use Case diagrams in detail with respect to their capability to express variability, with the result, that they only provide poor assistance. Consequently we introduce an enhancement of the Use Case meta-model to overcome this deficiency by integrating new modeling elements that allow to express the identified types of variability.

1 Introduction

Modeling variability is an essential task in developing Software Product Lines [PLP01]. Variability means that beside the common parts that are shared by all members of a Software Product Line, each product has its own specific parts. Defining the commonality and the variable parts of a Software Product Line is mainly done during product line scoping [Sch00]. But, variability exists and must be modeled in every phase of the product line development process which means in the requirements analyzing and in the design phase. Variability is finally realized in the implementation phase.

Besides considering variability in different phases it is very important to take into account that different types of variability exist. Halmans [Hal02] differentiates between *technical variability*, comprising all kinds of variability that exist in the system infrastructure, concretion and realization of the product line and *functional variability*, defining functional and quality characteristics of the system. Technical variability is defined in terms of "how" a product line can be implemented; functional variability is defined in terms of "what" the product line should be capable of.

The analysis of functional variability is necessary to identify characteristics that are mandatory, that means that they are common, and characteristics that are variable, that means, they are not mandatory but can be considered in a specific context or not. To communicate and negotiate common and variable characteristics with the stakeholders an appropriate notation must be chosen to guarantee that domain experts and developers understand each other.

In the next section we describe which types of variability exist and list requirements on a notation for modeling variability. Section 3 gives a brief overview on the feature based modeling approach and considers UML Use Cases in the context of modeling functional variability. In section 4 we present an extension of the UML Use Case meta-model, integrating variability-modeling elements. Finally, we draw some conclusions and give a brief outlook on our future research.

2 Modeling Functional Variability

2.1 Types of variability

Variable system characteristics are defined by means of so called *variation points* (see [Jac97]). At a variation point different specific variants can be chosen for each family member to resolve this variation point. The following types of functional variability must be considered:

- **Options**
  Optional characteristics of a system can be integrated or not. That means from a set of optional characteristics, any quantity of these characteristics can be chosen, including none or all. We distinguish between options that can only be chosen if a specific condition holds and options where one has a free choice to integrate them or not. Hence, optional aspects can be modeled by means of an or-relationship.
- **Alternatives**
  From a set of alternative characteristics, only one characteristic can be chosen - defining an exclusive-

or/xor-relationship, which means a "1 from n choice". Again, we distinguish between alternatives that are linked to a specific condition, or not.

- **Optional alternatives**
  At last a combination of optional and alternative characteristics must be considered. This is the case, if at a variation point alternatives are available, but it can be chosen if these alternatives are relevant at all – that means a "0 or 1 from n choice".

2.2 Levels of variability

Modeling variability can be done in different views and on different levels of abstraction. Whereas the level of abstraction determines the granularity of descriptions of characteristics, different views reveal information about perspectives on a system. In addition, variability can be examined in different scopes.

Figure 1 shows that variability exists among the different members of a product line designed for a specific domain. That means, the various members (products) provide different functionality. This variability is modeled at the domain level and is resolved when a concrete product is instantiated. Though the variability between the product line members is resolved, variability will occur for each product at runtime. Runtime variability should be modeled at the product level. It occurs if different application flows exists. The application flow is usually driven by the user or external systems.

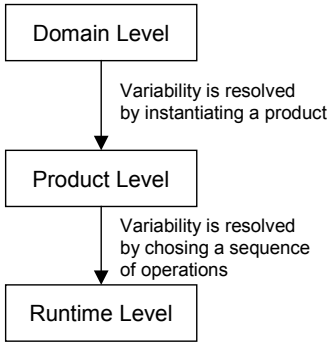


Figure 1: Resolving variability

2.3 Requirements on a modeling notation

To express variability it is necessary to model *variation points* and its variable parts [Jac97]. Modeling notations can be graphical, textual or a mixture of both. The advantage of modeling variability in a graphical notation is, that variation points are recognized much easier than in a pure textual description. In this section we present important requirements that have to be put on a notation for modeling variability. From the modeling point of view, the following requirements have to be considered:

- **Representation of common and variable parts**. It must be possible to express common and variable parts of a Software Product Line. Hence, distinct modeling elements are needed to explicitly express common characteristics belonging to the platform and variable characteristics. Variable characteristics should always be modeled in the context of a variation point.
- **Distinction between types of variability**. The notation must be able to explicitly express all different types of functional variability introduced in section 2.1.
- **Representation of dependencies between variable parts**. The description of dependencies is mandatory. Dependencies between variable parts are implied, equivalent and xor-relationships. An implied-relationship means, that if one characteristic is needed, than another characteristic must be taken into the system as well. An equivalent-relationship is an implied-relationship in both directions and a xor-relationship expresses that only one characteristic from a set of characteristics can be taken into the system.

Beside these modeling requirements the notation should support the following aspects:

- **Supporting model evolution**. Because models will evolve over time, new requirements and characteristics have to be integrated and the model must be changed easily.
- **Providing good tangibility and expressiveness**. To communicate the resulting models to the domain experts the models must be easy to read and to understand. In most cases domain experts are not able to understand formal textual notations. So a graphical notation might help to understand the relevant parts very easily. It is important to mention that the graphical notation should not replace natural language descriptions but to supplement them and to provide a different view on the same context.

3 Feature and Use Case Modeling

In this section two notations will be presented to model variability from different points of view: Feature graphs and UML Use Case diagrams. These two notations will be analyzed with respect to the requirements mentioned in section 2.3.

3.1 Feature graphs

The Feature Oriented Domain Analysis (FODA) is a common approach to model variability during domain analysis [Kan00]. In FODA, features are the central modeling element. A feature is defined as a "prominent or distinctive user-visible aspect, quality or characteristic of a software system or systems". The core of a feature model is a feature graph. A feature graph represents variability in a very compact and clear way, in that it presents the features in a tree of AND/OR nodes to identify the common and variable parts within the domain. Eisenecker describes the capability of feature graphs in detail [Eis00]. He points out, that a great advantage of feature graphs over UML class diagrams is that the representation is independent from implementation aspects and that features are easier building blocks than objects, because no mechanism like generalization, association or parameterization is used. The successful use of feature graphs is described in [Eis01] and [Phi00]. Figure 2 shows a small feature model in the domain of an automated teller machine, that provides several banking services and authorizes identification either through a chip card or fingerprint.

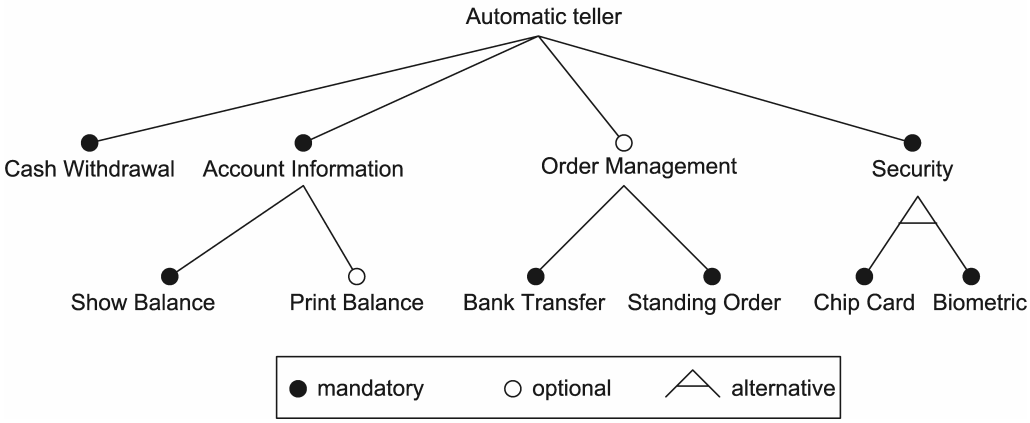


Figure 2: Example of a feature graph

A feature graph illustrates a structural view on a system. Characteristics of the system are structured hierarchically in a tree. A characteristic groups the functionality that constitutes this characteristic. Common characteristics can be modeled through mandatory features and variable parts can be modeled through optional or alternative features, which fulfill the requirements to distinguish between different types of variability. Therefore, feature graphs are able to express the identified variability, whereas conditions for optional or alternative types must be specified explicitly through a textual note. In addition, dependencies exist implicitly through the hierarchy of features, that means the existence of a child feature implies that the parent feature exists, as well. A xor-relationship exists between alternative features which have the same parent feature. Dependencies between features that don't have the same parent cannot be modeled directly in the feature graph, though it is possible to add composition rules that "supplement the feature model with mutual dependency and mutual exclusion relationships which are used to constrain the selection from optional or alternative features" [Kan02]. Unfortunately feature graphs and FODA are not widely known, accepted and used. Therefore, the practical use is constrained [Spe02].

In the next section we analyze UML use case diagrams with regard to their capability to model variability.

3.2 Use Case diagrams

During requirements analysis, Use Case diagrams help to identify the actors and to define by means of Use Cases the behavior of a system (see [Jac92]). Use Case modeling is accepted and widely used in industry. If Use Case diagrams support the modeling of functional variability, they can also be used to describe common and variable behavioral characteristics of a Software Product Line. Hence, we take a brief look on the UML Use Case modeling elements. Beside actors and Use Cases UML defines a small set of

relationships to structure actors and Use Cases. Use Cases may be related to other Use Cases by the following relationships:

- **Extend**
  An extend relationship implies that a Use Case may extend the behavior described in another Use Case, ruled by a condition.
- **Include**
  An include relationship means that a Use Case includes the behavior described in another Use Case.
- **Generalization**
  Generalization between Use Cases means that the child is a more specific form of the parent Use Case. The child inherits all features and associations of the parent, and may add new features and associations [UML01].

It is obvious that it is possible to model options that are linked to a condition through the extend relationship. All other types of functional variability cannot be modeled directly with Use Case diagrams. To illustrate this we examine the following example in the context of the automatic teller machine, mentioned above. Using the automatic teller machine an authentication procedure is necessary to get access to the banking services. The authentication can be performed by inserting a chip card or by giving a fingerprint. The authentication procedure is predetermined by the teller machine, that means the teller machine provides only one authentication procedure. A Use Case diagram where extend-relationships are used to model options is depicted in figure 3. Unfortunately this diagram doesn't model the context correctly. It is not clear that exactly only one extension Use Case must be executed. The two extension Use Cases are mutually exclusive, so.

Though it is possible to introduce stereotypes to mark relationships between Use Cases, stereotypes typically lack of a defined semantic and may have different meanings in different contexts. Furthermore it is possible to define constraints between Use Cases, but their semantics may vary in different contexts, too.
Therefore new Use Case modeling elements are needed to explicitly express all types of variability described above. It should be mentioned, that feature graphs, which represent characteristics of the system, can be used to complement Use Case modeling and to organize the results of the commonality and variability analysis in preparation for reuse [PLP01].

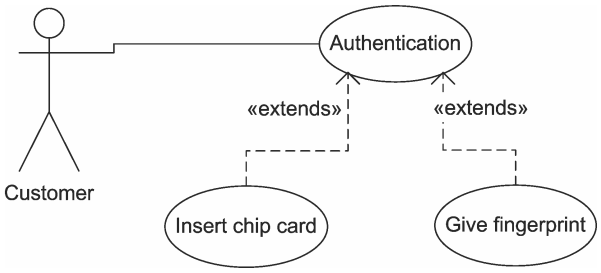


Figure 3: Modeling options by means of extend relationships

3.3 Comparison

These two notations allow to model variability in a domain from two different perspectives. Variability exists in features and in sequences of activities that means between product line members and at runtime of a specific product. Feature graphs are structure oriented because they describe the characteristics of a domain and the relationships between them. Furthermore they are intended to show hierarchies in structures and visualizes dependencies between characteristics, though in a restricted way because only dependencies between parent and child nodes can be expressed directly. Use Cases instead are suitable to model sequences of activities and they model therefore dynamic characteristics and dependencies between these activities.

A suitable process in capturing variability in an unknown domain is to start with capturing Use Cases to find essential activities that are executed in that domain. These essential activities can be structured further and can be extended on a more fine granular level. In a second step the variability in Use Cases can be summarized in features that can be modeled through feature graphs to identify common and variable characteristics. Thus feature graphs form a consolidation of variable sequences and performed activities.

To model variability in Use Cases the meta-model must be extended. This extension is introduced in the next chapter.

4 Extending the UML Use Case Meta-Model

4.1 Model extension

To explicitly model all types of variability described in section 2.1 the UML Use Case meta-model [UML01] has to be extended by two new relationships: *Option* and *Alternative*.

An option relationship between two Use Cases means that the behavior defined in the extending Use Case may be executed in the extension Use Case. This depends firstly on whether the functionality is provided by a potential product and not on a condition determined by the system or on earlier activities carried out by the actor. An optional behavior that depends on a condition can be modeled through the existing extend relationship.

An alternative relationship means that in the base Use Case exactly one alternative Use Case must be executed. Alternatives own an attribute *choice*, that consists of pairs of conditions and linked Use Cases. The conditions define under which circumstances an alternative Use Case is executed. The choice, which alternative to use may depend on a condition or not. The condition is formulated in a Boolean expression. This relationship exists always between one base Use Case and at least two alternative Use Cases. These new relationships can be integrated easily in the UML meta-model. Figure 4 shows the resulting meta-model.

To model variation points, an additional new model element *VariationPoint* has been included as a specialization of *ExtensionPoint*. A VariationPoint additionally has an enumeration of the alternative Use Cases. Alternatives and options are new relationships to express explicitly these types of variability.

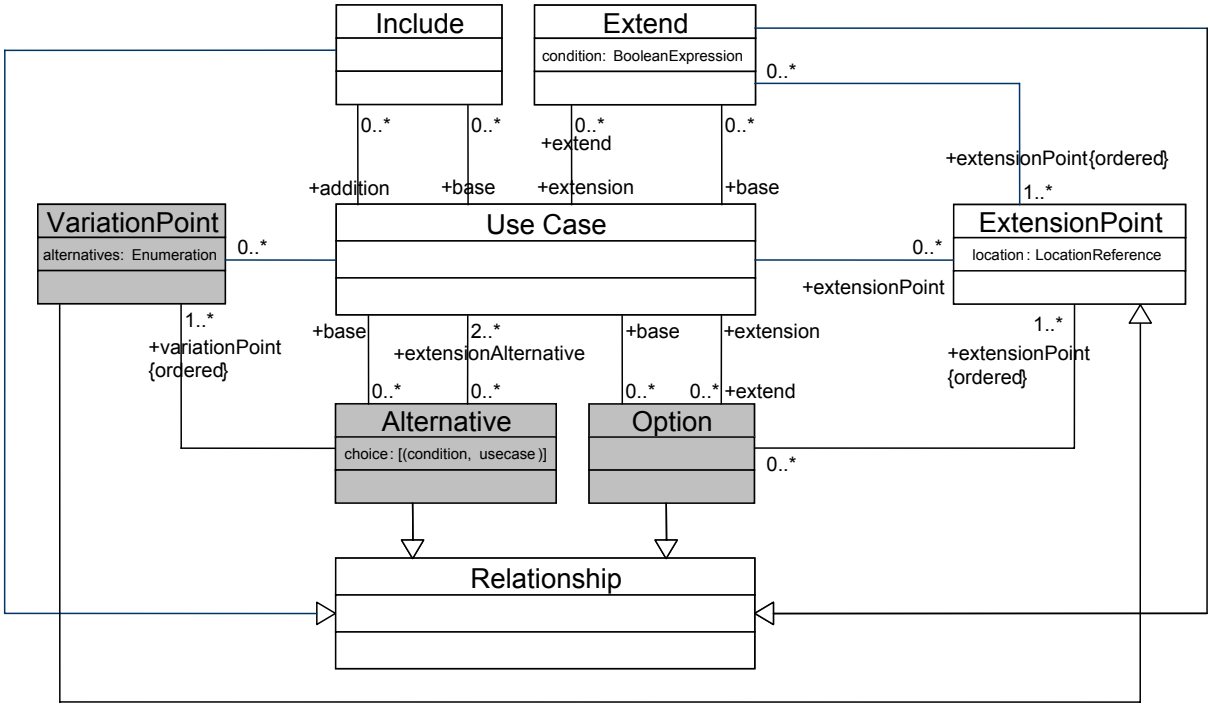


Figure 4: Extended Use Case meta-model

Table 1 shows the types of variability and the corresponding use case modeling element.

Type of variability	Modeling element
Options, linked to a condition	Extend relationship
Options, free choice	Option relationship
Alternatives, linked to a condition	Alternative relationship
Alternatives, free choice	Alternative relationship
Optional alternatives	Combination of optional and alternative relationships

Table 1: Modeling variability

Like in feature graphs, dependencies between variable and common parts are modeled implicitly. An including Use Case implies that the included Use Case is executed, too. A xor-relationship exists between alternative Use Cases that are part of the same variation point. In figure 5 an updated Use Case diagram for providing different forms of authentication is illustrated that uses the new alternative relationship to model the context correctly. Figure 6 shows the optional possibility to print the account balance.

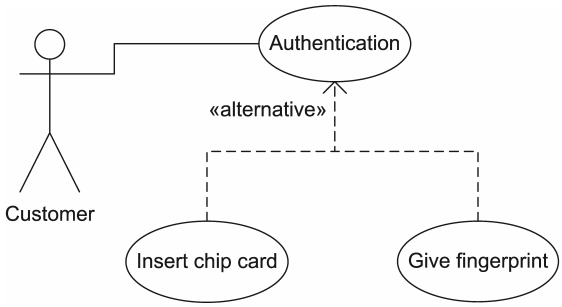


Figure 5: Example of an alternative relationship

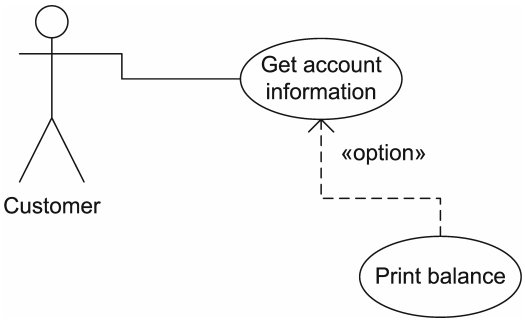


Figure 6: Example of an optional relationship

4.2 Domain variability vs. runtime variability

Considering the new variability modeling elements in the extended Use Case meta-model, the modeler is assigned to use them appropriately. That means the new modeling elements can be used on the one hand to express variability in the domain and therefore between members of the product line and on the other hand to express runtime variability (see section 3). A typical example of modeling runtime variability is illustrated in figure 7.

In this example the *Select order* procedure can be accomplished by to alternatives: either, the customer can choose to execute a bank transfer or a standing order – the customer is free of choice but can only perform one procedure at a time. Since both alternatives should be available in all applications in the domain of an automatic teller machine, this alternative relationship does not model variability between products. Though the new modeling elements give the modeler the choice to model runtime variability or variability between members of a product line, whereas a mixture of both should be avoided.

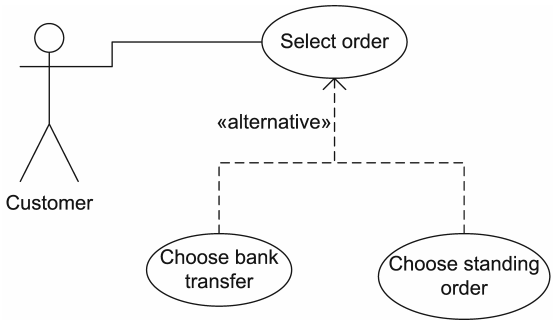


Figure 7: Modeling runtime variability with Use Case diagrams

5 Conclusion and future work

This study points out that one of the main tasks in requirements engineering for Software Product Lines is to identify and to express functional variability. This paper shows that several types of variability exist and have to be modeled. Therefore we examined briefly the feature graphs from FODA and in detail the UML Use Case diagrams with regard to their capability to express these types of variability.

Because UML Use Case diagrams are not able to model all kinds of variability, the meta-model had to be extended to fulfill the requirements that have been elicited. With the additional modeling elements the Use Case diagrams were able to express all identified types of variability with a defined semantic and in a readable and understandable way. Dependencies between Use Cases are modeled implicitly through include and alternative relationships.

It should be mentioned that Use Cases are only capable of modeling system behavior. They are not able to express static structures and characteristics of systems. Therefore a combination of feature graphs and extended Use Case diagrams can help to understand the complexity of extensive domains, processes and systems.

Our future research will be placed in validation and gaining experiences in praxis using the new modeling elements. Furthermore we are going to analyze dependencies between Use Cases that don't have direct relationships. The studies will show, whether it is necessary to consider these dependencies at all and if so, how they can be modeled in the Use Case diagrams. As a second step, we want to analyze in which way feature graph and Use Case modeling cohere and can be combined to help understanding complex processes.

References

[Eis00] Ulrich Eisenecker, Krzysztof Czarnecki. *Generative Programming – Methods, Tools, and Applications*. Addison-Wesley 2000.

[Eis01] Ulrich Eisenecker et. al. *Merkmalmodellierung für Softwaresystemfamilien*, in: OBJEKTspectrum 5/01, p. 23-30.

[Hal02] Günter Halmans, Klaus Pohl. *Modellierung der Variabilität einer Software-Produktfamilie*, Proceedings Modellierung 2002, p. 63-74. Lecture Notes in Informatics, 2002.

[Jac92] Ivar Jacobson. *Object-Oriented Software Engineering - A Use Case Driven Approach*. Addison-Wesley, 1992.

[Jac97] Ivar Jacobson, Martin Griss, Patrik Jonsson. *Software Reuse – Architecture, Process and Organization for Business success*. Addison-Wesley, 1997

[Kan00] Kyo Kang et al. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2000.

[Kan02] Kyo Kang et al. *Concepts and Guidelines of Feature Modeling for Product Line Software Engineering*. 7th International Conference on Software Reuse (ICSR), Austin, Texas, USA, pp. 62-77, April 15-19, 2002.

[Phi00] Ilka Philippow, Kai Böllert. *Erfahrungen bei der objektorientierten Modellierung von Produktlinien mit FeatuRSEB*. Technische Universität Ilmenau, 2000. Available at: http://www.theoinf.tu-ilmenau.de/ pld/pub/index.html.

[PLP01] Paul Clements, Linda M. Northrop. *A Framework for Product Line Practice - Version 3.0*. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2001. Available at: www.sei.cmu.edu/plp/framework.html.

[Sch00] Jean-Marc DeBaud, Klaus Schmidt. *A Systematic Approach to Derive the Scope of Software Product Lines*. Fraunhofer Institute for Experimental Software Engineering (IESE), Kaiserslautern 2000.

[Spe02] Andreas Speck, Matthias Clauss, Bogdan Franczyk. *Concerns of Variability in „bottom-up" Product-Lines*. Proceedings of the Workshop on Aspect-Oriented Software Development 2002.

[UML01] *OMG Unified Modeling Language*, Version 1.4, September 2001. Available at: http://www.uml.org.

Tailoring Use Cases for Product Line Modeling

Isabel John, Dirk Muthig

Fraunhofer Institute for Experimental Software Engineering (IESE)
Sauerwiesen 6, D-67661 Kaiserslautern, Germany
+49 (0) 6301 707 - 250
{Isabel.John, Dirk.Muthig}@iese.fhg.de

Abstract[1]

Use cases are used for single system requirements engineering to capture requirements from an external point of view. When utilizing use cases for product line modeling they cannot be used as is but they have to be extended with a variability mechanism. Stereotypes can be used as this variability mechanism for use case diagrams and tags can be used for textual use cases. In this paper we describe how to tailor use cases for product line modeling, describe in which situations the approach can be applied and illustrate the use case approach by an example.

1 Introduction

During the last decade reuse has been recognized as a key factor for improving software development efficiency. Product line engineering is a reuse approach providing methods to plan, control, and improve a reuse infrastructure for developing a family of similar products. The goal of product line approaches, such as PuLSE™ [3] is to achieve a planned domain-specific reuse by building a family of applications rather than developing products separately. Distinct from single-system software development, there are two main lifecycle phases: domain and application engineering. Domain Engineering constructs the reuse infrastructure, which is then used by application engineering to build the required products.

During the domain-analysis phase of domain engineering the common and varying requirements of the planned set of products are captured. Use cases are a widely accepted means to support domain understanding, find, and document user requirements but there is no generally accepted formalism that integrates variability modeling with use cases in order to do product line modeling.

Related Work

There are some approaches on how to extend use cases with variability and how to support reuse and genericity in use cases. Biddle et.al. [5] suggest to use patterns in use cases and to organize the use cases in a repository. However, they do not introduce variability. America and Wijnstra [1] describe how to use use cases in requirements engineering for product lines. Halmans and Pohl [9] show how application derifaction can be supported by product family specific use cases but they do not give a concrete notation. Gomaa [7] introduced in his method the stereotypes <<kernel>> and <<optional>> for use cases and other UML model elements for modeling families of systems. In his approach he focuses on the integration of features and use cases but does not say anything about how textual use cases should be represented. Jacobson et.al. [11] discuss how the text in use cases may involve variation points that can form the basis for a hierarchy of use cases from more abstract to more specific. They introduce “variation points” (notated as dots in use cases including a short description of the variation) into use case diagrams. They also use variation points in textual use cases (notated as highlighted text in curly brackets) to describe different ways of performing actions within a use case. They do not say anything about how variant or generic use cases can be instantiated. Our approach is related to the KobrA approach [2], which introduces variation points in the UML including use cases and supports the instantiation of generic models. The approach we describe here was already described in [12].

[1] This work has been partially funded by the Empress project (Eureka ?! 2023 Programme, ITEA project 00103)

Outline
In order to be suitable for product line modeling, commonality and variability has to be integrated and described in use-case diagrams and textual use-case descriptions. Furthermore decision modeling has to be supported. We illustrate the adaptations we made with an example, a “cruise control system” that is a part of the automotive domain. A cruise control system supports the driver in keeping a constant velocity and does real time monitoring and control of the cars speed. It exists in variants that have or have not a distance regulator, which controls the distance between the car to the car(s) ahead and behind.

The remainder of this paper is structured as follows: In section 2 we will describe how requirements engineering with use cases is done for single systems and introduce single-system use cases diagrams and use cases. In section 3, we will describe general product-line concepts and will describe how use cases can be tailored to support those concepts and in which situation the approach is used. The paper concludes with some final remarks.

2 Single System Use Cases

2.1 Requirements Engineering with Use Cases

Requirements Engineering for single systems is adressed in research and in practice since many years. Formalism which have been used for a long time are e.g. textual requirements, controlled languages, formal or semi-formal specification languages like SDL or scenarios. For some years, use cases [10] have been a means to understand, specify, and analyse user requirements that is rather often used. Use cases can document the requirements on a system from an outside or users point of view. Use cases cannot document all requirements, they normally do not deal in depth with non-functional requirements, with interfaces, and data formats. But use cases make it easier to understand what the system does and give a good means of communication about the system.

2.2 Use-Case Diagrams

A use case describes the system’s behavior under various conditions. Use cases are used during the analysis phase to identify and partition system functionality. A use case describes the actions of an actor when following a certain task while interacting with the system to be described. A use case diagram includes the actors, the system, and the use cases themselves. The set of functionality of a given system is determined through the study of the functional requirements of each actor, expressed in the use cases in the form of common interactions. So a use case diagram in UML 1.4 consists of [14]: the system to be described, the use cases within the system, the actors outside the system and the relationships between actors and use-cases or in between use cases (associations, generalization, include, and extend). Associations denote the participation of an actor in a use case, a generalization relation means that there is a specialization of one use case to another. An extend relationship indicates that an instance of a use case may be augmented by the behavior specified by another use case and the include relationship indicates that an instance of a use case will contain the behavior of another use case.Figure1 shows an example use case diagram for a cruise control system. The driver activates the cruise control system by choosing “`set velocity`”. He can also tell the system to “`keep velocity`” with help of the gas regulator if the velocity has already been set. He can also “`readopt velocity`” which will

bring the car to the fixed speed (e.g. after braking) and then continue keeping the velocity. In order to keep the velocity the system has to "`calculate velocity`" with different sensors.

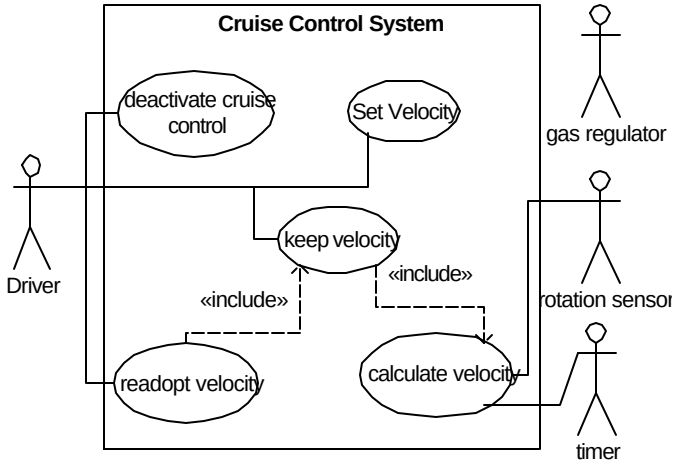


Figure. 1. A use case diagram for a cruise control system

2.3 Textual Use Cases

There is no standardized form for the content of a use case itself, the standard describes the graphical representation and the semantics of use case diagrams only. Use cases are fundamentally a text form although they can be written using flow charts, sequence charts or petri nets [6]. Use cases serve as a means of communication from one person to another, often among persons with no training in UML or software development. So writing use cases in simple text is usually a good choice. There is no general agreement on the attributes use cases should have and on the level of description of the use cases. Figure2 shows an example of a textual use case in the cruise control domain which describes the use case "`keep velocity`". The template used is a modification of the template suggested by Alistair Cockburn (see [6]). The use case is described with its actors, the triggers, which means the actors that can activate the use cases. The input and output of the use case are described and the post conditions and a success guarantee (what the user wants from the use case) and a minimal guarantee (what should in any case not go wrong) are given. The main part of the use case is the main success scenario which describes what the use case actually does.

Use Case Name: keep velocity
Short Description: keep the actual velocity value over gas regulator
Actors: driver, gas regulator
Trigger: actor driver
Precondition: --
Input: starting signal, velocity value vtarget
Output: infinit
Postcondition: vactual = vtarget
Success guarantee: vactual = vtarget
Minimal guarantee: The car keeps driving
Main Sucess Scenario:

1.) <keep velocity> is selected by actor driver
get vactual, vtarget (<Calculate Velocity>)
- compare vactual and vtarget
If vactual < vtarget : gas regulator increase velocity
- restart <keep velocity>
If vactual < vtarget : gas regulator decrease velocity
- restart <keep velocity>
else restart <keep velocity>

Figure. 2. A use case for "keep velocity" of the cruise control system

3 Tailoring Use Cases to Product Lines

In this section, the use case approach is extended to product families, that is, use cases do not longer describe the actions of an actor when following a certain task while interacting with a particular system only but summarize and integrate use cases describing analogous tasks for different products in a family into combined artifacts, product-line use cases. We describe how and why we tailored single system use cases to capture variability and commonality and describe how a decision model of those use cases can be built. Before describing in Section3.1 the required extensions to the two artifacts, use-case diagram and textual use-case descriptions, described above, the main product-line concepts are introduced.

3.1 Product Line Concepts

From an abstract point of view it is the concurrent consideration, planning, and comparison of similar systems that distinguishes product line engineering from single-system development. The intention is to systematically exploit common system characteristics and to share development and maintenance effort.

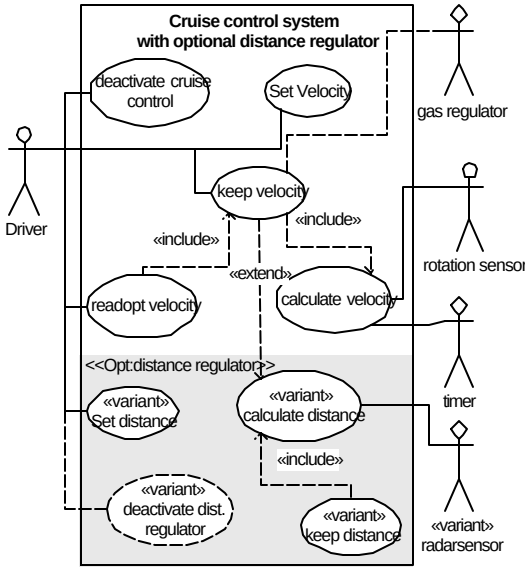


Figure. 3. Generic Use Case Diagram

In order to do so, the common and the varying aspects of the systems must be considered throughout all life-cycle stages and integrated into a common infrastructure that is the main focus of maintenance activities. Commonalities and variabilities are equally important: commonalities define the skeleton of systems in the product line, variabilities bound the space of required and anticipated variations of the common skeleton. In Product Line Modeling, modeling commonalities and Variabilities is often done with feature modeling (either with FODA [13] or FeatuRSEB [8]). But feature models often describe a static view on the systems capabilities. In order to get a dynamic view on the system use cases are a good choice as a modeling approach that should be used in addition to feature models.

When the concepts of commonalities and variabilities are applied to use cases, these concepts produce use cases that have a common story that is valid for all members of a system family with variation points that explicitly capture which actions are optional or alternatives. Of course, a use case as a whole may be optional, as well as use cases under the same label may realized totally different for some products. The common parts are modeled as all parts in a single-system context, to model variation, additional means are required. The variant use cases are instantiated during application engineering. The instantiation process is guided by a decision model, which captures the motivation and interdependencies of variation points, and produces use case artifacts as used in a single-system context (see previous section). We will illustrate, how decision modeling and instantiation can be done with use cases. These use cases approach we describe is used within PuLSE

CDA [4], the customizable domain analysis part of the PuLSE™ [3] product line framework. CDA is customizable which means, domain analysis is not always done with the same approach (like e.g. feature modeling) but customized to the context where the domain analysis approach is applied. Based on a set of fixed customization factors (c.f. [15]) we apply this use case approach in the following situation:

- **Domain characteristics:** use cases can be applied in almost all domains. They can be useful for embedded or information systems. The domain should address user-level information (e.g have a user interface). The number of possible variation points in the domain should be low to medium (2 to 100) otherwise the use cases are not readable anymore
- **Information sources**: Use cases can be derived from paper documents, experts or legacy systems. So they can be used with all information sources
- **Implementation characteristics:** As use cases are part of UML use case models are encouraged if an Object oriented modeling and implementation approach is chosen. For clarification they can also be used with other implementation approaches
- **Integratable software artifacts:** The use case approach is independent of existing software artifacts
- **Project context:** A complete set of use cases and use case diagrams makes sense in larger projects with a staff size bigger than 10. In smaller projects it might be more useful to make only an overview use case diagram but not all the use cases
- **Enterprise context:** The use case approach is independent of the organizational environment

So this approach is selected in situations where user-level information is essential for the domain model, where variability should be expressed early and explicit and where overview information (as it can be found in use case diagrams) is needed. Of course the use cases do not solely form the domain model, other approaches like feature modeling or textual requirements should be chosen as additional modeling formalisms depending on the customization..

3.2 Tailoring Use Case diagrams

In use case diagrams, any model element may potentially be variant in a product-line context. An actor is variant, for example, if a certain user class is not supported by a product. A use case is variant if it is not supported by some products in the family. A generic use case diagram for our cruise-control example is depicted by Figure 3. There, an optional distance regulator has been added, that is, four additional (and variant) use case have to be modeled by using the stereotype `<<variant>>`, as well as an additional actor, the optional radar sensor to measure the distance of a car in front. The additional feature has an impact on other use cases, which is modeled in the textual description of the affected use cases. An example for the use case "`keep velocity`" is given in the following subsection.

During application engineering, for each variant use case, it is decided whether the use case is (or is not) supported by the product to be built. The instantiation is done then with the help of the decision model. If a cruise control without distance regulator is built, all the variant use cases are removed, and the resulting use case diagram is the diagram shown in Figure1Tailoring Textual Use-Cases

In a textual use case description any text fragment may be variant. Variant text fragments are explicitly marked by pairs of the XML-like tags <variant> and </variant>. The decisions are integrated in the use case and underlined. Figure4 shows an example of this approach, the use case "`keep velocity`"Decision modeling and instantiation

Whether a use case in a use case diagram is an optional use case or whether it is an alternative to another use case is captured outside of the use-case diagram in a decision model. This is done simply because this information would overload the use-case diagram, make it less readable, and thus less useful.

The underlined questions in the use cases reflect the parts of the overall decision model that are relevant for this particular use case. This information is useful in both workproducts: integrated in the use case description, it helps to understand the use case's variability from the use case's point-of-view, in the decision model, it helps to understand the variability of the whole product family and what the impact of a particular variability is (e.g., it has an impact on this use case). Hence the overall decision model captures the relationship between the decisions related to the above generic use case diagram and the textual description in Figure4. That is, if the feature "`distance control`" is excluded the four variant use cases are removed and all variant

Use Case Name: keep velocity
Short Description: keep the actual velocity value over gas regulator <variant> by controlling the distance to cars in front </variant>
Actors: driver, gas regulator
Trigger: actor driver, <variant> actor distance regulator </variant>
Precondition: --
Input: starting signal, velocity value vtarget
Output: infinit
Postcondition: vactual = vtarget
Success guarantee: vactual = vtarget
Minimal guarantee: The car keeps driving
Main Sucess Scenario:

1.) <keep velocity> is selected by actor driver
2.) Does a distance regulator exist?
get vactual, vtarget (<Calculate Velocity>)
<variant OPT> get dactual, dtarget (<Calculate Distance>) </variant>
3.) Does a distance regulator exist?
<variant ALT 1: no; only cruise control>
- compare vactual and vtarget
If vactual < vtarget : gas regulator increase velocity
- restart <keep velocity>
If vactual < vtarget : gas regulator decrease velocity
- restart <keep velocity>
else restart <keep velocity>
</variant>
<variant ALT 2: yes, cruise control + distance regulator>
- - compare vactual and vtarget
If vactual < vtarget : gas regulator increase velocity
- restart <keep velocity>
If vactual < vtarget and atarget £ aactual: gas regulator decrease velocity
- restart <keep velocity>
If vactual < vtarget and atarget > aactual: gas regulator increase velocity
- restart <keep velocity>
else restart<keep velocity>
</variant>

Figure. 4. Generic Use Case Description

text fragments are removed from the description of the use case "`keep velocity`", as well as the first alternative for step 3 is selected. This instantiation leads to the use case description given in the previous sectionTable 1 shows an excerpt of the decision model in textual form for the use case diagram and the textual use case. As in this example the variation point is rather simple (Does a distance regulator exist or not?) the decision model also is very simple. If there is more and more complicated variability and hierarchical decisions, the decision model gets more complex and can really support the software engineers during application engineering.

4 Conclusions

In this paper, we described how use cases can be applied for modeling the requirements for a system family. Therefore, we showed how a particular single-system use case approach can be extended to capture product line information and especially variability. The approach has been illustrated by the running example "cruise control system". In our experience, use cases are a good means to elicit, structure, and represent user-level information during the requirements phase. Extended with variation points, they also enable people to easily switch from single-system requirements-engineering practices to domain analysis. The produced generic use-cases also support and guide application engineering (in particular, its requirements phase).

Thereby, each variation point must be instantiated, that is, each generic use-case with variation points is systematically transformed into a "normal" single-system use-case as it is expected by an individual customer. In general, the described approach pushes the explicit consideration of variability to the early phases of product-line development, which is required to systematically manage and evolve a product-line infrastructure. With the right decision model that documents the relationships and dependencies among variation points, the approach captures the traceability paths from variant use-case actions down to variant implementation elements..

Variation Point	Decision	Actions
1	The car has no distance regulator	Remove Use Case "`set distance`" from use case diagram
		Remove Actor "`radar sensor`" from use case diagram.....
		remove variant <variant Opt > from use case "`keep velocity`" point 2
		remove variant <Alt 2> from use case "`keep velocity`" point 3.....
	The car has a distance regulator	Remove the <<variant>> tag from all use cases in the use case diagram
		remove the <variant Opt > tag and the </variant> tag from use case "`keep velocity`" point 2.....

Table 1. Partial decision model

Acknowledgements

We want to thank Stefan Sollmann who provided the cruise control example and supported us in modeling the use case diagrams and the use cases.

References

[1] P. America and J. van Wijgerden. Requirements Modeling for Families of Complex Systems. In F. v. d. Linden, editor, Third International Workshop on Software Architectures for Product Families, LNCS 1951, Las Palmas de Gran Canaria, Spain, Mar. 2000. Springer.

[2] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wst, and J. Zettel. Component-based Product Line Engineering with UML. Component Software Series. Addison-Wesley, 2001.

[3] J. Bayer, O. Flege, P. Knauber, R. Laqua, D. Muthig, K. Schmid, T. Widen, and J.-M. DeBaud. PuLSE: A Methodology to Develop Software Product Lines. In Proceedings of SSR'99, Los Angeles, CA, USA, May 1999. ACM.

[4] J. Bayer, D. Muthig, and T. Widen. Customizable Domain Analysis. In Proceedings of the First International Symposium on Generative and Component-Based Software Engineering (GCSE '99), Erfurt, Germany, Sept. 1999.

[5] R. Biddle, J. Noble, and E. Tempero. Supporting Reusable Use Cases. In Proceedings of the Seventh International Conference on Software Reuse, Apr. 2002.

[6] A. Cockburn. Writing Effective Use Cases. Addison Wesley, 2001.

[7] H. Gomaa. Object Oriented Analysis and Modeling for Families of Systems with UML. In W. B. Frakes, editor, Proceedings of the Sixth International Conference on Software Reuse, June 2000.

[8] 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.

[9] G. Halmans and K. Pohl. Considering Product Family Assets when Defining Customer Requirements. Proceedings of PLEES'01, Erfurt, Sept. 2001. IESE-Report No. 050.01/E.

[10] I. Jacobson. Object-Oriented Software Engineering, A Use Case Driven Approach. Addison Wesley, 1992.

[11] I. Jacobson, M. Griss, and P. Jonsson. Software Reuse. Architecture, Process and Organization for Business Success. Addison-Wesley, 1997.

[12] I. John and D. Muthig. Product Line Modeling with Generic Use Cases. SPCL2 Workshop on Techniques for Exploiting Commonality Through Variability Management, August 2002, San Diego, California, 2002.

[13] K. C. Kang, K. Lee, J. Lee, and S. Kim. Feature Oriented Product Line Software Engineering: Principles and Guidelines. In Domain Oriented Systems Development – Practices and Perspectives. Gordon Breach Science Publishers, 2002.

[14] Object Management Group. OMG Unified Modeling Language Specification, Version 1.4, September 2001.

[15] K. Schmid and T. Widen. Customizing the PuLSE Product Line Approach to the Demands of an Organization. In Proceedings of EWSPT'2000, LNCS 1780, Kaprun, Austria, Feb. 2000. Springer.

Modeling behaviors in Product Lines*

Tewfik Ziadi, Loïc Hélouët, Jean-Marc Jézéquel
IRISA, Campus de Beaulieu
35042 Rennes Cedex, France
{tziadi,lhelouet,jezequel}@irisa.fr

abstract: This paper proposes a model for the definition of behavioral requirements in Product lines. These requirements are expressed by High-level Message Sequence Charts (HMSC) extended with constructs for handling variability. Then some clues for defining coherence between static and dynamic parts of the architecture with OCL are given.

1 Introduction

Recent proposals [3, 8, 2] for the development of software product lines (PL) define methodologies leveraging the UML for modeling commonalities and variability across the PL, using UML built-in extension mechanisms such as stereotypes. However, these proposals usually concentrate on static structural variability, and often neglect the behavioral aspects, that are only expressed by means of simple textual fragments. However, the analysis of the requirements domain could benefit from a formalized representation of interactions between constituents of a system.

This paper proposes the integration of behavioral aspects in a UML-based product line architecture. These interactions are described by means of High-level Message Sequence Charts (HMSC for short), a scenario language first standardized by ITU [5], that is bound to be integrated in UML 2.0. Hence, a product derived from a product line will be described by a UML model including all its features, but also by a set of scenarios describing the main interactions among parts of the system.

One of the main challenges of this work is to define behavioral variability on a scenario language. At first sight, HMSC do not seem to contain such notions. Hence, we propose to define variability through existing features such as inheritance, and to extend the notation with new constructs: variation points and optional behaviors. Another challenging task is to define a notion of coherent product lines. In order to be able to generate products from a product line, one has to ensure compatibility of features in the static model, between scenarios of the dynamic model, but also a global coherence between static and dynamic aspects of the product line.

This paper is organized as follows. Section 2 describes the static and dynamic aspects of our product line model, section 3 proposes some definitions for the coherence of a model, and section 4 concludes this work.

2 Models

This section presents a software Product Line Architecture model integrating dynamic aspects. HMSC are used to build dynamic assets for product line. In the literature many solutions (see for example [3]) are proposed for extending UML class diagram to support variability. The static model proposed hereafter uses templates, optionality, and inheritance to define variability. For illustrating the constructs introduced hereafter, we propose a small ad-hoc example: a product line

*This work has been partially supported by the ITEA project ip00004, CAFE in the Eureka Σ! 2023 Programme

for a digital camera. A digital camera comports an interface, a memory, a sensor, a display, and may propose a compression feature. The main variation points in this example concerns the presence of compression and the format of images stored, which can be parameterized.

2.1 Static Model

UML class diagrams describe sets of classes and their dependencies, but do not allow the definition of variability. Variation points can be expressed on this static model using mechanisms such as templates, stereotypes, and inheritance. A **template**[4] is the descriptor for a class with unbound formal parameters. It defines a family of classes that can be derived by binding the parameters to actual values. The **stereotype** concept provides a way of classifying model elements. A stereotype can be associated to any UML element. It allows the definition of new UML extensions without modification of the core of the language (See extension mechanisms p2-79 of the UML standard [4]).

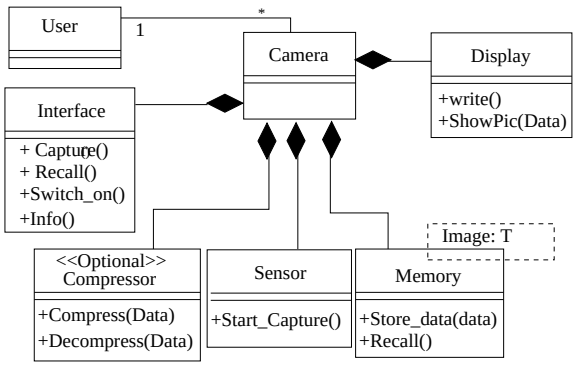


Figure 1: Class diagram for the digital camera

Variability may appear in products in different ways. Products may differ only on specific parameters (the size or color of a window, the memory available, ...). This kind of variability is called **Parameterization**. To implement parameterization, product lines must allow for the definition of generic assets with a set of parameters. Each product will then bind these parameters in a specific way. UML template classes can specify classes parameterized by a type.

Another kind of variability called **optionality** appears when a part of a product (feature, functionality,...) can be implemented or ignored by products from a family. Product line models gather information about a set of products. They must therefore be more generic than product models, and include elements from all the products of the family they represent. Some elements will be common to all products, but for each product, some **optional** elements will be omitted. To show optionality information we use a specific stereotype «Optional», that can be associated to any part of the class diagram.

In addition to parameterization and optionality, standard constructs such as inheritance can be used to define variants. In [6], for example, variants of a product are built from a model leveraging Creational Design Patterns. Example of Figure 1 shows the static model of a digital camera Product line, in which class *Compressor* is optional and class *Memory* is parameterized with image type. Other parameters, such as the size of a buffer, can be instantiated using model transformations.

2.2 Dynamic Model

In addition to the static aspects of products, a user of a product line may also want to express **behavioral variations**. A family of network softwares, for example, may use different commu-

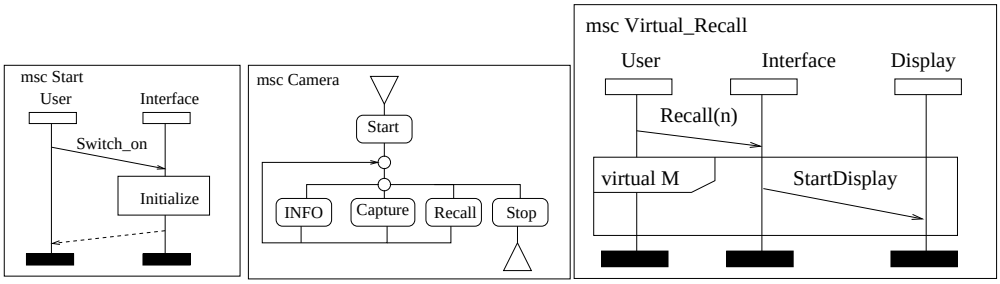


Figure 2: a- bMSC b- HMSC c- Virtual MSC

nication protocols, and components of a system may cooperate differently in different products. Furthermore, a certain coherence between static and dynamic aspects of a product model should be ensured by the product line. We propose to define dynamic aspects of products with Message Sequence Charts, and to use the OCL [11] for the expression of coherence properties.

Message Sequence Charts is a scenario language standardized by ITU [5]. The latest evolution of the language is very close from the future sequence diagrams of UML 2.0. MSC is composed of two kinds of diagrams, basic MSC (bMSC) and High-level MSC (HMSC). They define a set of basic charts composed by means of alternative, parallel composition, and iteration operators. A bMSC describes communications between entities of a system called instances. Diagram of Figure 2-a represents a bMSC defining communications between instances $User$ and $Interface$. HMSC $Camera$ of figure 2-b describes a set of scenarios starting with the interactions defined in bMSC $Start$, in which the communication patterns defined by $INFO$, $Capture$, and $Recall$ can be repeated an unlimited number of times before the occurrence of $Stop$. Interested readers can consult [10] for a complete presentation of the language.

Message Sequence Charts are equiped with a semantics based on process algebra [7], and their properties have been well studied during the last decade. Furthermore, it is interesting to describe behaviors at a high level of abstraction with a visual and intuitive formalism that allows formal manipulations. The main idea for integrating MSC into a PL approach is to associate a set of typical behaviors to a product model derived from the PL. Behaviors from a product to another differ only by small details. Hence managing different and disjoint sets of scenarios for each product would be a very heavy construction, and would not capture the notion of commonality and variability inherent to a product line approach.

Unfortunately, in its current form, MSC is not equiped with features allowing the expression of variability and commonality. We propose to use the inheritance mechanisms of [9], and to extend the language with some new constructs for expressing variability. Since HMSC are supposed to be a part of the new UML standard, variation constructs can be easily defined by specializing alternative and MSC references with ad hoc stereotypes. The main idea for integrating scenarios into a product line is to maintain a set of generic scenarios with variation points, and scenario assets. From this behavioral asset base, a set of "terminal" scenarios can be derived for each product by instanciation of variation points, and used for simulation, documentation, test purposes, etc. Note that the semantics of variable scenarios is defined as the semantics of derived scenarios. A terminal scenario derivation is obtained by syntactic replacement of a variable part, or by abstraction of an optional part. Hence, the semantics of a scenario can be very different from a product to another. If semantics preservation for some parts of a MSC is required in all variants, then additionnal constraints can be attached to variation points. One may for example require that all possible choice for variation point $Comp$ in MSC $Capture$ of Figure 4 impose an order from an event performed by the $Sensor$ to an event performed by the $memory$.

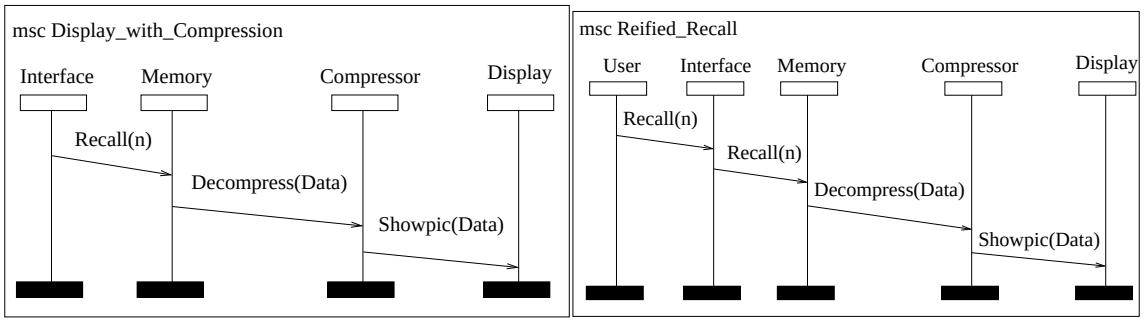


Figure 3: a) bmsc DisplayWithCompression b) bMSC Reified_Recall

A **virtual part** of a bMSC can be refined to obtain another bMSC. The example of Figure 2-c shows a MSC *Virtual_Recall* with a virtual part *M*. The derived MSC of Figure 3-b is obtained by applying the redefinition proposed by [9]: "`msc Redefined_Recall = Virtual_Recall with M redefined by Display_With_Compression`". In a product, a virtual part can be replaced by another MSC. When a virtual part is not redefined, the behavior contained in the virtual frame is supposed to be the default behavior.

A **variation point** in a MSC has the common meaning in PL approaches: for a given product, only one alternative defined by the variation point will be present in the scenario. Note that this variation can not be expressed by the choice operator of MSC. For example, the choice of Figure 2-b allows the conformance to the behavior described by *Capture* at the first occurrence of the choice, and then conform to *Recall* at the next occurrence of the same choice. Replacing this choice by an alternative between *Capture* and *Recall* would allow for the derivation of two products: one allowing image capture, and another allowing image displaying. A variation point is depicted by means of a rectangular frame, labeled by a variation name. Variable behaviors are separated by a dashed line. The example of Figure 4-b contains a variation point *Comp*, and two possible behaviors describing how data is stored depending on the presence of a compression device.

[3] defines **optionality** for instances and messages of sequence diagrams. We propose to introduce optional instances in MSC and a new construct called optional behaviors. Optional instances are represented by a dashed life line, and their name is tagged with the «optional» keyword. When an optional instance is not present, any incoming or outgoing message should be removed from the resulting behavior. An optional part can be included or removed from a bMSC the same way. Figure 4-b contains an optional instance *Compressor*. An optional behavior in defined by a rectangular frame labeled by an option name. Figure 4-a contains an optional part *M* that makes the memory information display optional.

3 Constraints on PL models

Bass et al [1] define product line architecture as a set of components, connectors, and additional constraints. Obviously, a product line is more than a collection of models, and has to ensure a certain coherence between it components. The PL architecture model we propose is composed of a static part (a stereotyped class diagram), of a set of behavioral requirements (HMSC) and of a set of constraints expressed in OCL. The model associated to a product can be considered as a sub-model of the PL class diagram, and as a set of HMSC, obtained by choosing a specific variant for each variation point, and satisfying some constraints. These constraints can be applied to both static and dynamic parts, and can be of different kind:

Generic constraints are constraints that apply to any product line, and define structural proper-

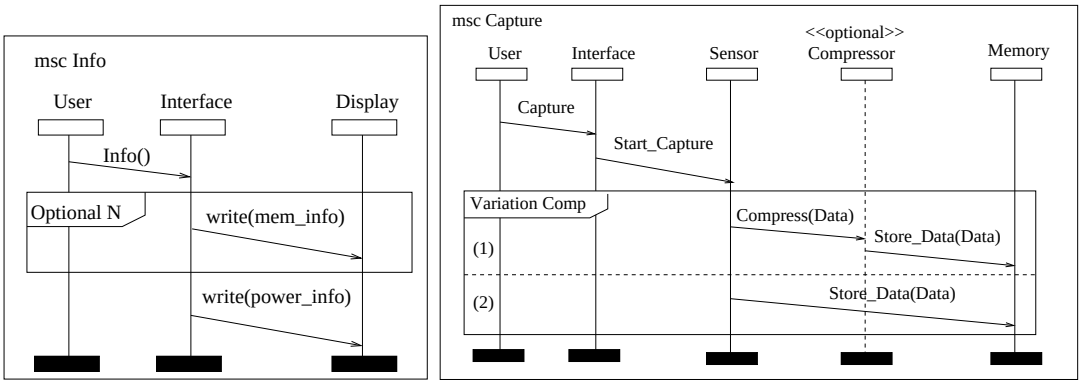


Figure 4: a) optional behavior b)optional instances and variation point

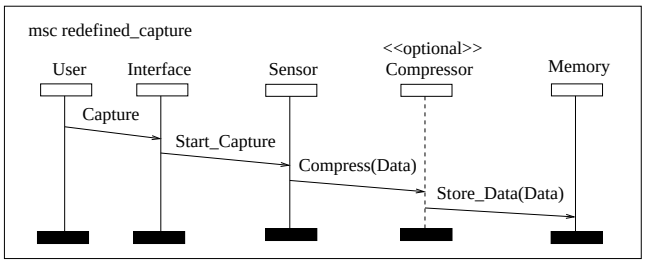


Figure 5: MSC with options and variation points fixed

ties. An example of such constraint is the dependency constraint, that forces non optional elements to depend only on non optional elements. This property can be expressed by the following OCL meta-level constraint, where `isStereotyped(S)` is an auxiliary OCL operation indicating if an element is stereotyped by a string S:

context Dependency
–Warning if a mandatory element depend on the optional one
inv:self.supplier → **exists**(S:ModelElement | S.isStereotyped("optional") **implies** self.client → **forAll**(C: ModelElement | C.isStereotyped("optional"))

Generic constraints can also concern the requirement part of the PL model. For example, instances that appear as optional in bMSCs should be instances of classes that are described as optional in the static model. When the UML 2.0 meta-model contains a definitive version of sequence diagrams it will be possible to express such a constraint as an OCL meta-level constraint. **PL specific constraints** specify properties and dependency relationships between architecture elements. Such a constraint can be for example a presence constraint between two optional classes $C1$ and $C2$. This can be expressed by the following OCL meta-level constraint:

context Namespace
– Presence of class C1 in a such Namespace requires the presence of class C2 in the same Namespace
inv : presenceClass(self, C1) **implies** presenceClass(self,C2)
presenceClass(N:Namespace, class:Class):boolean
post: result = N.ownedElement → **exists**(C:ModelEl ement|C.oclIsTypeOf(Class) **and** C.name=class.name)

PL specific constraint can also involve the dynamic part of the model. For example, on the digital camera Product Line, if class *compressor* is not instantiated, the virtual part of MSC

Virtual_Recall must not be redefined by MSC *Display_With_Compression*. Note that the coherence notion between static and dynamic part does not only concern optional instances, as the presence of components or functionalities of a system can influence the way communications are designed for the rest of the system.

4 Conclusions

We have introduced a formal dynamic representation of behaviors in UML-based product line architectures. Dynamic behaviors are represented by HMSC, where variability is introduced by constructs such as optional instances, optional parts, inheritance, and variation points. Constraint for the coherence of a model can be expressed with the OCL language. One of the main advantages of this approach is that static parts, dynamic parts, and constraints on a product line are expressed with UML elements, and therefore can be easily supported by a UML CASE tool. From this work, one of the challenging point is to derive automatically and efficiently a product model.

References

[1] L. Bass, P. Clements, and R. Kazman. *Software architecture in Practices*. The SEI Series in Software Engineering. Addison-Wesley, 1998.

[2] F. Boisbourdin, B. Pronk, J. Savolaien, and S. Salicki. System family requirements classification and formalisms. Technical Report WP2-0011-01, ESAPS-ITEA, november 2000. ESAPS deliverable.

[3] J.C Duenas, W. El Kaim, and C. Gacek. Style, structure and views for handling commonalities and variabilities - esaps deliverable (wg 2.2.3). Technical report, ESAPS Project, 2001.

[4] Object Management Group. Omg unified modeling language specification, version 1.4, September 2001.

[5] ITU-T. Z.120 : Message sequence charts (MSC), november 1999.

[6] J.M Jézéquel. Reifying variants in configuration management. *ACM Transaction on Software Engineering and Methodology*, 8(3):284–295, July 1999.

[7] M. Reniers and S. Mauw. High-level message sequence charts. In A. Cavalli and A. Sarma, editors, *SDL97: Time for Testing - SDL, MSC and Trends*, Proceedings of the Eighth SDL Forum, pages 291–306, Evry, France, Septembre 1997.

[8] Andreas Reuys, Klaus Pohl, Cristina Gacek, et al. System family process frameworks. Technical Report ESI-WP2-0002-04, ESI, december 2000. ESAPS technical report.

[9] E. Rudolph, P. Graubmann, and J. Grabowski. Message Sequence Chart: composition techniques versus OO-techniques - 'tema con variazioni'. In R. Bræk and A. Sarma, editors, *SDL'95 with MSC in CASE*, Proceedings of the Seventh SDL Forum, pages 77–88, Amsterdam, 1995.

[10] E. Rudolph, P. Graubmann, and J. Grabowski. Tutorial on Message Sequence Charts. *Computer Networks and ISDN Systems*, 28(12):1629–1641, 1996.

[11] J. Warmer and A. Kleppe. *The Object Constraint Language-Precise Modeling with UML*. Object Technology Series. Addison-Wesley, 1998.

DECIMAL: A Requirements Engineering Tool for Product Families

Prasanna Padmanabhan
Department of Computer Science
Iowa State University
Ames, IA, 50010
ppadmana@cs.iastate.edu

Robyn R. Lutz*
Computer Science Department
Iowa State University and
Jet Propulsion Laboratory
rlutz@cs.iastate.edu

Abstract

Today, many software organizations are utilizing product families as a way of improving productivity, improving quality and reducing development time. When a new member is added to a product family, there must be a way to verify whether the new member's specific requirements are met within the reuse constraints of its product family. The contribution of this paper is to demonstrate such a verification process by describing a requirements engineering tool called DECIMAL. DECIMAL is an interactive, automated, GUI-driven verification tool that automatically checks for completeness (checking to see if all commonalities are satisfied) and consistency (checking to see if dependencies between variabilities are satisfied) of the new member's requirements with the product family's requirements. DECIMAL also checks that variabilities are within the range and data type specified for the product family. The approach is to perform the verification using a database as the underlying analysis engine. A pilot study of a device-driver product family for virtual environments is also described which investigates the feasibility of this approach by evaluating the tool.

1. Background and Related Work

Software product families are becoming a key software production paradigm in the industry today. A *product family* is a set of systems with very similar requirements and a few key differences. Product families offer a high degree of product flexibility. In other words, companies can now build tailor-made systems to cater to the needs of particular market segments or customer groups. What makes product families succeed from the developer's point of view is that the common features shared by the products and well-defined variabilities can be exploited to achieve substantial cost savings through reuse [11].

Requirements verification for product families includes not only the process of verifying product-family-wide requirements, but also product-specific requirements when, later, a new product is added to the product family [11]. In this paper, we address the issue of verifying the new member's requirements against the product family's requirements. We have implemented our approach in an automated verification tool called DECIMAL (DECIsion Modeling AppLication). DECIMAL performs the following kinds of checks:

Completeness Checking: Completeness is defined as relative correctness [12]. In our work, we say that a new family member's requirements are complete with respect to the product family's requirements when the new member satisfies all the commonalities.

Consistency Checking: Two products are consistent when they exhibit the intended relationships [12]. In the case of product families, we check to see if the new member satisfies all dependency constraints among variabilities. For example, if the number of buffers must equal the number of sensors, then building a new system with three sensors requires a choice of three buffers.

Range and Type correctness checking: This ensures that values of variabilities chosen for the new member are within the range and are of the same data type as specified for the product family.

We believe that our work is important for the following reasons:

* This author's research was supported in part by National Science Foundation Grants CCR-0204139 and CCR-0205588.

- There is a strong need for easy-to-use tools to perform automated analysis. This has been pointed out by the 4th Product Line Practice Workshop [1] and by Zave [15]. While economic scoping tools exist, e.g. [3], [10], there are no tools available to automatically check for complex dependency relationships among variabilities, although these are often difficult to verify manually.
- Most existing approaches to decision modeling [6], [14] enforce an ordering on the decisions. Such an ordering can violate constraints if the choice of a variability violates the choice of a variability made earlier. DECIMAL does not impose such an ordering but also supports approaches that do.
- DECIMAL handles evolution of a product family by supporting near-commonalities (commonalities that are true for almost all members of the family).

The rest of the paper is organized as follows. Section 2 describes DECIMAL's capabilities. Section 3 provides an evaluation of DECIMAL. Section 4 presents conclusions and envisioned future work.

Previous work to check for consistency among product family requirements includes [4, 9]. Gomaa et al. [4] describe a domain model that captures mutually exclusive or mutually inclusive inter-feature and feature/object dependencies at the design level. Savolainen and Kuusela [9] describe a formal basis for consistency management based on a product requirements refinement hierarchy. Our approach to requirements engineering follows the FAST technique described by Weiss and Lai [14] and the domain modeling method prescribed by SPC [12].

2. Description of the Tool

DECIMAL is applicable to both domain engineering and application engineering of product families. DECIMAL is a Microsoft Windows application written entirely in Microsoft Visual Basic. It has a rich graphical user interface with wizards to drive the user through the analysis process.

The example used here [8] is that of a Virtual Reality device-driver product family called VRJuggler [2], [5], a VR research project at the Virtual Reality Application Center at Iowa State University.

2.1 DECIMAL in domain engineering

Product-line wide requirements are specified in the domain engineering phase. This includes specifying commonalities, variabilities and dependency constraints between variabilities.

In DECIMAL, each commonality is given a name and a short textual description and entered using a form. To add variabilities the user specifies for each parameter of variation [14] its associated data type, binding time and a default value, in addition to a name and a short textual description Fig 2.1). Constraints, which are dependencies among variabilities, are specified using predicate logic. Fig 2.2 shows an example of the Rules Editor for specifying constraints. The constraints can be complex rules with Boolean operators NOT, AND and OR.

2.2 DECIMAL in application engineering

Requirements for individual members of the family are specified in the application engineering phase and then verified. In DECIMAL, a user-friendly wizard drives the developer through this process. During this process, the user has an option of specifying a commonality as a near-commonality for that member.

Once the requirements for one or more new members have been specified (Fig. 2.3), the developer can subsequently perform the analysis process to verify their requirements. Fig 2.4 shows the results of the consistency checks for the new members (describes “what family member did not satisfy what constraint”). The other two analyses, not shown here, are completeness checking (describes “what family member did not satisfy what commonality”) and range and type correctness checking (describes “what family member contains variabilities out-of-range and/or of incorrect data type”).

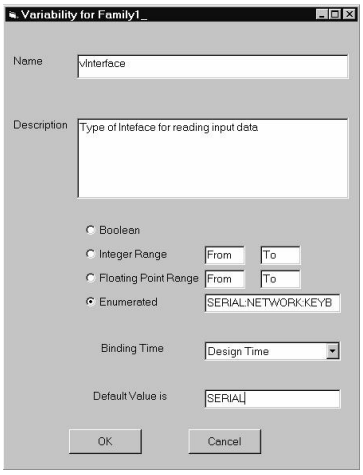


Figure 2.1: Variabilities Editor

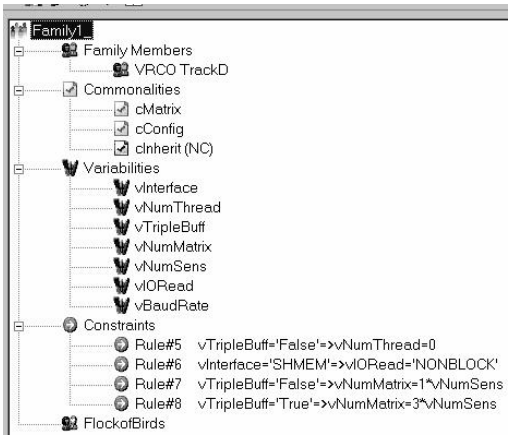


Figure 2.3: Requirements definition, after new members have been created.

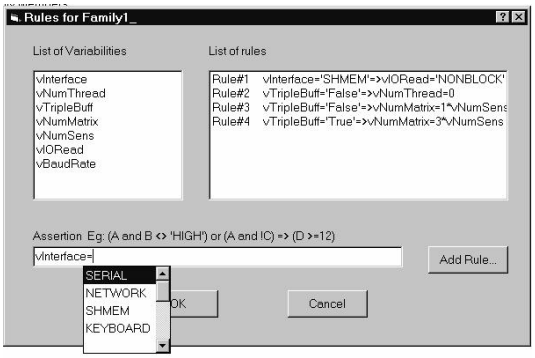


Figure 2.2: Rules Editor

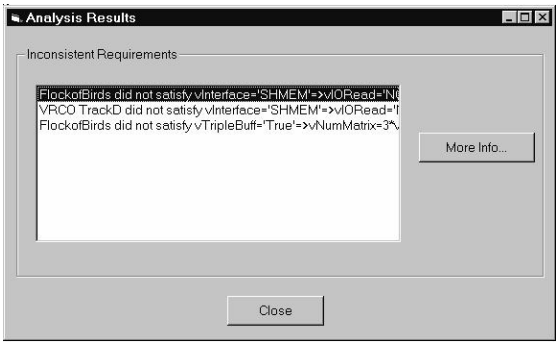


Figure 2.4: Analysis results

3. Approach

The previous section described DECIMAL as the user perceives it. In this section, we describe the implementation of DECIMAL, and the approach used to perform the analysis.

The artifacts produced by the domain engineering phase – i.e. the commonality analysis and the decision model – become the input for the application engineering phase, where the requirements of a new member are validated for consistency and completeness.

3.1 Description of Domain Engineering Activities

The initial domain engineering activities are the commonality analysis and the decision model. The commonality analysis of the VRJuggler positional device drivers (drivers for positional input devices such as gloves or wands that provide information about the location of the user in 3D space) is explained in detail in [8]. Table 3.1 shows some parameters of variation for the product family.

Dependency constraints among variabilities are also identified and specified in DECIMAL. Some examples of constraints for the Positional Device Driver Product Family are:

D1: Devices that do not use triple-buffering do not require device sampling threads to read input data.

```
vTripleBuff = 'False' => vNumThread = 0
```

D2: Drivers that read from shared memory do not use a blocking call (Instead, they poll periodically to see if data is available in the shared memory pool).

```
vInterface = "shared memory' => vIORead = 'Non-Blocking'
```

D3: If a triple buffering algorithm is not used, then the number of 4X4 matrices into which data is read equals the number of sensors. (Each sensor reads in data into just 1 matrix).

```
vTripleBuff = 'False' => (vNumMatrix = 1 * vNumSens)
```

D4: If a triple buffering algorithm is used, then the number of 4X4 matrices into which data is read is three times the number of sensors (Each sensor reads in data into 3 matrices).

```
vTripleBuff = 'True' => (vNumMatrix = 3 * vNumSens)
```

Table 3.1 Shows Parameters of Variation for the VRJuggler positional device drivers product family

Parameter	Meaning	Value Space	Binding Time	Default
vInterface	Type of interface for reading input data.	{SERIAL, NETWORK, SHMEM, KEYBOARD}	Specification	SERIAL
vNumThread	The number of threads to sample device data.	[0..2]	Specification	0
vNumMatrix	The number of 4x4 matrices to store the positional data read from the sensors.	[1..12]	Specification	4
vTripleBuff	Whether or not a triple buffering algorithm is used to read and store input data.	True, False	Specification	True
vNumSens	The number of sensors	[0..108]	Specification	0
vIOread	Type of system call used to read input data from the device.	{BLOCK, NONBLOCK}	Specification	BLOCK

3.1 Description of Application Engineering Activities

DECIMAL stores the domain model data input by the user, in a backend SQL Server database in three tables – a commonalities table, a variabilities table and a constraints table. The constraints table deserves special attention here. The constraints table not only contains a list of constraints in predicate logic as entered by the user, but also contains their corresponding SQL query representations.

For example, a rule that specifies that if the value chosen for variability V1 is 1, then the value chosen for V2 must be 'b' or the value chosen for V3 is 'False'. This rule has the form V1 = '1' => V2 = 'b' or V3 = 'False', and its corresponding SQL query notation is SELECT * FROM vrj WHERE not (not (V1 = '1') or (V2 = 'b' or V3 = 'False')). The name of Table 3.1 is "vrj"; VRCO TrackD and Flock of Birds are commercial VR tracking devices.

Table 3.1 Shows requirements for new members

Fno	Fmemname	vInterface	NumThread	vNumMatrix	vBaudRate	vTripleBuff	vNumSens	vIOread	c4X4	cConfig	cInherit
1	VRCO TrackD	SHMEM	4	6	38400	False	6	BLOCK	Yes	No	Yes
1	Flock of Birds	SERIAL	3	3	38400	True	2	NON-BLOCK	No	Yes	Yes

Once the data to be analyzed is loaded into the SQL database, the tool now performs three kinds of checks.

3.1.1 Consistency checking

To detect inconsistencies, the SQL query representations of the constraints are executed one by one. The rows returned by the query indicate the family members that are inconsistent. For example, to check for constraint D1 above, executing the query SELECT * FROM vrj WHERE (not (not (vTripleBuff = 'False') or vNumThread = 0)) returns the first row of the table. This indicates that the new member VRCO TrackD has inconsistent requirements in that it did not satisfy the rule vTripleBuff = 'False' => vNumThread = 0.

3.1.2 Completeness check

Detecting incompleteness (whether there exists a family member that did not satisfy a particular commonality), is very straightforward. For example, the commonality cConfig states, "All the configuration functions in all drivers request the following data - port data, baud rate and instance name." To check that this commonality is satisfied in all family members, an SQL query is executed. Executing the SQL query, SELECT * FROM vrj WHERE cConfig = 'No', returns the row corresponding to VRCO TrackD. This indicates that VRCO TrackD has an incomplete requirement with respect to the commonality cConfig.

3.1.3 Range and type correctness checking

DECIMAL performs range and type correctness checking to verify that the values of variabilities selected for the new member fall in the range and are of the same data type as specified for the variabilities in the requirements specification of the product family. This functionality is programmatically built into DECIMAL's front-end. For example, vNumSens must be an integer in the range 0-108. If a new member of the product family is being constructed with the value of the variability equal to 120.5, then DECIMAL will flag both an out-of-range and incorrect data type in error messages. The developer can then correct the error(s) and re-run DECIMAL to check the value.

3.1.4 Handling of near-commonalities in DECIMAL

Near-commonalities are commonalities that are true for almost all family members. A commonality usually becomes a near-commonality when the product family evolves. For example, at a later stage in the life of a product family, a new member might be added to the family (often a new prototype or testbed), that may not satisfy all the commonalities. Handling near-commonalities is not straightforward because near-commonalities can be handled either as variabilities or as constrained commonalities which are invariant over the domain [7].

DECIMAL supports the representation and checking of near-commonalities based on a solution proposed by Thompson and Heimdahl [13]. A new family member can designate a commonality as a near-commonality in its table, in which case the completeness check for that member will not be violated by failure to satisfy that near-commonality.

4. Evaluation of DECIMAL

DECIMAL was evaluated using two real life problems. The first application was the VRJuggler product family of positional device drivers described above. DECIMAL could adequately capture its commonalities, variabilities and constraints (explained in detail in [8]), and successfully performed the analyses.

The second problem was to see how DECIMAL modeled the feature interaction problem (detailed description is in [8]). Feature interaction is a concern when modeling telecom switches with many features as a product family, some of which might possibly interact in undesirable ways. We tried to model existing feature interaction resolution techniques in DECIMAL. We found that DECIMAL can represent several of the resolution policies as dependency constraints, such as mutual exclusion, dependency between features, conditional dependency between features and signaling conflicts between features (when two features react to the same signal). However, from our experiment, it became clear that DECIMAL cannot capture the temporal notions (features depending on time sequences of operations), nor certain run-time parameters like memory and processor usage.

5. Conclusions and future work

This paper describes DECIMAL, a tool for the requirements engineering of a product family that

supports completeness, consistency and range and type checking between a specified product family and an envisioned new system in the product family. The most significant contribution of the tool is its capability to check that dependency relationships among the values of the variabilities are maintained in the new system. In addition, DECIMAL handles near-commonalities which are often difficult to model. Currently, there are significant research results in product family engineering that have not been implemented in automated tool support. Industrial workshops [1] have indicated a strong need for improved tools. DECIMAL provides a step toward filling that need.

We envision future work in two areas. Firstly, we would like to extend DECIMAL to explicitly handle product sub-families (with additional commonalities or variabilities) [13]. Secondly, we would like to extend DECIMAL's capability to check that a set of dependency constraints are internally consistent before applying them.

6. References

[1] Bass, L., Clements, P., Donohoe, P., McGregor, J., Northrop, L., *Fourth Product Line Practice Workshop Report*, CMU/SEI-2000-TR-002, Software Engineering Institute, Carnegie Mellon University, 1999.

[2] Bierbaum, A., VR Juggler: "A Virtual Platform for Virtual Reality Application Development," M.S. thesis, Iowa State University, Ames, IA, 2000.

[3] DeBaud, J., TrueScope, http://www.truescope-tech.com/product_overview.htm

[4] Gomaa, H., Kerschberg, L., "Domain Modeling for Software Reuse and Evolution," *Proc. IEEE International CASE Conference*, Rio de Janeiro, Brazil, 1995

[5] Just, C., Bierbaum, A., Baker, A., Cruz-Neira, C., "VR Juggler: A Framework for Virtual Reality Development," presented at the *2nd Immersive Projection Technology Workshop (IPT98)*, Ames, May 1998.

[6] Lam, W., "A Case Study of Requirements through Product Families," *Annals of Software Engineering*, March 1998, pp. 253-277.

[7] Lutz, R.R., "Extending the Product Family Approach to Support Safe Reuse," *The Journal of Systems and Software*, Vol. 53, 3, Sept., 2000.

[8] Padmanabhan, P., "DECIMAL: A Requirements Engineering Tool for Product Families," M.S. thesis, Iowa State University, Ames, IA, USA, 2002.

[9] Savolainen, J., Kuusela, J., "Consistency Management of Product Line Requirements, Requirements Engineering," *Fifth IEEE International Symposium on Requirements Engineering*.

[10] Schmidt, K., Shank, M., "PuLSE-BEAT - A Decision Support Tool for Scoping Product Lines," 3rd *International Workshop on Software Architectures for Product Families*, Mar. 2000, pp. 197-203.

[11] Software Engineering Institute, "A Framework for Software Product Line Practice - Version 3.0," http://www.sei.cmu.edu/plp/frame_report/introduction.htm, Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, (current March 2002).

[12] Software Productivity Consortium, *Reuse-Driven Software Processes Guidebook*. SPC- 92019-CMC, v. 09.00.03., 1993

[13] Thompson, J. M, Heimdahl, M. P. E., "Extending the product family approach to support n-dimensional and hierarchical product lines," *Proc. Fifth International Symposium on Requirements*, Aug. 2001.

[14] Weiss, D. M., Lai, C. R, *Software Product Line Engineering: A Family-Based Software Development Process*, Addison Wesley, Reading, Mass., 1999.

[15] Zave, P., "Requirements for Evolving Systems: A Telecommunications Perspective," *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering*, IEEE Computer Society, 2001, pp. 2-9.

A Formal Method for the Analysis of Product Maps

Thomas Eisenbarth Rainer Koschke Daniel Simon
Universität Stuttgart
Breitwiesenstraße 20–22
70565 Stuttgart, Germany
{eisenbarth,koschke,simon}@informatik.uni-stuttgart.de

Abstract

During the initiation and evolution of a software product line, developers make use of *product maps* [3] for scoping and requirements engineering. We present how to utilize a formal mathematical method for analyzing the product maps for an anticipated software product line. The result of the method reveals variabilities and commonalities of products, as well as dependencies between products.

1 Introduction

During the initiation phase of a software product line, developers build a product map [3], i.e., a table that contains the features of the products of the product line. The manual analysis of a large product map is difficult. In this paper, we describe a mathematically founded method that helps in understanding the provided facts and the relations between individual products and features of the software product line.

2 Approach: Formal Concept Analysis

This section describes how formal concept analysis can be used to automatically identify commonalities and variabilities in product maps.

Formal concept analysis is a mathematical technique for analyzing binary relations (in our case, this is the product map). The mathematical foundation of concept analysis was laid by Birkhoff [2] in 1940. For a full description of the mathematical background of formal concept analysis, we refer to [6].

The product map lists the set of features for each product. It is basically a binary relation $\mathcal{M} \subseteq \mathcal{P} \times \mathcal{F}$ between a set of products $\mathcal{P}$ and a set of features $\mathcal{F}$. Figure 1 shows a product map.

Based on the product map, the *common features* of a set of products $P \subseteq \mathcal{P}$ can be identified as $\sigma(P)$:

$$\sigma(P) = \{f \in F \mid (p, f) \in \mathcal{M} \text{ for all } p \in P\} \quad (1)$$

For instance, the common features of *Siemens* and *Casio* in Fig. 1 are *mail*, *sms*, and *infra*.

	Features						
Products	mail	wap	sms	light	blue	infra	serial
Siemens	×	×	×			×	
Casio	×		×	×		×	
Handspring	×			×		×	×

Figure 1: A small product map.

Analogously, the set of *common products* $\tau(F)$ that possess a given set of features $F \subseteq \mathcal{F}$ can be described as:

$$\tau(F) = \{p \in P \mid (p, f) \in \mathcal{M} \text{ for all } f \in F\} \tag{2}$$

E.g., the products that share features *sms* and *light* in Fig. 1 are *Casio* and *Handspring.*

In product maps, we are interested—amongst other things—in sets of products that have all features in common. This leads us to the notion of *concept* in formal concept analysis. A pair $c = (P, F)$ is called *concept* iff $F = \sigma(P)$ and $P = \tau(F)$, i.e., every product has all features and every feature is offered by every product in this concept. Intuitively, this definition gives us a maximally large rectangle of filled product map cells, where row and column permutations are allowed. For example, *C1 = ({Handspring, Casio}, {mail, light, infra})* and *C2 = ({Handspring}, {mail, sms, light, infra, serial})* form concepts in Fig. 1.

Comparing the two concepts C1 and C2, we notice that the products in C2 are a subset of the products in C1, and the features in C1 are a subset of the features in C2. We state that concept C1 is more general than concept C2 because it has fewer features. Because fewer features are required, more products offer these features. Formally, we can define this relationship between two concepts $\leq$ as follows:

$$(P_1, F_1) \leq (P_2, F_2) \Leftrightarrow P_1 \subseteq P_2 \tag{3}$$

or, dually, by

$$(P_1, F_1) \leq (P_2, F_2) \Leftrightarrow F_1 \supseteq F_2 \tag{4}$$

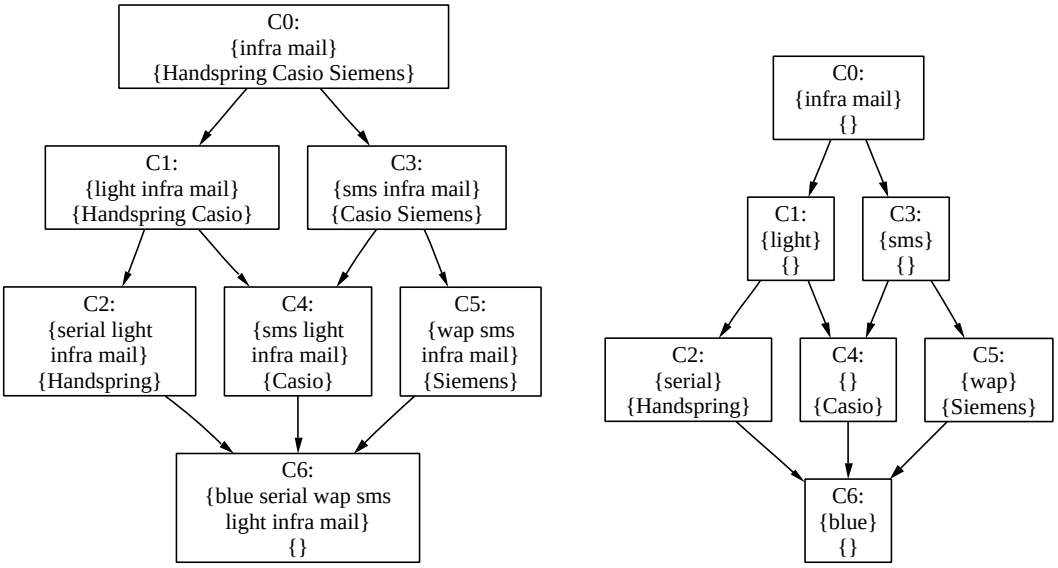
If we have $c_1 \leq c_2$, then c_1 is called a *subconcept* of c_2 and c_2 is called *superconcept* of c_1. The relationship $\leq$ is only a partial order because some concepts are incomparable. For example, $C2$ and $C5 = (\{Siemens\}, \{mail, wap, sms, infra\})$ are incomparable.

The set $\mathcal{L}$ of all concepts and the partial order $\leq$ form a complete lattice

$$\mathcal{L}(C) = \{(P, F) \in 2^{\mathcal{P}} \times 2^{\mathcal{F}} \mid F = \sigma(P) \text{ and } P = \tau(F)\} \tag{5}$$

which is called *concept lattice.* It can be depicted as a directed acyclic graph whose nodes represent the concepts and whose edges denote the relation $\leq$. For instance, Fig. 2(a) shows the lattice derived from the product map in Fig. 1.

To improve the legibility, we use the *sparse representation* of the lattice. Each product is attached to the unique concept c determined by $c = s(p)$, where $s(p)$ computes the



(a) Complete lattice

(b) Sparse representation

Figure 2: Concept lattices for the product map in Fig. 1.

largest (with respect to $\leq$) concept containing p. Likewise, each feature f is attached to the unique concept determined by $c = s(f)$, where $s(f)$ computes the smallest (with respect to $\leq$) concept containing f. Figure 2(b) shows the sparse representation of the lattice in Fig. 2(a). Note that, if required, the complete representation can be retrieved from the sparse representation: all features of a product p lie on the paths from $s(p)$ to the top concept. Further, all products offering a feature f can be found on the paths from $s(f)$ to the bottom concept.

Benefits from concept analysis

The concept lattice is a precise and semantically equivalent representation of the original product map, i.e., there is exactly one lattice that corresponds to the product map and vice versa—modulo row and column permutations. In other words, the product map and the lattice are two different views on the same relation. However, relationships between products and features are more obvious in the lattice than in the product map. This can be illustrated by comparing the product map in Fig. 3 to the corresponding lattice in Fig. 4. The product map in Fig. 3 lists the features of a number of PDAs[1].

We can derive the following facts more easily (i.e., automatically) from the lattice than from the table representation:

[1]PDA stands for Personal Digital Assistant. The product map is offered by `http://www.tariftip.de/default.asp?HB=1&BID=6&GID=82`. The web site is intended as a product comparison. We preselected "price less than 500€" and "infrared interface".

Product	mail	wap	sms	light	blue	infra	serial	usb	gsm	speaker	micro	line	module
Siemens	×	×	×			×							×
Oregon				×		×	×						
Palm m105	×	×	×	×		×	×						
Casio	×		×	×		×							
IBM c3	×			×		×	×						
Agenda	×			×		×	×			×	×		×
Palm m500	×		×	×		×		×					×
Handspring	×			×		×	×	×		×	×		×
IBM c500	×			×		×		×					
Ericsson	×	×	×	×		×	×	×	×				

Figure 3: A product map for a selected number of PDAs

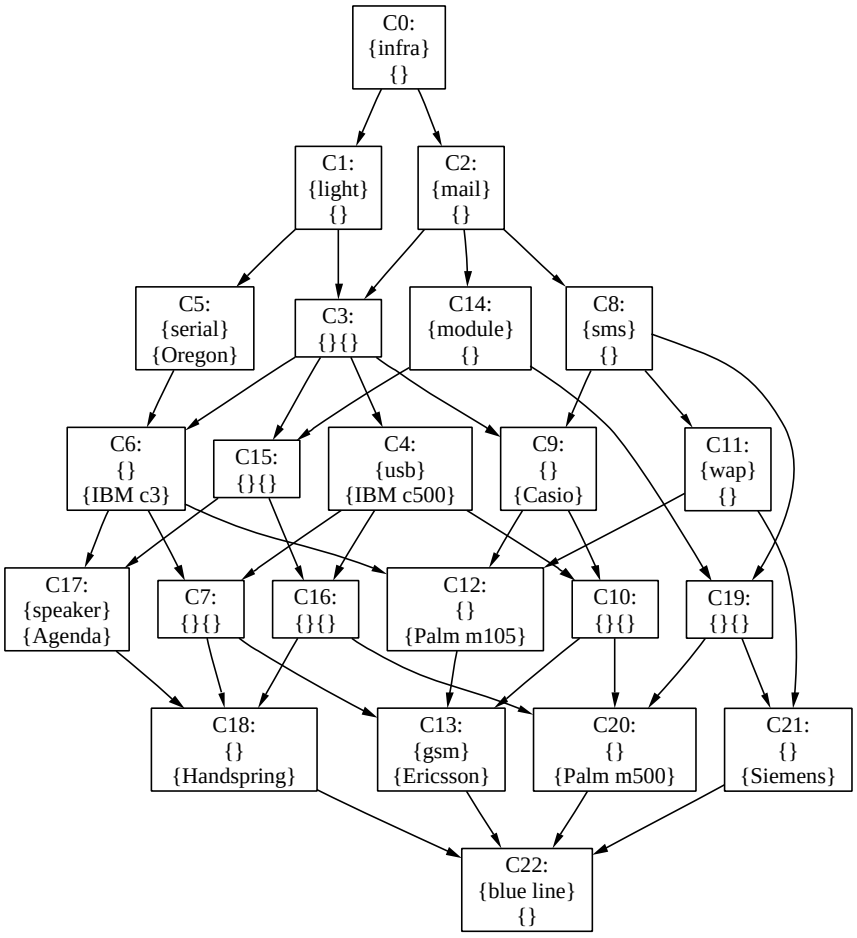


Figure 4: Full concept lattice for the product map in Fig. 3.

Shared features. Commonalities among a set of products $\{p_1, p_2, \ldots, p_n\}$ can be identified as the features of the concept that is the nearest common superconcept of the concepts $s(p_i)$. E.g., The nearest common superconcept of *IBM c3*, *IBM c500*, and *Casio* is $C3$ in Fig. 4 and, hence, the shared features are *light*, *mail*, and *infrared*.

Distinct features. Differences between a set of products $\{p_1, p_2, \ldots, p_n\}$ are features of the superconcepts of the concepts $s(p_i)$ not identified as shared features (as described above). E.g., the distinct feature of *IBM c500* w.r.t. *IBM c3* and *Casio* is its USB port.

Feature-offering products. For a given set of features $\{f_1, f_2, \ldots, f_n\}$ the products that offer these features can be identified as the products of the concept that is the nearest common subconcept of the nodes $s(f_i)$. E.g., the product that offers a *speaker* ($C17$) and a *USB port* ($C4$) is *Handspring* attached to $C18$.

Potential feature implications. Feature f_1 potentially implies feature f_2 if every product offering feature f_2 also offers f_1: $f_1 \rightarrow f_2$. This is the case if $s(f_2) \leq s(f_1)$. E.g., a *speaker* potentially implies a *serial port* according to Fig. 4. The example shows that the implication may be spurious and needs to be checked by an expert. Inversely, one can, however, automatically check an expert's assertion about a feature implication based on the lattice.

Feature categorization. The lattice reflects the specificity of features: The lower a feature in the lattice, the more special it is because fewer products offer it. Features that are common to all products will appear at the top concept. E.g., all products offer an infrared interface (this was the original selection criterion), but only *Ericsson* offers *GSM*. Note that features that are not offered by any product are at the bottom element of the concept lattice and no product will be attached to the bottom element. E.g., *bluetooth* and *line* are features offered by no product in the given price range.

Product categorization. Analogously to features, the lattice reflects the capability degree of products: The lower a product in the lattice, the more capable it is because it offers more features. E.g., *Ericsson*, *Handspring*, *Palm m500*, and *Siemens* are the most capable PDAs.

3 Related Work

Snelting [9] introduced formal concept analysis to software engineering. Since then, there is a growing number of applications of formal concept analysis. We used it to analyze features in existing code [4]. Based on that, we proposed a guided asset mining process [5] and a process for incrementally shifting from monolithic software to software product lines [8].

The product maps stem from PuLSE$^{\text{TM}}$-Eco [3], where they are used for scoping. Our method helps in the analysis of these maps.

4 Conclusions and Future Work

We presented a mathematical method to analyze product maps. The method is intuitive and easy to apply. The mathematical details can be hidden from the user of the method. It is also easy to implement our approach. Our prototypical implementation combines existing tools. The concept analysis is performed by Lindig's tool [7] and the lattices are visualized by AT&T's GraphViz [1]. To connect these tools, some simple Perl glue code was sufficient.

Future research will address transferring the already elaborated parts of the theory dealing with multi-value features (rather than "feature present or not"). Further, to provide support for evolution in product lines, we are working on an incremental application of our method.

References

[1] AT&T Labs-Research. GraphViz. Available at `http://www.research.att.com/sw/tools/graphviz/`, 2002.

[2] Garret Birkhoff. *Lattice Theory.* American Mathematical Society Colloquium Publications 25, Providence, RI, USA, first edition, 1940.

[3] Jean-Marc DeBaud and Klaus Schmid. A Systematic Approach to Derive the Scope of Software Product Lines. In *Proceedings of the 21st International Conference on Software Engineering*, pages 34–43, Los Angeles, CA, USA, May 1999. IEEE Computer Society Press.

[4] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Aiding Program Comprehension by Static and Dynamic Feature Analysis. In *Proceedings of the International Conference on Software Maintenance*, pages 602–611, Florence, Italy, November 2001. IEEE Computer Society Press.

[5] Thomas Eisenbarth and Daniel Simon. Guiding Feature Asset Mining for Software Product Line Development. In *Proceedings of the International Workshop on Product Line Engineering: The Early Steps: Planning, Modeling, and Managing*, pages 1–4, Erfurt, Germany, September 2001. Fraunhofer IESE.

[6] Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis—Mathematical Foundations.* Springer, 1999.

[7] Christian Lindig. Concepts 0.3e. Available at `http://www.gaertner.de/~lindig/software/`, 1999.

[8] Daniel Simon and Thomas Eisenbarth. Evolutionary Introduction of Software Product Lines. In *Proceedings of the 2nd Software Product Line Conference*, San Diego, CA, USA, August 2002. Springer. To appear.

[9] Gregor Snelting. Reengineering of Configurations Based on Mathematical Concept Analysis. *ACM Transactions on Software Engineering and Methodology*, 5(2):146–189, April 1996.

Reusable Test Requirements for UML-Modeled Product Lines[1]

Clémentine Nebut, Simon Pickin, Yves Le Traon and Jean-Marc Jézéquel
IRISA, Campus Universitaire de Beaulieu, 35042 Rennes Cedex, France
{Clementine.Nebut, Simon.Pickin, Yves.Le_Traon, Jean-Marc.Jezequel}@irisa.fr

Abstract

The object paradigm is increasingly being used in the construction of both centralized and distributed systems and is a key aspect of the current trend for model-driven architectures. In this paper, we present an approach to natural expression of test requirements and to formal validation in a UML-based development process which takes advantage of product lines (PL) specificities. We proceed by building behavioral test patterns (i.e. the test requirements) as combinations of use-case scenarios, these scenarios being product-independent and therefore constituting reusable PL assets. We then present a method for automated synthesis of test cases for specific products from these product-independent behavioral test patterns and product-specific design models, remaining entirely within the UML framework. We illustrate our approach using a virtual meeting PL case study.

1 Introduction

In the software field, object-orientation and component models are the main enabling technologies underpinning a product-line driven approach. In the development of a product line [1,2], the initial definition of the individual products depends mainly on marketing considerations. At this stage, functional requirements may be defined, in terms of use-cases and coarse scenarios. These use-cases/scenarios offer the natural basis for specifying test requirements. In our method, use-case scenarios (nominal and exceptional) are used to specify *behavioral test patterns* that then become reusable "test assets" of the product line. Use-case scenarios cannot be used directly for testing because they are generic and incomplete. That is, they are independent of the low-level design, which is specific to a particular product. They specify the general system functionality but not the exact way of exercising this functionality in terms of sequences of calls and responses. The complete sequence of method calls, the exact values of parameters in the method calls and even the exact types of the participating instances may not be known at this stage.

The main contribution of this paper is a method, supported by the Umlaut/TGV tool[3,4], for automatic generation of detailed test cases associated to specific products, from sets of incomplete and generic scenarios associated to a product line, i.e. *behavioral test patterns*. Of prime importance are the generic scenarios representing the test requirements (main expected behavior): the use-case scenarios. In this way, the method relates the principal test cases to the use-cases, contributing to the overall coherence of the development process.

We propose a two-step process, from test requirements to product-specific test cases. In step 1, test requirements are defined independently of the target final-product, in terms of behavioral test patterns. From use cases, we show how scenarios can be structured to produce reusable test patterns common to an entire product line and represented using the UML [5]. In step 2, when the final design is available and a product chosen as test target, we synthesize test cases for that particular product.

In Section 2, the issues of testing product lines are presented, as well as the testing requirements we propose, and our associated method dealing with behavioral test patterns. In Section 3, the test pattern methodology is detailed and exemplified.

2 Testing product lines : issues, requirements, method

We focus on exploiting test requirements, expressed as a combination of use cases scenarios. The product line testing we deal with in this paper is black-box testing: it is well-suited to dealing with variability in PLs. In this

[1] This work has been partially supported by the CAFE European project. Eureka Σ! 2023 Programme, ITEA project ip 0004 and the COTE project of the French RNTL program.

section, we present the issues involved in testing product lines, the test patterns we propose to use as testing requirements, and the underlying test methodology.

The philosophy for building product lines is build for reuse: from the first design of the PL, components are conceived, modeled and implemented to be reusable. This reuse philosophy can then be extended to the testing requirements of PLs. While the OO testing is becoming an important research domain[6, 7], to our knowledge, no study has been conducted dealing with the question of reusable, product-independent test assets in PLs, nor with the question of automatically synthesizing product-specific test cases from these product-independent assets.

As underlined in [8], product family requirements assist design evaluation. The idea we develop here is that the requirements and tests designers can take advantage of the presence of a common core, and consequently of a common behavior of products, to reuse both the tests and the requirements. In fact, we believe that the tests should be defined to be as independent as possible of the variants. At first sight, there is a contradiction in this approach:

- on the one hand, to be reusable, test scenarios must be expressed at a very high level to ensure that they are not dependent on the variants and on specific products,
- on the other hand, generic scenarios are too vague and incomplete to be directly used on a specific product, so that reuse of generic test scenarios does not seem possible.

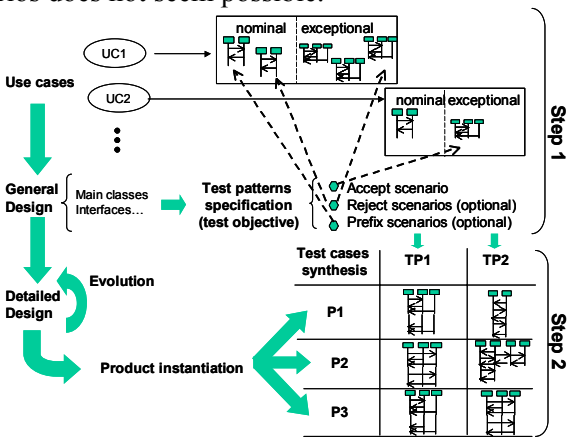


Figure 1. The PL testing methodology

This paper deals with this apparent contradiction by automatically deriving test cases dedicated to a specific product from a (set of) generic scenario(s). The methodology is illustrated in Figure 1. At the analysis stage, use cases are defined and sequence diagrams are attached to each of them. The sequence diagrams are of two types: those describing nominal scenarios, expressing the standard use-case occurrence, and those describing exceptional ones. The testing requirements are defined based on these scenarios in terms of *behavioral test patterns*.

A behavioral test pattern is here defined as a set of generic scenario structures representing a high-level view of some scenarios which the system under test (SUT) may engage in, according to its specification, and which we wish to test. The genericity may involve abstraction on instance names, method names or on method parameters. The scenario structures are elaborated with the aim of using them as selection criteria to derive a test case; they will usually include a subset of the communications of the corresponding test case. We suppose that in the derived test cases, the SUT environment role will be played by the tester. A test case derived from a test pattern specifies how to stimulate the SUT via the test interface, as well as specifying the expected responses at this interface, in such a way as to cause a conformant implementation to execute a scenario which fits the test pattern.

In our method, a test pattern comprises three parts, each part specified using sequence diagrams:

- the specification of the (visible or internal) behavior the test designer wants to test; the *accept* scenario structure serves to select the scenarios of the specification which are relevant for the test case;
- the specification of the (visible or internal) behavior the test designer wants to avoid in the test; the *reject* scenario structures serve to eliminate the scenarios of the specification which are irrelevant for the test case;

– the specification of the behavior needed to place the SUT in a state in which the accept behavior can take place; the *prefix* scenario serves to factorize the part of the accept scenario which may be common to several test patterns.

To specify a test pattern, the tester selects from among the nominal and exceptional scenarios an *accept scenario* describing the behavior that has to be tested, one or several (optional) *reject scenarios* describing the behaviors that are not significant for the test and one or several (optional) *prefix scenarios* that must precede the accept scenario.

Independently of the test development process, the PL design is developed and the final products instantiated. Further evolutions can also be carried out without changing the test patterns library, which is considered an asset of the PL. The most important aspect of our method is that the test patterns are defined once, and then kept unchanged and reused throughout the software life-cycle even if the code or the model evolve : a test pattern is a requirement for a product line, which is reusable for all the products of the PL. Test cases are derived for each specific product, taking into account all the choices and selections made during the product instantiation. For each product and for each test pattern, one or several test cases are synthesized.

The building of test patterns as requirements for testing (step 1) is explained and illustrated in the next section, and an overview is given on the derivation of test cases from test patterns (step 2).

3 Behavioral test patterns as requirements for testing product lines

3.1 The virtual meeting server example

The virtual meeting server PL offers simplified web conference services, that is, it permits several different kinds of work meetings on a distributed platform (a kind of generalized "chat" program).

When connected to the server, a user can enter or exit a meeting, speak, or plan new meetings. The use case diagram is given in Figure 2.

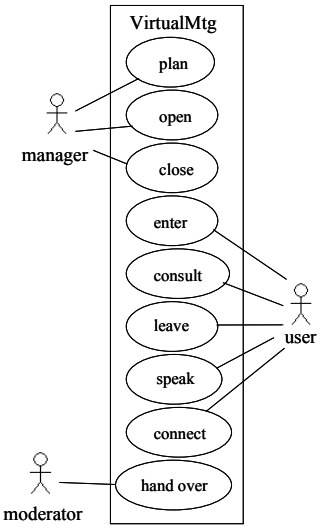


Figure 2 - Use case diagram for the virtual meeting

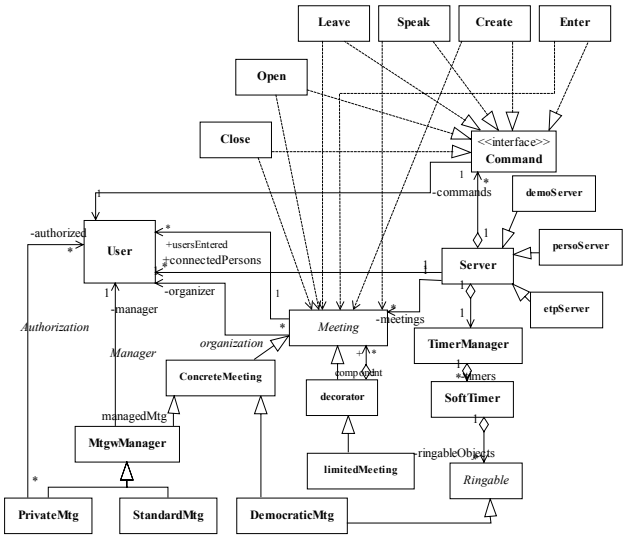


Figure 3 - Class diagram for the virtual meeting PL

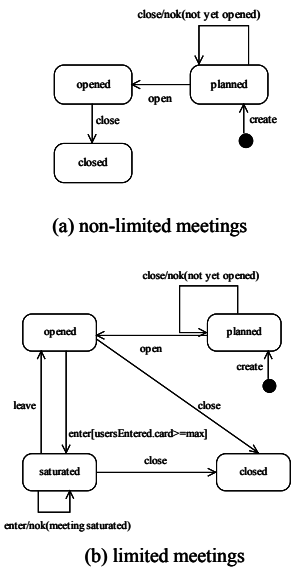


Figure 4 - Virtual meetings state machines

Three types of meetings exist:

- *standard* meetings where the current speaker is designated by a moderator (nominated by the organizer of the meeting)
- *democratic* meetings which are standard meetings where the moderator is a FIFO robot (the first client to ask for permission to speak is the first to speak)
- *private* meetings which are standard meetings with access limited to a certain set of users.

Marketing considerations mean that three products are to be derived in the virtual meeting PL : a demonstration edition, a personal edition, and an enterprise edition. The demonstration edition manages only standard meetings, limited to a certain number of participants. The personal edition provides the three kinds of meetings but still limits the number of participants of the meetings. The enterprise edition limits neither the type of meeting nor the number of participants in a meeting.

Two main variants are identified in the virtual meeting PL: firstly, the limitation of the number of participants in meetings, and secondly, the type of meetings available.

Figure 3 gives the partial class diagram for the Virtual Meeting PL. All the possible commands are reified and inherit the COMMAND interface. Only the main commands appear in Figure 3 to keep the diagram readable. To model behaviors, state diagrams are attached to the main classes that represent the users and the meetings. A state diagram is attached to the root of the concrete meetings hierarchy *concreteMeeting* and to the decorator *limitedMeeting*. Several design patterns are used but they are not described here, for the sake of conciseness.

3.2 Step 1: From use-case scenarios to behavioural test patterns

Building a test pattern consists in selecting – from among the high level scenarios – the behavior that the test designer wants to test, the behaviors that are irrelevant for testing it and, possibly, a prefix. This can be done as soon as the main classes of the product lines are defined: no design detail or state machines are used at this stage. This allow the test patterns to be reusable, because they are independent of the detailed design and the implementation.

The use case scenarios

The use-case scenarios employed to construct the test patterns are represented as generic sequence diagrams. They do not specify the exact parameter values of the methods, nor the exact name and type of the objects. They specify only general behaviors associated to a use case, and are defined very early in the conception (in the analysis phase). Wildcards and scenario parameters are used to achieve the genericity. As an example, in Figure 5 we give the scenarios associated to the *enter* use case. They are quite trivial: a user trying to enter a meeting can either be accepted or rejected.

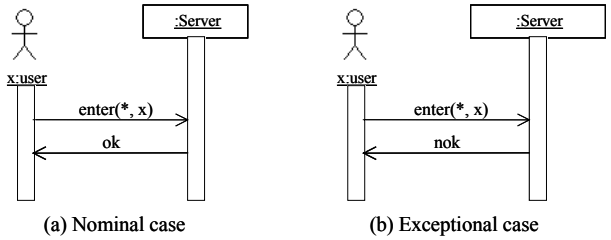


Figure 5 - Enter use case scenarios (x is a scenario parameter)

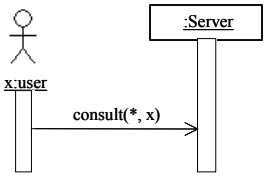


Figure 6 - A reject scenario for the *enter* use case

Reject scenarios

The reject scenarios describe the behaviors which, though correct, are unwanted in the test. Several reject scenarios can be associated to the same test pattern. They serve to limit the exploration required by the synthesis algorithms in order to find a test case that fits the test pattern, thereby improving performance. From a pragmatic point of view, if several test executions fit the accept part of the test pattern, reject scenarios can be used to guide the synthesis tool to produce the most suitable test case. Guiding the tool may be done to help minimize the synthesized test case by excluding calls which are known to be superfluous for the purposes of the test. This reduction of "noise" is particularly useful in testing concurrent applications. It may also be to exclude calls which are known to interfere with the test. We now give examples of these two cases.

To illustrate the case of superfluous calls, we consider deriving a test case from the nominal *enter* use case in our virtual meeting example. Any user can at any moment consult the state of the different meetings managed by the server. Calling the *consult* method is not relevant in testing of the functionality of the *enter* use case. The presence of a call to the *consult* method can be avoided in a synthesized test case by adding the reject scenario given in Figure 6 to the test pattern. This then means that the tester (taking the place of the system environment, ie. the users) will not call the *consult* method, at any moment, with any parameter values, during the test of the *enter* use case.

Note that this reject scenario corresponds to both the nominal and the exceptional scenarios attached to the use case *consult*, illustrating the reuse of scenarios in different test patterns. The use-case scenarios are defined once, and are then used either as reject or accept scenarios depending on which use case is being tested. In other words, once basic scenarios are defined, they are reused as building blocks to build test patterns.

To illustrate the case of interfering calls, we consider deriving a test case for the exceptional *enter* use-case scenario in our virtual meeting example. If the test designer wishes to check that an attempt to enter a meeting which is full will be refused, using the accept scenario as a test pattern is not sufficient. This is since the tool may instead produce a test in which the meeting is closed before the attempt takes place. Using the nominal scenario attached to the *close* use-case as a reject scenario will ensure that this does not occur.

Consider the case where the scenario associated to use case i is to be used as the accept scenario. Clearly, in this case, the scenarios associated to use cases which are independent of i can be used as reject scenarios. The task of adding reject scenarios can thus be partially automated, as soon as the designer of the PL has defined the dependencies between use cases. The dependencies are represented using a square boolean matrix of dimension the number of use cases, where Dep[i,j]=true if and only if use case i depends on use case j. When testing a particular use case i, all the scenarii attached to use cases j such that Dep[i,j]=false are used as reject scenarios. Such a matrix can be built in the early phases of conception.

Prefixes

The prefix is a high-level representation of the initialization of the behavior to be tested. It describes the preamble part of the test case, i.e. the behavior previous to that described in the accept scenario. The prefix serves to guide the synthesis towards the production of a minimal preamble. Like the reject scenarios, the prefix can be constructed from the other use-case scenarios. Unlike a reject scenario, a prefix may be composed of a sequence of such scenarios. Building the prefix is therefore a process of selecting use-case scenarios and composing them (by weak sequential composition).

Selection of the different parts of a test pattern

Once use-case scenarios have been built and placed in a test library, building a test pattern is a matter of selecting the accept scenario, the unwanted behaviors, and the prefix scenarios from this library. Figure 7 shows a test pattern for the virtual meeting example.

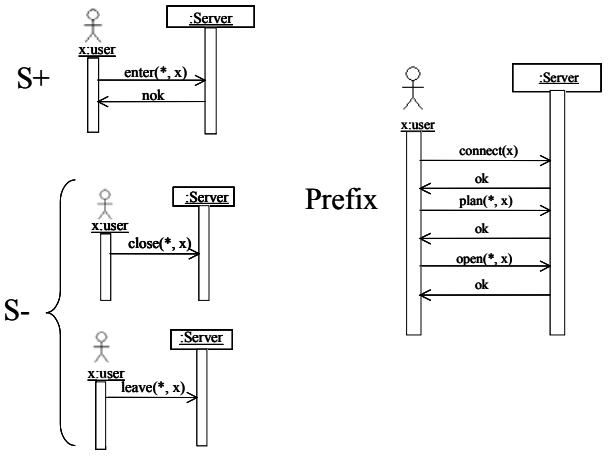


Figure 7 - A test pattern to check that an attempt to enter a meeting which is full is unsuccessful

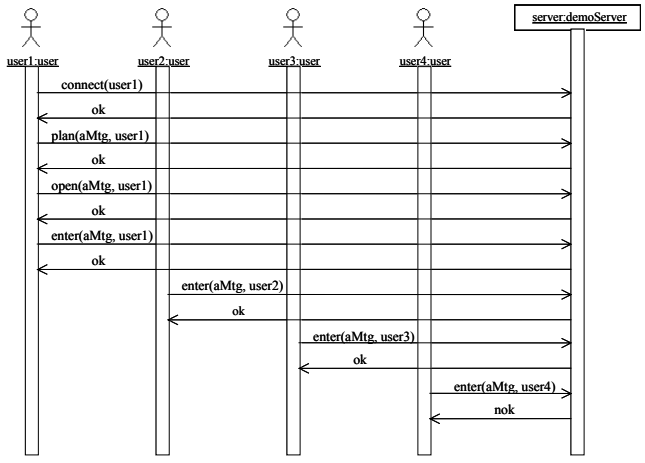


Figure 8 - A test case for the saturated meetings

The behavior to be tested, as described by the accept scenario, is: "an attempt by a user to enter a meeting is rejected". The unwanted behaviors are any closing or leaving action. The former since we do not want to synthesize a test in which an attempt to enter a meeting is rejected due to the meeting having been closed. The latter since we wish to synthesize a test in which other users may enter but not leave the meeting. Note that the *consult*, *hand over* and *speak* scenarios could also have been (automatically) added as reject scenarios. The prefix describes how a user

connects to the system, plans a meeting and then opens it. Of course, the opening action could be moved from the end of the prefix to the beginning of the accept scenario.

The test synthesis from a test pattern and a UML specification is summarized in the next following.

3.3 Step 2: The test synthesis from behavioral test patterns

Test patterns are independent of low-level design and implementation choices. While defining a high-level test scenario is not difficult when the main classes are identified, refining and adapting it to the final software product is an arduous process. However, the details of the low-level design are contained in the UML specification; completing the test pattern with these details is therefore a task which can be left to the automatic synthesis. Allowing test designers to work at the test pattern level rather than the test case level thus frees them from the need to specify the low-level detail, enabling them to concentrate on the essential aspects of the test.

Our process to generate the concrete test cases from the test requirements is based on the TGV generation tool [4]. The aim is not here to detail the whole process of the test synthesis. In short, the test pattern is first pre-compiled in order to treat the genericity, then compiled into a Labeled Transition System (LTS) [4]. A simulation API is built from the UML specifications, in order to be able to build lazily the LTS representing the operational semantics of the entire system. From those two LTSs and a set of other information such as the definition of the inputs and outputs, TGV builds an Input-Output LTS (IOLTS) representing the test case. This IOLTS is then transformed into a sequence diagram corresponding to the synthesized test case.

Figure 8 is an example of test case synthesized from the test pattern of Figure 7 and the final UML model whose simplified class diagram is given in Figure 3 and two of whose the state diagrams are given in Figure 4. The multi-instance representation of the tester (as several users) requires certain hypotheses concerning the concurrency.

4 Conclusion

In this paper, we have presented a methodology for testing product lines, from a natural way of specifying test requirements to the generation of concrete test cases. We have defined behavioral test patterns as reusable (and incomplete) testing scenarios expressed in UML, and explained how to build them from the use-case scenarios. We have also presented a methodology and tools for the automated production of test cases from these test patterns and a UML specification of the system under test.

The method, using prototype tools like Umlaut and TGV, is not yet completely tool-supported, but further work is underway to fully automate it.

From a methodological point of view, the contribution of this paper is to propose a whole process, from early modeling of requirements to test cases. This process allows concrete test cases to be produced for each specific product from reusable and generic test patterns and UML specifications

5 References

1. Atkinson, C., et al., *Component-based Product Line Engineering with UML*. Component Software Series, ed. C. Szyperski. Addison-Wesley, 2002.
2. Bosch, J., *Design and Use of Software Architectures: Adopting and evolving a product-line approach*. Addison-Wesley, 2000.
3. UMLAUT, *Unified Modeling Language All pUrposes Transformer*. 2002.
4. Jard, C. and T.Jéron. *TGV: theory, principles and algorithms*. in *6th world conference on integrated design and process technology*. 2002.
5. OMG, *Unified Modelling Language Specification, version 1.4. OMG Standard,*, in *Object Management Group*. 2001.
6. McGregor, J.D. and D.A. Sykes, *A practical guide to testing object-oriented software*. Addison Wesley ed, 2001.
7. McGregor, J.D., *Test patterns: please stand by.* JOOP, 1999. vol. 12 p. 14-19.
8. Lutz, R.R., *Extending the product family approach to support safe reuse.* Journal of systems and Software, 2000. vol. 53 p. 207-217.