LNCS 3714

Henk Obbink
Klaus Pohl (Eds.)

Software Product Lines

9th International Conference, SPLC 2005
Rennes, France, September 2005
Proceedings

SPLC 2005

Springer

Lecture Notes in Computer Science 3714

Henk Obbink   Klaus Pohl (Eds.)

Software Product Lines

9th International Conference, SPLC 2005
Rennes, France, September 26-29, 2005
Proceedings

Springer

Volume Editors

Henk Obbink
Philips Research Laboratories Eindhoven
Prof. Holstlaan 4, 5656 JA Eindhoven, The Netherlands
E-mail: Henk.Obbink@philips.com

Klaus Pohl
Universität Duisburg-Essen, Campus Essen
Institut für Informatik and Wirtschaftsinformatik
Software Systems Engineering
Schuetzenbahn 70, 45117 Essen, Germany
E-mail: Pohl@sse.uni-essen.de

Library of Congress Control Number: 2005932544

CR Subject Classification (1998): D.2, K.4.3, K.6

ISSN 0302-9743
ISBN-10 3-540-28936-4 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-28936-4 Springer Berlin Heidelberg New York

Springer is a part of Springer Science+Business Media

springeronline.com

Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11554844 06/3142 5 4 3 2 1 0

Preface

With SPLC 2005 we celebrated the formation of a new conference series, the *International* Software Product Line Conference (SPLC) which results from the "unification" of the former series of three SPLC (Software Product Line) Conferences launched in 2000 in the USA, and the former series of five PFE (Product Family Engineering) Workshops started in 1996 in Europe.

SPLC is now *the premier forum* for the growing community of software product line practitioners, researchers, and educators. SPLC offers a unique opportunity to present and discuss the most recent experiences, ideas, innovations, trends, and concerns in the area of *software product line engineering* and to build an international network of product line champions. An international SPLC Steering Committee has been established and it is the wish of this committee that from 2005 on, the SPLC conference will be held yearly in Europe, America, or Asia. The technical program of SPLC 2005 included.

- two keynotes from David Weiss (Avaya, USA) and Jan Bosch (Nokia, Finland), both leading experts with academic and industrial insights;
- 17 full and 3 short research papers organized around the following themes: feature modeling, re-engineering, strategies, validation, scoping and architecture, and product derivation;
- eight experience reports describing commercial application of product line practices;
- two panels focused on special topics in product line practice and product line research;
- tool demonstrations;
- a Hall of Fame session that continued the SPLC tradition in a slightly revised format.

In addition, the technical program was preceded by a tutorial and workshop day that included ten half-day tutorials presented by well-recognized experts and five workshops on specific areas of product line research.

The preparation of this programme would not have been possible without the help and support of many individuals. The role of the Program Committee was central in the achievement of this high-quality programme. We are indebted to each PC member for his or her commitment in reviewing the papers, participating in electronic consensus discussions and, finally, in actively taking part in the PC meeting, which was held in Essen on May 24, 2005.

Thanks also to the Organizing Committee and in particular to Jean-Marc Jézéquel for his continuous support at all stages and for making it possible to host SPLC 2005 in the beautiful city of Rennes. Most especially, we would like to thank all those who submitted their work to SPLC 2005. Without their willingness to publish and share their work SPLC 2005 would not have been possible.

June 2005 Henk Obbink and Klaus Pohl

Organization

General Co-chairs

Linda Northrop

Frank van der Linden

Program Co-chairs

Henk Obbink

Klaus Pohl

Organizing Committee

General Chairs	Frank van der Linden Philips, The Netherlands
	Linda Northrop Software Engineering Institute, USA

Program Chairs	Henk Obbink Philips, The Netherlands
	Klaus Pohl University of Duisburg-Essen, Germany
Local Organization Chair	Jean-Marc Jézéquel INRIA, France
Workshop Chairs	Svein Hallsteinsen SINTEF, Norway
	Benoit Baudry INRIA, France
Panel Chair	Charles W. Krueger BigLever Software, USA
Tool Demonstration Chair	Dirk Muthig Fraunhofer IESE, Germany
Hall of Fame	Paul Clements Software Engineering Institute, USA

Program Committee

Pierre America	Philips, The Netherlands
Joe Baumann	HP, USA
Sergio Bandinelli	ESI, Spain
Len Bass	SEI, USA
Günter Böckle	Siemens, Germany
Manfred Broy	TU Munich, Germany
Paul Clements	SEI, USA
Krzysztof Czarnecki	University of Waterloo, Canada
Juan Carlos Dueñas	University of Madrid, Spain
Birgit Geppert	Avaya, USA
Stefania Gnesi	IEI-CNR, Italy
Andre van der Hoek	University of California, USA
Kyo C. Kang	University of Pohang, Korea
Kari Känsälä	Nokia, Finland
Tomoji Kishi	JAIST, Japan
Stefan Kowalewski	RWTH Aachen, Germany
Philippe Kruchten	University of British Columbia, Canada
Charles W. Krueger	BigLever Software, USA

Tomi Männistö	HUT, Finland
John McGregor	Clemson University, USA
Nenad Medvidović	USC, USA
Dirk Muthig	Fraunhofer IESE, Germany
Robert Nord	Siemens, USA
Henk Obbink	Philips, The Netherlands (Chair)
Rob van Ommering	Philips, The Netherlands
Klaus Pohl	University of Duisburg-Essen, Germany (Chair)
Serge Salicki	Thales, France
Thomas Stauner	BMW, Germany
Steffen Thiel	Bosch, Germany
Martin Verlage	Market Maker, Germany
Matthias Weber	DaimlerChrysler, Germany

Additional Reviewers

Michal Antkiewicz
Jose L. Arciniegas
Somo Banerjee
Gerd Beneken
Jon Bentley
Ali Botorabi
Antonio Bucchiarone
Rodrigo Cerón
Alessandro Fantechi
Andreas Fleischmann
Gregory de Fombelle
Claudia Fritsch
Lars Geyer
Piergiorgio Di Giacomo
Pedro Gutierrez
Anna Ioschpe
Vladimir Jakobac
Eric Jouenne
Reinhard Klemm
Giuseppe Lami
Xabier Larrucea
Sam Malek
Jason Xabier Mansell
Chris Mattmann
Franco Mazzanti
Holt Mebane
Audris Mockus
David Morera
Ana R. Moya
Veronique Normand
Mark-Oliver Reiser
Laurent Rioux
Frank Roessler
Roshanak Roshandel
Kathrin Scheidemann
Tilman Seifert
Chiyoung Seo
Iratxe Gomez Susaeta
David Woollard

Table of Contents

Keynotes

Feature Modelling

Re-engineering

Short Papers

Strategies

Panels

Validation

Scoping and Architecture

Product Derivation

Next Generation Software Product Line Engineering

David M. Weiss

Avaya Labs,
233 Mt. Airy Rd.,
Basking Ridge, NJ 07920
weiss@avaya.com

Software product line engineering has advanced to the point where we know how to create software product lines on small to medium scales, and some organizations are having success on a larger scale. Success has come rather slowly, however, if one considers that many of the key ideas are 25-35 years old. For example, Dijkstra discussed the idea of program families in the late 1960s, David Parnas and others clarified the idea and showed how to apply it in real-time systems in the mid 1970s, and Jim Neighbors invented domain analysis in the early 1980s. Through the 1980s and 1990s we saw the systematization of product line engineering processes and their first applications. The first Software Product Lines Conference was held in 2000. Much of the development of the field has focused on technical aspects of creating product lines and producing applications. Indeed, most of the technical problems in creating product lines now seem solvable for many product lines. The Software Product Line Hall of Fame gives us examples of successful large scale product lines.

Institutionalizing the use of product lines in industrial organizations on a large scale may now require overcoming the obstacles in creating the right organizations and in quantifying the economics. Institutionalization often founders on the question of whether to create an organizational unit dedicated to domain engineering and developing the product line engineering environment, or whether to distribute the domain engineering task among different organizational units. Are there other organizational choices that we can make that solve this problem? How do other industries, which cannot survive without creating product lines, solve this problem? The economic justifications are typically cast in terms of a simple, cost-based model. What, then, is a good model to use?

The questions for the next generation of product lines focus on the following.

1. What are reliable, repeatable techniques for creating large scale product lines and the organizations that produce them?
2. What is the right economic model for an organization to use in deciding what product lines to create?
3. What is the next step in bringing organization to the way that we think about product lines?

I will discuss some possible avenues of approach for each of these problems.

H. Obbink and K. Pohl (Eds.): SPLC 2005, LNCS 3714, p. 1, 2005.

Software Product Families in Nokia

Jan Bosch

Nokia Research Center,
Software and Application Technologies Laboratory,
Helsinki, Finland
Jan.Bosch@nokia.com

Abstract. The level of software development and maintenance investment in embedded products has increased considerably over the last decade. As software product families are providing a proven approach to managing the cost and quality of software artefacts, Nokia has exploited this approach to software development for many years. This paper presents some lessons learned and the key challenges for the successful use and evolution of software artefacts.

1 Introduction

Reuse of existing software artefacts can be viewed as the holy grail of software engineering. For close to four decades, we have, as a software engineering community, evolved through an extended set of techniques for achieving higher productivity, more dynamic, responsive software development and lower maintenance cost. Techniques proposed in this context include modules, components, libraries, object-orientation, frameworks, architecture and, of course, software product families.

Software product families can be viewed as addressing a specific area of software reuse as most published product families are of an embedded nature, combining mechanical, hardware and software elements and less focused on information systems style functionality. Although this division has long been an accurate one, there is a clear trend towards blurring the distinctions between these two categories of systems. Embedded systems are becoming increasingly networked, upgradeable after their initial deployment, able to dynamically embed in new contexts and record, process and store increasing amounts of data. Examples of these kinds of systems can, among others, be found in the telecom, consumer electronics and automotive industry.

The transition from traditional, closed embedded systems to a world in which embedded systems provide platforms for deploying a wide variety of distributed, possibly peer-to-peer applications has a number of implications for research in the area of software product families as well. These implications include the increasing importance of hierarchy in product families, the increased complexity of variability management, the balance domain and product engineering and the role of open-source software.

The goal and contribution of this article is an analysis of the aforementioned implications for research in software product families. This analysis is performed from the perspective of Nokia, but also includes experiences from other companies that I have

H. Obbink and K. Pohl (Eds.): SPLC 2005, LNCS 3714, pp. 2 – 6, 2005.

worked with in the past and from earlier research performed at the University of Groningen. Consequently, the results should be relevant for software engineering organizations in general.

The remainder of this article is organized as follows. In the next section, an overview of the three main software product families for mobile terminals at Nokia is presented. Subsequently, in section 3, a set of challenges is presented that companies, including to various extent Nokia, are concerned with. Finally, the paper is concluded in section 5.

2 Overview of Product Families at Nokia

Nokia is a 55.000 person Fortune-500 company with revenue of around 30 billion euros. The company is organized in four business groups, i.e. Networks, primarily selling telecom infrastructure equipment and associated services, and Mobile Phones, Multimedia and Enterprise Solutions, addressing different segments of mobile devices with products and associated services.

The mobile devices business groups employ three main platforms in their products, i.e. Series 40, Series 60 and Maemo, an open-source Linux-based platform. The platforms address, with some overlap, mobile devices with different feature sets and price points. However, these platforms also share some components, so there is hierarchy in the shared artefacts.

In terms of the maturity model that I presented in [1], the platforms organizations typically employ the highest maturity model, i.e. the configurable product base approach. This means that most new features required for products under development typically are first developed as part of the platform. Once the platform is released the product configures the new platform release for use and inclusion in the product functionality.

Series 40

The Series 40 platform is a closed, proprietary platform consisting of a in-house developed operating system, a cellular subsystem managing wireless, cellular connectivity and a subsystem managing the applications and interface to the user. The Series 40 platform is primarily intended for mobile phones with restricted extended functionality, but can be extended with applications written in Java.

Series 60

The Series 60 platform is an open platform based on the Symbian operating system. The platform is explicitly intended for 3rd party application developers who can develop applications using native C/C++, Java or a scripting language such as Python or Perl. The architecture consists of an adaptation layer between the hardware and the Symbian OS, the Symbian OS, the Series 60 layer and a layer containing the core applications and extended application suite.

Maemo

The third platform, released during Q2 of 2005, used for Nokia mobile devices is Maemo, a Linux-based development platform using a large number of open-source components. The first product built based on the platform is the Nokia 770 Internet Tablet, planned for release during Q3 of 2005. The platform is open and can easily be used for application development by external developers, but even the platform itself can be changed and extended by external developers.

Concluding, in this section a brief overview of the three main platforms for Nokia mobile devices was presented. Every year, several (tens of) products are developed based on the Series 40 and Series 60 platforms. The Maemo platform is too novel to provide any information on the number of products.

3 Research Challenges

Software product families have, in Nokia as well as in many other companies, facilitated the increasing number of products released every year. The key challenge is obviously to exploit the commonalities between these products while as efficiently as possible managing their differences.

Despite these advantages, it is clear that not all problems have been solved and a number of key challenges remain that need to be addressed. These challenges are based on experiences from within Nokia, but also from organizations that I have worked with earlier.

In the list below, some of the key challenges of product-family based software development are discussed.

- **Hierarchical software product families**: In many cases, the initial presentation of a software product family is a relatively simple flat model with a common software architecture and set of components and a number of products derived from this architecture and, largely, populated with the shared components. In practice, almost all software product families are organized, in one way or another, in a hierarchical fashion. For instance, the infrastructure may be standardized for the complete company; a division has developed a platform on top of this infrastructure which is used by a business unit for a set of product family software artefacts that, in turn, are used to create multiple products from. In most research and theory development, the hierarchical nature of most product families is ignored, leading to solutions that do not encompass the actual complexity of our product development.
- **High skill requirements of staff**: Although the use of software product families can significantly improve productivity, it does require competent staff with significant skills to achieve these advantages. Due to the larger size of the overall system, the variation points and associated configuration tasks, validation of different use cases and several other factors, new R&D staff typically requires a significant amount of time before becoming productive. In addition, less competent staff often has challenges in operating efficiently in a product family context.

- **Variability and configurability management**: A key challenge that virtually every organization employing software product families experiences is managing the, frequently, large numbers of software variation points present in shared artefacts. The number of variation points easily ranges in the thousands and may even exceed ten-thousand in some product families. Especially without explicit management of the variation points, the sheer number may become a significant drain on the R&D resources. A second challenge is that the binding time and variant addition time of variation points typically evolves to later stages in the lifecycle. Especially the transition from pre-deployment to post-deployment binding of variation points often requires non-trivial development effort as the variation point should either become user-configurable or surrounded with functionality for automated variant selection.
- **Make, subcontract or license decisions**: One of the key trends in software engineering is the increasing importance of external software artefacts in the products or systems shipped to customers. Traditional products would perhaps use a 3rd party operating system, DBMS and GUI framework, but all product functionality would be built in-house. More recently the amount of smaller commercial components available for use in specific parts of the architecture has increased significantly. In addition, the amount of effort required for software development continues to increase, requiring companies to subcontract the development and maintenance of software components or to license the functionality. This complicates the decision for architects and product managers as the choices now include to build the software internal, to subcontract or to license components. Although the immediate decision is often relatively manageable, determining the long term consequences of these decisions is more difficult. For instance, subcontractors may purposely complicate and bloat their software to create a dependency and the cost for licensing software may increase significantly after the product family has become critically dependent on it.
- **Balancing domain- and application-engineering**: An issue that has raised concern in many organizations is where to place the boundary between functionality that is developed as part of the shared software artefacts and functionality developed as product-specific code. Although there obviously is an optimum from a technical perspective, business, organizational and other factors may cause significant deviations from that optimal point. The typical case is that product units are unwilling to relinquish power and the shared artefacts incorporate too little functionality. However, also cases exist where the balance went to far in the other direction and development resources are highly centralized and the central organization is trying to satisfy the needs of all product groups that lack resources to develop some of their product-specific, differentiating functionality.
- **Growing software size**: In embedded systems, the continuing growth of software as part of the overall R&D cost (and sometimes even of the bill of materials) is a concern for many companies because of the sheer complexity of software development and the lack of clear end to this trend. The consequences may, among others, include long lead times and quality concerns.

- **Role of open-source software**: One of the most interesting developments during the last decade is the emergence of open-source software and the societal trend, creative commons, associated with it. Of course, open-source software offers yet another solution to dealing with the challenges of the growing size of software in embedded systems and of software systems in general. Different from popular thought, open-source software is, in practice, not free. Just as any software component for which the source code is available, it needs to be managed, tested and integrated. Depending on the license, “pollution” of in-house developed software may occur. In most cases, any improvements developed by the company should be returned to the community, if not for legal reasons then for moral reasons. Finally, a concern is that the evolution of open-source software can typically not be predicted or steered.

4 Conclusion

For close to four decades, we have, as a software engineering community, evolved through an extended set of techniques for achieving higher productivity, more dynamic, responsive software development and lower maintenance cost. Techniques proposed in this context include modules, components, libraries, object-orientation, frameworks, architecture and, of course, software product families.

Software product families are often applied in the context of embedded systems, which are becoming increasingly networked, upgradeable after their initial deployment, able to dynamically embed in new contexts and record, process and store increasing amounts of data.

The transition from traditional, closed embedded systems to a world in which embedded systems provide platforms for deploying a wide variety of distributed, possibly peer-to-peer applications has a number of implications for research in the area of software product families as well. In this paper, we listed a number of these implications with the intention to raise the awareness of the research community with the obvious ambition to see these issues resolved. These implications include hierarchical software product families, high skill requirements of staff, variability and configurability management, make, subcontract or license decisions, balancing domain- and application-engineering, growing software size and the role of open-source software.

Although software product families have resulted in significant benefits for the organizations employing the technology, several research challenges remain. This paper has raised some of the most prominent challenges.

References

1. Bosch, J., Maturity and Evolution in Software Product Lines: Approaches, Artefacts and Organization, Proceedings of the Second International Conference on Software Product Lines (SPLC-2), San Diego, CA, USA. Springer LNCS 2379, pp. 257-271, August 2002.

Feature Models, Grammars, and Propositional Formulas

Don Batory

Department of Computer Sciences,
University of Texas at Austin,
Austin, Texas 78712
batory@cs.utexas.edu

Abstract. Feature models are used to specify members of a product-line. Despite years of progress, contemporary tools often provide limited support for feature constraints and offer little or no support for debugging feature models. We integrate prior results to connect feature models, grammars, and propositional formulas. This connection allows arbitrary propositional constraints to be defined among features and enables off-the-shelf satisfiability solvers to debug feature models. We also show how our ideas can generalize recent results on the staged configuration of feature models.

1 Introduction

A key technical innovation of software product-lines is the use of features to distinguish product-line members. A *feature* is an increment in program functionality [29]. A particular product-line member is defined by a unique combination of features. The set of all legal feature combinations defines the set of product-line members [23].

Feature models define features and their usage constraints in product-lines [12][20]. Current methodologies organize features into a tree, called a *feature diagram (FD)*, which is used to declaratively specify product-line members [2]. Relationships among FDs and grammars [21][13], and FDs and formal models/logic programming [7][24][26][27] have been noted in the past, but the potential of their integration is not yet fully realized.

Despite progress, tools for feature models often seem ad hoc; they exhibit odd limitations and provide little or no support for debugging feature models. This is to be expected when a fundamental underpinning of feature models is lacking. In this paper, we integrate prior results to connect FDs, grammars, and propositional formulas. This connection enables general-purpose, light-weight, and efficient *logic truth maintenance systems (LTMSs)*[17] to propagate constraints as users select features so that inconsistent product specifications are avoided — much like syntax-directed editors guarantee compilable programs [28]. This connection also allows us to use off-the-shelf tools, called *satisfiability solvers* or *SAT solvers* [16], to help debug feature models by confirming compatible and incomplete feature sets. To our knowledge, the use of LTMSs and SAT solvers in feature modeling tools is novel.

Our approach is tutorial. We believe it is important that researchers and practitioners clearly see the fundamental underpinnings of feature models, and that

H. Obbink and K. Pohl (Eds.): SPLC 2005, LNCS 3714, pp. 7 – 20, 2005.

light-weight and easy-to-build LTMS algorithms and easy-to-use SAT solvers can help address key weaknesses in existing feature model tools and theories

2 Feature Models

A *feature model* is a hierarchically arranged set of features. Relationships between a *parent* (or *compound*) feature and its *child* features (or *subfeatures*) are categorized as:

- *And* — all subfeatures must be selected,
- *Alternative* — only one subfeature can be selected,
- *Or* — one or more can be selected,
- *Mandatory* — features that required, and
- *Optional* — features that are optional.

Or relationships can have **`n:m`** cardinalities: a minimum of **`n`** features and at most **`m`** features can be selected [12]. More elaborate cardinalities are possible [13].

A *feature diagram* is a graphical representation of a feature model [23]. It is a tree where primitive features are leaves and compound features are interior nodes. Common graphical notations are depicted in Figure 1.

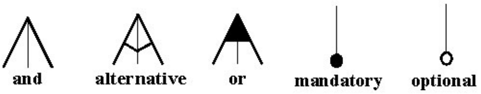
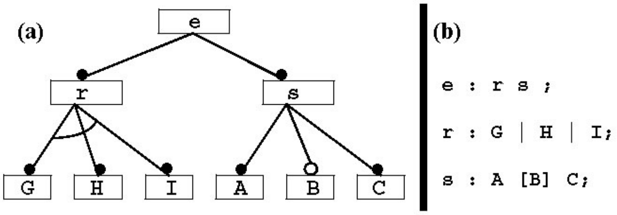


Fig. 1. Feature Diagram Notations

Figure 2a is a feature diagram. It defines a product-line where each application contains two features **`r`** and **`s`**, where **`r`** is an alternative feature: only one of **`G`**, **`H`**, and **`I`** can be present in an application. **`s`** is a compound feature that consists of mandatory features **A** and **C**, and optional feature **B**.



```
e : r s ;
r : G | H | I;
s : A [B] C;
```

Fig. 2. A Feature Diagram and its Grammar

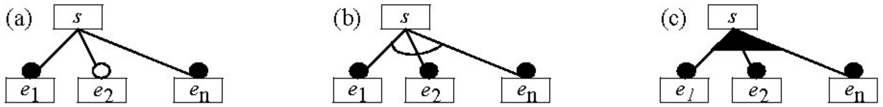


Fig. 3. Parent-Child Relationships in FDs

The connection between FDs and grammars is due to de Jong and Visser [21]. We will use iterative tree grammars. An *iterative grammar* uses iteration (e.g., one-or-more **`t+`** and zero-or-more **`t*`** constructs) rather than recursion, to express repetition.

A *tree grammar* requires every token to appear in exactly one pattern, and the name of every production to appear in exactly one pattern. The root production is an exception; it is not referenced in any pattern. More general grammars can be used, but iterative trees capture the minimum properties needed for our discussions.

Figure 3 enumerates the basic hierarchical relationships that can be expressed in a feature diagram. Each has a straightforward iterative tree grammar representation:

- Figure 3a is the production $\mathtt{s{:}e_1\ e_2...e_n}$ assuming all subfeatures are mandatory. If a subfeature is optional (as is $\mathtt{e_2}$), it is surrounded by `[`brackets`]`. Thus, the production for Figure 3a is $\mathtt{s{:}e_1\ [e_2]...e_n}$.
- Figure 3b is the production: $\mathtt{s{:}e_1\ |\ e_2\ |...|e_n}$.
- Figure 3c corresponds to a pair of rules: `s:t+;` and $\mathtt{t{:}e_1\ |e_2\ |...|e_n;}$ meaning one or more of the $\mathtt{e_i}$ are to be selected. In general, each non-terminal node of a feature diagram is a production. The root is the start production; leaves are tokens. Figure 2b is the grammar of Figure 2a. An application defined by the feature diagram of Figure 2a is a sentence of this grammar.

Henceforth, we use the following notation for grammars. Tokens are `UPPERCASE` and non-terminals are `lowercase`. `r+` denotes one or more instances of non-terminal `r`; `r*` denotes zero or more. `[r]` and `[R]` denote optional non-terminal `r` and optional token `R`. A pattern is a named sequence of (possibly optional or repeating) non-terminals and (possibly optional) terminals. Consider the production:

```
r : b+ A C          :: First
  | [D] E F         :: Second ;
```

The name of this production is `r`; it has two patterns `First` and `Second`. The `First` pattern has one or more instances of `b` followed by terminals `A` and `C`. The `Second` pattern has optional token `D` followed by terminals `E` and `F`.

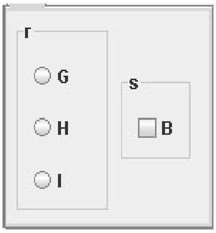


Fig. 4. GUI Specification

Grammars provide a graphics-neutral representation of feature models. For example, the grammar of Figure 2b could be displayed by the FD of Figure 2a or the GUI of Figure 4. (The GUI doesn't display features `E` and `F`, as they are mandatory — nothing needs to be selected). A popular Eclipse plug-in provides other possible graphical representations of FDs (tree and wizard-based), all of which are derived from a grammar-like specification [2].

In the next section, we show how iterative tree grammars (or equivalently feature diagrams) are mapped to propositional formulas.

3 Propositional Formulas

Mannion was the first to connect propositional formulas to product-lines [26]; we show how his results integrate with those of Section 2. A *propositional formula* is a set of boolean variables and a propositional logic predicate that constrains the values of these variables. Besides the standard $\wedge$, $\vee$, $\neg$, $\Rightarrow$, and $\Leftrightarrow$ operations of propositional logic, we also use $\mathtt{choose_1(e_1...e_k)}$ to mean at most one of the expressions $\mathbf{e}_1 \ldots \mathbf{e}_k$ is true. More generally, $\mathbf{choose}_{n,m}(\mathbf{e}_1 \ldots \mathbf{e}_k)$ means at least *n* and at most *m* of the expressions $\mathbf{e}_1 \ldots \mathbf{e}_k$ are true, where $0 \le n \le m \le k$.

A grammar is a compact representation of a propositional formula. A variable of the formula is either: a token, the name of a non-terminal, or the name of a pattern. For example, the production:

```
r : A B          :: P1
  | C [r1]       :: P2 ;
```

(1)

has seven variables: three **{A, B, C}** are tokens, two are non-terminals **{r, r1}**, and two are names of patterns **{P1, P2}**. Given these variables, the rules for mapping a grammar to a propositional formula are straightforward.

Mapping Productions. Consider production $\mathbf{r}{:}\mathbf{P}_1|\ldots|\mathbf{P}_n$, which has *n* patterns $\mathbf{P}_1 \ldots \mathbf{P}_n$. Production **r** can be referenced in one of three ways: **r** (choose one), **r+** (choose one or more), and **r*** (choose zero or more). As **r*** can be encoded as **[r+]** (optionally choose one or more), there are only two basic references: **r** and **r+**. The propositional formulas for both are listed below.

Pattern	**Formula**
r	$\mathbf{r} \Leftrightarrow \mathbf{choose}_1(\mathbf{P}_1, \ldots, \mathbf{P}_n)$
r+	$\mathbf{r} \Leftrightarrow (\mathbf{P}_1 \vee \ldots \vee \mathbf{P}_n)$

Mapping Patterns. A *basic term* is either a token or a production reference. A *pattern* is a sequence of one or more basic terms or optional basic terms. Let **P** be the name of a pattern and let $\mathbf{t}_1 \ldots \mathbf{t}_n$ be a sequence of basic terms. The formula for **P** is:

$$\mathbf{P} \Leftrightarrow \mathbf{t}_1 \wedge \mathbf{P} \Leftrightarrow \mathbf{t}_2 \wedge \ \ldots \ \wedge \mathbf{P} \Leftrightarrow \mathbf{t}_n \tag{2}$$

That is, if **P** is included in a design then terms $\mathbf{t}_1 \ldots \mathbf{t}_n$ are also included, and vice versa. Consider pattern **Q** whose second term is optional: $\mathbf{t}_1\ [\mathbf{t}_2] \ldots \mathbf{t}_n$. The formula for **Q** is:

$$\mathbf{Q} \Leftrightarrow \mathbf{t}_1 \wedge \mathbf{t}_2 \Rightarrow \mathbf{Q} \wedge \ \ldots \ \wedge \mathbf{Q} \Leftrightarrow \mathbf{t}_n \tag{3}$$

That is, if **Q** is included in a design then terms $\mathbf{t}_1$ and $\mathbf{t}_n$ are also included, and vice versa. In the case of optional term $\mathbf{t}_2$, if $\mathbf{t}_2$ is selected, **Q** is also selected; however, the converse is not true.

Using these rules, production **(1)** would be translated to the following formula:

$$\mathbf{r} \Leftrightarrow \mathbf{choose}_1(\mathbf{P1},\mathbf{P2}) \wedge \mathbf{P1} \Leftrightarrow \mathbf{A} \wedge \mathbf{P1} \Leftrightarrow \mathbf{B} \wedge \mathbf{P2} \Leftrightarrow \mathbf{C} \wedge \mathbf{r1} \Rightarrow \mathbf{P2}$$

Mapping Grammars. The propositional formula of a grammar is the conjunction of: (i) the formula for each production, (ii) the formula for each pattern, and (iii) the predicate **root=true**, where **root** is the grammar's start production. The propositional formula for the grammar of Figure 2b is:

$$\mathbf{e{=}true} \wedge \mathbf{e} \Leftrightarrow \mathbf{r} \wedge \mathbf{e} \Leftrightarrow \mathbf{s} \wedge \mathbf{r} \Leftrightarrow \mathbf{choose}_1(\mathbf{G},\mathbf{H},\mathbf{I}) \wedge \mathbf{s} \Leftrightarrow \mathbf{A} \wedge \mathbf{B} \Rightarrow \mathbf{s} \wedge \mathbf{s} \Leftrightarrow \mathbf{C} \tag{4}$$

Contrary to current literature, feature models are generally *not* context free grammars. There are often additional constraints, here called *non-grammar constraints*, that govern the compatibility of features. Current tools often limit non-grammar constraints to simple exclusion (choosing feature **I** automatically *excludes* a given feature list) and inclusion (choosing feature **I** *includes* or *requires* a given feature list). We argue exclusion and inclusion constraints are too simplistic. In earlier work [4], we implemented feature models as attribute grammars enabling us to write constraints of the form:

```
F implies A or B or C
```

This means **F** *needs* features **A**, **B**, or **C** or any combination thereof. More often, we found that preconditions for feature usage were based not on a single property but on *sets* of properties that could be satisfied by *combinations* of features, leading to predicates of the form:

```
F implies (A and X) or (B and (Y or Z)) or C
```

meaning **F** *needs* the feature pairs (**A**,**X**), (**B**,**Y**), (**B**,**Z**), or **C**, or any combination thereof. Exclusion constraints had a similar generality. For this reason, we concluded that non-grammar constraints should be *arbitrary* propositional formulas. By mapping a grammar to a propositional formula, we now can admit arbitrary propositional constraints by conjoining them onto the grammar's formula. In this way, a feature model (grammar + constraints) *is* a propositional formula.

An immediate application of these ideas may help resolve a pesky problem in that feature models do not have unique representations as feature diagrams. (That is, there are multiple ways of expressing the same constraints [12]). It is a daunting task to know if two FDs are equivalent; how tools handle redundant representations is left to tool implementors [14]. It is possible to show that two FDs are equivalent if their propositional formulas are equivalent. See [19] for details.

4 Logic Truth Maintenance Systems

Feature models are the basis for declarative domain-specific languages for product specifications. As users select features for a desired application, we want the implications of these selections to be propagated, so users cannot write incorrect specifications. A *Logic-Truth Maintenance Systems (LTMS)* can used for this purpose.

A LTMS is a classic AI program that maintains the consequences of a propositional formula. An LTMS application is defined by:

- a set of boolean *variables*,
- a set of *propositional logic predicates* to constrain the values of these variables,[1]
- *premises* (assignments to variables that hold universally),
- *assumptions* (assignments to variables that hold for the moment, but may be later retracted), and
- *inferences* (assignments to variables that follow from premises and assumptions).

[1] Equivalently, a single predicate can be used which is the conjunction of the input predicates.

The activities of an LTMS are to:

- compute inferences,
- provide a rationale for variable assignments,
- detect and report contradictions,
- retract and/or make new assumptions, and
- maintain a database of inferences for efficient backtracking.

A *SAT (propositional satisfiability)* solver relies on an LTMS to help it search the combinatorial space for a set of variable assignments that satisfy all predicates. The efficiency of SAT solvers relies on a database of knowledge of previously computed inferences to avoid redundant or unnecessary searches [17].

What makes an LTMS complicated is (a) how it is to be used (e.g., a SAT solver requires considerable support) and (b) the number of rules and variables. If the number is large, then it is computationally infeasible to recompute inferences from scratch; retractions and new assumptions require incremental updates to existing assignments. This requires a non-trivial amount of bookkeeping by an LTMS.

Fortunately, a particularly simple LTMS suffices for our needs. First, the number of rules and variables that arise in feature models isn't large enough (e.g, in the hundreds) for performance to be an issue. (Inferences can be recomputed from scratch in a fraction of a second). Second, searching the space of possible variable assignments is performed *manually* by feature model users as they select and deselect features. Thus, an LTMS that supports only the first three activities previously listed is needed. Better still, standard algorithms for implementing LTMSs are well-documented in AI texts [17]. The challenge is to adapt these algorithms to our needs.

The mapping of LTMS inputs to feature models is straightforward. The variables are the tokens, production names, and pattern names of a grammar. The propositional formula is derived from the feature model (grammar + constraints). There is a single premise: **`root=true`**. Assumptions are features that are manually selected by users. Inferences are variable assignments that follow from the premise and assumptions.

In the next sections, we outline the LTMS algorithms that we have used in building our feature modeling tool **`guidsl`**, whose use we illustrate in Section 5.

4.1 LTMS Algorithms

The *Boolean Constraint Propagation (BCP)* algorithm is the inference engine of an LTMS. Inputs to a BCP are a set of variables $\{v_1 \ldots v_m\}$ and a set of arbitrary propositional predicates $\{p_1 \ldots p_n\}$ whose conjunction $p_1 \wedge \ldots \wedge p_n$ defines the *global constraint (GC)* on variable assignments (i.e., the formula of a feature model). BCP algorithms require the GC to be in *conjunctive normal form (CNF)* [17]. Simple and efficient algorithms convert arbitrary p_j to a conjunction of clauses, where a *clause* is a disjunction of one or more *terms*, a term being a variable or its negation [17].

BCP uses three-value logic (**`true`**, **`false`**, **`unknown`**) for variable assignments. Initially, BCP assigns **`unknown`** to all variables, except for premises which it assigns **`true`**. Given a set of variable assignments, each clause **`C`** of GC is either:

- *satisfied*: some term is **`true`**.
- *violated*: all terms are **`false`**.

- *unit-open*: one term is **`unknown`**, the rest are **`false`**.
- *non-unit open*: more than one term is **`unknown`** and the rest are **`false`**.

A unit-open term enables the BCP to change the **`unknown`** assignment to **`true`**. Thus, if clause **`C`** is **`x`**∨¬**`y`** and **`x`** is **`false`** and **`y`** is **`unknown`**, BCP concludes **`y`** is **`false`**.

The BCP algorithm maintains a stack **`S`** of clauses to examine. Whenever it encounters a violated clause, it signals a contradiction (more on this later). Assume for now there are no contradictions. The BCP algorithm is simple: it marches through **`S`** finding unit-open clauses and setting their terms.

```
while (S is not empty) {
   c = S.pop();
   if (c.is_unit_open) {
       let t be term of c whose value is unknown;
       set(t);
    }
}
```

`set(t)` — setting a term — involves updating the term's variable's assignment (e.g., if **`t`** is ¬**`y`** then **`y`** is assigned **`false`**), pushing unit-open terms onto **`S`**, and signalling contradictions:

```
set variable of t so that t is true;
for each clause C of GC containing ¬t {
   if (C.is_unit_open) S.push(C);
   else
   if (C.is_violated) signal_contradiction();
}
```

Invoking BCP on its initial assignment to variables propagates the consequences of the premises. For each subsequent assumption, the variable assignment is made and BCP is invoked. Let **`L`** be the sequence of assumptions (i.e., user-made variable assignments). The consequences that follow from **`L`** are computed by:

```
for each variable l in L {
   set(l);
   BCP();
}
```

If an assumption is retracted, it is simply removed from **`L`**.

A contradiction reveals an inconsistency in the feature model. When contradictions are encountered, they (and their details) must be reported to the feature model designers for model repairs.

Example. Suppose a feature model has the contradictory predicates **`x`**⇒**`y`** and **`y`**⇒¬**`x`**. If **`x=true`** is a premise, BCP infers **`y=true`** (from clause **`x`**⇒**`y`**), and discovers clause (**`y`**⇒¬**`x`**) to be violated, thus signalling a contradiction.

Explanations for why a variable has a value (or why a contradiction occurs) requires extra bookkeeping. Each time BCP encounters a unit-open clause, it keeps a record of its conclusions by maintaining a 3-tuple of its actions **`<conclusion`**, **`reason`**, **`{antecedents}>`** where *conclusion* is a variable assignment, *reason* is the predicate (or clause) that lead to this inference, and *antecedents* are the 3-tuples of variables whose values were referenced. By traversing antecedents backwards, a justification for a conclusion can be presented in a human-understandable form.

Example. The prior example generates a pair of tuples: `#1:<x=true, premise, {}}` and `#2:<y=true, x⇒y, {#1}>`. The explanation for `y=true` is: `x=true` is a premise and `y=true` follows from `x⇒y`.

4.2 A Complete Specification

A product specification (or equivalently, a variable assignment) is *complete* if the GC predicate is satisfied. What makes this problem interesting is how the GC predicate is checked. Assume that a user specifies a product by selecting features from a GUI or FD. When a feature is selected, the variable for that feature is set to `true`; a deselection sets it to `unknown`. (Inferencing can set a variable to `true` or `false`). Under normal use, users can only declare the features that they want, *not what they don't want.*

At the time that a specification is to be output, all variables whose values are `unknown` are assumed `false` (i.e., these features are not to be in the target product). The GC is then evaluated with this variable assignment in mind. If the GC predicate is satisfied, a valid configuration of the feature model has been specified. However, if a clause of GC fails, then either a complete sentence has not yet been specified or certain non-grammar constraints are unsatisfied. In either case, the predicate that triggered the failure is reported thus providing guidance to the user on how to complete the specification. This guidance is usually helpful.

5 An Example

We have built a tool, called `guidsl`, that implements the ideas in the previous sections. `guidsl` is part of the AHEAD Tool Suite [5] a set of tools for product-line development that support feature modularizations and their compositions. In the following section, we describe a classical product line and the `guidsl` implementation of its feature model.

5.1 The Graph Product Line (GPL)

The *Graph Product-Line (GPL)* is a family of graph applications that was inspired by early work on modular software extensibility [29]. Each GPL application implements one or more graph algorithms. A `guidsl` feature model for GPL (i.e., its grammar + constraints) is listed in Figure 5, where token names are not capitalized.

The semantics of the GPL domain are straightforward. A graph is either `Directed` or `Undirected`. Edges can be `Weighted` with non-negative numbers or `Unweighted`. A graph application requires at most one search algorithm: depth-first search (`DFS`) or breadth-first search (`BFS`), and one or more of the following algorithms:

- **Vertex Numbering** (**`Number`**): A unique number is assigned to each vertex.
- **Connected Components (`Connected`):** Computes the *connected components* of an undirected graph, which are equivalence classes under the reachable-from relation. For every pair of vertices **`x`** and **`y`** in a component, there is a path from **`x`** to **`y`**.
- **Strongly Connected Components (`StrongC`):** Computes the *strongly connected components* of a directed graph, which are equivalence classes under the reachable relation. Vertex **`y`** is reachable from vertex **`x`** if there is a path from **`x`** to **`y`**.
- **Cycle Checking** (**`Cycle`**): Determines if there are cycles in a graph. A cycle in directed graphs must have at least 2 edges, while in undirected graphs it must have at least 3 edges.
- **Minimum Spanning Tree (`MSTPrim`, `MSTKruskal`):** Computes a *Minimum Spanning Tree (MST)*, which contains all the vertices in the graph such that the sum of the weights of the edges in the tree is minimal.
- **Single-Source Shortest Path (`Shortest`):** Computes the shortest path from a source vertex to all other vertices.

```
 // grammar

 GPL : Driver Alg+ [Src] [Wgt] Gtp :: MainGpl ;
 Gtp : Directed | Undirected ;
 Wgt : Weighted | Unweighted ;
 Src : BFS | DFS ;
 Alg : Number | Connected | Transpose StronglyConnected :: StrongC
       | Cycle | MSTPrim | MSTKruskal | Shortest ;
Driver : Prog Benchmark :: DriverProg ;

 %% // constraints

 Number implies Src ;
 Connected implies Undirected and Src ; StrongC implies Directed
 and DFS ;
 Cycle implies DFS ;
 MSTKruskal or MSTPrim implies Undirected and Weighted ;
 MSTKruskal or MSTPrim implies not (MSTKruskal and MSTPrim) ; //#
 Shortest implies Directed and Weighted ;
```

Fig. 5. GPL Model

The grammar that defines the order in which GPL features are composed is shown in Figure 5. Not all combinations of features are possible. The rules that govern compatibilities are taken directly from algorithm texts [11] and are listed in Figure 6. These constraints are listed as additional propositional formulas (below the **`%%`** in Figure 5). When combined with the GPL grammar, a feature model for GPL is defined. Note: **`MSTKruskal`** and **`MSTPrim`** are mutually exclusive (constraint **#** in Figure 5); at most one can be selected in a GPL product.

Algorithm	**Required Graph Type**	**Required Weight**	**Required Search**
Vertex Numbering	Any	Any	BFS, DFS
Connected Components	Undirected	Any	BFS, DFS
Strongly Connected Components	Directed	Any	DFS
Cycle Checking	Any	Any	DFS
Minimum Spanning Tree	Undirected	Weighted	None
Shortest Path	Directed	Weighted	None

Fig. 6. Feature Constraints in GPL

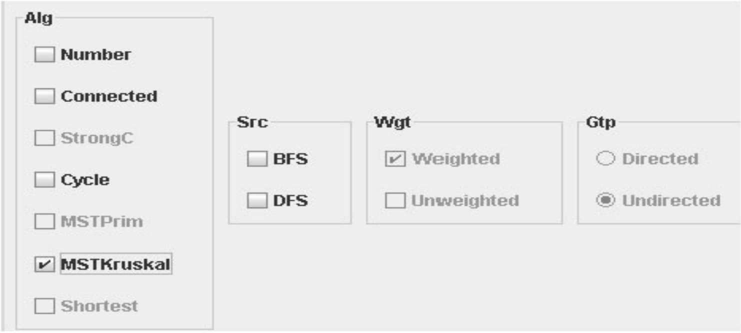
The GUI that is generated from Figure 5 is shown in Figure 7. The state that is shown results from the selection of **`MSTKruskal`** — the **`Weighted`** and **`Undirected`** features are automatically selected as a consequence of constraint propagation. Further, **`Shortest`**, **`MSTPrim`**, **`StrongC`**, **`Unweighted`**, and **`Directed`** are greyed out, meaning that they are no longer selectable as doing so would create an inconsistent specification. Using an LTMS to propagate constraints, users can only create correct specifications. In effect, the generated GUI is a declarative domain-specific language that acts as a syntax-directed editor which prevents users from making certain errors.

Although not illustrated, **`guidsl`** allows additional variables to be declared in the constraint section to define properties. Feature constraints can then be expressed in terms of properties, like that in [4], to support our observations in Section 3.

Another useful capability of LTMSs is to provide a justification for automatically selected/deselected features. We have incorporated this into **`guidsl`**: placing the mouse over a selected feature, a justification (in the form of a proof) is displayed. In the example of Figure 7, the justification for **`Undirected`** being selected is:

```
MSTKruskal because set by user
Undirected because ((MSTKruskal or MSTPrim)) implies
((Undirected and Weighted))
```

Meaning that **`MSTKruskal`** was set by the user, and **`Undirected`** is set because the selection of **`MSTKruskal`** implies **`Undirected`** and **`Weighted`**. More complex explanations are generated as additional selections are made.



Alg

☐ Number
☐ Connected
☐ StrongC
☐ Cycle
☐ MSTPrim
☑ MSTKruskal
☐ Shortest

Src

☐ BFS
☐ DFS

Wgt

☑ Weighted
☐ Unweighted

Gtp

○ Directed
● Undirected

Fig. 7. Generated GUI for the GPL Model

5.2 Debugging Feature Models

Debugging a feature model without tool support is notoriously difficult. When we debugged feature models prior to this work, it was a laborious, painstaking, and error-prone effort to enumerate feature combinations. By equating feature models with propositional formulas, the task of debugging is substantially simplified.

An LTMS is helpful in debugging feature models, but only to a limited extent. Only if users select the right combination of features will a contradiction be exposed. But models need not have contradictions to be wrong (e.g., **`Number implies`**

Weight). More help is needed. Given a propositional formula and a set of variable assignments, a SAT solver can determine whether there is a value assignment to the remaining variables that will satisfy the predicate. Thus, debugging scripts in **guidsl** are simply statements of the form **<S,L>** where **L** is a list of variable assignments and **S** is **true** or **false**. If **S** is **true**, then the SAT solver is expected to confirm that **L** is a compatible set of variable assignments; if **S** is **false**, the solver is expected to confirm that **L** is an incompatible set of assignments. Additional simple automatic tests, not requiring a SAT solver, is to verify that a given combination of features defines a product (i.e., a legal and complete program specification). Both the SAT solver and complete-specification-tests were instrumental in helping us debug the GPL feature model.

It is straightforward to list a large number of tests to validate a model; test suites can be run quickly. (SAT solvers have become very efficient, finding variable assignments for thousands of variables in minutes). Although we cannot prove a model is correct, we are comforted by the fact that we can now run a *much* more thorough set of tests on our models automatically than we could have performed previously.

6 Staged Configuration Models

Staged configuration has recently been proposed as an incremental way to progressively specialize feature models [13][14]. At each stage, different groups or developers make product configuration choices, rather than a configuration being specified by one person at one time. Specializations involve the selection or deselection of features and adding more constraints (e.g., converting a one-or-more selection to single selection).

Staged configuration is accomplished by (1) simplifying the grammar by eliminating choices or making optional choices mandatory, and (2) simplifying the non-grammar constraints. Both are required ([14] addresses grammar simplification). By limiting changes *only* to grammars, it is possible to preselect **MSTKruskal** and deselect **Unweighted** in a staged configuration and adjust the GPL grammar (making **MSTKruskal** mandatory and removing **Unweighted**). But the resulting model is unsatisfiable, as **MSTKruskal** requires **Unweighted**.

A generalization of the GUI presented earlier could be used to accomplish staged specifications. Each selectable feature will require a toggle that allows a feature to be selected (**true**), deselected (**false**), or to postpone its choice to a later stage (**unknown**). In this way, designers can distinguish features that are preselected from those that are permanently removed. The LTMS algorithm remains unchanged; constraints are propagated as before guaranteeing that the resulting model is consistent. Inferred feature selections and deselections can be used to further simplify the grammar and its non-grammar constraints.

More generally, where constraints on non-boolean variables (e.g. performance constraints) are part of a feature model, a more general logic, constraint propagation algorithms, and predicate simplification algorithms will be needed [15]. However, our work applies to many existing feature models, and we believe that current results on staged configuration can be improved for these cases with our suggestions.

7 Related Work

There is a great deal of prior work on feature modeling. For brevity, we focus on the key papers that are relevant. Some feature modeling tools support arbitrary propositional formulas [8][10], but these formulas are validated at product-build time, not incrementally as features are selected. We are aware that technologies that dynamically prune the design space — similar to that presented in this paper — may be known to pockets of researchers in industry (e.g., [1][7][18]), but the basic relationship of feature models, attribute grammars, and propositional formulas does not seem to be widely appreciated or understood.

The connection of feature models to grammars is not new. In 1992, Batory and O'Malley used grammars to specify feature models [3], and in 1997 showed how attribute grammars expressed non-grammar constraints [4]. In 2002, de Jonge and Visser recognized that feature diagrams were context free grammars. Czarnecki, Eisenecker, et al. have since used grammars to simplify feature models during staged configuration [12].

The connection of product-line configurations with propositional formulas is due to Mannion [26]. Beuche [7] and Pure::Variants [27] translate feature models into Prolog. Prolog is used as a constraint inference engine to accomplish the role of an LTMS. Non-grammar constraints are expressed by inclusion and exclusion predicates; while user-defined constraints (i.e., Prolog programs) could be arbitrary. We are unaware of tools that follow from [7] to debug feature models.

Neema, Sztipanovits, and Karsai represent design spaces as trees, where leaves are primitive components and interior nodes are design templates [24]. Constraints among nodes are expressed as OCL predicates, and so too are resource and performance constraints. *Ordered binary decision diagrams (OBDDs)* are used to encode this design space, and operations on OBDDs are used to find solutions (i.e., designs that satisfy constraints), possibly through user-interactions.

Concurrently and independently of our work, Benavides, Trinidad, and Ruiz-Cortes [6] also noted the connection between feature models and propositional formulas, and recognized that handling additional performance, resource, and other constraints is a general *constraint satisfaction problem (CSP)*, which is not limited to the boolean CSP techniques discussed in this paper. We believe their work is a valuable complement to our paper; read together, it is easy to imagine a new and powerful generation of feature modeling tools that leverage automated analyses.

8 Conclusions

In this paper, we integrated existing results to expose a fundamental connection between FDs, grammars, and propositional formulas. This connection has enabled us to leverage light-weight, efficient, and easy-to-build LTMSs and off-the-shelf SAT solvers to bring useful new capabilities to feature modeling tools. LTMSs provide a simple way to propagate constraints as users select features in product specifications. SAT solvers provide automated support to help debug feature models. We believe that the use of LTMSs and SAT solvers in feature model tools is novel. Further, we explained how work on staged configuration models could be improved by integrating non-grammar constraints into a staging process.

We believe that the foundations presented in this paper will be useful in future tools for product-line development.

Acknowledgements. I thank Ben Kuipers and Ray Mooney for directing me to the literature on LTMSs. I gratefully acknowledge the helpful comments from the referees, Mark Grechanik, Jack Sarvela, and Jack Greenfield. I thank Daniel Le Berre for his help with the SAT4J solver. I also thank Jun Yuan, Karim Jamal, and Maria Zolotova for their help in building `guidsl`.

References

[1] American Standard, http://www.americanstandard-us.com/planDesign/

[2] M. Antkiewicz and K. Czarnecki, “FeaturePlugIn: Feature Modeling Plug-In for Eclipse”, OOPSLA’04 Eclipse Technology eXchange (ETX) Workshop, 2004.

[3] D. Batory and S. O’Malley, “The Design and Implementation of Hierarchical Software Systems with Reusable Components”, ACM TOSEM, October 1992.

[4] D. Batory and B.J. Geraci, “Composition Validation and Subjectivity in GenVoca Generators”, IEEE TSE, February 1997, 67-82.

[5] D. Batory, AHEAD Tool Suite, www.cs.utexas.edu/users/schwartz/ATS.html

[6] D. Benavides, P. Trinidad, and A. Ruiz-Cortes, “Automated Reasoning on Feature Models”, Conference on Advanced Information Systems Engineering (CAISE), July 2005.

[7] D. Beuche, “Composition and Construction of Embedded Software Families”, Ph.D. thesis, Otto-von-Guericke-Universitaet, Magdeburg, Germany, 2003.

[8] Big Lever, GEARS tool, http://www.biglever.com/

[9] BMW, http://www.bmwusa.com/

[10] Captain Feature, https://sourceforge.net/projects/captainfeature/

[11] T.H. Cormen, C.E. Leiserson, and R.L.Rivest. Introduction to Algorithms, MIT Press,1990.

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

[13] K. Czarnecki, S. Helsen, and U. Eisenecker, “Formalizing Cardinality-based Feature Models and their Specialization”, Software Process Improvement and Practice, 2005 10(1).

[14] K. Czarnecki, S. Helsen, and U. Eisenecker, “Staged Configuration Through Specialization and Multi-Level Configuration of Feature Models”, Software Process Improvement and Practice, 10(2), 2005.

[15] K. Czarnecki, private correspondence, 2005.

[16] N. Eén and N. Sörensson, “An extensible SAT solver”. 6th International Conference on Theory and Applications of Satisfiability Testing, LNCS 2919, p 502-518, 2003.

[17] K.D. Forbus and J. de Kleer, Building Problem Solvers, MIT Press 1993.

[18] Gateway Computers. http://www.gateway.com/index.shtml

[19] M. Grechanik and D. Batory, “Verification of Dynamically Reconfigurable Applications”, in preparation 2005.

[20] J. Greenfield, K. Short, S. Cook, S. Kent, and J. Crupi, Software Factories: Assembling Applications with Patterns, models, Frameworks and Tools, Wiley, 2004.

[21] M. de Jong and J. Visser, “Grammars as Feature Diagrams”.

[22] D. Streitferdt, M. Riebisch, I. Philippow, “Details of Formalized Relations in Feature Models Using OCL”. ECBS 2003, IEEE Computer Society, 2003, p. 297-304.

[23] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. “Feature-Oriented Domain Analysis (FODA) Feasibility Study”. Technical Report, CMU/SEI-90TR-21, November 1990.

[24] S. Neema, J. Sztipanovits, and G. Karsai, “Constraint-Based Design Space Exploration and Model Synthesis”, EMSOFT 2003, LNCS 2855, p. 290-305.

[25] R.E. Lopez-Herrejon and D. Batory, “A Standard Problem for Evaluating Product-Line Methodologies”, GCSE 2001, September 9-13, 2001 Messe Erfurt, Erfurt, Germany.

[26] M. Mannion, “Using first-order logic for product line model validation”. 2nd Software Product Line Conf. (SPLC2), #2379 in LNCS, 176–187, 2002.

[27] Pure-Systems, “Technical White Paper: Variant Management with pure::variants”, www.pure-systems.com, 2003.

[28] T. Teitelbaum and T. Reps, “The Cornell Program Synthesizer: a Syntax-Directed Programming Environment”, CACM, v.24 n.9, p.563-573, Sept. 1981.

[29] P. Zave, “FAQ Sheet on Feature Interactions”, www.research.att.com/~pamela/faq.html

Using Product Sets to Define Complex Product Decisions

Mark-Oliver Reiser[1,2] and Matthias Weber[2]

[1] Technische Universität Berlin – Fachgebiet Softwaretechnik,
Sekretariat FR5-6, Franklinstraße 28/29, D-10587 Berlin, Germany
moreiser@cs.tu-berlin.de

[2] DaimlerChrysler AG – Research and Technology, REI/SM,
Alt-Moabit 96 A, D-10559 Berlin, Germany
{first.M.Reiser, Matthias.N.Weber}@DaimlerChrysler.com

Abstract. Product family engineering consists of several activities commonly separated into the areas of domain engineering and product engineering. The main part of product engineering is the definition of product decisions, which means in the context of feature modeling that for each feature the product engineer has to define in what products it will be included. In the automotive domain – and probably in many other embedded real-time domains as well – the considerations that influence these feature selections are extremely complex and, at the same time, need to be documented as closely as possible for later reference. In this paper, we (1) present a detailed description of this problem and (2) try to show that existing approaches do not sufficiently meet these concerns. We then (3) provide a detailed definition of product sets as a means to solve the problem and (4) show what methodological implications arise from the use of this concept.

1 Introduction

When applying product family concepts [3,8,13] to the automotive domain, some specific characteristics of embedded real-time systems in general and automotive control software in particular have to be taken into account [9,14]. One of the most important of these characteristics is related to the definition of feature selections, i.e. defining which features will be included in each product: Here, this definition is particularly intricate because it is influenced by a multitude of highly interrelated, complex considerations – ranging from marketing concerns to legislation issues to technical aspects. This is especially true if an automotive manufacturer's vehicle lines are represented as one large product family instead of several independent, smaller product lines.

We believe that current approaches to modeling feature selections – or product decisions – do not fully support this situation and that product sets as defined below can help solve this problem when applied in a certain methodological manner. To demonstrate this, we first give a summary of the state of the art of feature selection modeling in Section 2. Then, we describe the aforementioned

H. Obbink and K. Pohl (Eds.): SPLC 2005, LNCS 3714, pp. 21–32, 2005.

difficulty of feature selection modeling in the automotive domain in more detail (Section 3). Next, we provide a detailed description and definition of product sets (Section 4 and 5), followed by a discussion of the methodological implications of these concepts and their advantages and disadvantages (Section 6). The final two sections are devoted to discussion of related work and concluding remarks.

2 State of the Art

Feature modeling is a well-established means for product family domain engineering [4,7,10,11,12]. Each feature represents a certain characteristic that the individual products of the product family may have and can therefore be selected or unselected with respect to a certain product. In other words, an individual product is defined by the selection status of all features.

Features are commonly organized in a hierarchical manner in the form of feature trees. Figure 1 shows an example of a notation for such trees, as described in [4]. Each feature may have one or more child features which may be selected if the parent feature is selected. Child features can be marked as mandatory (filled circle) or optional (empty circle) and two or more children may be declared alternative, meaning that only one of them can be selected while the others have to be unselected (arc connecting the lines to the alternative children).

In addition, more complex restrictions may be defined: one feature may "need" another feature – indicating that it may only be selected if the other one is selected as well – or a feature may "exclude" another feature – meaning that the excluded feature may no longer be selected if the excluding feature is selected. Such constraints are usually depicted as arrows marked "needs" or "excludes", as shown in Figure 1.

There are many variations of the notation used here, and sometimes a greater degree of flexibility is provided for defining constraints and dependencies between features. For an overview, please refer to [1,4,5]. However, the discussion below is also applicable to such similar feature-modeling techniques.

One of the main purposes of feature modeling is to provide a basis for the production of individual products during product engineering. More precisely, depending on the selection of features, the software assets will be configured or

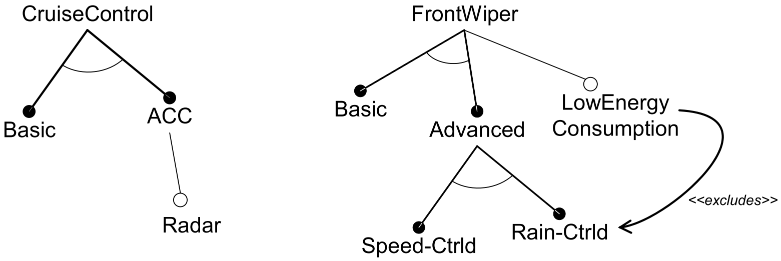


Fig. 1. Example of a feature tree

adapted and will then be composed to form the final product. The details of this production mechanism strongly depend on the type of assets in use – e.g. requirements data bases, user documentation, class diagrams, program code, test cases – and are beyond the scope of this paper. The important fact here is that the definition of feature selections is a key aspect of applying feature models during the product engineering phase.

Several techniques are currently in use to define such feature selections or product decisions. To provide an overview, we have attempted to divide them into five different basic approaches:

1. List of selected features
2. Selection criteria for features
3. Links from features to a product model
4. Links from a product model to features
5. Combination of 2. and either 3. or 4.

In some cases where there is only a small number of individual products, simply listing the selected features for each product is perfectly feasible. The product engineer explicitly states for each individual product and for each feature whether or not it will be included in the respective product. At first sight, this may seem to apply to only a small number of trivial cases because, if the number of products is small, there may appear to be no need for product family engineering concepts at all. But this is not true. Even if the number of delivered products is as low as three or four, there may be hundreds of features that need to be considered and therefore an elaborate domain engineering could well make sense. Another situation in which this first approach is often sufficient is where there is no distinction between a customer-driven configuration and an internal preconfiguration by development engineers and management personnel.

However, in more complex situations in which there is a huge number of individual products or in which a preconfiguration is required, feature selections cannot be defined for each product/feature combination explicitly. A very straightforward solution is to attribute each feature with a logical statement (e.g. [4]), which we will henceforth call "selection criterion". This selection criterion refers to attributes of the individual products such as Country (the country where the product will be offered), Chassis (station wagon, etc.) or Engine. If and only if the selection criterion is true will the corresponding feature be selected. This approach is highly flexible and scales very well and a slightly simplified form of it has proved viable in development projects at DaimlerChrysler. But it also has some severe methodological shortcomings, which are outlined in Section 3. Apart from these, there is the disadvantage that it is no longer defined what products will be on offer, i.e. what combinations of product attributes are valid.

The other two approaches (3. and 4. in the above list) solve this problem by providing a "product model" in addition to the feature tree. This is a model of all available individual products, usually organized in tree form, see Figure 2.

Feature selections can then either be defined by a link from the product element to an included feature (the link is called "includes") or vice versa (then

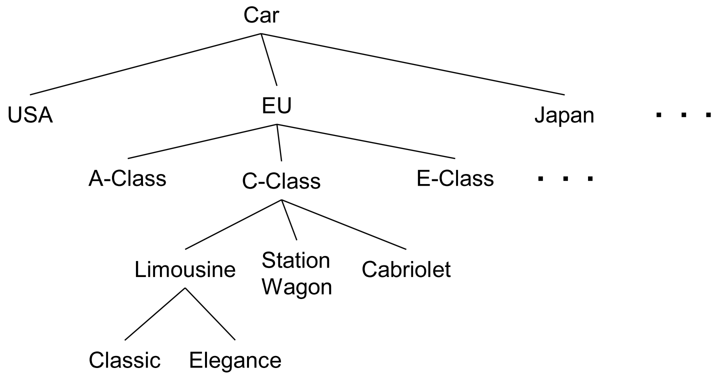


Fig. 2. Example of a product model in tree form

called "included in"), see Figure 3. Similarly, "excludes" or "excluded by" links can be used to state that a feature is not part of a product. Since lower-level product elements now inherit the "includes"/"excludes" links from their ancestors – or lower-level features inherit the "included in"/"excluded by" links – not all feature selections have to be defined explicitly. But, from that perspective, this approach is still less efficient than the selection criterion approach.

Note that the product tree can also be viewed as being part of the feature tree itself, which makes no difference from a conceptual point of view. The products then simply become features, i.e. a "U.S. station wagon" becomes a feature a certain product may have. The "includes" or "included in" links can be realized by "needs" links. The advantage is that this solution manages with fewer element and link types, while the separation of trees helps to prevent misunderstandings and enforces a certain methodological approach connected to the idea of product models.

Finally, the approaches can be combined by having the selection criteria refer to a product model in addition to (or instead of) product attributes.

3 Problem Description

Listing all selected features for each possible product individually is not feasible if the number of products is very large. This is especially true for the automotive domain, [2,14]. There, the products have to be distinguished at least by the market the car is being built for (e.g. EU, US, Japan, ...), the vehicle line (e.g. A-Class, C-Class, E-Class for Mercedes-Benz), the body type (e.g. Limousine, Station Wagon, Cabriolet), the engine types, the transmission types, and a style category (e.g. Classic, Elegance). If we assume that there are on average three values for each parameter, we get approx. three to the power of six, i.e. some 700 different products, which is still a conservative estimate. Of course, this figure must be reduced somewhat because not all combinations reflect products actually being offered. But even if this reduced the number by 50%, there would

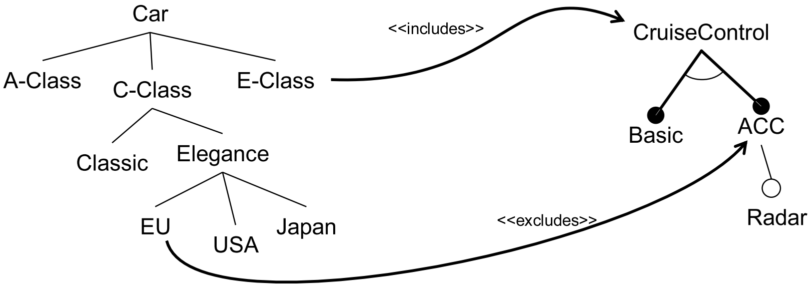


Fig. 3. Feature selection with "includes" links

still be about 350 products left to be configured. And this does not even take into account the customer configuration.

When using a product model in tree form, the various product criteria (such as country, body type, etc.) are put in a certain order depending on what criteria are applied on each of the tree's levels. This means that the elements representing the values of criteria on lower levels are spread over multiple branches of the tree.

Example 1. Let us assume that we have two product criteria: country (with values EU and USA) and body type (with Limousine, Station Wagon and Cabriolet). When the body type is used for distinction on the tree's first level and the country on the second level, we obtain multiple elements that only when taken together represent a certain market (see left tree in Figure 4). In the example, we thus have three elements representing the U.S. market and another three for the European market. No matter how the product tree is organized, this situation remains basically unchanged. The only thing that changes is the criterion for which the value elements are spread (compare left and right tree in Figure 4).

This separation leads to the problem that statements such as "all cars for the U.S. market have cruise control" cannot be defined directly but have to be partly defined at different locations in the product tree, i.e. an "includes" link has to be defined for each USA element. Technically, this makes no difference. But from a methodological point of view, it is desirable to document feature selection decisions as closely as possible because there is a rationale behind each of these decisions that has to be documented and taken into account when changing the

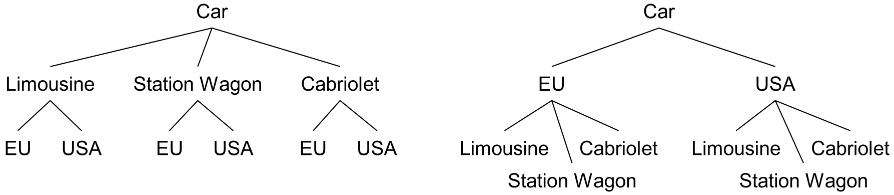


Fig. 4. Spreading of information in feature trees

product tree or the feature selections later on, e.g. when a new body type is being added to the product tree.

This problem of product trees can be avoided by using feature selection criteria that refer to product attributes (approach no. 2 in the above list). But then a similar problem arises. What if we wished to define that not only the feature CruiseControl is expected as standard equipment in all cars in the U.S., but also some other features such as automatic transmission ? Of course, we could define this in the selection criteria of the corresponding features. But the fact that all these selections share the same rationale is lost.

Moreover, the selection criteria of these features will also be influenced by other considerations and it is not documented how the different considerations for a single feature led to the feature's final selection criterion which was recorded.

Example 2. If we wished to state that a certain feature F will be included in all cars for the entire North American market for some reason (perhaps because all competitors offer it as standard equipment or it is traditionally expected by all customers there), and at the same time it has to be included in Canadian cars for some other reason (perhaps owing to special Canadian legislation), the final selection criterion for feature F will only state that the feature is included in the U.S. and Canada – at least in the optimal but unrealistic case that no other considerations influence the criterion even further. When the motivation for one of the individual selection statements is no longer justified or when a market is split in two (e.g. the French- and English-speaking parts of Canada), it is difficult – if not impossible – to decide how this change affects such a "combined" selection criterion.

The same applies to the other forms of feature selection definition. Of course, all this additional information could be put into some separate documentation on the feature model, but this would lead to a new source of inconsistency.

In the automotive domain, such complex and "orthogonal" considerations affecting a single feature's selection are very common in the preconfiguration of products and therefore a more sophisticated form of modeling is needed, [9].

4 Product Sets

In this section, we begin by describing the concept of product sets in an informal manner and then, in a second step, give a more precise and formal definition.

4.1 Concept

The notion of product sets is based on the distinction of a feature tree and a product tree. As outlined in Section 2, the feature and product trees may also be viewed as two branches of one large feature tree, but for the sake of clarity we will henceforth assume two separate data structures. Similarly, product attributes could be used instead of or in addition to the product tree without essentially changing the concept.

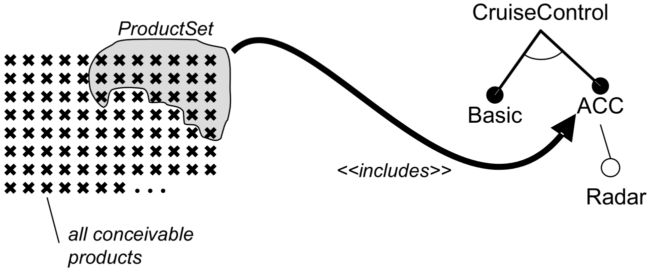


Fig. 5. Schematic description of product sets

Instead of directly linking the product elements of the product tree to features, or the other way round, a dedicated kind of element is used for feature selection definition: the *product set.* As the name suggests, each product set represents one or more individual products. Several product sets may represent the same set or intersecting sets of individual products. In other words, two product sets S_1 and S_2 may, for example, both refer to "all station wagons for the U.S. market". The purpose of a product set is to define properties of products, especially feature selection. To achieve this, product sets may be linked to features using directed "include" links, meaning that the corresponding feature and all its descendants will be included in all the products the product set refers to (see Figure 5). Correspondingly, "exclude" links may be used to show that a certain feature is not included in the products the product set refers to, even though "include" links may have been defined for this feature by other product sets referring to the same products. Put briefly, "exclude" links have a higher priority than "include" links with respect to one single feature. If there is a conflict between "include" and "exclude" links pointing to features on different levels of the feature tree (i.e. if the features pointed to are descendants and ancestors of each other), the link pointing to the lower-level feature has priority with respect to this feature and all its descendants.

It is an important characteristic of the product set concept that not all possible products necessarily need to be modeled explicitly, i.e. the small crosses in Figure 5 could represent the leaves of a product tree (all products explicitly modeled), but they could also simply represent certain combinations of product attribute values (actual products not explicitly modeled).

Finally, product sets also have a textual documentation containing the rationale for the corresponding inclusion or exclusion as well as other meta-information, e.g. the name of the person who created it or is responsible for it. The significance of this is described in detail in Section 6.

4.2 Definition

Let P be the set of all possible products and F the set of all features. Then a product set S, out of the set of all product sets PS, is defined as a 3-tuple $S = (R, I, E)$ with ...

$$R \subseteq P \tag{1}$$
$$I, E \subseteq F \tag{2}$$
$$I \cap E = \phi \tag{3}$$

The set of products affected by product set S, its *range*, is denoted by $R(S)$. The set I of features included by S is denoted by $Inc(S)$, the set E of excluded features by $Exc(S)$. As a short form, we define ...

$$p \in S \Leftrightarrow p \in R(S) \tag{4}$$

To keep the main definition below from becoming too complex, we need two auxiliary relations, defined as ...

$$Inc^* \subseteq P \times F \tag{5}$$
$$p\ Inc^* f \Leftrightarrow \exists\ S \in PS :\ \ p \in S\ \wedge\ f \in Inc(S) \tag{6}$$

$$Exc^* \subseteq P \times F \tag{7}$$
$$p\ Exc^* f \Leftrightarrow \exists\ S \in PS :\ \ p \in S\ \wedge\ f \in Exc(S) \tag{8}$$

This means that product p is Inc^*-related to feature f if and only if there exists a product set that *directly* defines f as being included in p. However, this does not prove whether f is really included in p. For this, the parents of f in the feature tree and the “excludes” relations need to be considered as well. This is the next relation’s task. But beforehand, a definition of the feature tree is needed. This is provided in the form of the following two (partial) functions:

$$parent : F \rightarrow F \tag{9}$$
$$isMandatory : F \rightarrow \{true, false\} \tag{10}$$

The relation Sel_{PS} (i.e. “selected in”) now specifies whether a certain feature is selected by way of product sets in a certain product.

$$Sel_{PS} \subseteq F \times P \tag{11}$$
$$\begin{aligned} f\ Sel_{PS}\ p \Leftrightarrow & [p\ Inc^* f\ \vee\ (parent(f)\ Sel_{PS}\ p\ \wedge\ isMandatory(f))] \\ & \wedge \neg\ p\ Exc^* f \end{aligned} \tag{12}$$

From that, we can deduce various conclusions. For example:

$$\forall\ p \in P, f \in F :\ \ p\ Inc^* f\ \wedge\ p\ Exc^* f \Rightarrow\ \neg\ f\ Sel_{PS}\ p \tag{13}$$

In this context, we are mainly interested in feature selection. But product sets may also be used to specify other information on certain individual products. For example, the range of products actually being offered could be defined with one or more product sets if product attributes are being used without a product model.

5 Combining Product Sets and Selection Criteria

Product sets alone are already a sufficient means for defining feature selection. But they are also very well suited for combination with selection criteria. In this section, we outline how this can be done technically, before describing the methodological benefits of such a combined approach in the next section.

A straightforward solution is to view product sets and selection criteria as two independent means of expression for the same thing: Will a certain feature be selected in a certain product ? The relation Sel_{SC} states whether a feature is selected through the mechanism of selection criteria and is defined as ...

$$Sel_{SC} \subseteq P \times F \tag{14}$$

$$\begin{aligned} f\ Sel_{SC}\ p \Leftrightarrow &(selectionCriterion(f)\ \vee\ isMandatory(f)) \\ &\wedge parent(f)\ Sel_{SC}\ p \end{aligned} \tag{15}$$

A feature's overall selection is then defined by relation Sel:

$$Sel \subseteq P \times F \tag{16}$$

$$f\ Sel\ p \Leftrightarrow f\ Sel_{PS}\ p\ \wedge\ f\ Sel_{SC}\ p \tag{17}$$

Alternatively, product sets could also be prioritized. If a feature was directly included by a product set, i.e. there existed some product set with an "includes" link leading to this feature, the feature would be selected, whether through its selection criterion or not. Correspondingly, a directly excluded feature would be excluded regardless of its selection criterion.

Then, the overall selection relation became a modified version of the product set definition in (12):

$$Sel \subseteq F \times P \tag{18}$$

$$\begin{aligned} f\ Sel\ p \Leftrightarrow [&p\ Inc^{*} f\ \vee\ (parent(f)\ Sel\ p\ \wedge \\ &selectionCriterion(f))]\ \wedge\ \neg\ p\ Exc^{*} f \end{aligned} \tag{19}$$

This means that the selection criteria played a role only with respect to the automatic selection of the descendants of an included feature and could be "overwritten" by product set inclusion and exclusion.

6 Discussion and Methodological Implications

When using product sets alone, most of the problems described in Section 3 can be solved. All considerations that influence the decision as to whether a certain feature is selected in a certain product, together with their precise impact on that decision, can be recorded for later reference. Not as a separate documentation that may become outdated and inconsistent if not maintained carefully, but as a constituent of the actual feature selection definition. How these different considerations are combined when affecting the same products and/or features is part of the definition of product sets as shown in Section 4. Thus, the considerations do not have to be "hidden" in a single "combined" selection definition for each product or feature (depending on the approach).

Example 3. When applying product sets to Example 2, we see that now two product sets are used to represent this situation: (a) one that states that feature F will be included in all North American cars and (b) one that states that it will be included in Canadian cars. Moreover, each product set will give a rationale for the feature selection decision expressed by the product set and will name a person who is responsible for it. This additional information then helps solving conflicts when incorporating changes to feature selection considerations: for example, when the rationale for (b) became invalid, (b) would be removed but nevertheless it would be clear that Canadian cars still have to include F because of (a).

However, these benefits cannot be obtained without a drawback: the process of manipulating the feature selection definition becomes somewhat more complex. But, with appropriate tool support we believe that this can be dealt with. For example, a tool that provides a table view of all product sets could offer a filter mechanism enabling the user to easily find all product sets that affect a certain product or a certain feature or a certain product/feature pair. Moreover, if the user were to create a new product set with one or more "include" links, the tool could pop up a warning if there were other product sets already defined that excluded the same feature(s) for all or some of the same products. In such a situation, the potential of product sets becomes obvious: the user would see exactly what other product sets are in conflict with what he had in mind and could then consult the attached descriptions of the rationale behind them or could get in touch with the person declared responsible for them.

All this applies regardless of whether product sets are used alone or in conjunction with selection criteria. But, in the latter case, additional implications have to be considered. Through the combination of the two concepts, two alternative means of expression are now available for each product/feature selection definition. Such redundancy in expressiveness only makes sense if it can be justified from a methodological point of view and if guidelines can be formulated specifying when to use which form of definition. To find an answer, it is useful to first examine what viewpoint the user is adopting while using them. When defining feature selections through product sets, he states that several specific products include or exclude several features. The view is directed from products to features, while having an outlook over all features. On the other hand, when using selection criteria to define feature selections, the engineer has to formulate a criterion for each single feature separately. Thus, he is always thinking in the context of a certain feature. We believe that the first situation – applying product sets – perfectly matches the marketing and management viewpoint, whereas the second situation – using selection criteria – fits the technical viewpoint very well.

Of course, this distinction is not clear in all cases and technically motivated feature selections can also be defined with product sets. But we believe that combining them with selection criteria is an interesting way to separate these two viewpoints. Another possibility would be to solely use product sets but enable the user to organize them into groups. These groups of product sets would not be taken into account in the product set definition and therefore would not have any direct impact on the feature selections. However, they would help separate different viewpoints and aspects that influence the selections.

7 Related Work

The idea of product sets as defined above originated in the ITEA project EAST-EEA, which ended in 2004, [6,15]. The work presented here is a refinement and extension of this initial idea. For example the precise definition has been formulated, "exclude" links have been added, and conflicts between several product sets and between product sets and selection criteria have been discussed. Moreover, the concept has been applied to several examples from the automotive domain.

Another approach for coping with the complexity of feature selection definition is *staged configuration*, as presented in [5]: configuration is organized as a number of consecutive steps, each further specializing the feature model or parts of it. But we believe that this approach is not sufficient to solve all the problems described in Section 3, particularly the spreading of selection considerations over several features' selection definitions and the need for an explicit documentation of "orthogonal" and overlapping considerations. This is especially true for non-technically motivated feature selection (driven by product strategy, marketing, legislation, etc.). However, it should be used as an advanced alternative to selection criteria and can be combined with product sets in a way similar to that described in Section 5. Whether only staged configuration or only product sets are applied, or a combination of the two, probably depends on the specific needs of the user during product engineering. But additional work is needed to reach definite conclusions on this.

8 Conclusion

In this paper, we have addressed the issue of product engineering in the form of feature selection definition. The motivation for doing so was the fact that in the automotive domain such a definition is extremely complex. Moreover, the different considerations influencing feature selection have to be documented in a way that makes them available for future reference when new or changed considerations have to be integrated.

To meet this challenge, we have presented a precise definition of product sets, which can be combined with another form of feature selection definition: selection criteria. Finally, we have described the methodological impact of these concepts and how they contribute to solving the above problem, especially when appropriate tool support is provided.

In the near future, we will further examine the concept of product sets, particularly the prioritization of product sets and the question how feature selection with product sets can be appropriately supported by tools.

References

1. G. Böckle, P. Knauber, K. Pohl, K. Schmidt: "Software Produktlinien". dpunkt Verlag, 2004.
2. S. Bühne, K. Lauenroth, K. Pohl, M. Weber: "Modeling Features for Multi-Criteria Product-Lines in Automotive Industry". Workshop on Software Engineering for Automotive Systems (SEAS), at ICSE 2004, Edinburgh, 2004.
3. P. Clements, L. Northrop: "Software Product Lines: Practices and Patterns". Addison-Wesley, 2002.
4. K. Czarnecki, U. W. Eisenecker: "Generative Programming – Methods, Tools and Applications". Addison-Wesley, 2000.
5. K. Czarnecki, S. Helsen, U. W. Eisenecker: "Staged Configuration Using Feature Models". In: *Proceedings of the Third International Conference, SPLC 2004, USA*, LNCS 3154, pp. 266-283, 2004.
6. ITEA EAST-EEA Project Web-Site: http://www.east-eea.net
7. D. Fey, R. Fajta, A. Boros: "Feature Modeling – A Meta-Model to Enhance Usability and Usefulness". In: *Proceedings of the Second International Conference, SPLC 2, USA*, LNCS 2379, pp. 198-216, 2002.
8. J. Greenfield, K. Short, et. al.: "Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools". Wiley, 2004.
9. K. Grimm: "Software Technology in an Automotive Company – Major Challenges". In: *Proceedings of the 25th International Conference on Software Engineering, May 3-10, 2003*, IEEE Computer Society, pp. 498-503, 2003
10. 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, 1990.
11. K. C. Kang, S. Kim, J. Lee, E. Shin, M. Huh: "FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures". Annals of Software Engineering 5, 1998.
12. K. C. Kang, J. Lee, P. Donohoe: "Feature-Oriented Product Line Enginering". In: IEEE Software, Vol. 19, pp. 58-65, 2002.
13. K. Pohl, G. Böckle, F. van der Linden: "Software Product Line Engineering: Foundations, Principles and Techniques". Springer, Heidelberg, 2005.
14. M. Weber, J. Weisbrod: "Requirements Engineering in Automotive Development – Experiences and Challenges". RE2002, pp. 331-340, 2002.
15. M. Weber, U. Freund, H. Lonn, et al.: "An Architecture Description Language for Developing Automotive ECU-Software". INCOSE 2004, Toulouse, France, 2004.

The PLUSS Approach - Domain Modeling with Features, Use Cases and Use Case Realizations

Magnus Eriksson[1], Jürgen Börstler[2], and Kjell Borg[1]

[1] Land Systems Hägglunds AB, SE-891 82 Örnsköldsvik, Sweden
{Magnus.Eriksson, Kjell.Borg}@baesystems.se
[2] Dept. of Computing Science, Umeå University, SE-901 87 Umeå, Sweden
jubo@cs.umu.se

Abstract. This paper describes a product line use case modeling approach tailored towards organizations developing and maintaining extremely long lived software intensive systems. We refer to the approach as the PLUSS approach, *Product Line Use case modeling for Systems and Software engineering*. An industrial case study is presented where PLUSS is applied and evaluated in the target domain. Based on the case study data we draw the conclusion that PLUSS performs better than modeling according to the styles and guidelines specified by the IBM-Rational Unified Process (RUP) in the current industrial context.

1 Introduction

Software intensive defense systems, for example vehicles, are developed in short series. They are always customized for different customer needs and they are expected to have an extremely long life span, often 30 years or longer. For an organization to be competitive in a market like this it is important to achieve high levels of reuse and effective maintenance. An interesting approach to address issues like these, which has gained considerable attention both by industry and academia over the last few years, is known as software product line development. The basic idea of this approach is to use domain knowledge to identify common parts within a family of products and to separate them from the differences between the products. The commonalties are then used to create a product platform that can be used as a common baseline for all products within the product family.

For embedded software we believe it is important that product line concepts such as domain modeling are also introduced into the systems engineering process, since embedded software requirements are for the most part not posed by customers or end users, but by systems engineering and the systems architecture. Due to earlier positive single system experiences with use cases, we are therefore interested in identifying a use case driven product line approach that can be applied by both our systems and software engineering teams. Unfortunately, we see a number of problems with existing approaches to product line use case modeling. To address issues in existing approaches we have developed a domain modeling approach that utilizes features [10], use cases and use case realizations [12]. For the remainder of this paper the

H. Obbink and K. Pohl (Eds.): SPLC 2005, LNCS 3714, pp. 33 – 44, 2005.

approach will be referred to as PLUSS (*Product Line Use case modeling for Systems and Software engineering*).

The UML Use case meta-model [19] provides poor assistance in modeling variability [16]. A number of suggestions addressing this issue are described in the literature. Von der Maβen and Lichter suggest that the UML use case meta-model should be extended by two new relationships, "Option" and "Alternative" [16]. Jacobson et al. suggest using the "generalization" and "extend" relationships to model variability in UML use case diagrams [9]. We do however see a fundamental problem with using use case diagrams for describing variants. Use case diagrams tend to get cluttered to a degree where it is impossible to get an overview of the variants within a family. It is furthermore not enough to only manage variability among whole use cases. It must also be possible to specify variant behavior within use cases. There have been some proposals on how to do this in the literature, for example the PLUC notation [5] and RSEB parameters [9]. However, like the UML approaches above these approaches do not have any means to provide a good overview of the variants within a family. Most existing product line use case modeling approaches also lack strong mechanisms to trace variant behavior to the system design and they are document, not model driven. Using documents instead of a common model is a major maintenance concern working on extremely long lived systems. Product instantiation in a document driven approach typically involves copying documents and removing variant information. This is not good from a long term maintenance perspective since information is being duplicated.

Our approach is based on the work by Griss et al. on FeatuRSEB [8]. Like Griss et al. we argue that feature models are better suited for domain modeling than UML use case diagrams and that a feature model therefore should be used as the high level view of a product family. In FeatuRSEB a feature model is added to the 4+1 view model adopted by Jacobson et al. in RSEB [9]. The feature model in FeatuRSEB takes "center stage" and provides a high-level view of the domain architecture and the reusable assets in the product family. Even though a feature model is also used in our approach to provide a high-level view of the variability within a product family, a fundamental difference exists between PLUSS and FeatuRSEB. In PLUSS the primary purpose of the feature model is not to take "center stage", but rather to be a tool for visualizing variants in our abstract product family use case model. We maintain one complete use case model for the whole system family and we use the feature model as a tool for instantiating that abstract family model into concrete product use case models for each system built within the family.

The main contributions of this paper are: An improved approach to manage variant behavior in use case models, stronger means to trace variant use case behavior to the system design and stronger means to generate product use case models from a common family model.

The remainder of the paper is organized as follows: Section 2 provides an introduction to PLUSS feature modeling. Section 3 describes PLUSS Use case modeling and how the PLUSS feature model relate to the use cases. Section 3 also describes the PLUSS notation for describing variants in use case scenarios and how product use case models are instantiated form a family model. Section 4 presents an industrial case study in which the PLUSS approach is applied and evaluated in its target domain. In section 5, we summarizes the paper and draw conclusions.

2 Feature Modeling

Kang et al. first proposed use of feature models in 1990 as part of the Feature Oriented Domain Analysis (FODA) [10]. A feature is defined as a prominent or distinctive user-visible aspect, quality or characteristic of a system in FODA. In feature models, features are organized into trees of AND and OR nodes that represent the commonalties and variations in the modeled domain. General features are located at the top of the tree and more refined features are located below. Originally, FODA described "Mandatory", "Optional" and "Alternative" features that may have the relations "requires" and "excludes" to other features. Mandatory features are available in all systems built within a family. Optional features represent variability within a family that may or may not be included in products. Alternative features represent an "exactly-one-out-of-many" selection that has to be made among a set of features. A "requires" relationship indicates that a feature depends on some other feature to make sense in a system. An "excludes" relationship between two features indicates that both features can not be included in the same system.

FODA has no defined mechanism to specify the relation "at-least-one-out-of-many" [6]. Our experience has shown that this is an important shortcoming. We address this issue by defining a new feature type called "Multiple Adaptor" in PLUSS. This feature type is similar to FODA's alternative features, but instead of representing the "exactly-one-out-of-many" relationship, it captures the missing relationship. Its name follows the naming scheme proposed by Mannion et al. for the equivalent relation in their work on reusable requirements [14]. We have also chosen to rename alternative features to "Single Adaptor" features following the same naming scheme. The feature modeling notation used in PLUSS is based on the FODA notation but it has been slightly modified to better suit our modeling needs as shown in **Fig. 1**. As in the original notation a filled black circle represents a mandatory feature and a non-filled circle represents an optional feature. Single and multiple adaptor features are represented by the letters 'S' and 'M' surrounded by a circle.

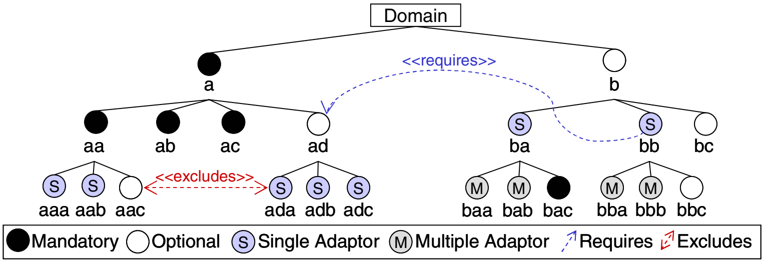


Fig. 1. An example feature model in the PLUSS notation

To further clarify the PLUSS notation, we have created a mapping between PLUSS feature constructs and multiplicities [19] as shown in **Fig. 2**. As shown in **Fig. 2** we have also identified a feature construct that should be avoided. Our experience has shown that this construct, a set containing only optional feature leaf nodes,

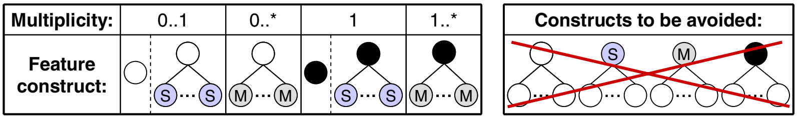


Fig. 2. Feature constructs vs. multiplicities, and constructs to be avoided in PLUSS

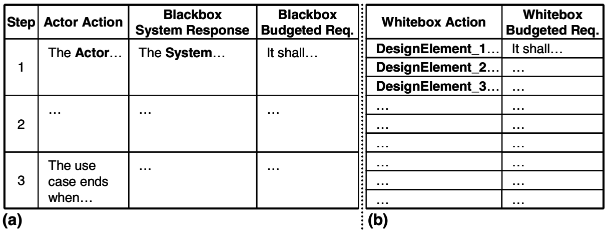
encourages misuse of the refinement relation used for building the feature tree. This construct typically appear when a set of multiple adaptor features is mistaken for a set of optional features.

One shortcoming of the PLUSS feature modeling notation, compared to for example Czarnecki et al. more expressive Cardinality-based notation [2], is the inability to model n..m multiplicity. Our experience has however shown that such constructs are not needed to capture the different types of variability the can exist in product family use case models. We therefore exclude cardinalities from our notation for the purpose of improved readability.

3 Use Case Modeling

As we described in [4], we have chosen to adopt the so called "Black Box Flow of Events" notation described in the Rational Unified Process for Systems Engineering (RUP-SE) [17] shown in **Fig. 3** (a) for describing use case scenarios. This notation is used for tabular descriptions of use case scenarios in natural language. We argue that the notation has two major advantages over tradition natural language scenario descriptions. It forces analysts to always think about interfaces since separate fields exist for describing actor and system actions. It also provides a strong mechanism to relate non-functional requirements to use cases using the "*Blackbox Budgeted Requirements"* column.

A use case realization describes how a particular use case is realized within the system design in terms of collaborating design elements [12]. As we described in [4], we have chosen to describe use case realizations in natural language description based on the RUP-SE "White Box Flow of Events" [17] shown in **Fig. 3** (b). We have chosen natural language descriptions of use case scenarios and use case realizations since the PLUSS approach must be applicable for both systems and software engineering. This increases the number and diversity of stakeholders interested in the models and thereby makes for example UML unsuitable for the purpose. Our natural language descriptions can however be supplemented with UML diagrams as needed.



Step	Actor Action	Blackbox System Response	Blackbox Budgeted Req.
1	The **Actor**...	The **System**...	It shall...
2	...	...	...
3	The use case ends when...	...	...

(a)

Whitebox Action	Whitebox Budgeted Req.
DesignElement_1...	It shall...
DesignElement_2...	...
DesignElement_3...	...
...	...
...	...
...	...
...	...
...	...
...	...

(b)

Fig. 3. The (a) Blackbox flow of events used for describing use case scenarios, and (b) the Whitebox flow of events used for describing use case realizations

3.1 The PLUSS Approach to Modeling Variants in Use Case Models

As mentioned in section 1, the basic idea of PLUSS is to maintain one common and complete use case model for whole product family. To do this, it must be possible to manage variability in the model. We have identified four types of variants that can exist in use case models for product families. The first type regards whole use case that can vary between systems built within a product family. We model this by relating one or more use cases with a feature of any type in the feature model. The second type of variability regards the set of included use case scenarios within each use case. We model this by relating one or more scenarios with a feature of any type in the feature model. The third type regards the set of included steps in each use case scenario. We model this by relating scenario steps with features of any type in the feature model. The fourth and final type of variability regards cross-cutting aspects that can affect several use cases on several levels. Cross-cutting aspects are modeled as use case parameters in PLUSS, these parameters must be related to a set of single adaptor features in the feature model. Gomaa [7] proposed to model each feature as a use case package. PLUSS extended this idea, saying that possibly a whole set of features compose a use case package. This have the advantage of enabling us to also visualize variants within use cases specifications using the feature model.

A meta-model for integration of features, use cases and use case realizations is shown in **Fig. 4.** It describes how use cases, scenarios and scenario steps are included by feature selections. This meta-model is an extension of the meta-model presented in [4] that also show how these included use case scenario steps prescribes a certain set of design element via use case realizations. Variant use case behavior is thereby traced to the system design.

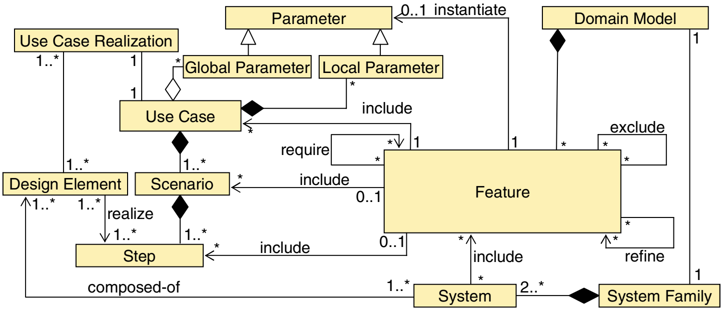


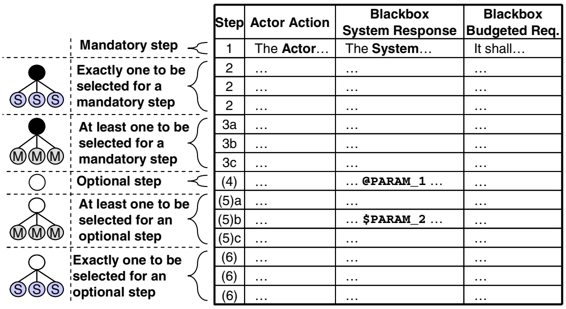
Fig. 4. The PLUSS Meta-model

Change cases, first proposed by Ecklund et al. [3], are basically use case that specifies anticipated changes to a system. Change cases also provide the relation "impact link" that creates traceability to use cases whose implementations are affected if the change case is implemented. In PLUSS, change cases are primarily used to mark proposed, but not yet accepted functionality in a domain. New requirements are first modeled as change cases, however once accepted for implementation in a system within a family, these change cases are transformed to use cases.

3.2 The PLUSS Notation for Describing Variants in Use Case Specifications

As we described in [4], the step identifier of the blackbox flow of events notation discussed in section 3 can be extended to describe variants in use case scenarios as shown in **Fig. 5**. A step identified by a number describes a mandatory step in the scenario, as it does in the original notation. Several steps identified with the same number identify a number of mutually exclusive alternatives for one mandatory step in the scenario. These steps must be related to a set of single adaptor features with a mandatory parent in the feature model. Several steps identified with the same number and a consecutive letter identify a number of alternatives for one mandatory step in the scenario out of which at least one must be selected. These steps must be related to a set of multiple adaptor features with a mandatory parent in the feature model. A step identified by a number within parenthesis identifies an optional step in the scenario. Optional steps must be related to an optional feature in the feature model. Several steps identified with the same number within parenthesis and a consecutive letter identify a number of alternatives for one optional step in the scenario out of which at least one must be selected. These steps must be related to a set of multiple adaptor features with an optional parent feature in the feature model. Several steps identified with the same number within parenthesis identify a number of mutually exclusive alternatives for one optional step in the scenario. These steps must be related to a set of single adaptor features with an optional parent in the feature model.

Jacobson et al. introduced the concept of use case parameters as part of the RSEB in [9]. Mannion et al. distinguished between local parameters and global parameters in their work on reusable natural language requirements [14]. We find this distinction useful also when working with use cases. In PLUSS, the scope of a local parameter is the use case in which it resides and the scope of a global parameter is the whole domain model. Like Mannion et al. we use the symbols ‘$’ and ‘@’ respectively to denote local and global parameters as shown in step ‘(4)’ and ‘(5)b’ of **Fig. 5**.


Mandatory step
S S S
Exactly one to be selected for a mandatory step
M M M
At least one to be selected for a mandatory step
Optional step
M M M
At least one to be selected for an optional step
S S S
Exactly one to be selected for an optional step

Step	Actor Action	Blackbox System Response	Blackbox Budgeted Req.
1	The **Actor**...	The **System**...	It shall...
2	...	...	...
2	...	...	...
2	...	...	...
3a	...	...	...
3b	...	...	...
3c	...	...	...
(4)	...	... @PARAM_1 ...	...
(5)a	...	...	...
(5)b	...	... $PARAM_2 ...	...
(5)c	...	...	...
(6)	...	...	...
(6)	...	...	...
(6)	...	...	...

Fig. 5. The PLUSS notation for describing variants in use case scenarios

3.3 Product Instantiation in the PLUSS Approach

Although the actual organization may vary, typically, when a new product is going to be added to a product family, initial requirements analysis is performed by a product team. This analysis will result in a set of change requests (CR) regarding new requirements (change cases) to be added to the domain model and regarding features that should be included in the new system. The domain engineering team is then responsible for performing change impact analyses on the change requests. A domain engineering change control board (CCB) may then decide if the requested set of requirements will be allowed in the product. Since a common use case model is maintaining for a whole product family in PLUSS, product instantiation is then basically done by adding any new requirements to the model and then using the feature model to choose among its variants. The set of included features directly correspond to a specific set of included use cases for the product. A product use case model is then generated by applying a filter to the domain model sorting out features not included in the current system. This will result in three types of reports: A "*Use Case Model Survey"* including all use cases included in the product, and "*Use Case Specifications",* and "*Use Case Realizations*" for all use case in the survey.

4 Case Study

The objective of this case study was to apply the PLUSS approach in the target domain to evaluate its feasibility. The hypothesis to be tested in the method evaluation and its null hypothesis were

H_1: *The PLUSS approach performs better than modeling according to the company process baseline in a product line setting.*

H_0: *The PLUSS approach performs equal to, or worse than the modeling according to company process baseline.*

A number of response variables relevant for measuring the performance of the approach were identified as part of the case study design. Examples are: *effort for learning and understanding notations* used; *effort for long term maintainability* of specifications; and *usefulness of the resulting models.*

4.1 Study Context

The case study was preformed with the Swedish defense contractor Land Systems Hägglunds. Land Systems Hägglunds is a leading manufacturer of combat vehicles, all terrain vehicles and a supplier of various turret systems. The company process baseline for software development, against which PLUSS was compared, is development according to the IBM-Rational Unified Process (RUP) [12].

The PLUSS approach was applied on the Vehicle Information System (VIS). The VIS subsystem is responsible for tasks such as displaying video, providing electronic manuals, performing onboard system test and diagnostics, displaying logs, displaying system status and reporting system alarms. The development of VIS has recently gone from clone-and-own reuse [1], to adopting a software product line approach. The transformation to software product line development was initiated by forming a domain engineering team which is now responsible for development and maintenance of the VIS core assets. At the time of the case study, the domain engineering team had successfully delivered core assets to their first customer project and was in the process of analyzing requirements for its second customer project.

The main CASE tools used for supporting the PLUSS approach were the requirements management tool Telelogic DOORS and the UML modeling tool IBM-Rational Rose. Rose was used for drawing feature graphs and UML Use case diagrams. DOORS was used for managing the overall domain model. Each feature was represented as an object in the database with a number of attributes; like feature type, products including the feature and a use case diagram. Each use case was represented as a module in DOORS. Scenario steps, both blackbox and whitebox, were represented as objects in those modules. Traceability links were used to relate features to use cases, scenarios and scenario steps according to the PLUSS meta-model shown in **Fig. 4**. A small number of scripts were written in DOORS to aid the modeling.

The domain modeling activity stared with a four hour introductory lecture on the PLUSS approach to the domain engineering team. After the lecture, the domain team had a four hour brainstorming session identifying and documenting features in the feature model. After this session, the domain engineering team split-up and only the product line analysis team continued the domain modeling for the reminder of the study. The product line analysis team consisted of three people, out of which two performed most of the modeling activities and the third mainly acted as a tool specialist, responsible for customizing DOORS to better support PLUSS.

4.2 Method

The case study involved collecting data from four different types of sources. The first type of data was collected by *examining documentation* [18]. Modeling artifacts from the early phases of the project were inspected to verify that they where used in the proper manner. The second type of data was collected by *participant observation* [18]. The research team assumed a mentoring role for the product line analysis team and could thereby get first hand information about any problems they ran into during the modeling activities. The third type of data was collected through *questionnaires* [11]. During the evaluation the product line analysis team filled out a questionnaire describing their experiences applying the approach. The questionnaire was designed to have both specific and open ended questions to also elicit unexpected types of information. The final type of data was collected trough *interviews* [18]. A total number of nine people, representing the domain engineering team, the product development team, the systems engineering team and technical management were interviewed to gather their views on the usefulness of the models and on possible pros and cons with the PLUSS approach. Interviews began with a short introduction to the research being performed. After the introduction, the VIS domain model and a product instance of the model were shown and discussed with each interviewee. Interviews proceeded in a semi-structured manner, trying to elicit as much information as possible about opinions and impressions regarding PLUSS.

The different types of data collected were first analyzed individually to find patterns and trends in the responses, then analyzed all together and conclusions were drawn about the case study hypothesis.

4.3 Threats to Validity

To minimize threats to the study's *construct validity,* the case study hypothesis and its null hypothesis were stated as clearly and as early as possible in the case study design to aid in identifying correct and relevant measures [11]. To minimize threats to the study's *internal validity,* the case study project was staffed using the organizations normal staff-allocation procedures. Everyone involved in the case study had good knowledge of modeling according to the company process baseline, against which the PLUSS approach was compared [11]. Furthermore, interviewees were chosen in collaboration with the organization's management to ensure that they properly represented their group of stakeholders. To avoid *Howthorne effect* [15], attitudes towards the company process baseline were collected from subjects and taken into account during data analysis. It was also pointed out to subjects that no "correct" answers existed, and that it was important that their answers correctly reflected their view. One confounding factor that may have affected the internal validity of the study is the close involvement of the research team with the product line analysis team. We do however judge this risk to be minor since the domain analysis team performed the actual modeling themselves and the mentoring activity mainly consisted of discussion meetings where possible problems were raised and discussed. To minimize threats to the study's *external validity,* the case study was conducted in the target domain of extremely long-lived software intensive systems and the pilot project was selected to be of typical size and complexity for the organization [11]. To minimize threats to the study's *conclusion validity,* results were triangulated by collecting data with four different methods from several different sources. Furthermore, results were discussed with the teams to assure that their opinions were represented correctly [18].

4.4 Results

Document examination indicated that the team understood and was able to apply all notations used after only the four hour introduction to the approach, even tough they had no earlier experience of feature modeling.

Participant observation revealed two initial problems applying PLUSS. During the first brainstorming session, the domain engineering team misused the feature model to "invent" variability that would force a "beautiful implementation", instead of focusing on creating a reusable requirements model. This problem was however resolved when the issue was discussed at the first mentoring meeting. The second problem regarded maintaining correct abstraction level. Even tough the team was to model only a certain subsystem (VIS), sometimes also system level functions appeared in the models. This problem was however resolved when the research team introduced a system context diagram [13] in the modeling process.

Questionnaires indicated that the product line analysis team gained a better understanding of the domain during the modeling activity. The team felt that applying PLUSS was an overall positive experience and that PLUSS has a number of positive characteristics, for example its way of providing a total overview of the product family and the possibility to maintain a common model for a whole family. A problem pointed at in the open ended questions was that the domain analysis team felt

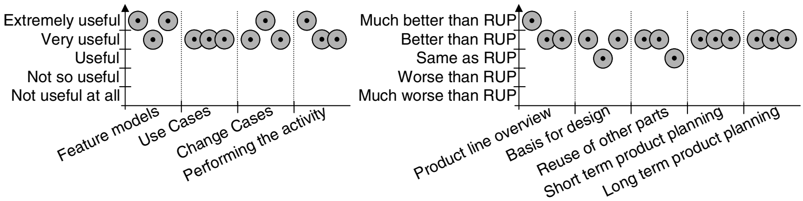


Fig. 6. Overview of questionnaire results, (a) usefulness of concepts / performing the modeling and (b) usefulness of resulting models compared to the company baseline

that DOORS and Rose were not integrated well enough, and that this resulted in time consuming manual synchronization of the models. However, as shown in **Fig. 6**, questionnaires indicated that the PLUSS approach performed better than the company baseline in the VIS context.

Interviews with *product line analysts* indicted that the PLUSS approach provides a better overview of the product line. The team also believed that the approach will improve the overall quality of the models and ease their maintenance. Experience of clone-and-own reuse [1] of use cases in earlier projects had pointed out a maintenance problem which they believed PLUSS addresses. They could not identify any scalability problems with the approach. However, they did believe that for it to work well, smart decisions from technical management regarding scooping and a strong configuration management function is needed. Analysts believed the initial extra investment related to applying the PLUSS approach would be returned in terms of reduced modeling costs already in the second or third project applying the approach.

Interviews with *product line designers* indicted that notations used were easy to understand and that the resulting models provided a good overview of dependencies within the model. They also felt that the approach made models more coherent and easier to find information in. They believed that the PLUSS approach will significantly increase the quality of specifications and ease their maintenance. Designers felt that change cases “might be good to keep in mind”, but a “probability of implementation” attribute would increase their usefulness. Designers could not identify any scalability problems with the approach. However, they did believe it to be important that technical management try to keep the number of variants down.

Interviews with the *product development team* indicated that the PLUSS approach offered product line mechanisms significantly stronger than anything the RUP has to offer. They believed that PLUSS will significantly reduce the effort needed for requirements analysis and that it has potential to largely reduce the amount of specification work. The team could not identify any scalability problems with the approach. They did however see a risk that the number of features might explode if too much new functionality is added in each project. They therefore believed a strong management function is needed keep the number of variants down. They also identified a risk that adding one or a few new features might create a dependency explosion in the feature graph, since the model is closely related to business rules. This thought could however not be further elaborated or illustrated by the team. The team also identified a need for obsolete management of features to prevent the feature tree from growing to infinity. The product development team believed the initial extra investment related to applying the PLUSS approach would be returned in terms of reduced modeling costs already in the second project applying the approach.

Interviews with the *systems engineering team* indicated that the notations used were easy to understand also for personnel with a non-software background. They liked the idea of a common model being a central source of information about a domain. They also found the use of change cases to tag unimplemented functionally very useful since it provides a good overview of what is new and what has been done before. They believed that the resulting models would be a good tool for early cost estimates and that the approach would encourage and produce high levels of reuse. The systems engineering team could not identify any problems with PLUSS. They did however see a risk with the whole concept domain modeling and requirements reuse. They believed that it might cause an organization to loose its visions and thereby cause products to stop evolving. Systems engineering also expressed a need for stronger means to document design rationale. This was however not seen as a problem with PLUSS, but as an important supplement to be further investigated.

Interviews with *Technical Management* indicate that the PLUSS approach provides significantly stronger support for product planning than traditional RUP. Management liked the fact that it is a use case driven approach, and the idea of a central source of information about a domain. Management also felt that feature models provided a good overview of the requirements space for the domain and that change cases provided a good overview of the current delta. However, to further improve the utility of change cases, management would like change cases to have attributes specifying planed platform release supporting them. Management also believed that PLUSS models could be a powerful means of communication towards other parts of the organization. Management believed the initial extra investment related to applying the PLUSS approach would be returned in terms of reduced modeling costs already in the second project, assuming the domain engineering team was able to produce models of required quality before the start of the second project.

5 Summary and Conclusions

We have described how a common use case model can be developed and maintained for a whole family of products in PLUSS. We have also described how product use case models can be generated from a family model by selecting features from a feature model. The approach was applied and evaluated in an industrial case study in the target domain. Triangulating on the collected case study data has led us to reject the case study null hypothesis. We thereby draw conclusion that the PLUSS approach performs better than modeling according to the styles and guidelines specified by the RUP in the current industrial context. Results did however also indicate that for PLUSS to be successfully applied, stronger configuration management and product planning functions than traditionally found in RUP projects are needed. Furthermore, results also pointed at a need for better tool support and stronger means to document design rationale. We consider these areas to be important areas of future work.

References

1. Bosch, J.: Design & Use of Software Architectures, Addison-Wesley (2000)
2. Czarnecki K., Helsen S., Eisenecker U.: Staged Configuration Using Feature Models, Proceedings of the Software Product Line Conference (SPLC 2004), LNCS 3154, Springer-Verlag, (2004) 266-283.
3. Ecklund E., Delcambre L., Freiling M.: Change Cases - Use Cases that Identify Future Requirements, Proceedings of OOPSLA'96, San Jose, Ca, October 6-10, (1996) 342-358.
4. Eriksson M., Börstler J., Borg K.: Marrying Features and Use Case for Product Line Requirements Modeling of Embedded Systems, Proceedings of the Fourth Conference on Software Engineering Research and Practice in Sweden SERPS'04, Institute of Technology, UniTryck, Linköping University, Sweden (2004) 73-82
5. Fantechi A., Gnesi S., Lambi G., Nesti E.: A Methodology for the Derivation and Verification of Use Cases for Product Lines, Proceedings of the International Conference on Software Product Lines, Lecture Notes in Computer Science, Vol. 3154, Springer-Verlag (2004) 255-265
6. Fey D., Fajta R., Boros A.: Feature Modeling: A Meta-model to enhance Usability and Usefulness, Proceedings of the International Conference on Software Product Lines, Lecture Notes in Computer Science, Vol. 2371, Springer-Verlag, (2002) 198-216.
7. Gomaa H.: Designing Software Product Lines with UML – From Use Cases to Pattern-Based Software Architectures, Addison-Wesley (2004)
8. Griss M., Favaro J., d'Alessandro M.: Integrating Feature Modeling with the RSEB, Proceedings of the Fifth International Conference on Software Reuse, Vancouver, BC, Canada, (1998) 76-85.
9. Jacobson I., Griss M., Jonsson P.: Software Reuse – Architecture, Process and Organization for Business success, Addison-Wesley (1997)
10. Kang K. Cohen S., Hess J., Novak W., Peterson A.: Feature Oriented Domain Analysis (FODA) Feasibility Study, Technical Report CMU/SEI-90-TR-021, Software Engineering Institue, Carnegie Mellon University, Pittsburgh, PA (1990)
11. Kitchenham B., Pickard L., Pfleeger S.: Case Studies for Method and Tool Evaluation, IEEE Software, Vol. 12 Nr. 45 (1995) 52-62
12. Kruchten P.: The Rational Unified Process - An Introduction, Second Edition, Addison-Wesley (2000)
13. Lykins H., Friedenthal S, Meilich A.: Adapting UML for an Object Oriented Sysyems Engineering Method (OOSEM), Proceedings of the 10'Th International INCOSE Symposium (2000)
14. Mannion M., Lewis O., Kaindl H., Montroni G., Wheadon J.: Representing Requirements on Generic Software in an Application Family Model, Proceedings of the International Conference on Software Reuse ICSR-6 (2000) 153-196.
15. Mayo E.: The human problems of an industrial civilization, New York: MacMillan (1933)
16. Von der Maβen T., Lichter H.: Modeling Variability by UML Use Case Diagrams, Proceedings of the International Workshop on Requirements Engineering for Product Lines (2002) 19-25
17. Rational Software: The Rational Unified Process for Systems Engineering Whitepaper, Ver. 1.1, Available at: http://www.rational.com/media/whitepapers/TP165.pdf, (2003)
18. Seaman C.: Qualitative Methods in Empirical Studies of Software Engineering, IEEE Transactions on Software Engineering, July/August (1999) 557-572
19. OMG: Unified Modeling Language Version 2.0, Available at: http://www.uml.org (2005)

Feature-Oriented Re-engineering of Legacy Systems into Product Line Assets – *a Case Study*

Kyo Chul Kang, Moonzoo Kim, Jaejoon Lee, and Byungkil Kim

Software Engineering Lab. Computer Science and Engineering Dept.,
Pohang University of Science and Technology, South Korea
{kck, moonzoo, gibman, dayfly}@postech.ac.kr
http://selab.postech.ac.kr/

Abstract. Home service robots have a wide range of potential applications, such as home security, patient caring, cleaning, etc. The services provided by the robots in each application area are being defined as markets are formed and, therefore, they change constantly. Thus, robot applications need to evolve both quickly and flexibly adopting frequently changing requirements. This makes software product line framework ideal for the domain of home service robots. Unfortunately, however, robot manufacturers often focus on developing technical components (e.g., vision recognizer and speech processor) and then attempt to develop robots by integrating these components in an ad-hoc way. This practice produces robot applications that are hard to re-use and evolve when requirements change. We believe that re-engineering legacy robot applications into product line assets can significantly enhance reusability and evolvability.

In this paper, we present our experience of re-engineering legacy home service robot applications into product line assets through feature modeling and analysis. First, through reverse engineering, we recovered architectures and components of the legacy applications. Second, based on the recovered information and domain knowledge, we reconstructed a feature model for the legacy applications. Anticipating changes in business opportunities or technologies, we restructured and refined the feature model to produce a feature model for the product line. Finally, based on the refined feature model and engineering principles we adopted for asset development, we designed a new architecture and components for robot applications.

1 Introduction

Home service robots utilize various technology-intensive components such as speech recognizers and vision processors to offer services. As markets for home service robots are still being formed, however, these technical components undergo frequent changes and new services are added and/or existing services are often removed or updated to address changing needs of the users. To compete in this rapidly changing market, robot manufacturers should be able to evolve robot products quickly with a minimal cost. The home service robot industry has strong needs for software development framework with which applications can be evolved easily. This situation makes software product line ideal for the home service robot industry.

H. Obbink and K. Pohl (Eds.): SPLC 2005, LNCS 3714, pp. 45 – 56, 2005.

Due to limited development resources, robot developers focused on technology intensive components at an early stage of product development without careful consideration of how software applications would evolve with changing requirements. Without a forethought architectural consideration, initial products have often been developed by integrating technology components in an ad-hoc way. Consequently, products suffered from feature interaction problems and maintenance of applications became costly. Re-engineering legacy robot applications into product line assets can enhance the competitive power of robot products by both decreasing development cost and increasing flexibility of robot applications. Jean-Marc at al [1][2][6] suggest an architecture-centric re-engineering process for initial product line asset recovery. This approach emphasizes a software architecture as a key to recovery of domain concept and relations. Bosch at al [3][4] consider a feature model as a core for creating product line assets from legacy products. These studies, however, do not suggest concrete design principles or guidelines for creating product line assets with adaptability.

In this paper, we describe our experience of re-engineering home service robot applications into product line assets via a feature-oriented methodology that is based on concrete principles and guidelines [5]. First, we extracted components and architectural information from legacy robot applications [7]. Second, based on the recovered information and domain knowledge, we discovered and modeled features of the robot applications. Anticipating future evolution of applications by considering potential business opportunities and technology changes, we refined the feature model adding additional features and variability information [8]. Finally, based on the refined feature model and three engineering principles we adopted to develop evolvable assets [9], we designed a new architecture and components for the product line. This re-engineering approach is depicted in Fig.1.

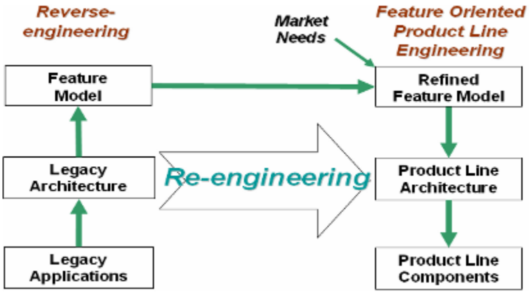


Fig. 1. Overview of re-engineering process

Sect. 2 gives an overview of home service robots. Sect. 3 explains the process of recovering architectural information from legacy applications. Sect. 4 describes recovery and refinement of a feature model from the legacy applications. Sect. 5 illustrates redesign of an architecture and asset components based on the refined feature model using the engineering principles we adopted. Sect. 6 validates the re-engineered product line assets. Finally, Sect. 7 describes the lessons learned from this project and Sect. 8 summarizes the paper and suggests future works.

2 Background on the Home Service Robot (HSR)

In this section, we briefly overview services of the home service robot (HSR) whose applications we re-engineered into product line assets. HSR is developed for daily home services such as home surveillance, cleaning, etc. From the HSR manufacturer, we received high level specifications of required HSR services such as “Call and Come” (locate and come to the user), “User Following” (continuously follow the user), “Security Monitoring” (home surveillance), and “Tele-presence” (control HSR remotely), etc. [1] In addition, we received two separate HSR applications each of which implements the “Call and Come” service and “User Following” service respectively. Of these primary services of HSR, we explain “Call and Come” and “User Following" services in detail.

*** Call and Come (CC)**

This service first analyzes audio data sampled from microphones attached to the surface of the robot and detects predefined sound patterns (e.g., hand clap or voice command). Currently, there are two commands “come” and “stop”. Once a “come” command is recognized, the robot detects the direction of a sound source. Then, the robot rotates to the direction of a sound source and tries to recognize a human face by analyzing video data captured through the front camera. If the caller's face is detected, the robot moves forward until it reaches within one meter from the caller (distance from the caller is measured by a structured light sensor). A “Stop” command simply makes the robot stop. If the following operation such as command recognition, sound source detection, or face recognition fails, CC resets to an initial state and waits for a new command.

*** User Following (UF)**

The robot uses a front camera and a structured light sensor to locate the user. Once UF is triggered, the robot constantly checks the vision data and sensor data from the structured light sensor to locate the user. The robot keeps following the user within one meter range. If the robot misses the user, the robot notifies the user by generating an audio message and UF terminates. The user may give a “come” command to let the robot recognize the user and restart UF.

Based on the given specifications and information extracted from the two legacy applications, we recovered a preliminary feature model covering both applications. The legacy HSR applications hard-coded most features without considering variation points for future extension or refinement. For example, the legacy HSR application has features such as “Face Detection Method” and “Object Recognition with SL” for user detection and user tracking. These features, however, do not have variations but have fixed implementations. For example, “Face Detection Method” is implemented based on “Color-based” method, not allowing other detection techniques to be adopted. For more detailed of features supported by the legacy HSR applications, see Fig. 5.

[1] For more information on HSR services and hardware, see [9].

3 Information Extraction from Legacy HSR Application

In this section, we explain how architectural information was extracted from the legacy applications and what potential problems were with the architecture.

3.1 Reverse Engineering Process

Fig. 2 describes the process of recovering a conceptual architecture as well as a process architecture from legacy applications.

1. From legacy applications, we obtain object relationship diagrams (see Fig. 3) mechanically, i.e., using the Rational-Rose[2] tool.
2. Based on the extracted object relationship diagram, we determine objects which constitute services (e.g., CC and UF services). This step needs heuristics based on domain knowledge and additional data flow analysis. Then, we identify *operational units* that the service consists of, by analyzing method invocations and data flows. By assigning operational units into architectural components, we recover a conceptual architecture.
3. From the object relationship diagram and identified service/operational units, we determine which objects (i.e. active objects) take initiative of invoking other objects' operations by creating processes/threads. Then, we identify interactions between active objects via a control flow analysis. By capturing these interactions between active objects, we recover a process architecture which shows assignment of software components to processes or thread synchronization relations.

How this process was applied to CC is explained in the following subsections.

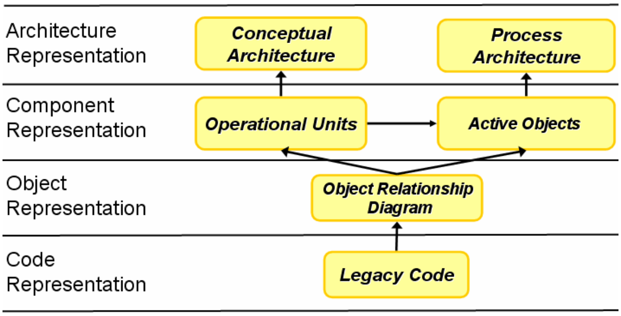


Fig. 2. Recovery of conceptual architecture and process architecture

3.2 Recovery of Operational Units

Fig. 3 illustrates recovery of operational units from the object relationship diagram for CC. Using functional cohesion as a criterion, we classified operational units into three categories – *sensor (input), controller (coordination), and actuator (output).* Using these categories as a guide, we identified five operational units as follows.

[2] Rational-Rose is a trademark of IBM corporation.

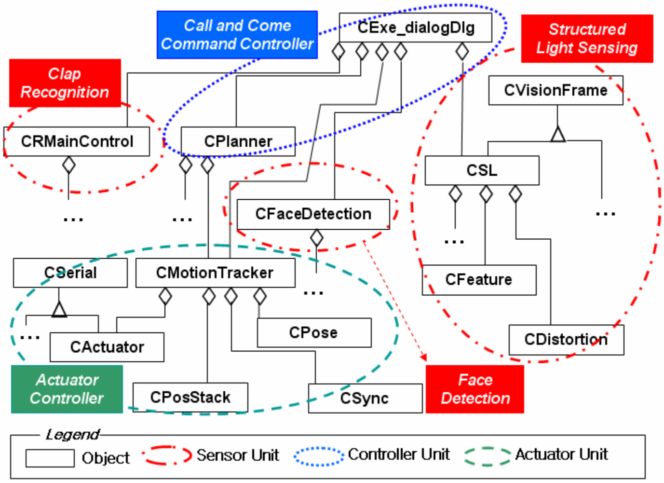


Fig. 3. Recovery of operational units for CC

- sensor units: "Face Detection", "Clap Recognition", and "SL Sensing"
- a controller unit: "CC Command Controller"
- an actuator unit: "Actuator Controller"

3.3 Recovery of Conceptual Architecture and Process Architecture

Through an additional data flow analysis, the identified operational units are configured into the conceptual architecture depicted in the Fig. 4.a). This conceptual architecture is hardly adequate for multi-service robots because all service units (e.g. CC

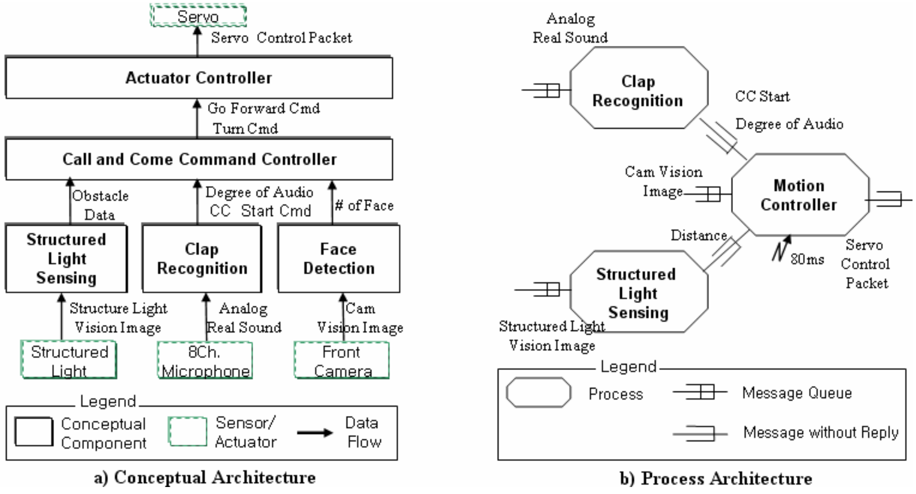


Fig. 4. Recovered conceptual architecture and process architecture

Command Controller) can access and control "Actuator Controller" directly. This architecture can allow services interfere with each other in an indirect way.

To recover a process architecture, we identified three active objects from the object relationship diagram depicted in Fig. 3 by detecting process creation code – `CEXE_dialogDlg`, `CRMainControl`, and `CSL`. These objects create three processes "Motion Controller (MC)" (consisting of "CC Command Controller", "Face Detection", and "Actuator Controller" operational units), "Clap Recognition (CR)" ("Clap Recognition" unit) and "SL Sensing (SLS)" ("Structured Light Sensing" unit) respectively as depicted in Fig.4.b). MC receives data such as the distance to an obstacle and the direction of clap sound from SLS and CR respectively. MC determines the moving direction based on these data. Thus, without a smart control logic in MC, feature interaction between CR and SLS may happen because both processes can control MC at the same time.

4 Refined Feature Model of HSR Product Line

In this section, we describe a refined feature model of HSR. First, we extracted features from the legacy application implementing CC service, which are indicated in bold font in Fig. 5. Newly added features and refined features are indicated in italic font in Fig. 5. The detailed explanation of the refined feature model is as follows.

First, we added new services targeted for different markets. For example, HSR supporting only CC service can be produced for a low-end market as a delivery robot, while HSR with CC, UF, Tele-presence, and Security Monitoring services can be

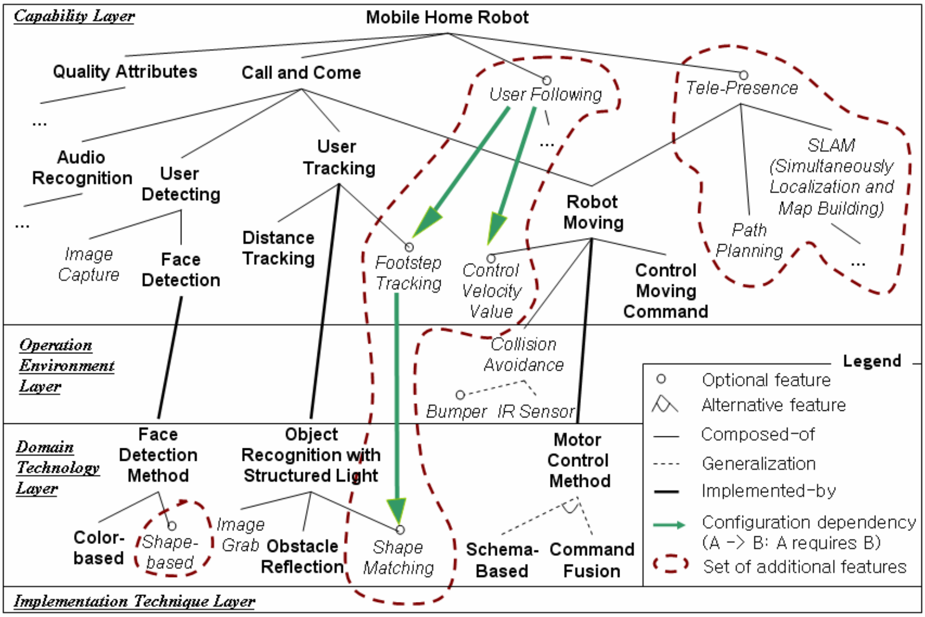


Fig. 5. Feature model for SH100 including CC service

produced for a high-end market as an intelligent home agent. Based on the legacy feature model for the CC service, we created a new model by adding features for new services, operations, and domain technologies, and also dependency relationships between features. Newly added services require operational features not included in the original feature model. For example, newly added UF service needs to follow user's footsteps ("Footstep Tracking"). In addition, to follow the user smoothly, UF service controls HSR in a velocity oriented way via "Control Velocity Value" (e.g. set the velocity of left wheel as 1 m/s, and the right wheel as 0.8 m/s). Furthermore, a new operational feature may require new domain technologies. For example, "Footstep Tracking" requires "Shape Matching" in order to recognize user's footsteps.

Second, we refined the feature model by including optional features to accommodate anticipated changes. For example, in the legacy CC application, "Face Detection Method" used only a color-based detection algorithm. We refined this feature by adding an optional feature "Shape-based" for its improved accuracy adequate for high-end markets, but at the cost of high computational resources.

Third, due to the advances of technologies, some features considered as important capabilities can simply be supported by the operational environment as SoC (System On Chip) or by OS. In the legacy CC application, "Collision Avoidance (CA)" feature was implemented in software and placed in the Capability Layer. We moved CA to the Operation Environment Layer because of CA SoCs available in the market.

5 New Architecture Design of HSR

One of the quality attributes with the new architecture is its *flexibility* in adding, removing, and/or replacing components as products evolve. For this purpose, we adopted C2 architectural style [10] for its substitutability of components. Also, we enforced *1:N* mapping from features to components whenever possible for easy inclusion/exclusion of features into/from products. Furthermore, through an analysis of legacy applications [11] and the refined feature model in Fig. 5, we decided to adopt three engineering principles in redesigning the architecture of HSR (for details on these principles, see [9]).

First, the legacy architecture intermixed control components with computational components, which caused difficulty in analyzing behaviors of applications. Therefore, we proposed the first principle – *separation of control aspects from computational aspects.* By separating the control plane which consists of control components from the data plane with computational components, we could separate data flows from control flows, thus making it possible to visualize and analyze behaviors of the system. As a consequence, addition/removal of components becomes easier because responsibilities of each component become clear.

Second, we aimed to minimize ripple effects caused when services are added or removed - simple integration of new services, without consideration of how features should be related with each other, has easily led to feature interaction problems. The legacy architecture did not provide careful coordination among service components, thus resulted in feature interaction problems when a new service was added. To address such problems, we proposed the second principle - *separation of global behaviors from local behaviors.* Service components are separated to be executed *locally*, i.e., independently from other service components. Therefore, effects from addition/removal of components to other components are localized, which helps implementing variation points. The coordination responsibility among different service components is assigned to a special component called *Mode Manager* which controls global system behavior such as interaction policies between service features.

Finally, we found that there existed hierarchy between some variable features. For example, “Object Recognition with SL” feature has three sub-features – “Image Grab”, “Obstacle Reflection”, and “Shape Matching” (see Fig. 5). “Image Grab” simply captures SL images whereas “Obstacle Reflection” detects objects in front of HSR by analyzing the SL images obtained by “Image Grab”. “Shape Matching” works more sophisticatedly by analyzing object images obtained from “Obstacle Reflection” to recognize user’s legs (e.g., footsteps). Therefore, we made three component layers corresponding to these variable features according to the third principle - *layering in accordance with data refinement hierarchy*. Different services may request operations from different layers of a single component. By adopting a layered architecture for computational components, addition/removal of variable features in the Domain Technology Layer could be implemented cleanly because the layered architecture provides well-defined interfaces between layers.

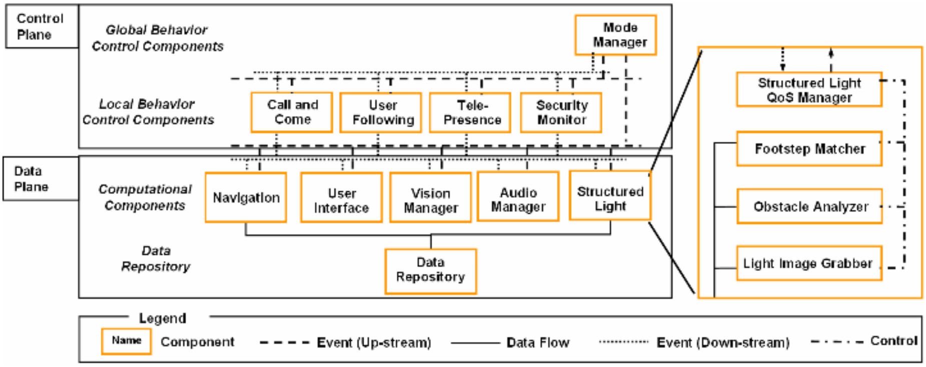
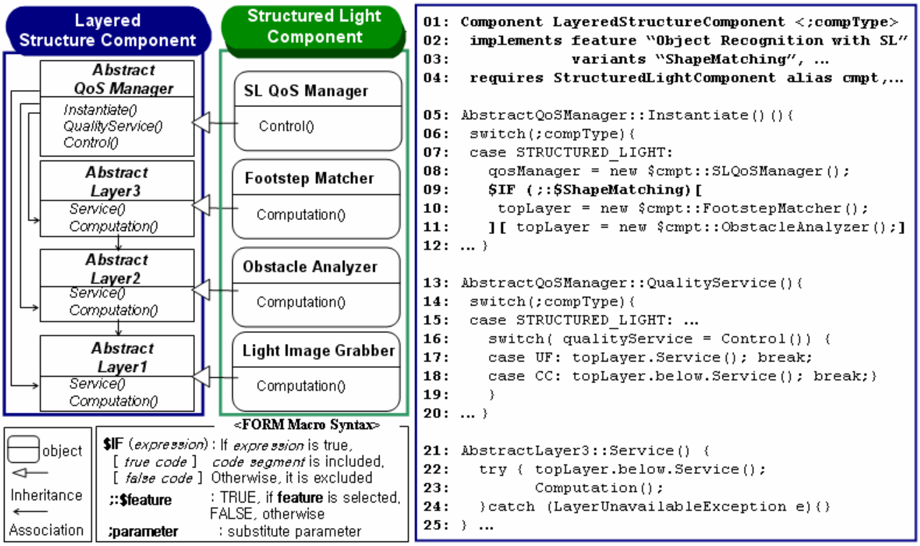


Fig. 6. New architecture for HSR

Fig.6 illustrates the new architecture designed according to the three re-engineering principles.[3] First, we identified four control components: CC, UF, Tele-presence, and Security Monitoring. And we identified five computational components: Navigation, Structured Light, User Interface, Vision Manager, and Audio Manager. Mode Manager was specified to control global behavior of HSR by receiving all up-stream events and managing the control components. Most computational components read raw input data from sensors and process them to generate outputs to other components. The generated outputs are transferred to the control component through a data connector/bus.

[3] This architecture reflects typical software architecture of embedded systems (especially application layer) such as network gateways or vehicle controllers which distinguish control data from computational data.



```
01: Component LayeredStructureComponent <;compType>
02:  implements feature "Object Recognition with SL"
03:             variants "ShapeMatching", ...
04:  requires StructuredLightComponent alias cmpt,...

05: AbstractQoSManager::Instantiate()(){
06:  switch(;compType){
07:  case STRUCTURED_LIGHT:
08:    qosManager = new $cmpt::SLQoSManager();
09:    $IF (;:$ShapeMatching)[
10:     topLayer = new $cmpt::FootstepMatcher();
11:    ][ topLayer = new $cmpt::ObstacleAnalyzer();]
12: ... }

13: AbstractQoSManager::QualityService(){
14:  switch(;compType){
15:  case STRUCTURED_LIGHT: ...
16:    switch( qualityService = Control()) {
17:    case UF: topLayer.Service(); break;
18:    case CC: topLayer.below.Service(); break;}
19:    }
20: ... }

21: AbstractLayer3::Service() {
22:   try { topLayer.below.Service();
23:         Computation();
24:   }catch (LayerUnavailableException e){}
25: } ...
```

Fig. 7. A design object model and component specification

Based on the new architecture, we designed components with a macro-processing mechanism (to incorporate variable features) [12]. In addition, we extracted sub-components from the existing code through refactoring techniques [13]. Fig. 7 illustrates the structured light component. The left part of Fig.7 shows a layered template for computational components and the structured light component instantiated from the template. The legacy structured light component was implemented as a long procedural function. Thus, we extracted reusable portion of the function into “Footstep Matcher”, “Obstacle Analyzer”, and “Light Image Grabber” components. These layered components were instantiated for the selected features using a component specification [14].

Lines 1-4 of the right part of Fig. 7 specify instantiation of `LayeredStructureComponent` implementing “Object Recognition with SL” feature (with variable feature “ShapeMatching”) from `StrcutredLightComponent`. Lines 5-12 describe how structured light and vision manager are instantiated. Especially, lines 9-11 specify that if a variant feature “Shape Matching” is selected, the instantiated component will have “Footstep Matcher” as its topmost layer; otherwise, “Obstacle Analyzer” as its topmost layer. Lines 13-20 illustrate how a service is selected for the service requestor. For example, at line 17, if UF requests service of structured light components, the service of topmost layer (i.e. “Footstep Matcher”) should be provided (with an assumption that “Footstep Matcher” feature is enabled). Lines 21-24 show a service chain between layers.

6 Validation of Product Line Assets

We have generated HSR applications using re-engineered product line assets. First, without difficulty, we have instantiated two applications supporting CC and UF respectively by selecting features required by the services. We could check that new applications worked successfully according to the given service specifications. For these two applications, Mode Manager does not enforce control on global behaviors because the HSR applications run only a single service.

Then, we have instantiated an application supporting both CC and UF services. The CC and UF services share computational components. Concurrent accesses to the computational components except “Navigation” did not cause any feature interaction problem between the CC and UF services; operations requested to the computational components by CC and UF are mainly reading analyzed data, not updating data. In addition, the layers accessed by the two services are different. For example, CC accesses the “Obstacle Analyzer” layer while UF accesses the “Footstep Matcher” layer of the “Structured Light” component. Operations requested by UF and CC to ”Navigation”, however, are mostly for controlling actuators. Thus, to prevent a feature interaction problem, Mode Manager coordinated CC and UF using a priority scheme. Code modification required for priority enforcement was not obstructive because CC and UF components except Mode Manager did not need to be modified. Therefore, we have shown that the re-engineered product line assets for HSR are suitable for creating applications of the home service robot.

7 Lessons Learned

In this section, we describe lessons we have learned from this re-engineering project.

7.1 Importance of Pre-planned Asset Integration

Hardware-oriented or technology-oriented organizations usually consider product development/instantiation as a last-minute task that can be achieved by simply integrating technology-intensive components. Without a fore-thought architectural consideration and component integration strategies, however, products often suffered from feature interaction problems and maintenance of applications became costly.

In this case study, we could alleviate these difficulties by providing an architectural framework based on the refined feature model and engineering principles we adopted for asset development. In addition, the explicit mapping between features and architectural components made the inclusion/exclusion of features visible. We also observed that a feature model could play a central role in identifying relationship between pre-existing features and new features. For example, for the addition of "User Following" feature, the feature model in Fig.5 shows additional new features such as “Footstep Tracking” and their relationships with the features of the legacy applications.

Based on the feature analysis results, we could determine component integration scheme. For the integration of the "Footstep Tracking" feature, for instance, the component that implemented "User Tacking" was modified to accommodate the "Footstep Tracking" feature and the modified component could confine the variations between "Distance Tracking" and "Footstep Tracking" by providing a common interface.

7.2 Benefit of a Feature Model in Architecture Layering

Through the case study, we found that the feature model provided a useful information for identifying layers in the component architecture. The feature model has features representing different levels of computation. Especially variation points show services of different levels. For example, “Shape Matching”, “Obstacle Reflection”, and “Image Grab” features (see Fig. 5) are used for UF, CC, and Tele-presence services respectively. These features altogether represent computational hierarchy, i.e., “Shape Matching” uses result from “Obstacle Reflection” and “Obstacle Reflection” from “Image Grab”. Accordingly, these features are implemented as a “Footstep Matcher” layer, an “Obstacle Analyzer” layer, and a “Light Image Grabber” layer of the structured light component (see Fig. 7). Similarly, we found that “Face Detection Method” feature also had a hierarchy among its sub-features and, thus, corresponding “Vision Manager” component was built as a layered structure. Therefore, layering based on the feature model was very helpful for creating component architecture for product line engineering.

7.3 Analysis Aid of Process Architecture

Process architecture can help finding possible feature interactions among concurrent processes. For example, from the process architecture in Fig. 4.b), we could guess that MC might suffer feature interaction problems due to concurrent input data from CR and SLS (see Sect. 3.3). Furthermore, process architecture also helps analyzing the legacy application design. For example, UF service implemented in the legacy application does not use the front camera, not following the UF service specification (see Sect. 2). We could find the reason based on the process architecture. In order to utilize the front camera for UF, the front camera should capture images continuously to detect user’s face. The “Face Detection” operational unit in the legacy application, however, was a sequential component of MC, not a separate process running concurrently (see Fig. 4.b)). That was the reason why legacy UF application did not use the front camera.

8 Conclusion

In this paper, we describe re-engineering legacy home service robot applications into product line assets via a feature-oriented method. We believe that feature-oriented re-engineering approach can help robot manufacturers to take advantage of product line framework – decrease in development cost and increase in application flexibility.

As a future work, based on the re-engineered HSR product assets, we plan to study evolution of HSR product line assets and evaluate both weaknesses and strengths of the current product line assets. Secondly, we will study and develop guidelines for evaluating product line assets.

References

1. DeBaud, J.M., Girard, J.F.: The relationship between the Product Line Development Entry Points and Reengineering, 2nd International Workshop on Development and Evolution of Software Architectures for Product Families, LNCS 1492, pp. 132-139, (1998)
2. Bayer, J., Girard, J.F, Wuerthner, M., DeBaud, J.M., Apel, M.: Transitioning Legacy Assets to a Product Line Architecture, 7th European Software Engineering Conference (ESEC/FSE'99), LNCS-1687, pages 446-463, (1999)
3. Bosch, J., Ran, A.: Evolution of Software Product Families, Software Architectures for Product Families: International Workshop(IW-SAPF-3), LNCS 1952, pp. 168-183, (2000)
4. Maccari, A., Riva, C.: Architectural Evolution of Legacy Product Families, Software Product Family Engineering: 4th International Workshop (PFE-4 2001), LNCS 2290, pp 64-69, (2002)
5. Kang, K., Lee, J., Donohoe, P.: Feature Oriented Product Line Engineering, IEEE Software, 19(4), July/August, pp. 58-65, (2002)
6. Eixelsberger, W., Kalan, M., Ogris, M., Beckman, H., Bellay, B., Gall, H.: Recovery of Architectural Structure: A Case Study, 2nd International ESPRIT ARES Workshop on Development and Evolution of Software Architectures for Product Families LNCS Vol. 1429. Springer-Verlag, Berlin Heidelberg New York (2002)
7. Bergey, J., O'Brien, L., Smith, D.: Option Analysis for Reengineering (OAR): A Method for Mining Legacy Assets (CMU/SEI-2001-TN-013). Pittsburgh, PA:Software Engineering Institute, Carnegie Mellon University (2001)
8. Lee, K., Kang, K., Lee, J.: Concepts and Guidelines of Feature Modeling for Product LineSoftware Engineering. In: Gacek, C. (eds.): Software Reuse: Methods, Techniques, and Tools.Lecture Notes in Computer Science, Vol. 2319. Springer-Verlag, Berlin Heidelberg
9. Kim, M., Lee, J., Kang, K., Hong, Y., Bang., S.: Re-engineering Software Architecture of Home Service Robots: A Case Study, International Conference on Software Engineering, Missouri, USA, pp.505-513, (2005)
10. Medvidovic, N., Taylor, R. N.: Exploiting architectural style to develop a family of applications, Software Engineering. IEE Proceeding, Vol. 144, No 5-6. October/December (1997)
11. Lago, P., Vliet, H.: Observations from the Recovery of a Software Product Family, Software Product Line Conference 2004, LNCS Vol 3154 Springer-Verlag, Berlin Heidelberg New York (2004)
12. Basset, P.G.: Framing Software Reuse: Lessons from the Real World. Prentice Hall, Yourdon Press (1997)
13. Fowler, M., Beck, K., Brant.,J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code, Addison-Wesley (2000)
14. Bosch, J., Hogstrom, M.: Product Instantiation in Software Product Lines: A Case Study. Second International Symposium on Generative and Component-Based Software Engineering LNCS 2177 (2001)

Reuse without Compromising Performance: Industrial Experience from RPG Software Product Line for Mobile Devices

Weishan Zhang[1] and Stan Jarzabek[2]

[1] School of Software Engineering, Tongji University,
No. 4800 Cao'an Highway, Shanghai, 201804, China
{zhangws}@mail.tongji.edu.cn
[2] Department of Computer Science, School of Computing,
3 science drive 2, 117543, Singapore
{stan}@comp.nus.edu.sg

Abstract. It is often believed that reusable solutions, being generic, must necessarily compromise performance. In this paper, we consider a family of Role-Playing Games (RPGs). We analyzed similarities and differences among four RPGs. By applying a reuse technique of XVCL, we built an RPG product line architecture (RPG-PLA) from which we could derive any of the four RPGs. We built into the RPG-PLA a number of performance optimization strategies that could benefit any of the four (and possibly other similar) RPGs. By comparing the original vs. the new RPGs derived from the RPG-PLA, we demonstrated that reuse allowed us to achieve improved performance, both speed and memory utilization, as compared to each game developed individually. At the same time, our solution facilitated rapid development of new games, for new mobile devices, as well as ease of evolving with new features the RPG-PLA and custom games already in use.

1 Introduction

Mobile games have become an important trend in the mobile phone industry. Role-Playing Game (RPG) is one kind of a mobile game suitable for mobile devices and attractive to players. With an RPG, the players take the roles of fictional characters and participate in an interactive story. All RPGs share basic Role-Playing concepts and differ in certain functional requirements. RPGs are further differentiated by the properties of a specific mobile device platform on which they run, and which affect RPG's design and implementation. This includes high end mobile devices with 640x200 colorful screen and up to 80M memory versus lower end devices with 100x80 mono display and less than 100kb memory; the new mobile devices J2ME MIDP2.0 compliant versus the old ones MIDP1.0 enabled. RPGs must perform well across all these different devices.

Given the above similarities and differences, RPGs form an interesting and potentially useful product line [2][4]. However, to be attractive and practical, reuse in mobile device, and also in embedded software sectors, must not compromise the performance. This problem has been frequently mentioned in many sources [5], but we

H. Obbink and K. Pohl (Eds.): SPLC 2005, LNCS 3714, pp. 57 – 69, 2005.

have not come across many examples of successful solutions. On contrary, in our discussions with embedded software vendors, it has been often mentioned that reusable design, being generic, may need compromise performance.

In this paper, we report on a mobile device industrial project in which product line approach not only achieved development/maintenance productivity gains, but also enhanced software performance. We applied extractive approach [9] combined with generalizations revealed by domain analysis, to convert four RPGs into an RPG product line architecture (RPG-PLA) from which we could derive the four RPGs, as well as more similar ones. Upon exanimation of the original RPGs, we found certain shortcomings in their design. We also found that certain optimizations used in one RPG could be also useful in other RPGs. We encoded best design solutions we learned from four RPGs into the RPG product line architecture (RPG-PLA). From there, we could propagate common optimization strategies to all the RPGs built based on the RPG-PLA. We used XVCL [18] to build and manage the RPG-PLA.

The rest of the paper is structured as follows: In the next two sections, we introduce Dig Gem as an example of an RPG and then analyze the RPG domain. Section 4 gives an overview of the RPG product line experiment. In Section 5, we analyze design of the four original RPGs. Section 6 discusses optimization strategies for RPG software. Then we present the building of the RPG-PLA incorporated optimization strategies. In section 8, we show the derivation of specific RPGs from the RPG-PLA. Section 9 presents the discussion of the experiment results. Related work and concluding remarks end the paper.

2 Dig Gem - An Example of an RPG

In Dig Gem, a hero digs around the map to look for gems. Scores for finding different gems are added up and listed (Figure 1). A hero faces various traps (e.g., bombs) that obstruct his efforts. The elapsed and remaining time for the game is displayed on a bar on top of the screen. The main concepts behind the Dig Gem are depicted in Figure 2.

Fig. 1. A snapshot of Dig Gem

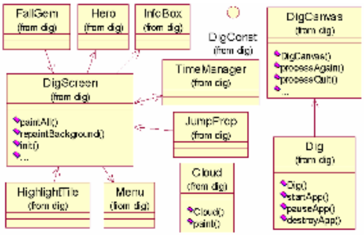


Fig. 2. Conceptual class diagram of Dig Gem

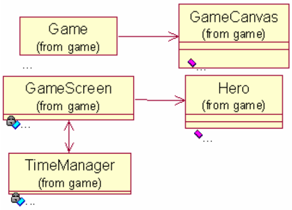


Fig. 3. Concepts in the RPG domain

3 An RPG Domain

RPGs come in many different incarnations and can be very complex. Here, for start, we consider *Dig Gem* introduced in Section 2; *Climb* where the hero walks and jumps on the floor to avoid falling down to the mountain; *Feeding* where the hero tries to pick up as much food as possible; and *Hunt*, where the hero shoots animals and monsters with arrows. Figure 3 shows the generic concepts in RPG domain.

3.1 Commonalities in the Mobile RPG Domain

The four RPGs share the following commonalities:

1. They all use MIDlet application model [7] of the J2ME platform.
2. The execution scenario and its control flow for all the games is similar:
   a) There is a main class that extends MIDlet (for example Dig class in Figure 2), as required by MIDP. In the constructor of the main class, an instance of the game Canvas (for example DigCanvas) is created. Using a getInitialScreen() method, this instance creates a game starting screen object (for example DigScreen).
   b) In the starting screen, all initialization data are loaded, including map, sound, etc.
3. The heroes of the game move according to a predefined pattern, which differs from game to game.
4. The game scores are displayed as the game goes on.

3.2 Variants in the RPG Domain

The following are some of the variant features in the RPG domain:

1. The details of game stories are different from one RPG to another. In particular:
   a) Initial position of the hero.
   b) The scenarios for hero's lives (e.g., the number of lives the hero can have) depend on the game, hero's actions and the underlying context.
   c) The number and types of heroes in the game.
   d) Types of weapons, number of bullets, etc.
2. The hero's movement style and mode (the hero may go up and down, or right and left; jump or even fly).
3. Different games may use different algorithms for movement.
4. Some of the games need time manager to count the time elapsed in order to manage the time-related behavior.
5. The show time and time spent before the splash screen disappears; Different splash images can be used from game to game; An option to skip the splash screen.
6. User interface variants are most profound and plentiful. They include menus, energy bars, map styles, and position and size of widgets.

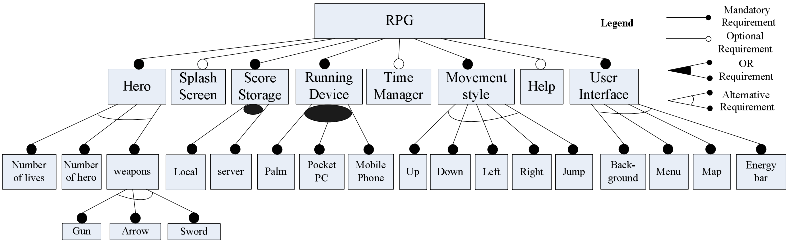


Fig. 4. A feature diagram for the mobile RPG domain

Figure 4 depicts common and variant features in the RPG domain, as a feature diagram [8]. Variant features are often dependent on each other, in the sense that one variant can be a prerequisite for other variants. A *legal configuration of variants* is any variant selection that can appear in a specific product line member.

4 An Overview of the Experiment

We studied four RPGs developed for Motorola E680 by Sanjie Team in Meitong Co. Ltd. The four developers in the team had a good grasp of the RPG domain, platform characteristics, and object-oriented program design techniques. First, we analyzed the design of RPGs in terms of quality metrics, design solutions and optimization strategies used in them. Then, we designed a "generic RPG", that is, an RPG product line architecture, that included design and optimization strategies that could benefit any of the four games, and possibly similar games to be developed in the future.

We started with Dig Gem and applied a combination of extractive and pro-active approaches to design and evolve the RPG-PLA [5][9]. Our strategy was a variant of an incremental reengineering of existing system family into a product line developed in our earlier experiment [17]. We could derive from the RPG-PLA four original RPGs developed by Sanjie Team, as well as many similar ones. The steps involved in the experiment are depicted in Figure 5.

Finally, we compared our reuse-based solution to the original one from the point of view of both the development/maintenance productivity and performance.

We designed the RPG-PLA with XVCL [18], a static meta-programming technique to create parameterized, generic meta-components. Meta-components form a hierarchical structure, called an x-framework in XVCL jargon, which, in the context of the RPG project, is an implementation of a product line architecture concept. The XVCL Processor synthesizes custom programs, members of the product line, based on specifications of their required properties provided in a special, top-most meta-component, called SPC. In our experiment, the process of applying XVCL to produce product line members, (i.e. RPGs) from the RPG-PLA is depicted in Figure 6.

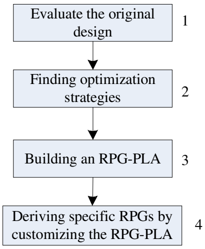

Evaluate the original design
1
Finding optimization strategies
2
Building an RPG-PLA
3
Deriving specific RPGs by customizing the RPG-PLA
4

Fig. 5. Steps leading to an RPGs product line

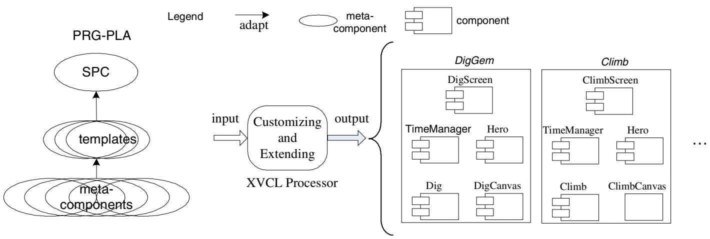


Fig. 6. Project application of XVCL

5 Evaluation of the Original RPGs

We evaluated the quality of the existing RPGs using typical OO metrics [3][10], as shown in **Table 1**.

Table 1. OO metrics used for evaluation

coupling	inheritance	inheritance based coupling	complexity of the design	polymorphism
• CBO - Coupling Between Objects • VOD -Violations of Demeters Law • FO – FanOut • MIC-Method Invocation Coupling	• DOIH-Depth of Inheritance Hierarchy	• TRAp - Total Reuse from Ancestors percentage • TRAu –Total Reuse from Ancestor unitary • TRDp -Total Reuse in Descendant percentage • TRDu -Total Reuse in Descendant unitary	• MSOO - Maximum Size of Operation • MNOL - Maximum Number Of Levels • MNOP - Maximum Number Of Parameters • NOM - Number Of Members • NORM - Number Of Remote Methods	• NOOM - Number Of Overridden Methods

Table 2 and Table 3 show the summary of metrics-based quality analysis. According to commonly accepted thresholds (as indicated by the default value set in Borland Together), we can see that on overall the design conforms to the norms, with the exception of the MSOO and NORM metrics which are below the recommended thresholds (shown in italics in Table 2 and Table 3).

Table 2. OO metrics for game packages

Item	CBO	DOIH	FO	MIC	MNOL	MNOP	MSOO	NOM	NOOM	NORM	TRAp	TRAu	TRDp	TRDu	VOD
hunt	27	3	*20*	18	*6*	*5*	*19*	52	7	*68*	27	100	0	0	10
feeding	14	2	10	8	3	2	*13*	32	7	*37*	9	80	0	0	4
dig	24	2	*16*	18	4	*5*	*19*	53	7	*72*	27	100	0	0	11
climb	18	2	12	8	6	*4*	*31*	38	7	*60*	11	80	0	0	6

Table 3. OO metrics for game Dig Gem

Item	CBO	DOIH	FO	MIC	MNOL	MNOP	MSOO	NOM	NOOM	NORM	TRAp	TRAu	TRDp	TRDu	VOD
TimeManager	6	2	4	3	1	2	2	21	1	4	6	22	0	0	1
Menu	5	2	4	2	1	1	3	13	1	2	12	33	0	0	1
JumpProp	5	2	4	3	2	*5*	5	13	1	4	18	67	0	0	1
InfoBox	6	2	4	3	4	*4*	8	21	1	4	7	22	0	0	1
HighlightTile	6	2	4	3	2	*5*	5	6	1	4	27	100	0	0	2
Hero	5	2	0	3	1	1	2	4	2	10	9	100	0	0	3
FallGem	6	2	4	3	1	*5*	2	7	1	5	27	100	0	0	2
DigScreen	24	2	*16*	18	4	2	*19*	53	5	*72*	3	44	0	0	11
DigConst	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
DigCanvas	11	2	6	6	1	2	2	13	7	12	3	90	0	0	3
Dig	2	1	2	2	0	1	1	5	0	2	0	0	0	0	1
Cloud	5	2	4	2	4	3	6	8	1	4	27	100	0	0	1

6 Optimization Strategies for RPGs

An initial design is never perfect. This especially holds for applications developed with multiple design goals in mind which is the case for the resource limited mobile device software. Although developers of the four RPGs under our study were experienced professionals, we found that there was a room for performance improvement in their design. We list common problems and improvements below.

a. **Remove the constant interface.** There are some constant interfaces, each for every game. Interfaces should be used solely for type definitions [11], therefore we remove all constant interfaces for the game engine and all four games.

b. **Change class members to local variables.** For example, the following buffer size is defined in class GameMedia: public final static int BUFFER_SIZE = 1024;

As this attribute is used only in method LoadData(), buffer size definition should be moved and defined as a LoadData() local variable. Then, the memory can be reclaimed after the execution of LoadData() completes.

c. **Remove redundant inheritance relationships.** In some cases, some classes that had very little in common were related by inheritance. We removed this kind of inheritance relationship along with unnecessary class members and methods.

d. **Iterate loops down to zero.** Comparing against zero is faster than comparing against other numbers. For example, we changed loops like: for(int i = 0 ; i < 6 ; i ++) to faster for(int i = 5 ; i >=0 ; i--).

e. **Remove unnecessary classes.** We re-allocated functionalities so that some classes, especially those with very few methods, could be removed.

f. **Remove obsolete class methods.** In some cases, we found never used class methods. For example in GameMedia class, there were two methods for loading data from file or input stream, remaining from previous implementation, but never used in the current implementation. We remove such obsolete methods.

g. **Remove constant definitions.** As we planned to use XVCL, in many cases we could delegate constant management to the XVCL level. This helped us decrease the heap size, as there was no need to define them in the code anymore.

The following are other optimizations that we applied:

h. **Avoid slow string comparisons**

i. **Declare the method and variable final for faster access**

j. **Replace resizable Vectors with arrays, if possible**

k. **Return a null object instead of throwing exceptions wherever possible**

We built the above strategies into an RPG-PLA so that they could be propagated to all the RPGs.

7 Building an RPG Product Line Architecture

We applied a combination of extractive and pro-active approaches to design and evolve the RPG-PLA [5][9], using experiences from our earlier project on incremental reengineering of an existing system into a product line [17].

Design of the RPG-PLA started with a typical game – Dig Gem in our case. We evaluated the impact of variant features on Dig Gem components. Some variants had localized impact on one component only, but other variants had a wider impact on many components. For example, the initial position of the hero only affects the Hero component. But the number and types of heroes affect many components.

Components affected by variant features were typical candidates for meta-components: Instead of having many similar components, each implementing some combination of variant features, we designed a small number of generic meta-components capable of producing components implementing any combination of variant features, as required in some member of the RPG product line. For example, Canvas components and MIDlet classes are quite similar across RPGs and all their variant forms can be obtained from generic meta-components.

Then, in iterations, we refined the RPG-PLA with new variant features, and extended its functionality, to support more RPGs.

7.1 Initial RPG-PL Based on Dig Gem

Figure 7 depicts meta-components forming our first-cut RPG-PLA based on the Dig Gem. Before addressing variant features related to other RPGs, we implemented optimizations described in the Section 6 into the RPG-PLA, with the intention to propagate them to other games to-be-built based on he RPG-PLA.

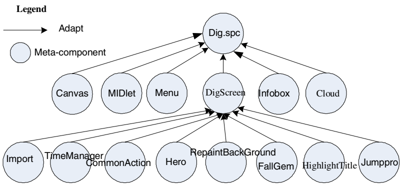


Fig. 7. A first-cut RPG-PLA based on Dig Gem

Meta-variables and meta-expressions are the basic means for parameterization. Meta-variables have global scope in the x-framework and their values are propagated across the underneath meta-components. An important role of meta-variables is to chain together modifications of multiple components (at multiple program points) related to variant features of a product line. Parameterization also provides effective means to fulfill some of the optimization strategies outlined in Section 6.

```
name :Hero.xvcl
set      Hero = Hero
text     class @Hero  extends Nlayer{
                 public ?@Hero?(){
                    setPosition(@HeroInitX, @HeroInitY);
                    … …
while    Using-items-in=LoopImageX
         while    Using-items-in=LoopImageY
                  iImg[@LoopImageX][@LoopImageY]=GameMedia.loadImage("/@GameName/@Hero@LoopImageX?@LoopImageY?.png");
break    MotionMode
         ifdef       Stand
                  case @Stand :
                       drawImage(iImg[2][2],this.getX(),this.getY(),Graphics.LEFT | Graphics.TOP  );
                            break;
……
while    Using-items-in=MotionState
         select   option= MotionState
                  Stand   while   Using-items-in=MotionMode
                                  getMotion().addAction(@MotionMode, @Stand, new int[]{@Stand });
                  Walk    while   Using-items-in=MotionMode
                                  getMotion().addAction(@MotionMode,@Walk,new int[]{@Walk });
break    MoreMethods
```

Fig. 8. Hero meta-component

Consider the Hero meta-component (Figure 8) as an example. We removed all the constant interfaces (strategy *a*) and other constants (strategy *g*), replacing them with XVCL meta-variables. For example, hero's initial position, originally defined in DigConst interface, have been replaced with meta-variables HeroInitX and HeroInitY. There were many other situations of removing constant definitions (yet another is shown in the *case* clauses).

In Figure 8, multi-value meta-variables *LoopImageX* and *LoopImageY* are used to initialize the image array that controls the movement of the hero. The <*while*> command iterates over *LoopImageX* and *LoopImageY*. The n'th loop iteration uses the n'th value of *LoopImageX* and *LoopImageY* to define the name of the corresponding image file. For example, in the first iteration, meta-expression: *@Hero@LoopImageX?@LoopImageY?.png* is computed as follows:

- Reference *@Hero* is replaced with meta-variable's value "Hero", yielding the intermediate result: *Hero@LoopImageX?@LoopImageY?.png*.
- Reference *@LoopImageX* is replaced with "0" (first value of *LoopImageX* defined in Dig.SPC), yielding the intermediate result *Hero0?@LoopImageY?.png*.
- Reference *@LoopImageY* is replaced with "0" (first value of *LoopImageY* defined in Dig.SPC), yielding the final result *Hero00.png,* which is a required image file.

In the original code, there were two sets of variables and methods to define the moving speed. We applied strategy *f* to remove these and other duplicated methods.

We removed all the unnecessary inheritance relationships (strategy *c*). For example, the TimeManager class is a subclass of Nlayer class, but in fact they share no commons. Therefore this inheritance relationship is removed, the unnecessary class members and method are also deleted.

Optimization strategies *a*, *g* working at XVCL level in which it can provide unique optimization power which could not be achieved by other reuse technologies. Other optimization strategies are embodied into the meta-components at the code level.

7.2 Subsequent Refinements of the RPG-PLA

In the next phase, we extended the initial RPG-PLA to accommodate features of the remaining three games. In the Climb game, the hero can go to the Left and Right, and can Jump on the floor to avoid falling down the mountain. Having examined the Hero component in Dig Gem and Climb games, we found that the differences were pretty minor, limited to handling extra movements of the hero in the Climb game. We added more option values (Figure 9) to the <*select*>/<*option*> in Figure 8 after the <*option*> value *Walk*. And an extra <*ifdef*> command (Figure 10) is also added after the <*break*> named *MoreMethods*. The <*ifdef*> command is similar to cpp's #ifdef: If meta-variable Jumping is defined, then the enclosed part of the <*ifdef*> command is processed, otherwise the enclosed part is ignored.

We applied similar procedures to extend our meta-components to cater for the remaining games, as well as to incorporate optimization strategies. Having addressed all the variants for four RPGs, we obtained the RPG-PLA shown in Figure 11.

Jumping	**while**	*Using-items-in=MotionMode*
		getMotion().addAction(*@MotionMode*, *@ Jumping,* new int[]{*@Jumping* });

Fig. 9. Jumping related actions

ifdef	*Jumping*
	private boolean bJumping=false; ……

Fig. 10. Jumping related methods

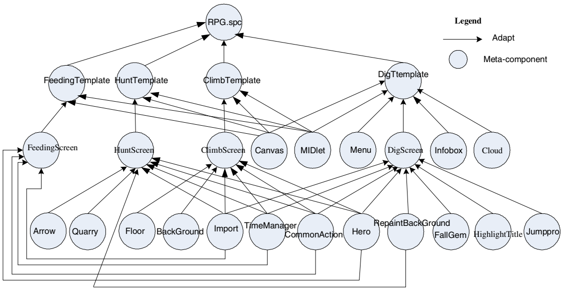


Fig. 11. An optimized X-framework for the mobile RPG product line

8 Deriving RPGs by Customizing the RPG-PLA

By customizing and extending the RPG-PLA, we could derive from the RPG-PLA any of the four RPGs and other similar games. (Term "derivation" is used in [5] to mean reuse-based development of a product line member from the product line architecture). RPGs derived from the RPG-PLA could benefit from optimization strategies.

To develop a new game, we first select the required variant features from the feature diagram (Figure 4). Then we write a suitable SPC and template to define

name : *KongfuTemplate.xvcl*	
set-multi	*MotionMode=<3,7>*
set	*Left= 3*
set	*Right= 7*
set-multi	*MotionState = <Stand, Walk >*
set	*Stand = 1*
set	*Walk = 2*
……	
adapt	*MIDlet.xvcl outfile=?@GameName?.java outdir=games/@GameName*
adapt	*?@GameName?Screen.xvcl outfile=?@GameName?Screen.java outdir=games/@GameName*
adapt	*Canvas.xvcl outfile=?@GameName?Canvas.java outdir=games/@GameName*

Fig. 12. KongfuTemplate for the Kongfu game

name :*KongfuScreen.xvcl*		
set	*Hero = Hero*	
set	*HeroInitY=-50*	
set-multi	*LoopImageX=<0,1,2,3,4>*	
set-multi	*LoopImageY=<0,1,2,3>*	
adapt	*Hero.xvcl*	
	insert	*MoreMethods*
		private boolean bMotiondo; ……
	insert	*MotionMode*
		if(!bMotiondo) { ……
set	*Hero = Master*	
set	*HeroInitY=100*	
set-multi	*LoopImageX=<0,1,2,3,4>*	
adapt	*Hero.xvcl*	
	insert	*MoreMethods*
		private boolean bFirstTouch; ……
	insert	*MotionMode*
		if(!this.isTimeOver()) { ……

Fig. 14. screenshot of the Kongfu game on A6288

Fig. 13. KongfuScreen meta-component

necessary customizations of the existing meta-components. We may also need to develop new meta-components, to address extensions not catered for by the RPG-PLA.

We show the processes with a new game named Kongfu running on Motorola 6288. A ‘learner’ learns Kongfu skills from his ‘master’. Therefore we have two heroes in this game. The game has time manager to limit the time spent on learning. We can reuse the TimeManger, MIDlet and Canvas meta-components from the RPG-PLA without any changes. Figure 12 is the template meta-component for Kongfu.

With <*insert*> command, the original code contained in <break> body can be replaced by the modified code matched with the corresponding <*break*>s’ name.

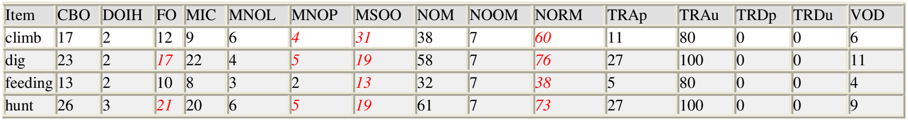
We use <*insert*> to extend the Hero component (Figure 8) with new requirements (e.g., to control the time the learner spent playing the taught action). We <*adapt*> the Hero meta-component in two different ways, to produce components for the master and the learner respectively (Figure 13). In the Kongfu game, the learner should follow his master to learn Kongfu skills. The motion mode is quite different with that of Dig Gem and of other games. Therefore new functions have to be <insert>ed to the <break> named MotionMode (Figure 8) to overwrite the original code (Figure 13).

The running result of Kongfu game derived from RPG-PLA is shown in Figure 14.

9 Design Quality, Performance, and Productivity Evaluation

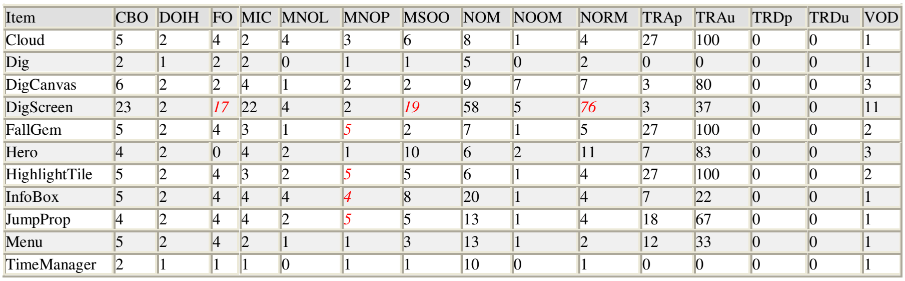
From the OO metrics in Table 4 and we can see that the design quality of original games was very much the same as the quality of games derived from the RPG-PLA. We improved the coupling between objects metrics, while the complexity was increased a bit as the MSOO and MSOR value increased after the using of XVCL.

Table 4. OO metrics for game packages after using XVCL



Item	CBO	DOIH	FO	MIC	MNOL	MNOP	MSOO	NOM	NOOM	NORM	TRAp	TRAu	TRDp	TRDu	VOD
climb	17	2	12	9	6	*4*	*31*	38	7	*60*	11	80	0	0	6
dig	23	2	*17*	22	4	*5*	*19*	58	7	*76*	27	100	0	0	11
feeding	13	2	10	8	3	2	*13*	32	7	*38*	5	80	0	0	4
hunt	26	3	*21*	20	6	*5*	*19*	61	7	*73*	27	100	0	0	9

Table 5. OO metrics for Dig Gem generated from XVCL meta-components



Item	CBO	DOIH	FO	MIC	MNOL	MNOP	MSOO	NOM	NOOM	NORM	TRAp	TRAu	TRDp	TRDu	VOD
Cloud	5	2	4	2	4	3	6	8	1	4	27	100	0	0	1
Dig	2	1	2	2	0	1	1	5	0	2	0	0	0	0	1
DigCanvas	6	2	2	4	1	2	2	9	7	7	3	80	0	0	3
DigScreen	23	2	*17*	22	4	2	*19*	58	5	*76*	3	37	0	0	11
FallGem	5	2	4	3	1	*5*	2	7	1	5	27	100	0	0	2
Hero	4	2	0	4	2	1	10	6	2	11	7	83	0	0	3
HighlightTile	5	2	4	3	2	*5*	5	6	1	4	27	100	0	0	2
InfoBox	5	2	4	4	4	*4*	8	20	1	4	7	22	0	0	1
JumpProp	4	2	4	4	2	*5*	5	13	1	4	18	67	0	0	1
Menu	5	2	4	2	1	1	3	13	1	2	12	33	0	0	1
TimeManager	2	1	1	1	0	1	1	10	0	1	0	0	0	0	1

Table 6. The performance comparison before and after applying XVCL

Name		Memory usage for 3 runnings (Bytes)					Running time (second)				
		1	2	3	average	Improvements	1	2	3	average	Improvements
Climb	Before	325784	325856	325848	325829	2.8%	149	148.6	148.3	148.6	9.9%
	After	316890	316684	316760	316778		133.4	134.2	134	133.9	
Feeding	Before	356624	356640	356604	356623	0.6%	Not available as there is only one scenario from the beginning to the end in the game				
	After	354564	354592	354554	354570						
Hunt	Before	140700	141304	141612	141205	18.9%	168.1	169	173.5	170.2	2.7%
	After	114324	114296	114324	114524		165.8	163.8	167	165.5	
Dig	Before	274856	269088	266532	270158	11.3%	532.5	541.8	512	528.8	9.9%
	After	242336	233732	242900	239656		470.5	490	468.4	476.3	

Performance results are shown in Table 6. For all the games, we measured the running time with memory monitor and profiler turned on in the Wireless Toolkit. This allowed us to slow down the games in order to make more accurate measurements and comparisons. The experiment environment was the Wireless Toolkit 2.2 Beta on Windows XP, Pentium IV 1.4G with 512M memory.

From the we can see that in games derived from the RPG-PLA, the memory usage decreased by almost 19% and the game run almost 10% faster than the corresponding original games, both are the best cases. These performance improvements are due to applying optimization strategies across all the games, further boosted by XVCL's ability to fine tune the strategies in the context of specific games.

Table 7. Line of code comparisons

	Original LOC	Meta-components LOC	Reduced LOC	Reduced Percentage
Total	4526	3325	1201	26.5%

The product line approach also improved maintainability of games. From Table 7, we can see that 26% of the code was reduced. While reduced code size need not automatically mean reduced maintenance effort, in case of our solution it does translate into savings on maintenance. When adding a new game feature, solutions already represented in the RPG-PLA help understand how the new feature is to be implemented. The RPG-PLA provides explicit patterns to address many types of changes which makes evolution easier and kept in a systematic way.

As expected, the product line approach improved productivity of building new games via reuse of the RPG-PLA. The average effort for the development of each of the original four games was 90 man-days. For various business reasons, the Kongfu game was developed twice by two separate Teams A and B, each consisting of two Sanjie developers of comparable skills. Team A developed the Kongfu game in very much the same way as the original four games had been developed, which took the effort of 88 man-days. Group B used RPG-PLA to develop the Kongfu game, which took the effort of 28 man-days. Because the project is still in an initial phase, we do not have more statistics. However, in addition to the reduced effort, the feedback from Team B regarding the ease of reuse was quite encouraging.

10 Related Work

The problem of variability in product lines and product line architectures have been discussed in many sources [2][4][5][9]. Authors of [15] present a way of integrating of variability into the product line architecture to enhance traceability of variant features. The importance of applying a proper variability realization mechanism has been also pointed out in the above mentioned sources, and specific problems (e.g., explosion of component versions) have been discussed in [5].

Various approaches to design and evolve the product line architecture have been described in [2][4][5][9]. In our project, we applied a combination of extractive and pro-active approach. Our strategy was also based on the incremental reengineering of existing system family into a product line developed in our earlier experiments [17]. We applied XVCL in J2EE [16] and ASP [14] Web Portal product lines achieving similar simplification and related productivity gains as we observed in the project described in this paper. To aid in re-engineering of legacy code into product line architectures, we developed a tool called Clone Miner capable of finding similarity patterns in existing programs [1].

Performance is an important consideration for the successful application of reuse technologies [13]. The method described in [6] uses a meta-level architecture to separate domain descriptions from technical code, in which it is similar to XVCL. As the domain discussed in [6] is different from the resource constrained device domain, performance issues are not addressed. A solution to handling variability in technical concerns such as data persistence, screen management, session management with a container abstraction, is described in MobCon [12]. However, MobCon can only handle variants of those predefined technical concerns, while the solution described in this paper can deal with any kind of variability.

11 Conclusions

We described a project in which we applied product line approach to aid in implementing mobile device Role-Playing Games (RPGs). We started with four existing games, and applied a combination of extractive and proactive approaches to build an RPG Product Line Architecture (RPG-PLA). Due to reuse, we shortened the effort to develop a new game from 88 man-days to 28 man-days in the initial applying.

As mobile devices require highly optimized solutions, we paid particular attention to performance. An interesting result of our experiment is that games derived from the RPG-PLA performed better, in terms of both speed and memory consumption, than the original games, developed as custom products. We achieved this result by implementing optimization strategies into the RPG-PLA that could be propagated to games derived from it. In the paper, we presented technical details of our solution, as well as statistics illustrating the results.

In the future work, we plan to accommodate in the RPG Product Line Architecture more variations in game rules, and to address the characteristics of specific devices that have to do with performance. We also plan to apply the product line techniques to other types of mobile device software.

Acknowledgements. This research is sponsored by “Excellent Teacher Funds of Tongji University”. Thanks to Liu Wei and other authors from Meitong Co. Ltd. who have implemented the original games and participated in the project.

References

[1] Basit, A.H. and Jarzabek, S. “Detecting Higher-level Similarity Patterns in Programs,” to appear in ESEC-FSE'05, European Software Eng. Conf. and ACM SIGSOFT Symp. on the Foundations of Software Eng., Sept. 2005, Lisbon

[2] Bosch, J. Design and Use of Software Architectures – Adopting and evolving a product-line approach, Addison-Welsey, 2000

[3] Chidamber S. R. and Kemerer C. F.,“A metrics suite for object oriented design,” IEEE Trans. Software Eng., vol. 20, pp. 476–493, 1994

[4] Clements, P. & Northrop, L. “Software Product Lines: Practices and Patterns”. Boston, MA: Addison-Wesley, 2001.

[5] Deelstra, S., Sinnema, M. and Bosch, J. “Experiences in Software Product Families: Problems and Issues during Product Derivation”. Proceedings of SPLC2004, Boston, Aug. 2004, LNCS3154, Springer-Verlag, pp. 165-182

[6] Fritsch C. and Renz B. Four Mechanisms for Adaptable Systems: A Meta-level Approach to Building a Software Product Line. Proceedings of SPLC2004, Boston, Aug. 2004, LNCS 3154: 51-72

[7] Giguere E. Understanding J2ME Application Models. October 2002.

[8] Kang, K et al. “Feature-Oriented Domain Analysis (FODA) Feasibility Study”. CMU/SEI-90-TR-21, SEI, Carnegie Mellon University, Pittsburgh.

[9] Krueger, C. “Eliminating the Adoption Barrier,” Point-Counter Point Column, in IEEE Software July/August 2002, pages 28-31

[10] Marinescu R. An Object Oriented Metrics Suite on Coupling. Master’s thesis, ”Politehnica” University of Timisoara, 1998.

[11] Michael C. Daconta, et al. More java pitfalls. Wiley, 2003

[12] MobCon www.st.informatik.tu-darmstadt.de/static/pages/projects/mobcon/

[13] Opdahl, A. Sindre, G. Vetland, V. Performance considerations in object-oriented reuse. Proceedings Advances in Software Reuse, Mar. 1993. pp:142–151

[14] Pettersson, U., and Jarzabek, S. “Industrial Experience with Building a Web Portal Product Line using a Lightweight, Reactive Approach,” to appear in ESEC-FSE'05, European Software Eng. Conf. and ACM SIGSOFT Symp. on the Foundations of Software Eng., Sept. 2005, Lisbon

[15] Thiel S. and Hein A: Systematic Integration of Variability into Product Line Architecture Design. Proceedings of SPLC2002; LNCS 2379, San Diego, California, August, 2002 : 130-153

[16] Yang, J. and Jarzabek, S. “Applying a Generative Technique for Enhanced Reuse on J2EE Platform,” accepted for 4th Int. Conf. on Generative Programming and Component Engineering, *GPCE'05*, Sep 29 - Oct 1, 2005, Tallinn, Estonia

[17] Zhang W. at al. Reengineering a PC-based System into the Mobile Device Product Line. Proc. IWPSE03. Helsinki, Finland. Sept. 2003, PP. 149-161

[18] XVCL http://fxvcl.sourceforge.net

Extracting and Evolving Mobile Games Product Lines

Vander Alves, Pedro Matos Jr., Leonardo Cole,
Paulo Borba, and Geber Ramalho

Informatics Center, Federal University of Pernambuco,
P.O. Box 7851 - 50.732-970 Recife PE, Brazil
{vra,poamj,lcn,phmb,glr}@cin.ufpe.br

Abstract. For some organizations, the proactive approach to product lines may be inadequate due to prohibitively high investment and risks. As an alternative, the extractive and the reactive approaches are incremental, offering moderate costs and risks, and therefore sometimes may be more appropriate. However, combining these two approaches demands a more detailed process at the implementation level. This paper presents a method for extracting a product line and evolving it, relying on a strategy that uses refactorings expressed in terms of simpler programming laws. The approach is evaluated with a case study in the domain of games for mobile devices, where variations are handled with aspect-oriented constructs.

1 Introduction

There are several approaches for developing software Product Lines (PL) [5]: proactive, reactive, and extractive [13]. Since the proactive approach supports the full scope of products needed on the foreseeable horizon, it demands a high upfront investment and offers more risks; therefore, it may be unsuitable for some organizations, particularly for small to medium-sized software development companies with projects under tight schedules. In contrast, the other two approaches have reduced scope and therefore require a lower investment; they are incremental and thus can be more suitable for such organizations. An interesting possibility is to combine the last two approaches. But, to our knowledge, this alternative has not been addressed systematically at the architectural and at the implementation levels.

In all approaches, variability management must be addressed in the domain: while focusing on exploiting the commonality within the products, adequate support must be available for customizing the PL core in order to derive a particular PL instance. The more diverse the domain, the harder it is to accomplish this task. This, in some cases, may outweigh the cost of developing the PL core itself.

This paper addresses the issues of structuring and evolving product lines in highly variant domains. In particular, we present a method that relies on the combination of the extractive and the reactive approaches, by initially extracting variation from an existing application and then reactively adapting the newly

H. Obbink and K. Pohl (Eds.): SPLC 2005, LNCS 3714, pp. 70–81, 2005.

created PL to encompass other variant products. The method systematically supports both the extractive and the reactive tasks by defining refactorings that are derived from simple Aspect-Oriented Programming (AOP) [12] laws. Further, we evaluate our approach in the context of an industrial-strength mobile game product line.

Indeed, there are a number of techniques for managing variability from requirements to code level. Most techniques rely on object-oriented concepts. These techniques, however, are well-known for failing to capture crosscutting concerns, which often appear in highly variant domains. Mobile games, in particular, must comply with strict portability requirements that are considerably crosscutting, thereby suggesting AOP to handle variation, which is explored in our method.

The next section provides the background needed for describing our approach. The section briefly explains variability issues in the mobile games domain and also introduces AOP. Section 3 describes our approach, including its strategy and both extractive and reactive refactorings. The industrial case study evaluating the approach is presented in Section 4. We discuss related work in Section 5 and offer concluding remarks in Section 6.

2 J2ME Games and Aspects

Mobile games (and mobile applications, in general) must adhere to strong portability requirements. This stems from business constraints: in order to target more users, owning different kinds of devices, service carriers typically demand that a single application be deployed in a dozen or more platforms. Each platform generally provides vendor-specific Application Programming Interfaces (APIs) with mandatory or optional advanced features, which the developer is likely to use in order to improve game quality. In addition, devices have memory and display constraints, which further requires the developer to optimize the application. In either case, adapting the game for each platform is mandatory.

In this work, we focus on game development for mobile phones using J2ME's MIDP profile, which is targeted at mobile devices with constrained resources [14]. We analyze and manage the specific kinds of variations arising from platform variation, where *platform* means a combination of MIDP, vendor-specific API, and hardware constraints. Accordingly, some of the specific challenges for managing variation in this domain are the following: UI features (such as screen size, number of colors, pixel depth, sound, keypad display); available memory and maximum application size; different profile versions (MIDP 1.0 and MIDP 2.0); different implementation of the same profile; proprietary APIs and optional packages; known device-specific *bugs*; different idioms.

These specific kinds of variation tend to be considerably fine-grained such that they generally crosscut the game core and are tangled with other kinds of variation. This suggests AOP as a suitable candidate for modularizing these variations.

Aspect-oriented languages support the modular definition of concerns that are generally spread throughout the system and tangled with core features. These are called crosscutting concerns and their separation promotes the construction of a modular system, avoiding code tangling and scattering.

AspectJ [1] is the most widely used aspect-oriented extension to Java. Programming with AspectJ involves both aspects and classes to separate concerns. Concepts which are well defined with object-oriented constructs are implemented in classes. Crosscutting concerns are usually separated using units called aspects, which are integrated with classes through a process called weaving. Thus, an AspectJ application is composed of both classes and aspects. Therefore, each AspectJ aspect defines a functionality that affects different parts of the system.

Aspects may define pointcuts, advice and inter-type declarations. Pointcuts describe join points, which are sets of points of the program execution flow. Code to be executed at join points is declared as advice. Inter-type declarations are structures that allow the introduction of fields and methods into a class.

3 Method

Contrary to the proactive approach, which is more like the waterfall model, we rely here on a combination of the extractive and the reactive approaches. There are a number of reasons for this. First, small to medium-sized organizations, which still want to benefit from PLs, cannot afford the high cost incurred in adopting the proactive approach. Second, in domains such as mobile game development, the development cycle is so short that proactive planning cannot be completed. Third, there are risks associated in the proactive approach, because the scope may become invalid due to new requirements.

Our method first bootstraps the PL and then evolves it with a reactive approach. Initially, there may be one or more independent products, which are refactored in order to expose variations to bootstrap the PL. Next, the PL scope is extended to encompass another product: the PL reacts to accommodate the new variant. During this step, refactorings are performed to maintain the existing product, and a PL extension is used to add a new variant. The PL may react to further extension or refactoring.

The method is systematic because it relies on a collection of provided refactorings. Such refactorings are described in terms of templates, which are a concise and declarative way to specify program transformations. In addition, refactoring preconditions (a frequently subtle issue) are more clearly organized and not tangled with the transformation itself. Furthermore, the refactorings can be systematically derived from more elementary and simpler programming laws [6]. These laws are appropriate because they are considerably simpler than most refactorings, involving only localized program changes, with each one focusing on a specific language construct.

3.1 Extraction

The first step of our method is to extract the PL: from one or more existing product variants, we extract a common core and corresponding product-specific adaptation constructs. According to the variability nature of our domain, these constructs correspond to AspectJ constructs. The left-hand side of Figure 1 depicts this approach.

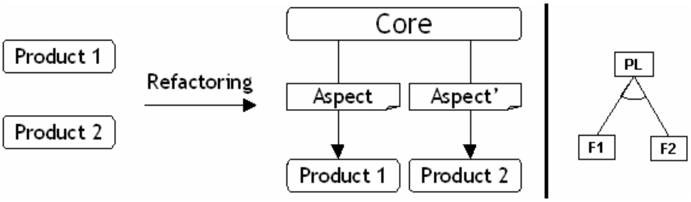


Fig. 1. Bootstrapping the Product Line

Product1 and *Product2* are existing applications in the same domain (for example, versions of a J2ME game for two platforms). *Core* represents the commonality within these applications. The core is composed with the *Aspect* and *Aspect'* aspects in order to instantiate the original products. These aspects thus encapsulate product-specific code.

The feature diagram [8,4] for the PL is shown on the right-hand side of Figure 1. The diagram shows that the new PL is composed of two alternative subfeatures, *F1* and *F2*, representing *Product1* and *Product2*, respectively. The mapping between features and aspects is specified by a configuration knowledge mechanism [7], which imposes constraints on features and aspect combinations like dependencies, illegal combinations, and default combinations. Constraints involving only feature combinations are also specified in the feature model. The feature diagram is simple, since the PL has just been bootstrapped. However, as the PL evolves, either to accommodate more products or to explore further reuse opportunities, the diagram becomes more complex (Section 3.2).

In order to extract the variation within *Product1* and *Product2* — thus defining *Aspect* and *Aspect'*— we must first identify it in the existing code base. When more than one variant exists, diff-like tools provide an alternative. In either case, however, such a view is too detailed at this point. Indeed, the developer first needs to determine the general concerns involved. This could be described more concisely and abstractly with concern graphs, whose construction is supported by a specific tool [16]. Concern graphs localize an abstracted representation of the program elements contributing to the implementation of a concern, making the dependencies between the contributing elements explicit. Therefore, the actual first step in identifying these variations is to build a concern graph corresponding to known variability issues. In the case study described in Section 4, such issues would be the ones discussed in Section 2.

Once the concern graph is constructed, the developer should analyze the variability pattern within that concern. Depending on the pattern, a refactoring may be applied in order to extract it from the core. By analyzing applications in the domain of mobile games, we observed a number of recurring variability patterns, for which the corresponding refactorings are listed in Table 1.

Table 1. Summary of Refactorings

Refactoring	Name
1	Extract Method to Aspect
2	Extract Resource to Aspect - after
3	Extract Context
4	Extract Before Block
5	Extract After Block
6	Extract Argument Function
7	Change Class Hierarchy
8	Extract Aspect Commonality

Some of the refactorings in Table 1, such as *Change Class Hierarchy*, are coarse-grained; others, such as *Extract Argument Function*, are fine-grained; some, such as *Extract Method to Aspect*, have medium granularity. Part of their names refers to an AspectJ construct that encapsulates the variation. For example, the *Extract Method to Aspect* refactoring is intended to extract the variant part of a concern, appearing in the middle of a method body, into AspectJ's inter-type declaration construct. Such declaration can then be implemented according to the specific variant. The refactoring structure is shown next:

Refactoring 1 ⟨Extract Method to Aspect⟩

```
ts
class C {
  fs
  ms
  T m(ps) {
   body
   body′
   body″
  }
}
```

→

```
ts
class C {
  fs
  ms
  T m(ps) {
   body
   newm(αps′);
   body″
  }
}
privileged aspect A {
  T′ C.newm(ps′) {
   body′
  }
}
```

provided

- A cannot be defined in ts;
- $body'$ does not change more than one local variable;
- A does not introduce any field to C with the same name of a C field used in $body'$.

On the left-hand side, *body′* denotes the variability to be extracted. On the right-hand side, such variability is extracted into aspect A's inter-type declaration; thus a different aspect may provide a different variant implementation with that construct. We denote the set of type declarations (classes and aspects) by *ts*. Also, *fs* and *ms* denote field declarations and method declarations, respectively. Finally, we use α preceding a list of parameters to denote only the names of those parameters.

The refactoring provides preconditions to ensure that the program is valid after the transformation. Another use of the preconditions is to guarantee that the transformation preserves behavior. Refactoring 1 has preconditions arising from simpler transformations and refactorings, whose composition yields the whole refactoring.

The first precondition guarantees validity: since the refactoring creates an aspect A, such aspect cannot be defined in *ts*. For the second precondition, as we rely on the *Extract Method* refactoring [9], we need a precondition stating that the piece of code extracted into its own method does not change more than one local variable. Otherwise, the extracted code would need to return two values, and that would not be possible. Regarding the third precondition, visibility modifiers of inter-type declarations are related to the aspect and not to the affected class, according to the AspectJ semantics. Hence, it is possible to declare a private field as a class member and as an inter-type declaration at the same time using the same name. As a consequence, transforming a member method that uses this field into an inter-type declaration implies that the method now uses the aspect inter-typed field. This leads to a change in behavior. A precondition is thus necessary to avoid this problem.

As mentioned, the application of the refactoring creates a new aspect (A), which is related to a variant concern. Further application of other refactorings may refine A, incorporating additional elements of this concern, possibly using other constructs such as pointcuts and advice. In fact, except for Refactorings 1 and 7, all the others in Table 1 deal with pointcuts and advice constructs. A slight variation of Refactoring 1 would consider the pre-existence of aspect A in order to make the refactoring available for repeated applications. Additionally, even though aspect A is privileged, this constraint can be removed later, after moving, with intermediate refactorings, other pieces of the variant concern into the aspect, for example by using the *Extract Resource to Aspect* refactoring.

Indeed, after applying Refactoring 1, there may remain other variabilities to be extracted from the core. The strategy is to apply the refactorings in Table 1 repeatedly, such that the product line core and the variant aspects are built progressively. Section 4 illustrates this with a case study.

3.2 Evolution

Once the product line has been bootstrapped, it can evolve to encompass additional products. In this process, a new aspect is created to adapt the core to the new variant. Moreover, a new feature is added to the feature diagram in order to

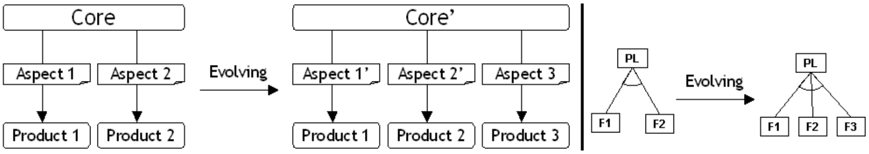


Fig. 2. Evolving the Product Line

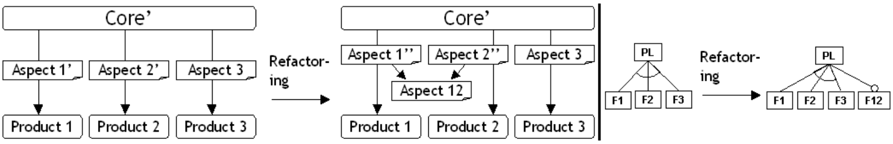


Fig. 3. Refactoring the Product Line

represent the new product, and the configuration knowledge is updated to map the new feature to the new aspect (Figure 2).

The refactorings in Table 1 can also be used for evolution. As Figure 2 also indicates, the core itself may evolve because features common to *Product1* and *Product2* might not be shared by *Product3*. This may trigger further adaptation of the previously existing aspects, too. However, AspectJ tools can identify parts of the core on which these previous aspects depend, and some refactorings are also aspect-aware [10], thereby minimizing the need to revisit such previous aspects.

Another evolution scenario involves restructuring the product line to explore commonality within aspects. Such commonality would not be in the core when it is not shared by all products, but only by a subset. The feature diagram is also changed to show the commonality extraction (Figure 3). The existing commonality is extracted from *F1* and *F2* and is represented as a new optional feature, *F12*. Further, the feature model is augmented with the constraint that *F1* and *F2* depend on *F12*, and the configuration knowledge with the mapping of *F12* to *Aspect12*. An alternative approach would not update the feature model, but then the configuration knowledge would have to map *F1* to $\{Aspect1'', Aspect12\}$ and *F2* to $\{Aspect2'', Aspect12\}$. The former alternative should be used when it is meaningful to have the *F12* feature; the latter when the extracted commonality is meaningful only at the code level.

Figure 3 can become more complex with the addition of new platforms and identification of reusable aspects. However, constraints in the feature model as well as the configuration knowledge (the mapping of features to aspects) limit aspect combinations, thereby providing support for scalability.

4 Method Evaluation

We performed a case study to evaluate variability in J2ME games, which are mainstream mobile applications of considerable complexity in comparison with other mobile applications. In particular, we investigated how the same game GM was adapted to run in three platforms (P_1, P_2, and P_3)[1]. P_1 relies solely on MIDP 1.0, whereas P_2 and P_3 rely on MIDP 1.0 and a proprietary API. GM is a game currently offered by service carriers in South America and Asia.

The variability issues within these products are as follows: optional images, proprietary API, application size limit, screen dimensions, and additional keys. One important remark is that these features are not independent. Indeed, application size constrains other features, such as optional images and additional keys.

In order to evaluate our approach, we created a PL implementation of the three products and then compared the PL version with the original implementation of these products. To create and evolve the PL, we first identified the variabilities (such as optional images) with concern graphs and then moved their definition to aspects using the *Extract Resource to Aspect* refactoring. In another step, we addressed method body variability within the platforms. Accordingly, we made extensive use of the *Extract Method to Aspect* refactoring. The *Extract After Block* and *Extract Before Block* refactorings were used when the variant code appeared at the end or beginning of the method body. On the other hand, the *Extract Context* refactoring was used when the variation surrounded common code, representing a context to it. The *Extract Argument Function* refactoring was used when variation appeared as an argument for a method call. Finally, we used the *Change Class Hierarchy* refactoring to deal with class hierarchy variability.

As mentioned in Section 3.1, in order to better identify and understand some variations, we can use concern graphs, which are created iteratively by querying a model of the program, and by determining which elements (class, methods, and fields) and relationships returned as part of the queries contribute to the implementation of the concern. The querying process starts with a *seed* [16], usually a class found with a lexical tool. From this class, the remaining elements are added with tool support. For example, the concern graph C for the optional images concern (oi) in P_1 would be as follows:

$$C_{p1,oi} = (V_{p1,oi}, V^{*}_{p1,oi}, E_{p1,oi}),\ V^{*}_{p1,oi} = \varnothing$$

$$V_{p1,oi} = \left\{ \begin{array}{l} Resources, GameScreen, Resources.dragonRight, \\ Resources.loadImages(), GameScreen.wakeEnemy() \end{array} \right\},$$

$$E_{p1,oi} = \left\{ \begin{array}{l} (reads, GameScreen.wakeEnemy(), Resources.dragonRight), \\ (writes, Resources.loadImages(), Resources.dragonRight), \\ (declares, Resources, Resources.dragonRight), \\ (declares, Resources, loadImages()), \\ (declares, GameScreen, wakeEnemy()) \end{array} \right\},$$

The set $V_{p1,oi}$ describes the vertices (classes, methods, attributes) partially implementing the concern. Set $V^{*}_{p1,oi}$ consists of vertices (classes, methods) solely

[1] The actual names are not relevant here.

dedicated to the concern implementation. Finally, set $E_{p1,oi}$ groups edges relating elements from the previous sets.

During the evolution of the PL to include P_3, we had to deal with the *load images on demand* concern. This concern was specific to this platform, as it had constrained memory and processing power. To implement this concern, we had to define a method for each screen that could be loaded. Before a screen was loaded, the corresponding method was called. In contrast, in P_1 and P_2 implementations, the images were loaded only once, during game start-up. In this case, there was only one method that loaded all the images into memory. This situation illustrates the scenario in Figure 2.

We addressed this by applying a sequence of *Extract Method* refactorings in the core to break the single method loading all images into finer-grained methods loading images for each screen; the call of this single method was then moved from the core to P_1's and P_2's aspects, and the calls to such smaller methods were moved to P_3's aspect by the *Extract Before Block* refactoring.

Another evolution scenario took place when we realized that some commonality existed between P_1 and P_2 with respect to the *flip* feature[2]: these two platforms are from the same vendor and share this feature, which is not shared by P_3, from another vendor. Therefore, the flip feature is isolated in the corresponding aspects of P_1 and P_2, but it would be useful to extract this commonality into a single module. In fact, we were able to factor this out into a single generic aspect with the *Extract Aspect Commonality* refactoring, thus illustrating the scenario in Figure 3.

After creation and evolution of the PL, we analyzed code metrics. Table 2 shows the number of Lines of Code (LOC) for each product in the original implementation, in contrast with the PL implementation. We calculate the LOC of a PL instance as the sum of the core's LOC and the LOC of all aspects necessary to instantiate this specific product.

Table 2. LOC in original and PL implementations

Original Implementation				**PL Implementation**				
P_1	P_2	P_3	Total	Core	P_1	P_2	P_3	Total
2965	2968	3143	9076	2549	3042	3047	3210	4405

Table 2 shows that LOC is slightly higher when comparing each PL instance with the corresponding product in the original implementation. This is caused by the extraction of methods and aspects, which increase code size due to new declarations. On the other hand, there is a 48% reduction in the total LOC of the PL implementation, when compared to the sum of LOCs of the single original versions. This was possible because the core, which represents 57% of the PL LOC, is reused in all instances, thus eliminating most of code repetition

[2] Proprietary graphic API allowing an image object to be drawn in the reverse direction, without the need for an additional image.

occurring when there are three independent implementations. Another factor that contributes to the reduction in PL LOC is the existence of reusable aspects.

Another analyzed metric was the packaged application (jar files) sizes of the original and of PL implementations (Table 3). The jar files include not only the bytecode files, but also every resource necessary to execute the application, such as images and sound files. The jar file size is a very important factor in games for mobile devices, due to memory constraints.

We can notice a jar size increase from original versions to PL instances. The reason for this is the overhead generated by the AspectJ weaver on the bytecode files. We also noticed that very general pointcuts intercepting many join points can lead to greater increases in bytecode file sizes. This considerably influenced us in the definition and use of the refactorings. Moreover, we can gain a significant reduction in the jar size when using a bytecode optimization tool [2]. The reduced size of each original version and PL instance are shown in Table 3.

Table 3. Jar size (kbytes) in original and PL implementations

	Original Implementation		**PL Implementation**	
	size	reduced size	size	reduced size
P_1	61,9	58,5	97,0	67,9
P_2	61,7	57,3	97,6	61,8
P_3	56,1	52,4	93,5	56,7
Total	179,8	168,2	288,2	186,3

5 Related Work

Prior research also evaluated the use of AOP for building J2ME product lines [3]. We complement this work by considering the implementation of more features in an industrial-strength application, explicitly specifying the refactorings to build and evolve the PL, and raising issues in AspectJ that need to be addressed in order to foster widespread application in this domain.

AOP refactorings have also been described elsewhere [15,11]. The former proposes a catalog for object-to-aspect and aspect-to-aspect refactorings, whereas the latter provides an abstract representation of object-to-aspect refactorings as roles. However, their use in the PL setting is not explored, and the refactorings format follows the imperative style [9]; in contrast, our approach is template-oriented, abstract, concise, and thus does not bind a specific implementation, which could be done, for instance, with a transformation systems receiving as input refactoring templates.

Concern graphs provide a more concise and abstract description of concerns than source code [16]. We rely on concern graphs to identify variant features. Once the concern is identified, we extract it into an aspect and may further revisit it during PL evolution.

In previous work, a language-independent way to represent variability is provided, and it is shown how it can be used to port J2SE applications to a J2ME product line [17]. Our approach differs from such work because, although ours relies on language-specific constructs, it has the advantage of not having to specify join points in the base code.

6 Conclusions

We present a method for creating and evolving product lines combining the reactive and extractive approaches. Our method uses a set of refactorings, which can be extended when necessary. These refactorings can be derived from a combination of programming laws that allow us to better understand these refactorings and increase the confidence that they are correct. Our refactorings rely on AOP to modularize crosscutting concerns and to generalize the implementations of these concerns in order to increase code reuse. Constraints in the feature model and in the configuration knowledge limit aspect combination and thus promote scalability of the process.

Our evaluation with an existing mobile game shows that we can benefit from extensive code reuse and easily evolve the PL to encompass other products while still maintaining code reliability. It also shows that the sequence of applied refactorings must be strategically chosen. This strategy can be influenced by some factors like desirable reuse level and application size restrictions. Although the evaluation is in the mobile game domain, we argue that the method and the issues addressed here are valid for mobile applications in general, of which mobile games are representative. We also believe that other highly variant domains could benefit from our method.

Acknowledgements

We thank Meantime Mobile Creations for granting us access to the game used in our case study and for fruitful discussions with lead developers Alexandre Damasceno and Pedro Sampaio. We also appreciate valuable feedback from the members of SPG, André Santos, Jeff Gray, the anonymous referees, and our shepherd Pierre America. This work was partially supported by CNPq and FACEPE.

References

1. *AspectJ project.* http://www.eclipse.org/aspectj/, 2005.
2. *ProGuard.* http://proguard.sourceforge.net, 2005.
3. M. Anastasopoulos and D. Muthig. An evaluation of aspect-oriented programming as a product line implementation technology. In *Proceedings of the International Conference on Software Reuse (ICSR)*, 2004.
4. T. Bednasch, K. Czarnecki, U. Eisenecker, and M. Lang. *Captain Feature.* https://sourceforge.net/projects/captainfeature/, 2005.

5. P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns.* Addison-Wesley, 2002.
6. L. Cole and P. Borba. Deriving refactorings for AspectJ. In *AOSD'05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 123–134, 2005.
7. K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley, 2000.
8. K. Czarnecki, S. Helsen, and U. Eisenecker. Staged configuration using feature models. In *SPLC*, pages 266–283, 2004.
9. M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code.* Addison–Wesley, 1999.
10. Oberschulte C. Hanenberg S. and Unland R. Refactoring of aspect-oriented software. In *Net.ObjectDays*, Erfurt, Germany, September 2003.
11. J. Hannemann, G. Murphy, and G. Kiczales. Role-based refactoring of crosscutting concerns. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 135–146, 2005.
12. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect–Oriented Programming. In *European Conference on Object–Oriented Programming, ECOOP'97*, LNCS 1241, pages 220–242, 1997.
13. C. Krueger. Easing the transition to software mass customization. In *Proceedings of the 4th International Workshop on Software Product-Family Engineering*, pages 282–293, 2001.
14. Sun Microsystems. *JSR-37 Mobile Information Device Profile (MIDP).* `http://jcp.org/aboutJava/communityprocess/final/jsr037/index.html`, 2000.
15. M. Monteiro and J. Fernandes. Towards a catalog of aspect-oriented refactorings. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 111–122, 2005.
16. M. Robillard and G. Murphy. Concern graphs: Finding and describing concerns using structural program dependencies. In *Proceedings of the 24th International Conference on Software Engineering*, pages 406–416, 2002.
17. W. Zhang, S. Jarzabek, N. Loughran, and A. Rashid. Reengineering a PC-based system into the mobile device product line. In *Proceedings of the Sixth International Workshop on Principles of Software Evolution (IWPSE'03)*, 2003.

Determining the Variation Degree of Feature Models

Thomas von der Maßen and Horst Lichter

Research Group Software Construction,
RWTH Aachen University
{vdmass, lichter}@cs.rwth-aachen.de

Abstract. When developing a product line the knowledge about the variation degree is of vital importance for development, maintenance and evolution of a product line. In this paper we focus on the variation degree of product line feature models, considering different types of variability and dependency relationships between features.

1 Introduction

Feature based domain modeling is a well-known technique in requirements engineering of product lines. The overall objective is to model commonality and variability in a product line feature model (PLFM) [2]. From the PLFM, dedicated product feature models (PFM) can be derived. The process of deriving a PFM is called *product instantiation*. Instantiation is done by resolving the variation points of the PLFM correctly, especially obeying the dependencies between features. The basis for instantiation is a normalized feature tree, which we discussed in detail in [3].

In this paper we focus on the *variation degree* of a PLFM representing the number of valid PFMs that can be instantiated. Only little work has already been done in formalizing feature models and in investigating how the variation degree of feature models can be calculated. Besides Eisenecker et al. [4], an important contribution is provided by van Deursen and Klient [5] who presented a Feature Description Language to formulate features and feature models in a textual representation. They introduced rules for computing variability. These rules are almost equivalent to the formulas we present in section 2.1. Unfortunately they ignored the presence of dependencies. Based on their approach we propose how to calculate the variation degree of a PLFM considering dependencies.

2 Variation Degree

The *variation degree* reveals the instantiation space of the product line on basis of the feature model. The variation degree of a feature in the feature tree represents the number of possible instantiations below this feature. Therefore, the variation degree of the root feature states the number of possible instantiations of the PLFM. This information is of vital importance for deciding whether the PLFM captures the desired variability adequately. In a proactive product line approach the variation degree is necessary for planing the product line. It reveals information about the flexibility and complexity of the product line, having great impact on every development phase like design, implementation and quality assurance. In a reactive approach and especially in merging several individual products into a product line, the variation degree is even more important,

H. Obbink and K. Pohl (Eds.): SPLC 2005, LNCS 3714, pp. 82 – 88, 2005.

because this information was not required yet and is therefore not known. The necessary product line oriented reengineering of the individual products, benefits from the variation degree. Furthermore, the impact on the variation degree can be assessed, when introducing a new feature.

At first, we consider only domain relationships, ignoring dependencies, to determine the variation degree of a single feature. In a second step we regard dependencies as well and analyze how dependencies influence the instantiation space.

2.1 Determining the Variation Degree Considering Domain Relationships

In the following $var(F)$ represents the variation degree of a feature *F*. The variation degree of a feature representing a leaf in the feature tree is set to 1. The variation degree of a non-leaf feature (father feature) is influenced by the variation degrees of all its children and the domain relationship type that connects the children to the father feature.

At first, we analyze the variation degree with respect to different domain relationship types, offered by most feature modeling approaches (e.g. FODA [1]).

Mandatory. A mandatory-relationship between the features *F* and *CF* means that if the father feature *F* is selected for a PFM, the child feature *CF* must be selected as well. The variation degree of a feature *F* with *n* mandatory child features CF_i is the product of the variation degrees of all mandatory child features.

$$var(F) = \prod_{i=1}^{n} var(CF_i)$$

Option. An option-relationship between the features *F* and *CF* means that if the father feature *F* is selected for a PFM, the child feature *CF* can but needs not to be selected. The variation degree of a feature *F* with *n* optional child features CF_i is the product of the variation degrees of all optional child features. The variation degree of an optional child feature is increased by 1 because the optional child feature can be selected or not.

$$var(F) = \prod_{i=1}^{n} (var(CF_i) + 1)$$

Alternative. An alternative-relationship between the features *F* and *CF* means that if the father feature *F* is selected for a PFM, exactly one feature of the alternative child features must be selected. The variation degree of a feature *F* with *n* alternative child features CF_i is determined by the sum of the variation degrees of the alternative child features because exactly one alternative child feature can be selected.

$$var(F) = \sum_{i=1}^{n} var(CF_i)$$

Or. An or-relationship between the features *F* and *CF* means that if the father feature *F* is selected, at least one of the or-child features must be selected. The variation degree of a feature *F* with *n* or-child features CF_i is equal to the variation degree of a feature *F* with optional child features decreased by 1 because at least one or-child feature has to be selected. The case that no child feature is selected must be subtracted from the number of possible instantiations.

$$var(F) = \left(\prod_{i=1}^{n}(var(CF_i) + 1)\right) - 1$$

Figure 1shows an example feature model to illustrate the formulas given above.

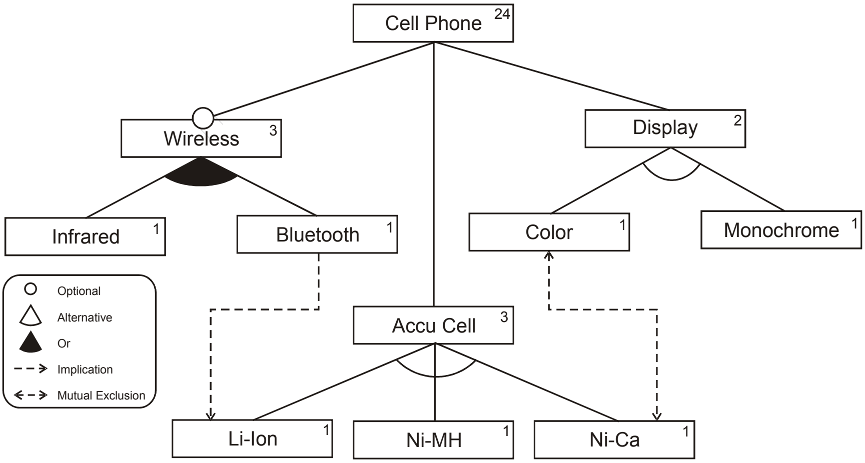


Fig. 1. Variation degrees in a feature tree

The variation degree of all leaf features is 1. *var(Accu Cell)* is 3, because it is the sum of its child feature variation degrees; *var(Wireless)* is $(1 + 1) \cdot (1 + 1) - 1 = 3$; *var(Display)* is 2 and *var(Cell Phone)* is determined by $(3 + 1) \cdot 3 \cdot 2 = 24$. Hence, the number of valid PFMs (ignoring the modeled dependencies) is 24. The valid PFMs can be easily determined. They are not listed here because of space restrictions.

2.2 Determining the Variation Degree Considering Dependencies

Dependencies constrain the binding of variation points. To be more precise, the existence of a feature F_1 in a PFM determines the existence or non-existence of another feature F_2, if a dependency has been modeled between these features. Therefore, dependencies constrain the number of possible instantiations and thus they reduce the variation degree of a PLFM. In this section we analyze the impact of dependencies on the variation degree. We are considering the dependencies *implication* and *mutual exclusion*, as these dependency types are offered by most feature modeling approaches. An implication states that if the source feature is selected for a PFM the target feature must be selected as well. A mutual exclusion between two features states that not both features can be selected for the same PFM.

2.2.1 Considering a Single Dependency

First of all, we show by means of our example how a single dependency influences the variation degree of a PLFM. If we consider e.g. the implication between the features *Bluetooth* and *Li-Ion*, the PFMs containing *Bluetooth* but not *Li-Ion* become invalid. Hence, the number of valid instantiations is reduced by 8, because 8 PFMs that do contain *Bluetooth* but not *Li-Ion* become invalid.

If we consider e.g. the mutual exclusion between features *Color* and *Ni-Ca* all instantiations containing both features become invalid, reducing the number of valid instantiations in our example from 24 to 20. To determine the variation degree of the PLFM we define:

- Ω = Set of valid PFMs, ignoring all dependencies
- ω = $|\Omega|$ (variation degree of the root feature ignoring dependencies)
- Δ_{Dep_i} = Set of invalid PFMs, considering only dependency *i*
- $\delta_{Dep_i} = |\Delta_{Dep_i}|$

We can always apply the following procedure to determine the number of valid PFMs considering a single dependency Dep_i:

1. Determine the number of all valid PFMs ω ignoring the dependency *i*
2. Determine the number of invalid PFMs δ_{Dep_i} resulting from dependency *i*
3. The number of valid PFMs considering the dependency *i* is the difference of these the numbers $\omega_{Dep_i} = \omega - \delta_{Dep_i}$

Table 1. Variation degree with selected / not selected child feature

Domain relationship type	Variation degree
Mandatory	$var(F)_{CF_i} = \prod_{j=1}^{n} var(CF_j)$ $var(F)_{\neg CF_i} = 0$
Option	$var(F)_{CF_i} = var(CF_i) \cdot \prod_{j=1}^{n} (var(CF_j) + 1), \forall(j \neq i)$ $var(F)_{\neg CF_i} = \prod_{j=1}^{n} (var(CF_j) + 1), \forall(j \neq i)$
Alternative	$var(F)_{CF_i} = var(CF_i)$ $var(F)_{\neg CF_i} = \sum_{j=1}^{n} var(CF_j), \forall(j \neq i)$

Table 1. Variation degree with selected / not selected child feature

Domain relationship type	Variation degree
Or	$var(F)_{CF_i} = var(CF_i) \cdot \prod_{j=1}^{n}(var(CF_j)+1), \forall(j \neq i)$ $var(F)_{\neg CF_i} = \prod_{j=1}^{n}(var(CF_j)+1) - 1, \forall(j \neq i)$

The number of invalid PFMs resulting from a dependency is influenced by the dependency type and the variation degrees of the features that are connected by the dependency. Thus, it is necessary to establish the invalid PFMs, concerning the dependency. Therefore, the variation degree of a feature must be determined if a dedicated child feature has to be part, respectively must not be part of a PFM. Table 1 depicts how to calculate the variation degree of a feature *F*, considering the various domain relationship types. Hereby $var(F)_{CF_i}$ represents the variation degree of feature *F* if child feature CF_i is selected and $var(F_i)_{\neg CF_i}$ represents the variation degree of feature *F* if child feature CF_i is discarded.

2.2.2 Considering Multiple Dependencies

The formulas given above hold true if only one dependency is modeled in the PLFM. If the feature tree contains multiple dependencies (which is the normal case), the calculation becomes more difficult because of correlations between the constraints resulting from dependencies.

Two dependencies Dep_1 and Dep_2 are *correlated* if the intersection set of invalid PFMs resulting from these dependencies is not empty. To get exact values of the variation degrees, information about all sets of invalid PFMs Δ_{Dep_i} is needed. If these sets are known (e.g. by applying derivation and configuration approaches) ω_{Dep} can be exactly calculated by

$$\omega_{Dep} = |\Omega_{Dep}| \text{ whereas } \Omega_{Dep} = \Omega - \bigcup_{i=1}^{n} \Delta_{Dep_i} \text{ and}$$

- Ω_{Dep} = Set of valid PFMs, considering all dependencies
- $\omega_{Dep} = |\Omega_{Dep}|$ (variation degree of the root feature considering dependencies)

If the invalid PFMs Δ_{Dep_i} are not known, the following upper and lower bounds can be given as a first approximation for the number of valid instantiations of a PLFM considering all dependencies. The upper bound is:

$$\overline{\omega}_{Dep} = \omega - max(\delta_{Dep_i})$$

The lower bound is:

$$\underline{\omega}_{Dep} = max\left(0, \omega - \sum_{i=1}^{n} \delta_{Dep_i}\right)$$

For calculating the upper bound we consider only the dependency with the greatest impact on the number of valid PFMs. Therefore, the number of valid PFMs is reduced by at least δ_{Dep_i} (where Dep_i leads to the largest set of invalid PFMs). ω_{Dep} can be lower if there are other sets of invalid PFMs Δ_{Dep_j} that are no subsets of Δ_{Dep_i}. That means,these dependencies lead to invalid PFMs that are not covered by the dependency *i*.

For calculating the lower bound, we consider all dependencies. If a dependency *i* correlates with a dependency *j*, we subtract the PFMs of the intersection set of Δ_{Dep_i} and Δ_{Dep_j} twice from the number of valid PFMs. Therefore the sum of the δ_{Dep_i} can become greater than ω. In this case we set the lower bound to 0, as the number of valid PFMs cannot become negative.

The upper bound gives a good approximation if the dependencies are strong correlated with each other. It represents the exact variation degree if $\Delta_{Dep_i} \supseteq \Delta_{Dep_j}, \forall (j \neq i)$. That means, the set of invalid PFMs resulting from a dependency *i* is the super-set of all other sets. The lower bound gives a good approximation if the correlation of the dependencies is low. It represents the exact variation degree if there is no correlation between all modeled dependencies, i.e.

$$\Delta_{A_i} \cap \Delta_{A_j} = \varnothing \qquad \forall((i,j \in \{1, \ldots, n\}) \wedge i \neq j)$$

Though the given bounds may represent the exact number of valid PFMs concerning all dependencies, in most cases they are just approximations.

Concerning the *Cell Phone* example, the following values can be determined: $\omega = 24$, $\delta_{Dep_{MutExcl}} = 4$, $\delta_{Dep_{Impl}} = 8$, $\omega_{Dep} = 14$, $\overline{\omega}_{Dep} = 16$, $\underline{\omega}_{Dep} = 12$. The bounds are just approximations, because the dependencies correlate in two invalid PFMs, namely: {*Cell Phone, Wireless, Bluetooth, Accu Cell, Ni-Ca, Display, Color Display*} and {*Cell Phone, Wireless, Infrared, Bluetooth, Accu, Ni-Ca, Display, Color Display*}.

3 Conclusion

Originally introduced for modeling application domains, feature modeling has been successfully applied in the context of product line engineering, because of its expressiveness of common and variable characteristics. Typically the variant characteristics and their relationships express the flexibility of a product line and how many different products can be derived. Therefore, we introduced the variation degree, which is important for a reactive product line approach and during evolution. In contrast to existing approaches we considered not only domain relationships but also dependencies between features. Especially, determining the variation degree if multiple dependencies are present, is a hard task, because of correlations between the dependencies. Furthermore, for an exact calculation, the invalid PFMs must be known. As these sets are normally not known, we introduced an upper and a lower bound for the variation degree of a PLFM and discussed under which conditions the bounds provide good and bad results.

References

1. Kyo C. Kang et al., *Feature-Oriented Domain Analysis (FODA) Feasibility Study*, Technical report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, 2000.
2. Horst Lichter, Alexander Nyßen, Thomas von der Maßen, Thomas Weiler, *Vergleich von Ansätzen zur Feature Modellierung bei der Softwareproduktlinienentwicklung*, Aachener Informatik Berichte, Aachen 2003.
3. T. von der Massen, H. Lichter, *Deficiencies in Feature Models*, Proceedings of the Software Variability Management for Product Derivation - Towards Tool Support, International Workshop of SPLC 2004, Boston, 2004.
4. U. Eisenecker, M. Selbig, F. Blinn, K. Czarnecki, *Merkmalmodellierung für Softwaresystemfamilien*, ObjectSpektrum, 5/2001, pp. 23-30, 2001.
5. A. van Deursen, P. Klint, *Domain-Specific Language Design Requires Feature Descriptions*, Journal of Computing and Information Technology, 2001.

Modeling Architectural Value: Cash Flow, Time and Uncertainty

J.H. Wesselius

Philips Medical Systems, Business Unit Cardiovascular X-ray Systems,
P.O. Box 10,000 5680 DA Best, The Netherlands
Jacco.Wesselius@philips.com

Abstract. Developing a product family (architecture) means making early investments. The product family architecture roadmap has to be considered in a business context: how to optimize the expected business value? Estimating the business value of (architectural) investments is a key step. This paper proposes an extension to existing value estimating approaches by combining an NPV-based approach with strategic scenarios to deal with uncertainty and expectations about the future.

1 Introduction

Product family development is costly. The economical justification for product family architecture development is based on *future* cost reduction or *future* value creation. This means that product family architecture planning is done in the context of expectations and assumptions about future developments: How do we expect the future to be? How should the product family architecture evolve in order to maximize its benefits? What is the business case for product family development?

To make the right choices for investments in product line development (investments in architecture, organization, asset development *etc.*) estimating the value of the investment is an essential step. To compensate for the effect of time (income generated today is worth more than income generated in a couple of years), Net Present Value (NPV) calculations are commonly used to make business decisions. In this paper, we combine NPV-based value calculations with a scenario-based approach to modeling expectations about the future. This way, expectations are made explicit and probability assessments of scenarios can be used to estimate the expected value of the investments.

First, a brief overview of economical factors in product line development will be given. What are the effects of time, and what are the effects of uncertainty about the future? After that, the effects of time and uncertainty will be combined into some a formula combining NPV-calculations with scenario probabilities. This formula explicitly expresses the way time and uncertainty influence the expected value of investments. Finally, a simple example will be presented to clarify the effect of the factors considered in the formula.

H. Obbink and K. Pohl (Eds.): SPLC 2005, LNCS 3714, pp. 89 – 95, 2005.

2 Cost Saving and Value Creation Opportunities

In [1][2] a series of case studies is described to calculate the cost/benefits of product line development. In these studies a formula is used that captures the components of the cost of product line development:

$$C_{org} + C_{cab} + \sum_{i=1}^{n_1} (C_{unique}(P_i) + C_{reuse}(P_i)) \tag{1}$$

C_{org} = organizational cost to adopt product line development

C_{cab} = development cost of core asset base suited to support the product line being developed

$C_{unique}(P)$ = the cost of developing unique software for product *P*, (software that is not based on the product line platform)

$C_{reuse}(P)$ = the development cost to reuse core assets for the development of product *P*.

The organizational cost and the cost of core asset base development represent the initial investments. For the investments in product line development to be economically sound, the overall cost should be less than the cost of developing the products on-by-one. In that case, product line development helps to reduce development cost.

Development cost is just one factor, however. To optimize the value of the investments they need to be well planned and well aligned with the organization's business drivers. Although different organizations will have different drivers, many common drivers have been identified for which product line development can make positive contributions: time to market, development efficiency, cross-product compatibility, cost of product upgrades, cost of product manufacturing, life cycle management costs etc. These are abstract terms that need to be translated into economical terms to judge the value of the investments in the product line (organization development, architecture and other assets).

From an economical point of view, the value of an investment is high if the expected benefit from the investment is very *probable* and the *expected time* between making the investment and getting the return on investment is short. To judge the value of investments, both aspects should be taken into account. Making the relation between the investment and the expected return on investment explicit is necessary to justify the investment. Take for example "time to market". This is in many cases an important driver. But is the claim of reduced TTM always justified? To justify this claim, assumptions are made about the future, e.g., which future products will be developed? when will these products be developed? when will the market ask for these products? what would the TTM reduction be if we make the investment? would a valuable market opportunity be lost if we would not make the investment? *etc.*

3 Modeling the Value of (Architectural) Investments

To quantify the value of investments, Net Present Value calculations are commonly used (see also [3] for an example of using NPV-calculations to quantify the value of architectural investments). In NPV calculations, a discount rate is used to compute the value of *future* cost and *future* income: cash flow generated today is worth more than cash flow generated later.

$$NPV = \frac{cashflow}{(1+discountRate)^{time}} \tag{2}$$

To use NPV-calculations for evaluating the value of architectural investments, *architectural scenarios* (term taken from [4]) can be used. An architectural scenario is a sequence of events characterized by the associated cash flow and the moment in time the cash flow will be generated; a positive cash flow for income, and a negative cash flow for investments. The NPV for an architectural investment can easily by calculated with the NPV formula (1) by summing the NPV for the individual events.

Doing an NPV-calculation for architectural scenarios is not straightforward however, since the value of an investment can only be judged in view of assumptions about the future: market developments, application developments, and technology developments. In [4], these assumptions are captured in *strategic scenarios*. For different strategic scenarios the value of an architectural scenario will be different:

- if an architectural scenario creates value by enabling easy development of certain features, the value of the architectural scenario is high in strategic scenarios that predict a high business value for these features;
- if the enabled features prove to have no business value in another strategic scenario, the value of the architectural scenario will prove low in that strategic scenario.

The value of architectural investment scenarios is never for 100% certain, because the future is not certain. We therefore speak of the *expected* NPV. The expected NPV can be evaluated in the context of a set of strategic scenarios, which make assumptions and expectations about the future explicit. Using these strategic scenarios, we propose to determine the expected value of architectural scenarios in four steps:

1. draw up the architectural scenarios;
2. draw up the most important strategic scenarios;
3. estimate the cash flow for the architectural scenarios in combination with the strategic scenarios:
   a. estimate the investments needed to realize the architectural scenarios;
   b. estimate the expected income for the architectural scenario if combined with the strategic scenario;
4. calculate the *expected* NPV as follows:

$$\begin{aligned} &NPV_{Expected}(ArchScenario, StratScenario[1..n]) = \\ &\quad \sum_{i=1}^{n} NPV(ArchScenario, StratScenario[i]) * probability(StratScenario[i]) \end{aligned} \tag{3}$$

This approach makes explicit which factors contribute to the economical justification of investments in architectural features of the product line:

1. a high *probability* of actually creating value based on the architectural investments;
2. a *short time interval* between making the architectural investment and realizing the benefits of the investment.

Note that it is not possible to judge the quality of the investment decisions by the outcome: a choice that is bad in view of the expected future could result in a very positive outcome when things do not go as expected. The quality of the decision should be judged in view of the information available at the moment of making the decision. In hindsight, anyone can be a genius. Formula (3) gives insight in the way uncertainty could be dealt with. The way uncertainty is dealt with makes the difference between quality investments and plain gambling.

4 An Example

Suppose that a company wants to build a product *A*. When looking into the future, one might expect demand for two similar products *A'* and *A''*. When looking into the design of these products, the three variants of the product could be built by using a common part (called A_{common}) and three extensions of this common part (A_{ext}, A'_{ext}, and A''_{ext}). Suppose that an investment has to be made to separate the common part from the extensions.
Assume that[1]:

1. Income from the products will be €4000 in the first year and €2000 in the second to fourth year. After 4 years the products will not be sold anymore.
2. Each year at most €1000 can be spent on product development.
3. Developing the 3 products from scratch costs €1000 per product.
4. Fully preparing the architecture from the start for the A'_{ext}, and A''_{ext} development requires an initial investment of €600. This will delay the introduction of product *A* with one year (missing the high income in the first year). After the architecture has been fully prepared, completing product *A'* and *A''* only costs €50 per product. Since only a small development effort is needed, the two products can be offered immediately when the market demands them (reduced time to market!!)
5. Partially preparing the architecture after product *A* has been released requires investments of €400, which can be made in 2 consecutive years after product *A* has been completed. When this investment is made, completing *A'* and *A''* costs only €300 per product.

What would be the right choice for the product line architecture roadmap?
Consider three architectural scenarios:

1. Just build product *A*: do not invest in future cost saving.
   Total cost of the three products will be: €3000

[1] Multiply cash flow numbers with any factor you like to make it more realistic.

2. Partially prepare for *A*’ and *A*’’: first build product *A*, and make some investments into the architecture in the next few years (after the product has been put into the market).

   Total cost of the three products will be: €1000 + €400 + 2*€300 = €2000.
3. Fully prepare for *A*’ and *A*’’: make the investment into the architecture from the start (accept the initial cost + delayed product introduction).

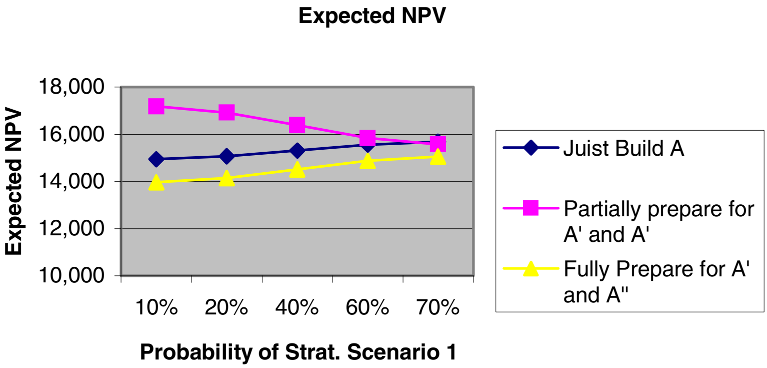
   Total cost of the three products will be: €1000 + €600 + 2*€50 = €1700.

When comparing the total development cost (as in formula (1)), architectural scenario 3 would be the preferred scenario. But what happens when we take NPV and strategic scenarios into account (as in formula (3))? Much depends on the expected timing of the market demand for product *A’*, and *A’’*.

Assume the following strategic scenarios:

1.	Product *A’*, and *A’’* will be demanded in 2007	(likelihood: *X%*)
2.	Product *A’*, and *A’’* will be demanded in 2009	(likelihood: 100 - 30 - *X%*)
3.	Product *A’*, and *A’’* will never be demanded	(likelihood: 30%)

If we vary the likelihood of strategic scenario 1 (*X%*), the expected NPV develops as sketched in the chart below (see Appendix A for the numbers used in the example).



This chart indicates that it would be wise to just build product *A* if scenario 1 has a probability of 70%. It also indicates that partially preparing can be expected to result in a much higher NPV if scenario 1 would be improbable. Furthermore, it indicates that the NPV of the three architectural scenarios is expected to be more or less the same when the likelihood of scenario 1 would be high.

From this, one would conclude that it would be wise to partially prepare the architecture, since the potential benefit from this can be high, and worst-case, the outcome is almost as high as “just building product *A*”.

When just looking at the total development cost, architectural scenario 3 (fully prepare the architecture) would have been the preferred scenario. When taking the NPV-effect and three strategic scenarios into account, scenario 3 is not the preferred scenario. Although this is a simplified example, the relevance of taking NPV-effects and the probability of strategic scenarios into account is evident.

5 Conclusion

The combination of scenario definition, probability estimations and NPV-calculations offers a framework that identifies the aspects to be considered for evaluating the value of investments during product line roadmapping.

By means of an example, it has been shown that just counting the development cost does not give the same result. The NPV-effect and the effect of considering the probability of strategic scenarios improve the value estimation of product line investments (and hence the quality of the business case for product line development), which is an essential step in product line roadmapping.

References

[1] Sholom Cohen, ***Predicting When Product Line Investments Pays***, The Software Engineering Institute/ Carnegie Mellon University, Technical Note CMU/SEI-2003-TN-017, http://www.sei.cmu.edu/publications/documents/03.reports/03tn017.html

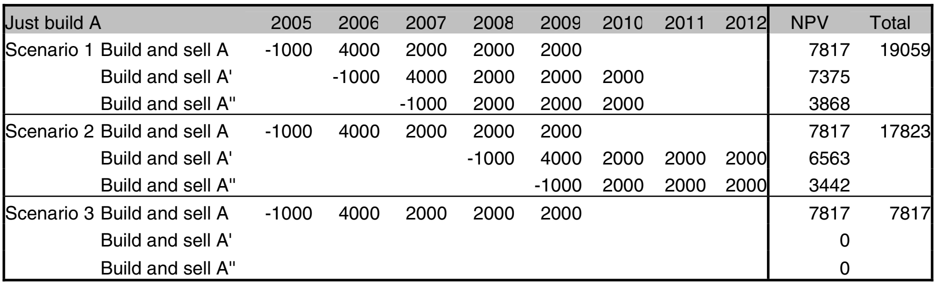
[2] ***Economics of Software Product Lines***, http://www.sei.cmu.edu/productlines/economics_spl.html

[3] Klaus Schmid, ***A Quantitative Model of the Value of Architecture in Product Line Adoption***, in: Frank van der Linden, ed.: PFE-5: Fifth International Workshop on Product Family Engineering, Siena, Italy, pp. 32-43,November 4-6 2003, Springer, LNCS

[4] Pierre America, Dieter Hammer, Murugel T. Ionita, Henk Obbink, Eelco Rommes, ***Scenario-Based Decision Making for Architectural Variability in Product Families***, accepted for SPLC 2004: Third Software Product Line Conference, Boston, MA, USA, August 30-September 2, 2004

Appendix: Numbers Used for Example

For each architectural scenario, the expected cash flow for the three products is given for the three strategic scenarios mentioned in the example. In the column "Total", the total NPV is given. These values are multiplied by the scenario probability as in formula (3)



Just build A		2005	2006	2007	2008	2009	2010	2011	2012	NPV	Total
Scenario 1	Build and sell A	-1000	4000	2000	2000	2000				7817	19059
	Build and sell A'		-1000	4000	2000	2000	2000			7375	
	Build and sell A"			-1000	2000	2000	2000			3868	
Scenario 2	Build and sell A	-1000	4000	2000	2000	2000				7817	17823
	Build and sell A'				-1000	4000	2000	2000	2000	6563	
	Build and sell A"					-1000	2000	2000	2000	3442	
Scenario 3	Build and sell A	-1000	4000	2000	2000	2000				7817	7817
	Build and sell A'									0	
	Build and sell A"									0	

Partially prepare for product A' and A"											
Scenario 1	Build and sell A	-1000	4000	2000	2000	2000				7817	19059
	Build and sell A'		-1000	4000	2000	2000	2000			7375	
	Build and sell A"			-1000	2000	2000	2000			3868	
Scenario 2	Build and sell A	-1000	4000	2000	2000	2000				7817	21752
	Build and sell A'		-100	-100	-300	4000	2000	2000	2000	6968	
	Build and sell A"		-100	-100	-300	4000	2000	2000	2000	6968	
Scenario 3	Build and sell A	-1000	4000	2000	2000	2000				7817	7450
	Build and sell A'		-100	-100						-183	
	Build and sell A"		-100	-100						-183	

Fully prepare for product A' and A"											
Scenario 1	Build and sell A	-1000	-600	2000	2000	2000				3477	20019
	Build and sell A'		-50	4000	2000	2000	2000			8271	
	Build and sell A"		-50	4000	2000	2000	2000			8271	
Scenario 2	Build and sell A	-1000	-600	2000	2000	2000				3477	18199
	Build and sell A'				-50	4000	2000	2000	2000	7361	
	Build and sell A"				-50	4000	2000	2000	2000	7361	
Scenario 3	Build and sell A	-1000	-600	2000	2000	2000				3477	3477
	Build and sell A'									0	
	Build and sell A"									0	

Note: The discount rate used = 6% per year.

A Knowledge-Based Perspective for Preparing the Transition to a Software Product Line Approach

Gerardo Matturro[1] and Andrés Silva[2]

[1] Universidad ORT Uruguay, Campus Centro, Cuareim 1451,
11200 Montevideo, Uruguay
gerardo.matturro@universidad.ort.edu.uy
[2] Universidad Politécnica de Madrid, Campus de Montegancedo,
28660 Boadilla del Monte, Madrid, Spain
asilva@fi.upm.es

Abstract. The adoption of a Software Product Line approach implies a series of changes in the way an organization develops software and runs its whole business. This change in the organization's business strategy can lead to knowledge gaps between the knowledge the organization has at present and the knowledge it must have in the future in order to implement its new strategy. In this article we propose to consider the transition to a Product Lines approach as a Knowledge Management problem, and we also introduce a method for identifying and assessing the aforementioned knowledge gaps.

1 Introduction

The adoption of the product line approach for software development involves changes of magnitude not only in the way an organization develops software, but also in many other areas of its business activities. To succeed with Software Product Lines an organization must alter its technical practices, management practices, organizational structure and personnel and business approach [1].

The differences between what an organization is already doing about its business and what it will have to do in the future, commonly called "strategy gap", can lead to a "knowledge gap" between what the organization knows at present and what it must know in the future to implement its new strategy [2]. In preparing its transition to a product line approach, an organization should identify and assess these knowledge gaps, and take some actions in order to acquire or develop the knowledge needed to close them, so this enhanced set of knowledge and skills becomes aligned with its new business strategy. From this point of view, preparing the transition to the product line approach can be considered as a knowledge management problem.

This article is structured as follows. In section 2 a few definitions about "knowledge" are introduced and a relationship between different kinds of knowledge in the field of software engineering and the practice areas of the SEI's framework for product lines is presented. In section 3, we introduce Zack's framework for analysing the relationship between knowledge and business strategy and we show how it can be applied to the specific situation of the transition to the product line approach. In

H. Obbink and K. Pohl (Eds.): SPLC 2005, LNCS 3714, pp. 96 – 101, 2005.

section 4 we present our proposed method to identify and assess the potential knowledge gaps derived from the adoption of the product line approach. Finally, in section 5 we present some topics we consider that require further research.

2 Product Lines Knowledge

In the Knowledge Management literature there is a broad variety of definitions and characterizations about what knowledge is. Probst et al. proposes that knowledge is the whole body of cognition's and skills individuals use to solve problems [3]. Bollinger and Smith define knowledge as the understanding, awareness or familiarity acquired through study, investigation, observation or experience over the course of the time [4]. Knowledge can be classified into different categories and according to different criteria [2], [5]. In the field of software engineering, Rus et al. establish that depending on the set of activities in software engineering to which knowledge pertains, there can be different kinds of knowledge, such as: organizational, managerial, technical and domain knowledge [6]. These kinds of knowledge can be put in correspondence with the three categories of practice areas defined in the SEI's Framework for Software Product Lines Practices [7], as shown in Table 1:

Table 1. Kinds of knowledge related with the three categories of product lines practice areas

	Software Engineering	Technical Management	Organizational Management
Organizational Knowledge			X
Managerial Knowledge		X	
Technical Knowledge	X		
Domain Knowledge	X		

Thus, the 29 practices areas of the SEI's Framework can be seen as a refinement of those four classes of knowledge, in the sense that they provide a more detailed approximation about what type of knowledge we refer to when we talk about "managerial knowledge" or about "domain knowledge". These practice areas will guide the three main activities our method is structured on: defining the knowledge the organization must have to implement its product line strategy, establishing the knowledge the organization already has, and identifying the knowledge gaps.

3 Aligning Knowledge with a Product Line Strategy

When an organization defines or redefines its business strategy, it will need a set of knowledge and skills that enable the organization to put in practice its new strategy. The strategic choice an organization makes regarding technology, markets, product, services and processes has a direct impact on the knowledge, skills and competences that it needs to compete in its intended markets [8]. As a consequence, those knowledge, skills and competences became strategic as they are necessary for the organization to develop its intended strategy and to deploy it at the operational level.

Zack has presented a framework for analysing the relation between knowledge and business strategy [2]. According to Zack, the gap between what an organization must do to compete and what it actually is doing represents a strategic gap. At the same time, underlying an organization's strategic gap there is a potential knowledge gap. That is, given a gap between what an organization must do and what it can do, there may also be a gap between what the organization must know and what it actually knows. Then, after defining its new business strategy and having performed a strategic evaluation of its knowledge-based resources and capabilities, an organization can determine which knowledge should be developed or acquired.

The adoption of the product line approach is a special case of strategic change because it is not just a different technical way to develop software but also a different way of running the whole business, and involves changes in many other areas of its business activities. We can, then, apply Zack's framework to this situation, as depicted in Figure 1:

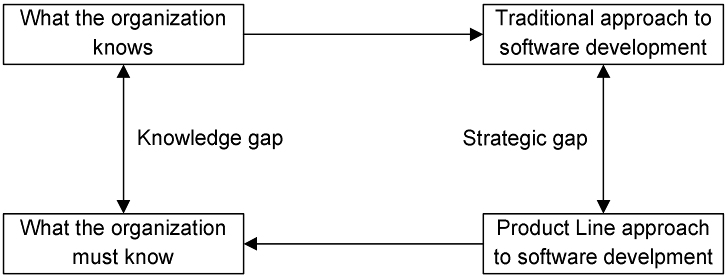

What the organization knows
Traditional approach to software development
Knowledge gap
Strategic gap
What the organization must know
Product Line approach to software develpment

Fig. 1. Knowledge gaps derived from the adoption of the product line approach

To identify the knowledge gaps we will base our method on the 29 practice areas of the SEI's Framework for Software Product Lines Practices. Given any of these 29 practices areas, identifying the knowledge gaps for that area means to identify what the organization knows and what it must know in relation to that practice area.

4 Product Lines Knowledge Assessment

For each practice area, there exist a set of concepts, methods and tools that represent the knowledge the practitioners use when they perform the different activities related to that practice area. This knowledge can be classified in the following three categories: **General**: knowledge and practices that are considered generally accepted in relation to that practice area, **Particular**: knowledge and practices developed and applied by the proper organization and that are variations or "customizations" of the ones included in the previous category, and **Specific**: knowledge and practices that are specific to the product line that the organization is going to start.

Following the definition of knowledge given by Bollinger and Smith [4], to define what to assess we will focus our attention on two elements that are: formal training and study, and working experience. To assess the breadth and depth of the potential knowledge gaps we propose the following four-point scales (based on a scale presented by Mayo [9]) to rate formal Training & Study and Working Experience:

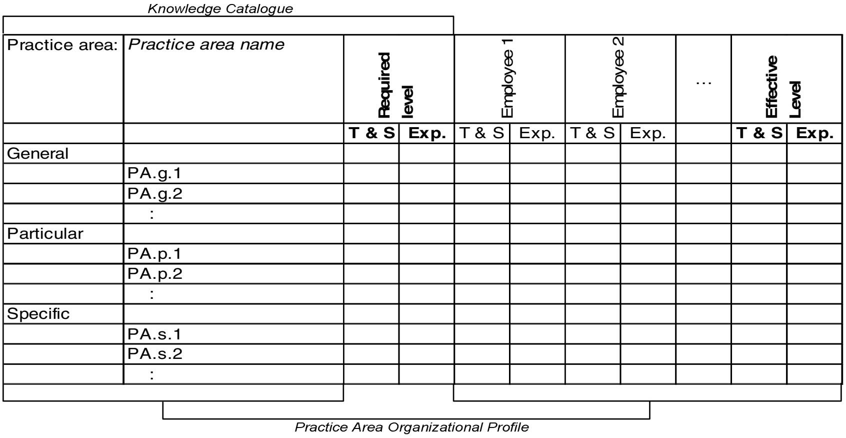
Table 2. Four-point scales to rate Training & Study and Working Experience

Level	Training & Study	Working Experience
1	Has a rudimentary knowledge of the field	Less than 1 year
2	Is able to discuss and work competently	Between 1 and 2 years
3	Is one to whom work colleagues turn to advice	Between 2 and 5 years
4	Is known within the organization for her/his expertise	More than 5 years

4.1 Defining What the Organization Must Know

With the expression "what the organization must know" we mean the knowledge the organization must have and the practices the organization must apply in order to properly initiate and evolve its planned product line.

The template for a practice area Knowledge Catalogue is shown in Figure 2:


Knowledge Catalogue
Practice Area Organizational Profile

Practice area:	*Practice area name*	**Required level**		Employee 1		Employee 2		…	**Effective Level**	
		T & S	**Exp.**	T & S	Exp.	T & S	Exp.		**T & S**	**Exp.**
General										
	PA.g.1									
	PA.g.2									
	:									
Particular										
	PA.p.1									
	PA.p.2									
	:									
Specific										
	PA.s.1									
	PA.s.2									
	:									

Fig. 2. Template for the Knowledge Catalogue and Practice Areas Organizational Profile

The PA.x.y represents knowledge elements such as concepts, methods and techniques for that practice area that the organization considers it will need to initiate its product line. The columns T&S (Training and Study) and Exp. (Experience) under the **Required Level** heading are the places where the expected required levels of knowledge for each knowledge element are initially set. These initial levels can be adjusted later, as the product line evolves and more insight is gained about it.

4.2 Establishing What the Organization Knows

With the expression "what the organization knows" we mean the knowledge the organization has and the practices the organization applies in its current way of developing software and running its business.

To identify this knowledge, we propose to take a bottom-up approach (from the individual to the organizational level) to build an inventory of the knowledge and experience the employees have, taking into account the knowledge elements included in the Knowledge Catalogue. To build this inventory, two steps must be followed:

1. Construction of the General Personal Profile of each employee
2. Construction of the Practice Areas Organizational Profile

4.2.1 The General Personal Profile

The General Personal Profile is a record of the Training & Study events (courses, seminaries, etc.) and of the Working Experience events (roles in a project, position in a company) an employee has taken part. To gather this information, a set of forms based on the template shown in Figure 3 can be given to each employee. For each event, the practice areas it applies are recorded by marking in the corresponding cell.

General Personal Profile	Product line practice areas																														
Name: *employee's name*	Archtecture Definition	Architecture Evaluation	Component Development	COTS Utilization	Mining Existing Assets	Requirements Engineering	Software System Integration	Testing	Understanding Relevant Domains		Configuration Management	Data Collection, Metrics and Tracking	Make/Buy/Mine/Commission Analysis	Process Definition	Scoping	Technical Planning	Technical Risk Maangement	Tool Support		Building a Business Case	Customer Interface Management	Developing an Acquisition Strategy	Funding	Launching and Institutionalizing	Market Analysis	Operations	Organizational Planning	Organizational Risk Management	Structuring the Organization	Technology Forecasting	Training
Training and Study events																															
Course 1																															
Course 2																															
:																															
Working Experience events																															
Project 1																															
Project 2																															
:																															

Fig. 3. Template for the General Personal profile

4.2.2 Construction of the Practice Areas Organizational Profile

The information contained in the employees' General Personal Profiles is the base for building the Practice Areas Organizational profile. These profiles are a more detailed view of the previous ones, taking into account the knowledge elements defined in the Knowledge Catalogue. The corresponding template is also shown in Fig. 2.

Given a practice area, only the employees that recorded a Training & Study event or an Experience event in his/her General Personal profile must be included. For each employee, the corresponding cells at the intersection of each knowledge element are the places where his/her knowledge levels are set. To make a better judgment about these levels, detailed information can be gathered by conducting an interview with each the employee, in which the interviewee explains the characteristics of the training events or the specific task he/she was assigned in the previous projects.

The column headed **Effective Level** is where the overall level for the practice area is established.

4.3 Identifying the Knowledge Gaps

The identification of the knowledge gaps for each practice area is made by comparing the Required levels defined in the Knowledge Catalogue and the Effective levels established in the Practice Areas Organizational profile. By making this comparison, both for Training & Study and for Experience, the knowledge elements for which the effective level is lower than the required level correspond to the knowledge gaps we are looking for.

5 Further work

We are already working on the following subjects that, from the previous exposition, we consider deserve further analysis and research.

When an organization decides to adopt the product line approach, it does it with specific business goals in mind [1]. The question here is what influences these business goals can have in the required levels that are initially set for each knowledge element included in the Knowledge Catalogue. Along with this, a more accurate way to define those expected required levels will lead to a more accurate assessment of the knowledge gaps found, which is the main goal of the proposed method.

The second topic we want to consider here refers to other forms of knowledge an organization usually have such as the knowledge embedded in documents or repositories as well as in organizational routines, processes, practices and norms [10]. These processes, practices and routines will also be affected by the adoption of the product line approach and for them, the corresponding knowledge gaps also need to be identified, assessed and resolved.

References

1. Northrop, L.: SEI's Software Product Line Tenets. IEEE Software 4 (2002) 32–40
2. Zack, M.: Developing a Knowledge Strategy. California Management Review 3 (1999) 125–145
3. Probst, G., Raub, S., Romhardt, K: Managing Knowledge, Chichester, J. Wiley & Sons Ltd. (2000)
4. Bollinger, A., Smith, R.: Managing Organizational Knowledge as a Strategic Asset. Journal of Knowledge Management 1 (2001) 8–16
5. Wiig, K.: Knowledge Management Methods. Arlington, Schema Press (1995)
6. Rus, I., Lindvall, M., Sinha, S.: Knowledge Management in Software Engineering. Technical Report, New York, DoD Data Analysis Center for Software (2001)
7. Clements, P., Northrop, L.: Software Product Lines. Practices and Patterns. Boston, Addison-Wesley (2002)
8. Tiwana, A.: The Knowledge Management Toolkit. Upper Saddle River, Prentice Hall (2002)
9. Mayo, A.: The Human Value of the Enterprise. London, Nicholas Brealey Publishing (2001)
10. Davenport, T., Prusak, L.: Working Knowledge. How Organizations Manage what they Know. Boston, Harvard Business School Press (1998)

Comparison of System Family Modeling Approaches

Øystein Haugen[1], Birger Møller-Pedersen[1], and Jon Oldevik[2]

[1]Department of Informatics, University of Oslo, Norway
{oysteinh, birger}@ifi.uio.no
[2]Sintef ICT, Oslo, Norway
jon.oldevik@sintef.no

Abstract. A reference model for the comparison of system family modeling approaches is presented. Three main approaches to system family modeling are illustrated with a simple example and compared relative to the reference model.

1 Introduction

Many of the challenges of system family engineering are organizational [1], and for these organizational issues, comparisons of approaches seem to be based upon an agreed set of criteria (or framework/reference model) for system family engineering processes.

When it comes to the *modeling* of system families, there is no such agreed comparison framework – no established reference model for comparing and evaluating approaches. In [2] a taxonomy for software product lines is presented, but it is not used to compare different approaches.

This paper presents a reference model for comparing system family modeling approaches, applies it to three broad categories of approaches, and compares the different approaches using the reference model.

2 Reference Model

The reference model makes a distinction between the *generic* sphere and the *specific* sphere. In the generic sphere we have Feature Models and System Family/Product Line models, and within the specific sphere we have Feature selection and System/Product models (Fig. 1).

A Feature Model is a model of the potential features of Systems within the System Family, and it will typically be expressed in terms of possible feature selections. The System Family model is a model corresponding to the potential features. The Production is a process that produces a specific System model from a System Family model and a selection of features from the Feature Model.

The reference model is restricted to cover only *models* and the *process* that produces *models*, i.e. the reference model does not cover variations controlled at execution time.

Note that if we had system family engineering with just programming (i.e. no modeling), we would have the situation illustrated in Fig. 2.

H. Obbink and K. Pohl (Eds.): SPLC 2005, LNCS 3714, pp. 102 – 112, 2005.

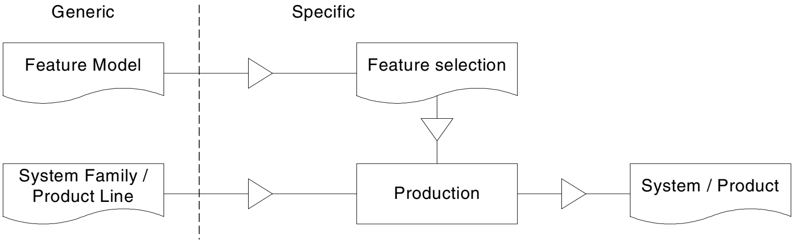


Fig. 1. Separation of spheres in model-driven system family engineering

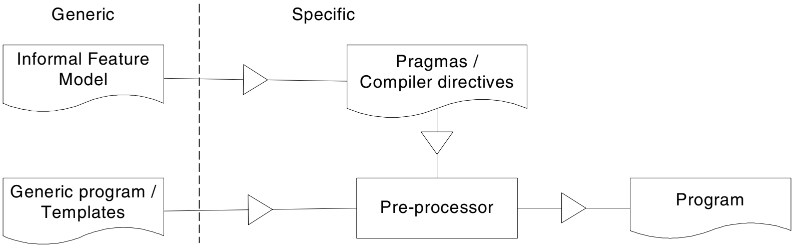


Fig. 2. Separation of spheres in programming-oriented system family engineering

This is close to how generative programming [3] is illustrated. The Production is a generator (or pre-processor) and the feature selection takes the syntactic form of pragmas (compiler directives).

3 The Approaches

We identify three approaches to the modeling of system families. The main defining distinction between the approaches is the kind of language used to model the system family. System families may be modeled by some standard, general language using generic mechanisms of that language. Alternatively the variabilities of a system family may be modeled through annotations to a general language and resolved at system production time. Finally a system family may be modeled by a dedicated domain-specific language. The three different approaches are ideal types, and we shall see that there may be approaches that do not fit exactly these three categories. A pragmatic approach will sometimes use a combination of these three categories.

3.1 Using Standard Languages: Framework / Configuration

This is a category of approaches where domain concepts are represented by predefined components/classes in a standard (modeling) language, and System Families are modeled by frameworks and composition of predefined components with well-defined interfaces [4]. System models are obtained by specializing and configuring a framework, composing specialized components and binding generic type parameters.

As shown in Fig. 3, a Requirements Model represents the Feature Model, and the Feature Selection is expressed through Requirements Resolution. The Framework / configuration approach will make a generic System Family model that seeks to cover the Requirements Model. The process of specialization/ composition/ configuration in Fig. 3 is not an automatic process driven by the requirements resolution, but rather a modeling process.

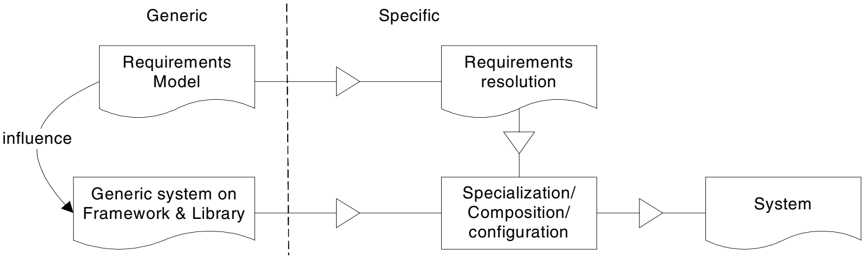

Generic
Specific
Requirements Model
Requirements resolution
influence
Generic system on Framework & Library
Specialization/ Composition/ configuration
System

Fig. 3. Framework / configuration approach

The Framework / configuration approach models variations only by means of existing mechanisms in the modeling language, such as composition, specialization, and (template/generic) parameters.

3.2 Using Annotations: Family-as-the-Union-of-All-Systems

This approach is characterized by having a System Family model containing model elements representing variability often depicted by annotations to model elements of a base language. In UML such annotations are called “stereotypes”. Feature Models are sometimes called Decision Models, and Feature Selection is called Resolution Model. There is a tight relation between the variability model elements and elements of a Decision Model. The Decision Model is used as the basis for the feature selection in the form of a Resolution Model where the decisions described by the Decision Model have been resolved. The approach is illustrated in Fig. 4.

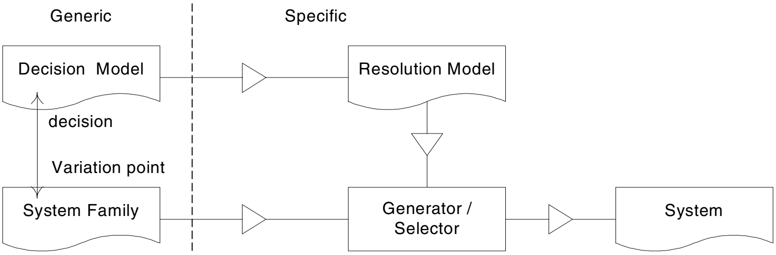


Fig. 4. Family-as-the-union-of-all-systems

A System Family model is a model that is the *union* of all potential System models, marking some elements of this model as variable model elements. The specific system models are *generated*, i.e. there is no modeling involved in producing the system models. The system family model is not executable, but the generated systems may be executable.

Note also that Decision Models may be seen as separate from Feature Models. As the term “decision” implies some kind of process, it is possible to view Decision models as the combination of feature models (defining requirements) and strategies (recipes for how to reach resolutions).

This approach has explicit variability as part of the Family model, and the Family model has model elements that are mapped to elements of the Feature Model. While the Framework / configuration approach requires that one consults the Family model and potential specializations, components or parameters in order to get a picture of all the variations, the Family-as-the-union-of-all-systems approach will produce Family models where the variations can be seen by inspecting one model.

3.3 Using Special Domain Specific Languages

The definition and use of Domain Specific Languages (DSLs) have been proposed as the solution to product line modeling [5, 6]. While general modeling languages (and programming languages) represent domain concepts by means of libraries of classes/components, DSLs represent these as language constructs. There is thus really no System Family model, but the DSL gives the potential of making models that are guaranteed to adhere to restrictions that are wise to have in a domain (Fig. 5).

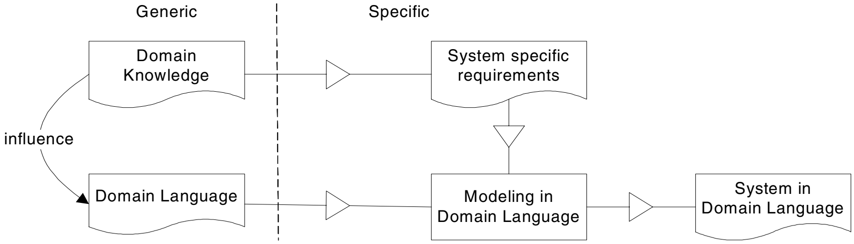

Generic
Specific
Domain Knowledge
System specific requirements
influence
Domain Language
Modeling in Domain Language
System in Domain Language

Fig. 5. Domain specific language approach

The primary input to the DSL is the domain knowledge, and not a specified feature model. A System Family is thereby the set of all systems that may be modeled with this language. This set will potentially be a larger set than the union of all system models (see 0). In this approach Production amounts to Modeling in the DSL, i.e. it is not automatic, but pure modeling.

This approach has no special means for modeling variation, except for the capabilities built into the DSL. Variation point elements are not modeled, but potential features are represented by the possibilities (and constraints) of the DSL.

4 The Comparison

For our comparison we use our reference model (Fig. 1) as guide. We start with the Feature selection and assess how commonalities and variabilities are handled in the different approaches. We then turn to the system family and its development and consider how the production of actual systems is performed. Finally, we review how the different approaches handle systems that span more than one domain.

As seen from the description of the three different approaches, they are rather different when it comes to how variations are modeled and how features are represented in the Family models. To make this clearer and to illustrate our analysis we use a toy example of specifying a digital watch with some mandatory components (buttons, display) and one variable part, namely a speaker that may be either a plain speaker or a polyphonic speaker. The example has been used in the Families project [7].

4.1 How Are Variabilities and Commonalities Modeled?

Complimentary to modeling variability is the modeling of commonality. The three different approaches have distinct attitudes toward defining properties common to all systems.

Table 1. How are commonalities and variabilities modeled?

Framework / configuration	**Family-as-the-union-of-all-systems**	**Domain Specific Languages**
The system family model *is* a model of the common properties of all systems. This model is also a valid system model.	Implicitly defined by all the model elements that are mandatory. This model may not be a valid system model.	As there is no system family model, commonalities are defined through the semantics of the DSL constructs
Variability modeled by generic mechanisms of the language	Variability modeled as annotations to a standard base language	Variability is modeled through specific language mechanisms in the created language

In Fig. 6 we have shown the watch model according to the Framework / configuration approach. In figure a) we show the general watch framework. The speaker is given only to be a part typed by an abstract class. Only necessary common properties may be modeled. Figure b) shows the specific specialization that may appear later as part of the modeling of the specific system.

The Framework / configuration approach models the commonalities explicitly as a system with default structure and default behavior. Fig. 6a) gives the structure of all watches; and if one were to make a system according to the Watch family, one would get a watch with a Button, a Display and a Speaker. Provided that e.g. Button is defined to have some default behavior that is executed when pressed, this behavior will be part of such a watch. As specified here, the Speaker will not specify any default behavior, but only a required interface towards the audio port, while the different behavior of speakers are defined in the two subclasses.

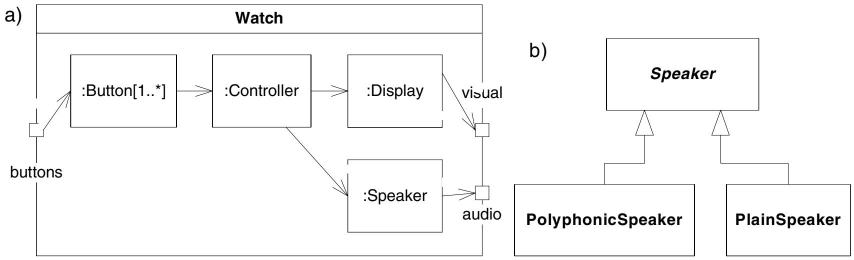


Fig. 6. Watch System family with framework / configuration

In Fig. 7 we have sketched how a small extract of the digital watch may be modeled using the Family-as-a-union-of-all-systems approach. Figure a) shows a UML-like composite structure modeling the generic system family where the choice between a plain speaker and a polyphonic speaker has been made explicit. The feature model shown in figure b) depicts the generic decision model in a notation given in [3] indicating that one choice exactly out of the given alternatives shall be chosen.

In Fig. 8 we have shown how the watch structure could have been modeled in a DSL. The DSL itself is not shown in the figure, but it would contain a palette of concepts and their interrelationships. For a more elaborate description of a watch in a DSL, see Pohjonen and Kelly [8].

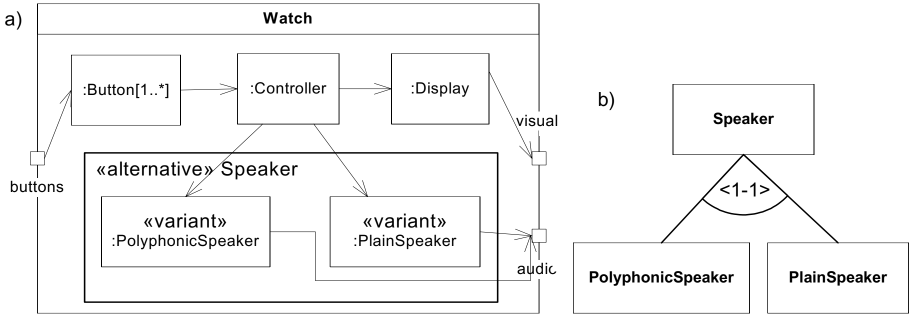


Fig. 7. Digital watch with Family-as-a-union-of-all-systems

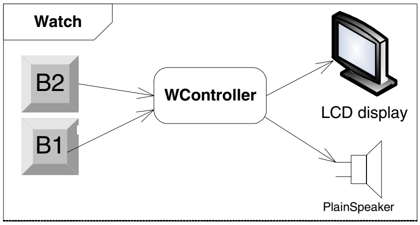


Fig. 8. Watch modeled in a DSL

4.2 Support for Iterative and Incremental System Family Development?

In order for an approach to support iterative and incremental development, it shall be possible to *analyze* (formally, testing, reviewing, etc) system family models, have *partial* system (product) models, and to handle *unforeseen* requirements/features.

4.2.1 Can the System Family Model Be Analyzed?

By being analyzed we mean that it is possible to establish certain properties of the system family that will prevail in all the systems derived from it.

If the system family model cannot be analyzed, analysis has to be repeated for each specific system model. In analysis we include all kinds of techniques that establish some properties, both formal analyses, reviewing, and testing.

Table 2. Can the system family model be analyzed?

Framework / configuration	Family-as-the-union-of-all-systems	Domain Specific Languages
Yes: the family model is a model of a general system with default structure and behavior specified and can as such be analyzed, e.g. by executing the model. Specialization and binding of parameters may ensure that properties of the family are preserved	No, the family model with all variations and all model annotations included cannot be given an execution semantics and therefore cannot be analyzed	No, as there is no system family model, it cannot be analyzed

The two extremes here are the Family-as-the-union-of-all-systems approach and the Framework / configuration approach. The illustrative model in the Framework / configuration approach is in Fig. 6a). The Speaker class defines all the common properties of all speakers and the interfaces to the rest of the watch architecture. The Speaker may either be abstract, in which case analyzing the family model will simply check that interfaces match the rest of the architectures, or it may have some minimal behavior, in which case the effect of that behavior can be analyzed, too.

In the Family-as-the-union-of-all-systems approach, illustrated in Fig. 7, the total family model with all possible variations is modeled in one system familymodel. The resulting model contains information covering more than one system. The system family model can, therefore, not be analyzed using means of analysis used in single system development. Rather, additional analyses are necessary to analyze system family models that take into account the generic nature of the models. Another option is to first generate the specific system models, and then use single-system analyses on the resulting system model. This is similar to traditional macro-expansion and other generic descriptions that are in principle not meaningful until the generics have been bound.

Although the DSL approach does not have the notion of family model so that it cannot be analyzed, it may still benefit from analysis performed on the DSL and its implementation. This is in fact one of the arguments in favor of DSLs: domain experts and language implementation experts together guarantee that users of the DSL get the best implementation of the right concepts.

4.2.2 Are Partial System Family Models Supported?

Partial system models are representations of a system family with a smaller scope than the original system family model, i.e. the space of variability and thus the space of possible different kinds of systems is narrowed. This is very useful in order to specify categories of systems that are more specific than the complete system family yet more specific than a single system.

Table 3. Are partial system family models supported?

Framework / configuration	**Family-as-the-union-of-all-systems**	**Domain Specific Languages**
Yes. Partial system models are specified as specializations and extensions of the system family model. There can be arbitrary levels of specializations. There is low risk of model inconsistencies.	Yes. A partial system model can be given as a new system family model (a copy), in which some variabilities have been resolved, thus defining a more limited space of systems. There is a risk of model inconsistencies.	No. Since there is no system family model, partial models cannot be specified as such. A similar effect can, however, be achieved by constraining the domain-specific language for the specific case.

Partial system models can be supported both in the Framework / configuration and the Family-as-the-union-of-all-systems approach. In the latter approach, however, there is no established way of handling refinements of system family models, the risk being inconsistent model refinements. In the former approach, traditional mechanisms for specialization and extensions are used, facilitating consistent model refinements.

4.2.3 How Are Unforeseen Features Handled (Maintenance, Evolution)?

Maintenance and evolution are important parts of systems' development. For system families this means that it is important to handle unforeseen features.

Table 4. How are unforeseen features handled (maintenance, evolution)?

Framework / configuration	**Family-as-the-union-of-all-systems**	**Domain Specific Languages**
The unforeseen features are just added to the model, by specialization or composition	The system family model has to be changed, or one shall allow additions to the automatic generated system model	If they can be expressed in the DSL, express them there. If not expressible, make a new language.

Note that the feature model has to be changed in all three approaches.

Unforeseen features come in two variants: features that belong to the family and features that are required for a specific system. The Framework / configuration approach allows adding properties for specific systems, while the Family-as-the-union-of-all-systems approach treats these in the same way: as family features. As indicated above, it is with the Family-as-the-union-of-all-systems approach possible to add properties after the system model has been generated, but it is not wise. The Framework / configuration approach may choose to let the unforeseen properties become properties of a new (specialized) family model, instead of just of a specific system model.

The need for making a new domain specific language for the purpose of supporting new features reveals some challenges. Can new constructs be added without corrupting existing constructs (are they orthogonal or are there any dependencies)? Can a DSL be defined as a specialization of another (inheriting the semantics of the super language and adding what is needed for the new features)?

4.3 The Production of Individual Systems

Does making specific systems involve the professional skills of modeling or simply taking decisions based upon the feature model and then let the system model be generated? A follow-up question would be whether there is a way after the generation where more model elements can be added?

Associated with the production of the individual systems is also another question: What kind of code generator may be used on the resulting system model?

As indicated in the table above, two of the approaches are similar when it comes to how they obtain specific system models. Both the Framework / configuration approach and the DSL approach model the specific systems: The Framework / configuration approach models them by specializing frameworks, composing components and/or applying actual parameters to parameterized models, while with DSLs they are in principle modeled from scratch, although this approach may also use predefined components.

Along the same distinction, relying on generation of specific system models (as with the Family-as-the-union-of-all-systems approach), one should rather not to this generated system model *add* model elements for these specific systems, as this will cause problems if repeated generations are required: The generated system models should not be touched. The other two approaches have a different approach: In principle, a DSL model is a specific system model (i.e. all model elements are added),

Table 5. Are individual systems modeled or are they generated?

Framework / configuration	Family-as-the-union-of-all-systems	Domain Specific Languages
Modeled in the standard language, using pre-defined domain specific elements and framework.	*Generated* from applying a decision model to the system family model.	*Modeled* in the domain language
Model elements *may be added*	More model elements should *not* be added.	Model elements *may be added.*
From system model, *standard* code generator may be used.	From the system model, *standard* code generator may be used.	From the system model, a *tailored* code generator is used.

while specialization and composition encourage the adding of specific model elements (specializations inherit the general family properties and may add properties, composition is often more than just the composition of components).

When one has obtained the System model, the Family-as-the-union-of-all-systems and the Framework / configuration approaches are similar in that they can use standard code generators. Models in the Family-as-the-union-of-all-systems approach will often be made in some standard modeling languages with annotations for variation model elements (e.g. stereotypes in UML). In the system models these annotations are gone, and therefore standard code generators may be used.

4.4 Systems Spanning More Than One Domain

System family engineering is often equated with domain engineering, and most often a system family belongs to one domain. We are here not considering user interfaces and interfaces to some underlying data repositories as separate domains (could be called implementation domains), so in order for a system family to span more than one domain, the main application model has to be based upon concepts from more than one 'real' domain.

The reason for asking this question is that conventional systems often span more than one domain, so the answer to this question will tell how easy it is for the approach to cater for large classes of systems.

Table 6. What about systems spanning more than one domain?

Framework / configuration	**Family-as-the-union-of-all-systems**	**Domain Specific Languages**
Use classes/components from different libraries/frameworks	Use classes/components from different libraries/frameworks	Contrary to the idea of DSLs

For the DSL approach this is of course only a problem if there are existing DSLs for the different domains. It will be a (costly) solution to define a DSL from scratch from each combination of domains.

5 Summary and Conclusions

A reference model for the comparison of system family/product line modeling has been presented. The application of this reference model to three main approaches has revealed that these approaches have different qualities regarding essential issues in systems modeling.

Some of the most distinct differences are: the DSL approach is the only approach that does not even have a System Family Model (but rather a language that allows potentially many systems to be modeled); the Framework / configuration approach is the only approach that allows for analysis of Family Models; the Family-as-union-of-all-systems is the only approach in which all the variations are present in the system family model.

Some of the similarities are: The Framework / configuration and Family-as-the-union-of-all-systems approaches can benefit from standard modeling languages and tools (e.g. code generators), while the DSL approach has to make specific tools for each language.

A concrete system family may apply more than one ideal approach. A system family made with annotations may also benefit from generic and component mechanisms of the base language, and thereby apply the framework/ configuration approach. A DSL may include generic and component mechanisms, and even annotations to be resolved by a preprocessing phase may be added to such a language. It is reasonable, however, that there is one dominant approach in a given concrete system family.

References

1. Clements, P. and L. Northrop, *Software Product Lines: Practices and Patterns*. 2002: Addison Wesley Professional. 608.
2. Krueger, C.W. *Towards a Taxonomy for Software Product Lines*. in *5th International Workshop, PFE 2003,*. 2003. Siena, Italy: Springer Lecture Notes in Computer Science LNCS 3014 (2004) 3-540-21941-2.
3. Czarnecki, K. and U. Eisenecker, *Generative Programming: Methods, Tools, and Applications*. 2000: Addison-Wesley Professional. 864.
4. D'Souza, D.F. and A.C. Wills, *Objects, Components and Frameworks with UML: The Catalysis Approach.* 1998: Addison-Wesley. 785.
5. Greenfield, J. and K. Short, *Software Factories*. 2004, Indianapolis: Wiley. 666.
6. Gray, J., M. Rossi, and J.-P. Tolvanen, eds. *Domain-Specific Modeling with Visual Languages*. Journal of Visual Languages & Computing. Vol. 15, Issues 3-4. 2004, Elsevier.
7. Families, *Families*. 2004, http://www.esi.es/en/Projects/Families/. p. Eureka Σ! 2023 Programme, ITEA project ip02009.
8. Pohjonen, R. and S. Kelly, *Improving productivity and time to market*, in *Dr. Dobb's Journal*. 2002.

Cost Estimation for Product Line Engineering Using COTS Components

Sana Ben Abdallah Ben Lamine[1], Lamia Labed Jilani[2],
and Henda Hajjami Ben Ghezala[3]

Laboratoire Riadi-Gdl, Ecole Nationale des Sciences de l'Informatique,
Campus Universitaire la Manouba, La Manouba, 2010, Tunisia
[1] Sana.Benabdallah@riadi.rnu.tn, [2] Lamia.Labed@isg.rnu.tn
[3] Henda.BG@cck.rnu.tn

Abstract. Economic models for reuse are very important to organizations aiming to develop software with large scale reuse approaches. In fact, the initial investment is so important that it can discourage managers to commit to those approaches. Thus, economic models can help them to assess the worthiness of such an investment.

Product Line Engineering (PLE) seems to be an attractive reuse approach in matter of product quality and time-to-market. Using Commercial Off The Shelf (COTS) in a PLE approach may have a positive impact.

This paper reports on the need for an economic model to quantify the predicted benefits of the PLE software development with the use of COTS components. We introduce a Model for Software Cost Estimation in a Product Line Engineering approach that we denote SoCoEMo-PLE 2. This latter includes the usage of COTS components. The potential benefits of the model are described.

1 Introduction

Many economic models are achieved to quantify software development costs in reuse. The most known models for reuse are studied in [1, 2, 3, and 4]. The measurement of the Return On Investment (ROI) of a project using a Product Line Engineering (PLE) approach is given in [5]. Since PLE seems to be very attractive in matter of quality and time to market, and since very few models deal with PLE, we are particularly interested in economic models that help to estimate the pretended benefits achieved by the adoption of a software development approach using product lines.

In our previous work on cost estimation models for PLE [6, and 7], we based our research on two models: the integrated cost estimation model for reuse, presented in [3, and 4], and Poulin's economic model for PLE, presented in [5]. We chose specifically those models because the first is proved to be a generic model for reuse and seems to be applicable to PLE after some adaptations, while the second proposes a global formula that estimates the ROI of a project using PLE. We obtained the SoCoEMo-PLE model which is a cost estimation model for a product line engineering approach, where all the reusable components are supposed to be developed internally in the corporation.

H. Obbink and K. Pohl (Eds.): SPLC 2005, LNCS 3714, pp. 113 – 123, 2005.

In the present work, there are two fundamental assumptions. The first one is that a reuse organization contains four engineering cycles that feed costs and benefits into each other: the component, the domain, the application, and the corporate engineering cycles. The second one is that components of the product line can include COTS components.

The contribution of this paper is a model for software cost estimation in a PLE software development approach that uses COTS components.

In the next section we provide a concise background on economic models for reuse that are the basis of this work. The following two sections define the model SoCoEMo-PLE 2, and give a synthesis and potential benefits respectively. We conclude with a look at how this model can be bettered and even extended to the PLE development that uses PLE.

2 Background: Software Cost Estimation in Reuse

2.1 The Integrated Cost Estimation Model for Reuse

The integrated cost estimation model for software reuse, presented in [3, and 4] is characterized by:

Variety of Investment Cycles. The model defines four distinct investment cycles and identifies how they feed into each other: The corporate engineering cycle, the domain engineering cycle, the component engineering cycle and the application engineering cycle. In fact, there are four parties in the software reuse process: the corporate management, which has a stake in seeing the reuse program reap benefits for the corporation; the domain engineering team, which has a stake in seeing its domain engineering products reused; the application engineering teams, which has a stake in producing applications with low cost, high quality, and short time to market; and the component developers, who have a stake in seeing their components reused widely.

Variety of cost factors. The cost factors used to define the various economic functions are quantified for each investment cycle and they include:

- Investment Cycle (Y), in years.
- Start Date of the investment (SD).
- Discount Rate (d), which is an abstract quantity that reflects the time value of money.
- Investment Cost (IC), in person months (PM).
- Periodic Benefits (B(y)), at year y, for SD+1≤y≤ SD+Y, in PM.
- Periodic Costs (C (y)), at year y, for SD+1≤y≤ SD+Y, in PM.

Variety of Economic Functions. The economic functions used to assess the worthiness of the investment after the estimation of cost factors are: Net Present Value (NPV), Return on Investment (ROI), Profitability Index (PI), Average Rate of Return (ARR), Average Return on Book Value (ARBV), Internal Rate of Return (IRR), and Payback Value (PB).

Variety of Viewpoints. The model analyzes the cost factors for each stakeholder (corporate managers, domain engineering teams, application engineering teams, and producers of reusable assets).

2.2 Poulin's Model for PLE

Poulin, in [5], defines the ROI of a PLE project by the formula:

$$ROI = \sum_{i=1}^{n} RCAi - ADC \,. \tag{1}$$

RCA is the Reuse Cost Avoidance by the reuse of components *i* in the project:

$$RCA = DCA + SCA \,. \tag{2}$$

DCA is the Development Cost Avoidance by the reuse of a component:

$$DCA = RSI \times (1 - RCR) \times (NewCodeCost) \,. \tag{3}$$

RSI is Reused Source Instructions.

RCR is the Relative Cost of Reuse. It represents the ratio of the effort that it takes to reuse software without modification to the cost incurred to develop it to use once.

SCA is Service Cost Avoidance by the reuse of a component:

$$SCA = RSI \times (errorRate) \times (errorCost) \,. \tag{4}$$

ADC is the Additional Development Costs assumed by the reuse:

$$ADC = (RCWR - 1)(CodeWrittenFor\,\mathrm{Re}\,useByOthers)(NewCodeCost) \,. \tag{5}$$

RCWR is the Relative Cost of Writing for Reuse. It represents the ratio of the effort that it takes to develop reusable software to the cost of writing it to use once.

2.3 SoCoEMo-PLE

SoCoEMo-PLE is a Software Cost Estimation Model for PLE detailed in [6, and 7] and based on the strong features of both the integrated cost estimation model for reuse and Poulin's model for PLE. SoCoEMo-PLE uses the two pre-cited models as basis and tries to palliate their insufficiency regarding to the PLE development. In fact, the integrated cost estimation model for reuse considers reuse in general. It doesn't consider specifically the PLE development life cycle. Poulin's economic model for PLE is a rapid and simple model. It doesn't consider many cost drivers like the discount rate. It proposes a global formula that estimates the ROI of a project using PLE, without detailing costs and benefits for each co-operant in the reuse program.

Notational conventions for SoCoEMo-PLE are γ, δ, α, and ρ which denote respectively component, domain, application and corporate engineering factors. Cost factors used are Y, d, SD, IC, C(y), and B(y). Economic functions used are NPV, ROI, PI, ARBV, and PB.SoCoEMo-PLE supposes that the reusable components of the product line are developed internally in the corporation (in the component engineering cycle).

3 SoCoEMo-PLE 2

The SoCoEMo-PLE 2 model is an extension of the SoCoEMo-PLE model. In fact, SoCoEMo-PLE has a main assumption that the reusable components of the product line are developed internally in the corporation. But the SoCoEMo-PLE 2 model considers that COTS components can be bought. Thus, the cost cascade between the cycles changes and obviously new cost components appear in the equations of the previous model (SoCoEMo-PLE) to show the costs (and the benefits) incurred by the use of COTS components. In this section we detail the SoCoEMo-PLE 2 model's equations for four engineering cycles. For each cycle, estimations are done for three cost factors: IC, C(y), and B(y), since Y, d, and SD are uniform within a corporation.

Notational conventions, cost factors, and economic functions are the same used in SoCoEMo-PLE.

In this work, we adopt the definition of a COTS component which is given in [8]. A COTS product is an executable software product that has the following characteristics:

- It is sold, leased, or licensed to the general public.
- Buyers, lessees, and licensees have no access to the source code; hence can only use the product as a black box.
- It is offered by a vendor who has created it and is typically responsible for its maintenance and its upgrades.

It is available in multiple identical copies (within the same version) on the market.

3.1 Component Engineering Cycle

Investment Cost. The investment cost of the component engineering cycle is estimated by:

$$IC_{\gamma} \;=\; C_{\gamma}(SD) = ER + LI \quad . \tag{6}$$

LI is the certification and Library Insertion cost. It is determined by expert judgment.

ER is the Estimation of the development cost for Reuse. It is formulated by:

$$ER = E(RCWR)Pay\,. \tag{7}$$

E is the Estimation of the development cost without reuse and to use once. It is estimated by COCOMO in organic mode:

$$E = 3S^{1.12}\,. \tag{8}$$

Pay is the average monthly salary of the developer.

RCWR is the Relative Cost of Writing for Reuse.

Periodic Cost. The periodic cost of a reusable component γ of the product line is estimated in year y by:

$$C_{\gamma}(y) = OC(y)Pay_{l} + MN(y)Pay_{d}\,. \tag{9}$$

OC(y) is the Operating Cost of the library, given by:

$$OC(y) = \frac{TotalOperationnelCostOfLibrary}{ComponentsNumber} . \qquad (10)$$

Payl, and Payd are the average monthly salaries respectively of the librarian and the developer.

MN(y) is the MaiNtenance cost, estimated by COCOMO by:

$$MN(y) = E(ACT) . \qquad (11)$$

ACT is the Annual Change Traffic (the ratio of the yearly maintenance cost to the development cost).

Periodic Benefit. The periodic benefit of a reusable component γ of the product line is estimated in year y by:

$$B_{\gamma}(y) = freq_{BB}(y)BP(y) + freq_{WB}(y)WP(y) . \qquad (12)$$

freqBB(y) and freqWB(y) are respectively the component's black box and white box reuse frequencies in year y. They are determined by existing data or by expert judgment.

BP(y) and WP(y) are respectively Black box and White box Prices of the component, given respectively by:

$$BP = (RBP)E . \qquad (13)$$

$$WP = (RWP)E . \qquad (14)$$

RBP and RWP are respectively Relative Black box and Relative White box Prices, which are determined by expert judgment.

3.2 Domain Engineering Cycle

Investment Cost. The investment cost of the domain engineering cycle is estimated by:

$$IC_{\delta} = PLADC + \sum_{\gamma \in \delta} C_{\gamma}(SD) + \sum_{i=1}^{N_{SD}} CCOTS_{i} . \qquad (15)$$

Cγ(SD) is the investment cost of the component γ.

PLADC is the Product Line Architecture Development Cost which comprises costs relative to the different steps to build a PLA, described by [9]:

- BCAC (Business Case Analysis Cost),
- SC (Scoping Cost),
- PFPC (Product and Feature Planning Cost),
- DPLA (Design of Product Line Architecture Cost),
- CRSC (Component Requirement Specification Cost),
- VC (Validation Cost).

These costs are determined by expert judgment.

$$PLADC = BCAC + SC + PFPC + DPLA + CRSC + VC\,. \tag{16}$$

CCOTS$_i$ is the Cost of buying a COTS components *i* in year SD. We suppose that the cost of a COTS component is less then the cost of the same component developed by the component engineering cycle internally in the corporation. In fact, to [8], using COTS components allows gain in cost, because the product is produced once and used multiple times. Then, it can be sold for an arbitrarily small fraction of its development cost.

N$_{SD}$ is the total number of COTS components bought in year SD.

Periodic Cost. The periodic cost for the domain engineering cycle is estimated by:

$$C_\delta(y) = AEC(y) + \sum_{\gamma\in\delta} C_\gamma(y) + \sum_{i=1}^{Ny} CCOTS_i\,. \tag{17}$$

AEC(y) is the Architecture Evolution Cost in year y. In [9], evolution includes changes to components of the product line, to the relations between them, etc. AEC is determined by expert judgment.

Cγ(y) is the investment cost of the component γ, if y is the year where γ is developed, because Cγ(y=SD)=ICγ. Else if y>SD, then Cγ(y) is the periodic cost of γ in year y.

CCOTS$_i$ is the Cost of buying a COTS component *i* in year y. The same gain in cost thanks to the use of COTS components (see Investment Cost.).

N$_y$ is the total number of COTS components bought in year y.

Periodic Benefit. The periodic benefit of the domain engineering cycle is estimated in year y by:

$$B_\delta(y) = \sum_{\gamma\in\delta} B_\gamma(y) + \sum_{j=1}^{Nsell} CCOTS_j\,. \tag{18}$$

Bγ(y) is periodic benefit of component γ in year y.

CCOTS$_j$ is the Cost of selling a COTS component *j* to the application engineering cycle in year y. We suppose that the domain engineering cycle sells COTS to the application engineering cycle at the same price it bought it.

N$_{sell}$ is the total number of COTS components sold in year y.

3.3 Application Engineering Cycle

Investment Cost. The investment cost of the application engineering cycle is estimated by:

$$IC_\alpha = C_\alpha(SD) = \sum_{i=1}^{NC} PR_i + INCOTS + Glue\,. \tag{19}$$

PR$_i$ is the price of the component *i* used in application α. The component *i* can be developed internally in the component engineering cycle or it can be a COTS component. In the first case, PR$_i$ is estimated by BP$_i$ or WP$_i$, respectively Black box and White box Prices of the component. BP_i and WP_i are determined by expert judgment. In the second case, PR_i is the price of the COTS component.

NC is the total number of components used in application α (components of the product line or COTS components).

INCOTS is the cost of integration of COTS components used in application α. These costs are determined by expert judgment. We suppose that the integration costs of components developed in the component engineering cycle and used in application α are determined by the cost of the glue code needed.

Glue is the cost of glue code developed in the application α in order to integrate the product line components (developed in the component engineering cycle) used in application α. Glue is estimated by COCOMO (see equation (8)).

Periodic Cost. We suppose that the periodic cost of an application is null because it is achieved in a year.

$$C_{\alpha}(y) = 0 \,. \tag{20}$$

Periodic Benefit. The periodic benefit of an application in year SD is estimated using RCA (see equation (2)) to quantify cost economies for an application α in year SD.

$$B\alpha(SD) = \sum_{\gamma \in \alpha} RCA_{\gamma} + \sum_{\cot s \in \alpha} RCA_{\cot s} \,. \tag{21}$$

RCA_{γ} is reuse cost avoided by the use of component γ developed in the component engineering cycle.

RCA_{cots} is reuse cost avoided by the use of a COTS component *cots* in the application α. We suppose that the reuse cost avoided by the use of a COTS component (RCA_{cots}) is greater then the reuse cost avoided by the same component (RCAγ) developed by the component engineering cycle internally in the corporation. In the same way we suppose that $DCA_{cots}>DCA_{\gamma}$ and $SCA_{cots}>SCA_{\gamma}$ (see equations (3) and (4)). In fact:

$DCA_{cots}>DCA_{\gamma}$ because, to [8] the use of COTS components permits gain in cost (the multiple users of a COTS product share the cost of developing the product).

$SCA_{cots}>SCA_{\gamma}$ because, to [8] the use of COTS components permits gain in operational quality because the product is widely used by a broad segment of users, then, it is typically thoroughly tested and debugged, hence it typically has much better quality than any one user can afford. In addition, the use of COTS components permits gain in maintenance overhead, because the multiple users of a COTS product not only share the cost of developing the product, but they also share the cost of its long term operation and maintenance. The vendor is typically responsible for its corrective, perfective, and adaptive maintenance.

For years y after SD, we consider that benefits of an application α in the product line come from cost economies achieved by the use of high quality reuse components, pretended to need less maintenance:

$$B\alpha(y) = \sum_{\gamma \in \alpha} SCA\gamma + \sum_{\cot s \in \alpha} SCA_{\cot s} \,. \tag{22}$$

SCA_{γ} is service cost avoided by the use of component γ developed in the component engineering cycle.

SCA_{cots} is service cost avoided by the use of a COTS component *cots* in the application α. In the same way, $SCA_{cots}>SCA_{\gamma}$.

3.4 Corporate Engineering Cycle

Investment Cost. The investment cost of the corporate engineering cycle is estimated by:

$$IC_{\rho} = INF + C_{\delta}(SD) . \tag{23}$$

INF is the infrastructure cost. It is determined by expert judgment.

$C_{\delta}(SD)$ is the investment cost of the domain δ of the product line.

Periodic Cost. The periodic cost in the corporate engineering cycle is estimated by:

$$C_{\rho}(y) = C_{\delta}(y) . \tag{24}$$

$C\delta(y)$ is the periodic cost of the domain engineering cycle in year y.

Periodic Benefit. The periodic benefit of the corporation is given by:

$$B_{\rho}(y) = \sum_{\alpha \in \rho} B_{\alpha}(y) . \tag{25}$$

$B\alpha(y)$ is the periodic benefit of application α in year y.

4 Synthesis and Potential Benefits

SoCoEMo-PLE 2 estimates costs and benefits of software development with a PLE approach using COTS components for four investment cycles. All of them cooperate to ensure their own interest and also the collective one of the corporation that adopts a PLE approach to develop software. Some costs are determined by expert judgment [10].

Both theory (see sect. 3) and preliminary experiments of the model (we did not present the example here for constraints of space) have shown the potential benefits of the use of COTS components in a PLE development approach since these components permit gains in cost, quality and then maintenance costs.

The main assumption in this work is that the core assets of the product line can include COTS components. Thus, the cost cascade between the four cycles changes with regard to the cost cascade of SoCoEMo-PLE (see fig. 1 and 2).

We notice that the symbol "+" on some costs (or benefits) in figures 1 and 2 indicates that the cost (or benefit) doesn't come only from the cycle indicated by an arrow, but it has also other source(s).

The cost balances of the SoCoEMo-PLE 2 model for the different engineering cycles are given in tables 1 to 4. We emphasize the costs incurred by the use of COTS components (*italic*) to clarify the difference with the SoCoEMo-PLE model.

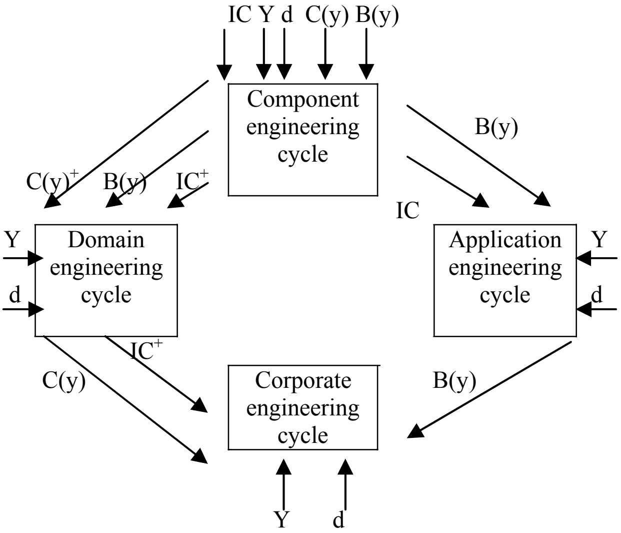

IC Y d C(y) B(y)
Component engineering cycle
B(y)
C(y)+
B(y)
IC+
IC
Y
d
Domain engineering cycle
Application engineering cycle
Y
d
IC+
C(y)
Corporate engineering cycle
B(y)
Y
d

Fig. 1. Cost cascade for SoCoEMo-PLE. The different cycles feed costs and benefits to each other.

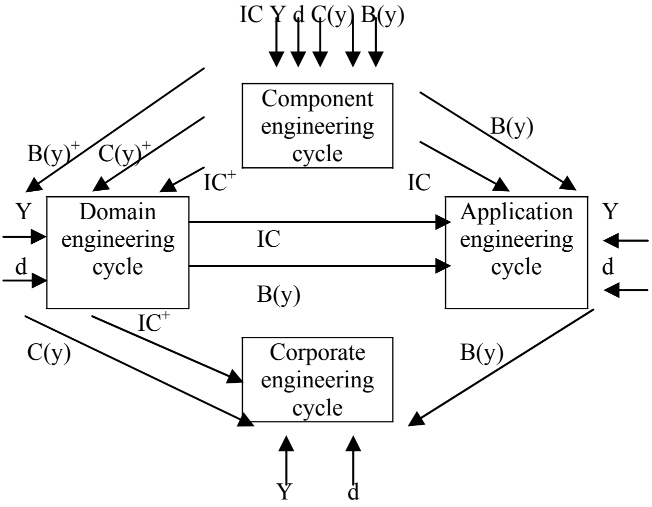

IC Y d C(y) B(y)
Component engineering cycle
B(y)
B(y)+
C(y)+
IC+
IC
Y
d
Domain engineering cycle
IC
B(y)
Application engineering cycle
Y
d
IC+
C(y)
Corporate engineering cycle
B(y)
Y
d

Fig. 2. Cost cascade for SoCoEMo-PLE 2 (see sect. 3)

Table 1. Cost balance for the component engineering cycle

Year y	Cost $C_\gamma(y)$	Benefit $B_\gamma(y)$
y = SD	Cost of development for reuse + Cost of certification and library insertion.	
y > SD	Costs of maintenance and library residence.	Sell of components internally.

Table 2. Cost balance for the domain engineering cycle

Year y	Cost C_δ (y)	Benefit B_δ (y)
y = SD	Cost of development of the PL architecture + Cost of development and residence of components + *Cost of buying COTS components in year SD.*	
y > SD	Cost of evolution of the PL architecture + Cost of development and residence of components + *Cost of buying COTS components in year y.*	Sell of components (*COTS* or developed internally).

Table 3. Cost balance for the application engineering cycle

Year y	Cost C_α (y)	Benefit B_α (y)
y = SD	Cost of buying reusable components (*COTS* or developed internally) + *Cost of integration of COTS* + Glue code cost	Economies on development and maintenance costs through the use of reusable components (*COTS* or developed internally)
y > SD	$C_\alpha(y)=0$	Quality gains (maintenance cost economies) through the use of reusable components (*COTS* or developed internally).

Table 4. Cost balance for the corporate engineering cycle

Year y	Cost $C_\gamma(y)$	Benefit $B_\gamma(y)$
y = SD	Infrastructure cost + Domain cost in year SD.	
y > SD	Domain periodic costs.	Application periodic benefits.

5 Conclusion

In this paper we presented a software cost estimation model for product line engineering using COTS components: SoCoEMo-PLE 2.

This model is based on the calculus of costs and benefits for four investment cycles in a corporation: the component engineering cycle, the domain engineering cycle, the application engineering cycle, and the corporate engineering cycle. It is based on the SoCoEMo-PLE model which doesn't consider the use of COTS components. The model SoCoEMo-PLE 2 has proved potential benefits (compared with results obtained with the SoCoEMo-PLE model) both theoretically and by the application of the model on an example. Our future work focuses on the integration cost of COTS components (which are determined by expert judgment in the current model). Later, we will focus on the costs of PLE use in a PLE development approach.

References

1. Lim, W.: Reuse Economics: A Comparison of Seventeen Models and Directions for Future Research. Proceedings, International Conference on Software Reuse. Orlando FL (1996) 41-50
2. Wiles, E.: Economic Models of Software Reuse: A Survey, Comparison and Partial Validation. Version 2.1, Release, Report Reference: UWA-DCS-99-032, Department of Computer Science, University of Wales, Aberstwyth Ceredigion SY23 3DB U.K. (1999)
3. Chmiel, S. F.: An Integrated Cost Model for Software Reuse. Thesis for the obtention of PhD degree in computer science, Morgantown West Virginia (2000)
4. Mili, A., Chmiel, S.F., Gottumukkala, R., Zhang, L.: Managing Software Reuse Economics: An Integrated ROI-based Model. CSEE Department, West Virginia University Morgantown WV 26506-6109 USA (2000)
5. Poulin, J.: The Economics of Software Product Lines. International Journal of Applied Software Technology (1997)
6. Ben Abdallah Ben Lamine, S. : Modèle d'estimation de coûts pour le développement logiciel basé sur la réutilisation: Cas de l'approche PLE. Computer science Master, RIADI Laboratory, National School of Computer Science, Tunis Tunisia (2004)
7. Ben Abdallah Ben Lamine, S., Labed Jilani, L., Hajjami Ben Ghezala, H.: A Software Cost Estimation Model for Product Line Engineering: SoCoEMo-PLE. Proceedings, The 2005 International MultiConference in Computer Science and Computer Engineering, Software Engineering Research and Practice conference SERP 2005, Las Vegas Nevada USA (2005)
8. Mili, H., Mili, A., Yacoub, S., Edward A.: Reuse Based Software Engineering: Techniques, Organizations, and Measurement. John Wiley & Sons, Inc., ISBN: 0-471-39819-5 (2001) 672 p
9. Bosch, J.: Design and Use of Software Architectures: Adopting and Evolving a Product Line Approach. Addison-Wesley, Great Britain, ISBN 0-201-67494-7 (2000) 354 p
10. Ben Abdallah Ben Lamine, S., Labed Jilani, L., Hajjami Ben Ghezala, H.: Importance of Knowledge Engineering in Cost Estimation Models for Software Reuse: Case of a Software Cost Estimation Model for Product Line Engineering. Proceedings, The Seventeenth International Conference on Software Engineering and Knowledge Engineering, Taipei Taiwan China (2005)

Innovation Management for Product Line Engineering Organizations

Günter Böckle

Siemens AG, Otto Hahn Ring 6, 81730 Munich, Germany
guenter.boeckle@siemens.com

Abstract. Active innovation management is performed by companies to create an environment that fosters innovation. In a product line environment, platform and predefined variability restrict innovation because the development artifacts in the platform and the variation are prescribed. An analysis of innovation projects in literature shows that moderate innovations like introducing a new member of a product line yield only a small return on investment. This paper introduces a series of measures that can help to prevent a lock-in of a product line organization with respect to innovation. We take a look at various aspects of innovation – personnel, customer and market, technology and engineering, organization and process. Organizations may pick the best-suited measures for their current situation.

1 Introduction

We cannot command or order innovation. However, we can create an environment that fosters innovation. This is being done in companies already, often by a planned innovation management process. In a product line engineering organization, however, innovation may be blocked. We have platforms that provide certain artifacts for reuse in products and we have predefined variability. This means that particular innovations may be blocked from consideration or from introduction. Platform artifacts may be considered as fixed because a change may affect many existing and planned products. In many cases there is also a platform evolution plan that restricts changes of platform artifacts. The reference architecture determines the structure of all products, and changes to the architecture mostly require considerable effort for existing and planned products. Variability is pre-planned in a variability model, thus changes that go beyond this pre-planned variability may also require considerable effort; thus, innovations that affect the variability model may not be considered. So, there seems to be a lock-in of product lines that prevents innovations that go beyond pre-planned platform assets, their evolution, reference architecture, and variability model. There is the danger that innovation is only allowed or put into practice as far as the platform (and its evolution plan) and the variability model allow.

This paper presents a series of measures that can be taken to prevent such a lock-in. The list of measures is quite large and no organization will be able to perform all of them. These measures are just proposals and an organization may select some of them that are adequate for their particular situation.

H. Obbink and K. Pohl (Eds.): SPLC 2005, LNCS 3714, pp. 124 – 134, 2005.

2 Motivation

Kleinschmidt et al. [1] made analyses of the success of innovations. They categorized innovations into three classes according to low, moderate and high degree of innovation and determined the return on investment (ROI) and the success rate for each class.

Table 1. Return on investment for three classes of innovation (from [1])

Degree of innovation	Characteristics	Reasons for high or low ROI and success	Return on investment	Success rate
Low	Product modifications or new market positioning	- Low investments - Market and technology know how	124%	68%
Moderate	New products of existing product lines or new product line for the company with products that exist already in the market	- Moderate product advantage - Underestimation of existing market and technology challenges	31%	51%
High	New product lines for the company with new products for the market	- High product advantage compensates for lacking routine in market and technology	75%	78%

The findings, deduced from analyses of 203 projects in 125 international companies, show that moderate innovations yield the lowest return on investment. The rather low success rate of moderate innovations is rather surprising. So, for planning innovations we have to consider the potential ROI carefully. A very critical result from the analyses that gave the motivation for this paper is that a new product in an existing product line yields a low ROI. This indicates the innovation lock-in mentioned above, although this is not explicitly mentioned in [1]. As conclusion from these analyses we have to take special care for innovation in a product line organization and should consider measures to do this as part of a planned innovation management process. Such measures are put together in this paper.

There are some measures for innovation management in product-line organizations that shall be taken in any case:

- Innovation management shall be performed as a planned process.
- Roles for innovation management shall be defined and responsibilities assigned. Innovation will only be realized if someone is responsible for performing the adequate measures.
- For all non-trivial innovations an impact analysis, effort estimation and ROI shall be determined.
- The evolution of the product portfolio, platform, variability model, and reference architecture shall be planned with further innovations in mind.

3 Innovation Aspects

When we speak about innovation, we typically think of technological innovation, especially revolutionary innovations. However, for creating an innovation-friendly environment we have to consider several aspects of innovation, not just technology. And there are many innovations that are not revolutionary but still very successful, as Table 1 shows. In this paper, the following aspects of innovation are considered:

- Personnel aspects
- Customer and market aspects
- Technology and engineering aspects
- Organization aspects
- Process aspects.

Below, for each of these aspects a series of measures is listed from which an organization may select those that are appropriate for its situation. Many of these measures are already used in common innovation management; here they are adapted to a product line environment.

3.1 Personnel Aspects of Innovation

Innovation management offers encouragement for personnel e.g. in creativity workshops, for lateral thinking etc. Some of those workshops should be focused in a product-line organization especially on product line portfolio evolution, platform evolution, and variability evolution. For the identification of new features, creativity workshops on ideas about new features may be conducted for experts from marketing, product management, architects, engineering, maintenance, and others. The topic "thinking beyond our platform and current variability" may also be included in personnel-training sessions.

An intranet discussion forum and bulletin board for innovation and topics of platform, reference architecture, variability, etc. can help to support the exchange of ideas. For identifying daily practices that hinder innovation, people shall be encouraged to report those.

A reward system for innovation regarding the product line shall be established.

A role must be specified to pick up ideas from such workshops and discussion forums and check how these ideas can be used for product portfolio evolution, platform evolution, and variability evolution. These ideas shall be analyzed to determine how they can be put into practice, considering restrictions by the platform and the variability model. The ROI for making changes that put such ideas into practice shall be determine for selected ideas.

In PLE, cross-functional teams are used for many product-line related purposes. Such teams, where expertise from different fields meets are good places for innovation. These teams may be encouraged for using their meetings for "innovation sessions" where their different domain knowledge can further innovative ideas.

3.2 Customer and Market Aspects of Innovation

The market analysis has to identify new markets for the product line, trends in markets, new usage patterns, new features, competitors' offers, etc. There are two approaches to be taken: the first one determines the new features required (or wished) by the new markets and then analyses if and how platform and variability model support those. The second approach analyses the possibilities offered by platform and variability model and analyses how they can be used to support particular features (including determining how company and product line image fit in these markets). Market analyses shall also analyze the product line's success in different cultures to answer questions like: Which features are exciting[1] in different cultures? In which directions shall the product portfolio, platform and the variability model be extended for different cultures? Market analysis shall include identifying potential strengthening or damage to the company image and product line brand identity if certain features are added, new markets approached, etc.

The platform may be marketed as a brand of its own. The organization may create a product line brand, not just a company image and product brand. The platform flexibility may be marketed as virtue of the product line brand.

For some customer groups, companies provide help desks for reporting problems. In some cases it will make sense to categorize and analyze customer-reported problems to find out which ones are due to the platform and predefined variability. The analysis shall also find out if changes to overcome these problems will improve or hinder platform development. It shall be determined if new features customers ask for fit into the feature model and the variability model and which changes will be necessary.

In cases where an organization knows their customers, platform and variability shall be revealed to certain customers and they shall be encouraged to make proposals for innovation: new features, improved user interface, more or different variants, new variation points, etc. Creativity workshops can be performed with them on ideas about new features. The results of these workshops shall be analyzed, partly together with the customers, to find out if and how these features can be supported by the platform and the variability model.

Some organizations provide usability labs where customers can play with new products and where ideas and complaints are collected. These ideas and complaints can be related to the feature model, platform and variability model. An analysis can show how the ideas can be supported by them and how feature model, platform and variability model can help to overcome the usage problems, and where changes need to be made.

Product management shall update product and innovation roadmaps regularly. They shall review exciting[1] features for planned products relative to new market insight and customer wishes. The results shall be analyzed with respect to support by platform and variability model.

The objectives proposed in [3] for regular assessments of the innovation portfolio shall be complemented by product-line specific objectives:

[1] According to the Kano categorization of requirements.

- Assessing the innovation portfolio w.r.t. the platform and variability model. Besides assessing each initiative individually for risk, investment, return, and timing, the total product line portfolio shall be assessed to ensure that we have the right initiatives in it.
- Stretch and strategic fit. How much does the portfolio push the industry frontiers, and how well does it fit with the business goals, business strategy, product line goals and strategy? How can the product line strategy (including platform strategy) support the business strategy?
- Capabilities and capacity. Do we have the required capabilities to execute the product line portfolio and do you have enough of them?
- Leverage and risk. Did we leverage our investments for product line and the individual products in it (i.e., domain engineering and application engineering) so that we have a productivity advantage, while keeping risk within acceptable bounds?

3.3 Technology and Engineering Aspects of Innovation

Engineers shall watch new technology, new ideas and approaches. They shall present those ideas and approaches that they categorize as useful to a person responsible for assessing them if they can be applied for the product line, if they can improve business and if they can provide sufficient ROI.

Technological innovations encompass not only breakthrough innovations, but also new modeling technologies (for instance MDA, MDD, etc.), new templates for documents, new design and programming approaches (e.g. aspect-oriented programming), new ideas for architectural structuring, new technological features, new support for quality characteristics (e.g. for making something more reliable or faster), new algorithms for performing some functionality better or adding some functionality.

Joining existing platforms may enlarge the functionality and increase the quality of the platforms. A risk analysis and ROI have to be made. Unnecessary and double artifacts in the joined platform have to be removed; a new platform evolution plan is necessary. Similarly, if a platform becomes too large and hard to handle, a platform split shall be considered. The connectivity of certain parts of the platform and the usage in different products shall be determined to identify mostly independent parts.

Innovative COTS[2]: Watch the market for innovative COTS that can replace COTS in the product line, especially in the platform. Prioritize them, especially with respect to their ROI (considering potential reference architecture changes).

The usage of prototypes for new products and new approaches shall be encouraged. It shall be assessed which new artifacts based on the prototypes should be put into the platform, which existing ones can be used from the platform, and which ones shall be product-specific.

3.4 Organizational Aspects of Innovation

For supporting innovation, cross-functional teams are essential. These are needed for product-line engineering, anyway; they consist mostly of people from domain engineering and application engineering with different expertise (product management,

[2] COTS = Components Off The Shelf.

architecture, design, test, maintenance, etc.). Such cross-functional teams should get the tasks to devote a certain amount of their meeting time to innovation.

Cooperative development with companies and research institutes that have specific expertise in innovative development approaches and innovative products is a way to support innovation. Corresponding roles, contracts, processes and environment for cooperative work are necessary for this.

Opening a platform for other organizational units inside a company may support innovation, new ideas, and new applications. A platform may also be opened for other companies; an example is the Symbian operating system. Different participating companies may offer different new ideas, features, requirements, development approaches, etc.

New business models: Innovation can also come from new business models. Virtual integration of organization units, various kinds of partnerships, strategic alliances, joint ventures, open enterprises and extended enterprises are becoming common parts of a competitive strategy. Cooperation of different partners around one or more common platforms and around a common variability model can support innovation. Opening platform and variability model in new business models is an important aspect of organizational innovation.

3.5 Process Aspects of Innovation

For all kinds of innovation it has to be checked if they affect domain engineering (i.e. the platform) or application engineering (product-specific topics) or both.

Process and organization balancing: For innovation support it is important to remain flexible. Depending on the kind of innovation (new market, new features, new quality, etc.) a prototype or lead product may be produced. This means shifting personnel and other resources to an application engineering project for this. When a lead product or prototype is considered successful, personnel and other resources have to be shifted to domain engineering so that artifacts can be adapted for being incorporated into the platform. The corresponding managers must get the responsibility and decision power to perform such shifts.

A defined process should make measurements. This means for innovation management:

- Note whenever a competitor makes an innovation that the organization does not have.
- Note whenever an innovation cannot be made because of insufficient ROI and note the reasons (e.g. high effort for architecture change).
- Note whenever an employee does not find an adequate variability in the variability model or an adequate artifact in the platform.

These notes shall be analyzed regularly and the project leader (or other responsible role) shall try to define actions based on the results.

For all changes that are considered for innovation the 80/20 (sometimes also denoted as 90/10) principle may be applied. It is determined if a new artifact belongs to the 10% or 20% that are differentiating, i.e. that distinguish the company from all its competitors and that contain the organization's knowledge of their customers' wishes and their expertise of the market and technology. Otherwise, it may be of the

80 – 90% commodity that an organization builds, but that could be built by some other company, too. For all innovations concerning the differentiating part, extra accuracy shall be put in innovations there; this will be most effective. Many of the differentiating parts of an organization will be in its platforms, but also in the variability model. So, these should be considered first for innovation. It may be considered to use different platforms for differentiating and commodity artifacts, so that the latter may be offered to other organizations for cooperation.

Quality management: In markets where quality matters, innovations that improve quality are especially important. Changes that improve quality in platform artifacts and in the reference architecture will have most effect, because they affect more than one product. In cases where Six-Sigma or some other quality management method is applied, quality improving innovations for the platform and the variability model shall be considered first.

Innovation in teaching: All kinds of innovations in training, teaching, coaching shall be considered for the platform and variability model first, because they bring mostly more effect than the product-specific parts. Teaching about product line processes, platform and variability for new and experienced people can also bring new, innovative ideas. All training shall also be used for gathering new ideas for improving the product line.

Innovations in maintenance: Making things similar or the same for all products of a product line will make maintenance much easier and faster. The maintenance people have direct contacts to customers and learn from them about problems with the products. The complaints and ideas from maintenance personnel shall be collected and analyzed for improving the product line.

New roles: For innovation management we have to introduce new roles with responsibilities for tracking data about innovations and for analyzing and assessing these data.

4 The Analysis

In many of the cases described above, data has to be analyzed with respect to the influence on the product line. In this section, we consider the analysis of technological aspects.

One kind of innovation concerns new technology for developing artifacts of the products. These may be new templates, new modeling techniques, new programming languages, new interface descriptions, new tools, etc. In these cases we have to check if the artifacts created with the new technology can be used together with the old ones or if the old artifacts have to be changed to the new technology. Such a change may entail an adaptation or a rewriting of existing artifacts and corresponding updates in the configuration management. Several solutions for such changes may be possible and have to be compared with respect to cost and benefit. These innovations may affect the architectural texture (e.g., new rules for describing architectural artifacts). The effort for performing these changes has to be determined and the resulting cost has to be compared with the benefit from doing it; i.e., the return on investment (ROI) has to be determined for supporting the decision about applying the new technology and for the selection of appropriate changes.

Another kind of innovation concerns the product technology. These may be new algorithms, support for new hardware, new features and requirements and the artifacts for their realization, etc. The innovation has to be checked if it means just an addition of new artifacts or changes to existing artifacts in the platform. For each innovation we have to determine its impact on the reference architecture. The innovation may just mean an addition to the reference architecture, or it may mean a minor change to existing parts of the architecture. This may result in changing existing artifacts. Thus, new versions of existing products may have to be developed and integrated in the platform and the configuration management system. The innovation may involve a change to the reference architecture, which may entail many other changes. Any of these changes may entail in addition to technology changes also changing training, e.g. for maintenance people and customers. The effort for performing the required changes has to be determined and compared with the benefits.

An innovation has to be checked if it affects the variability model in the various kinds of artifacts - features, requirements, architecture, design, and test. A change to the variability in features or requirements will mostly cause changes in the architecture variability, the design and test variability. A change in the architecture variability will mostly cause changes in design and test variability, and a change in design variability may cause changes in the test variability. But also other dependencies have to be checked. In some cases an architecture change can influence requirements.

For analyzing impacts on the variability model we take a look at the terminology used here for variability.

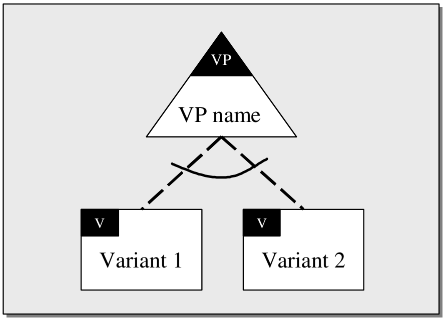


Fig. 1. Variability notation

The variability may be described as part of the artifact models – features, requirements, architecture, or design. Or it may be described in a separate, orthogonal variability model, independent from the artifact models, but related to them (typically via dependency links). The latter allows for easy traceability of relations between variability in different artifacts. In any case, we have variation points that define what varies. Connected to a variation point are variants that represent the selections to choose from at a variation point ("how it varies"). Between variation point and

variants we have variability dependencies – *mandatory*, if we have to select a certain variant; *alternative*, if one of several variants shall be selected, and *optional* if a variant may be selected or not. We may have '*requires*' or '*excludes*' dependencies between variants, variation points, and between either. For further information about variability, see [2]; a detailed introduction to variability is provided there, together with descriptions about variability in the different artifacts.
An innovation may have several kinds of impacts on the variability model:

- *Introduction of a new variant to an existing variation point.* This is mostly an easy case with minor changes only. However, it may change platform artifacts and may need configuration management effort. If we have an orthogonal variability model, the sub-graph of all dependent variation points and variants can be constructed to identify any changes induced by the introduction of the new variant. Otherwise, the induced changes have to be determined mostly manually. Tests for the new variant have to be created and included in existing test cases or new test cases have to be introduced. Maintenance plans have to be adapted.
- *Deletion of a variant.* This may happen if a specific variant shall no longer be offered. In this case it has to be checked if any future and current products need this variant and if there are any dependencies to other variants (e.g., another variant needs this one). Also former products that still are supported by maintenance have to be checked. In cases where we still need the variant, it must not be deleted or it has to be changed accordingly. Otherwise, it may be deleted but still changes to other variants may be necessary. Configuration management, tests, and maintenance plans may have to be adapted.
- *Changes to existing variants.* In this case all dependent variation points and variants have to be determined (e.g., by constructing the sub-graph of dependent variation points and variants in an orthogonal variability model) to identify any changes related to this change of one or more variants. Usually, this means that configuration management has to introduce new versions of the variants, and that tests have to be adapted.
- *Introduction of a new variation point.* Here, the reference architecture will mostly be affected. An artifact that has so far been fixed becomes variable by introducing a variation point. This may introduce changes to several artifacts; i.e., if a variation point is introduced in requirements or features, this may mean introducing variation points in architecture, design and test. The dependencies of the new variation point on other variation points have to be determined and modeled. Configuration management, tests and maintenance plans have to be considered for changes. This case may cause significant effort for realization. Also the platform evolution plan may be affected.
- *Deletion of a variation point.* This may happen if only one variant of this variation point shall be offered in future, or even none. Current and future products (and former ones for which we still offer maintenance) have to be checked if they will need the variation point, and all variants and variation points that have a constraint dependency on this variation point have to be identified for checking if the deletion can be performed. This deletion may cause changes in other artifacts, too (require-

ments, architecture …). Configuration management and tests have to be checked for changes and also the platform evolution plan may be affected.

- *Changes of dependencies between variants.* This may be a 'requires' dependency where the selection of a variant at a variation point requires the selection of a particular variant at another variation point. Or it may be an 'excludes' dependency where the selection of a variant at a variation point requires that a particular variant at another variation point must not be selected. A change to such a dependency may either be a change (from 'requires' to 'excludes' or vice versa) or it is deleted or a new dependency is introduced. The first case will rarely happen, the second case may cause changes to tests, while the third case may cause changes to other artifacts (if a new 'excludes' dependency is introduced between requirements, this may cause a change to the reference architecture). It has to be checked if tests need to be changed.
- *Changes of dependencies between variation points.* Again we may have a 'requires' dependency (the fact that we offer a selection at one point requires that we offer a selection at another point) or an 'excludes' dependency (the fact that we offer a selection at one point excludes selection at another point i.e., requires that the other point is fixed). Again, changing such a dependency from 'requires' to 'excludes' or vice versa will rarely happen, but an existing dependency may be deleted or a new one introduced. The second and third case may cause changes to tests, and the third case may also cause changes to other artifacts (as above).
- *Changes of dependencies between variation points and variants.* Here we may have the variability dependency between a variation point and its variants; this comprises the three cases 'mandatory', 'optional' or 'alternative'. Again, a change may either mean that the kind of dependency changes or that new dependencies are introduced or that existing ones are deleted. Mostly, such a change will not cause much effort, except for tests that check the dependencies. There may also be constraint dependencies between variants and variation points; this is the case when the selection of a variant at a variation point requires or excludes the availability of another variation point. Changes to that may in some cases affect changes in other artifacts (like above).

In most cases we have to check who will be affected – engineers, marketing people, testers, maintenance people, or customers. This means that we may have to consider changes in marketing strategy and material, training for customers, engineers and other personnel, changes to the product line portfolio, to the platform administration and evolution, and to the development process.

5 Conclusion

In a product line environment the paths an innovation may take are restricted by the platform and the predefined variability. The platform requires using predefined reference architecture and other predefined artifacts. The variability model restricts the variation for products of the product line. This may lead to a lock-in situation where the innovation that is necessary to increase market share is blocked. This paper presents a series of measures to overcome this problem. An organization may select the most appropriate ones for its current situation.

For making these measures a categorization of innovation measures into personnel, customer and market, technology and engineering, organization and process aspects is used that helps for the selection.

References

1. Kleinschmidt, E.J., Geschka, H., Cooper, R.G.: Erfolgsfaktor Markt – Kundenorientierte Produktinnovation. Springer-Verlag, Berlin Heidelberg New York (1996)
2. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering – Foundations, Principle & Techniques. Springer-Verlag, Berlin Heidelberg New York (to appear 2005)
3. http://www.1000ventures.com/business_guide/innovation_strategy.html; as of April, 2005.

Panel: Change is Good. You Go First

Charles W. Krueger

BigLever Software,
10500 Laurel Hill Cove,
Austin TX 78730 USA
ckrueger@biglever.com

Abstract. There is ample evidence that for many software development organizations, a *change* to software product line practice would be *good*. But there is also a reluctance by many of these software development organizations to *go first*. This panel explores what remains to be done by the SPLC community and others to overcome the inhibitors and to facilitate SPL adoption.

Overview

Of two things we can be certain. The first is that software product line practice offers some of the largest software engineering improvements seen in over four decades – numerous case studies have proven this. The second is that software product line challenges and opportunities exist in most commercial software development organizations – few markets call for just one product without variants.

From these two *givens*, the obvious expectation is a wholesale rush by the software development industry to embrace software product line practice. However, there is little evidence that such a trend is yet underway. In fact, very few organizations are coming forward to SPLC or other public forums with adoption or success stories.

This panel will explore the reluctance of software development organizations to be among the first to adopt and gain the benefits of software product line technologies and methodologies. The panel is comprised of software product line leaders with firsthand, in-depth experience in promoting and championing software product line practice, from inside and outside of candidate software development organizations.

What are the economic, temporal, cultural, and technological barriers – real or perceived – that prevent organizations from doing something that on the surface seems so obvious? What are the objections? What are the excuses? What are the inhibitors? Why do organizations think it doesn't apply to them?

Insights from this panel will help the audience to understand what needs to be done, both in research and in practice, to facilitate widespread adoption of software product line approaches in commercial practice.

H. Obbink and K. Pohl (Eds.): SPLC 2005, LNCS 3714, p. 135, 2005.

Panel: A Competition of Software Product Line Economic Models

Paul Clements

Software Engineering Institute, Carnegie Mellon University,
Pittsburgh, PA 15213-3890 USA
clements@sei.cmu.edu

Abstract. Proponents of different software product line and software reuse economic models will be given a real-world software product line scenario and asked to predict alternate outcomes and to justify – with hard data from their models – some of the difficult choices that need to be made in the scenario. The audience will have a chance to compare, side by side, the predictions, recommendations, insights, intuitive fidelity and ease-of-use of the different models.

Overview

There are numerous compelling success stories and abundant anecdotal evidence to suggest the tactical and strategic benefits of software product line practice. However, the stories and anecdotes also illustrate that there can be significant cost and risk.

In order for a software development organization to make the fundamental decision to embrace a software product line practice, real economic justification is needed, based on accurate economic models and engineering data. Without a realistic and accurate prediction of the prerequisites, risks, costs, returns and timeframes, management will be reticent to support a transition to software product lines.

There are published economic models for software reuse and software product lines that promise a solution to this problem. But how do these different economic models compare? This panel will offer insights in the form of a head-to-head competition. It will allow the audience to:

- learn how to apply the different economic models in a real-world scenario
- see what predictions the models can and cannot provide
- identify tradeoffs among the different models
- see which models are most intuitive and easy to use
- decide which ones they like best

H. Obbink and K. Pohl (Eds.): SPLC 2005, LNCS 3714, p. 136, 2005.

Enabling the Smooth Integration of Core Assets: Defining and Packaging Architectural Rules for a Family of Embedded Products

Tim Trew

Information Processing Architectures Group,
Philips Research Laboratories
Tim.Trew@philips.com

Abstract. One of the remaining challenges in product line engineering is how to establish the quality of the reusable assets so that we can be confident that they can be configured and composed reliably. This is desirable, both to avoid having to completely re-test each product and to avoid integration faults only being detected late in product development. One of the diversity mechanisms of Philips' high-end TV product line is the selection and composition of sub-systems, so different sub-system variants must integrate reliably if the aims of the product line are to be realized. An earlier study of integration testing obligations in Philips products concluded that certain design policies must be imposed if integration testing is to be feasible, but it did not describe how relevant policies could be identified at the earliest stages of design. This paper addresses how a set of architectural rules were established for the TV product line through a root-cause analysis of problem reports, and packaged so that developers can recognize when they should be applied. The approach builds on other work on the impact of design choices on non-func-tional requirements to ensure that all quality attributes are addressed.

1 Introduction

Many well-known product lines[1] are for families of embedded systems, where hardware diversity is a source of product variability. Embedded systems often share other properties:

- They use concurrent or multi-threaded designs to ensure that they can meet all real-time deadlines.
- They are developed by multiple groups, with limited communication between them, each with skills in particular aspects of the system and its development.
- New hardware and software core assets are developed concurrently.

In developing a product line, we aim to create a repository of core assets, which can be rapidly composed into product instances. This paper addresses the experience of reducing the faults found during the integration of high-end TVs in Philips Consumer Electronics, in particular by identifying architectural rules that should be applied to make this feasible. This section continues with an overview of the TV

H. Obbink and K. Pohl (Eds.): SPLC 2005, LNCS 3714, pp. 137 – 149, 2005.

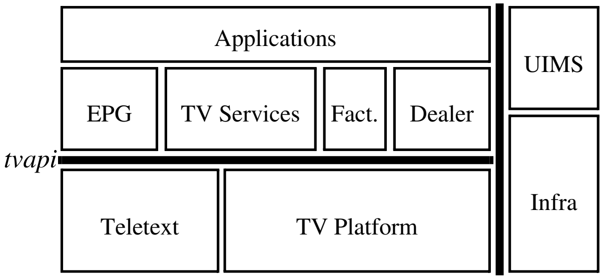


Fig. 1. High-end TV product line sub-systems

architecture and its development process. Section 2 summarizes the issues of integration and testing, providing the motivation for "Design for 'Plug and Play'", described in section 3, which identifies rules that eliminate many classes of integration faults, shifts the balance of test responsibility towards the core assets and makes the residual integration test obligations attainable. Finally, section 4 discusses the deployment of such rules in the product line development organization.

1.1 Philips High-End TV Architecture

The product line architecture, together with its supporting Koala component model, has been documented by van Ommering[8]. Figure 1 shows its principal sub-system decomposition. Considering its sub-system structure, it has a hardware abstraction layer, whose interface is defined by the *tvapi*, several service sub-systems, which are generic to all TVs, and a general computing infrastructure and user-interface management system (UIMS). Finally, the applications structure the behavior as perceived by the user. The sub-systems and their constituent components are characterized in terms of their *required* and *provided* interfaces.

A variety of diversity mechanisms are used[8], principally:

- Selection of sub-systems. In particular, the *tvapi* has been implemented by three entirely-different TV Platforms, supported by very different hardware.
- Diversity interfaces, through which components query parameters through their *required* interfaces. Components are nested recursively as compound components until the top-level component, representing the complete product, is reached. Diversity interfaces can be propagated outwards, with explicit settings being made at appropriate levels in the hierarchy, depending on the nature and scope of the product diversity.
- Product-specific glue modules, which adapt between mismatched interfaces or implement small fragments of product-specific functionality not supported by the reusable sub-systems. A *switch* is a particular type of glue that provides alternative component bindings that can be set at compile- or run-time.

Consequently, a product is instantiated by selecting the appropriate sub-systems, binding them, using product-specific glue where necessary, and satisfying the diversity interfaces at all levels of the component hierarchy.

1.2 Philips High-End TV Development Processes and Organization

The product line is developed across multiple sites and the architecture is designed so that a specific site is responsible for the development of all variants of a particular subsystem. Additionally, specialized sites are responsible for the final product integration, i.e. binding the sub-systems, instantiating diversity parameters and creating product-specific glue.

The development process can be summarized as follows:

Product line definition: the product line scope, commonality and variability are identified and elements of features are allocated to sub-systems. Some key axes of variability are geographical region, display technology (CRT, plasma, LCD) and level of features. This activity is repeated periodically as completely new features, e.g. digital TV or wireless connectivity, are added to the product line scope.

Sub-system specification: sub-systems are defined by their *provided* and *required* interfaces, whose specifications are placed under a global change control board to ensure stability.

Sub-system design: sub-system architects decompose the sub-systems into finer-grained components, again defined by their interfaces, which can be allocated to small groups of developers. When a new variant is required, it is created through a combination of existing components and newly developed ones.

Sub-system implementation and test: the components are created and composed into sub-systems. These are intended to be tested such that they can be integrated into a product, using the diversity mechanisms described in section 1.1. Testing and integration are discussed further in section 2.

Product instantiation: sub-systems are delivered to the integrating site, then composed using glue modules as required, and tested. The complexity of this process depends on the phase in a generation of the product line. Periodically, substantial changes are made to the functionality offered or the technology employed. In this case, incremental development is used, in which new functionality is partitioned into *feature blocks*, and the corresponding functionality is added to all relevant sub-systems in a synchronized manner. This brings the advantages of better risk management inherent in an iterative life cycle and has successfully brought new features to market, despite constrained development times. The delivery of features blocks is followed by several *maturity releases*, during which problems are resolved. It should be noted that there is considerable tension between quality and schedule during the development of feature blocks since such major changes in functionality are often accompanied by substantial changes in hardware and early versions of the software are required to validate overall system performance, irrespective of their reliability.

Once the first instance has been integrated successfully, other members of the product line are instantiated without significant changes to the constituent sub-systems, as illustrated in simplified form in figure 2. A consequence of this process is that difficult integration problems have only been addressed during the maturity releases, when their solutions may be overly-tuned to a particular composition of sub-system instances.

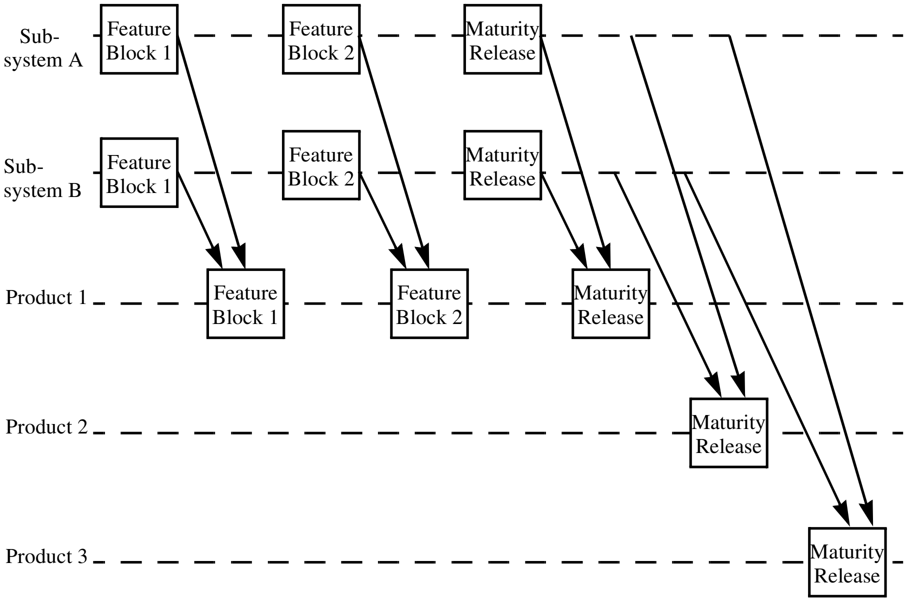


Fig. 2. Simplified view of the incremental development of new functionality for a newgeneration of a product line, followed by the creation of further product instances

2 Testing and Integration Issues

When the TV product line development was started in 1998, it was widely believed that it should be possible to test a sub-system against its interface specifications with sufficient rigor that sub-systems could be composed into products without knowledge of the internal design of other sub-systems and without many global rules. Consequently, problems encountered during the integration of early products were considered to be the result of inadequate sub-system testing and there was pressure on the test teams to develop new methods. Initially, the focus was on testing individual interfaces using a bottom-up integration strategy and many cases were found in which such tests passed, but the system tests still failed, so an improved sub-system testing strategy was sought.

Two main obstacles to achieving adequate sub-system testing were identified:

1. The obligation of component testing, of which sub-system testing is an example, is to show that the component satisfies its *provided* interfaces for any correct implementation of its *required* interfaces. In doing so, it must show that there are no undocumented constraints between the interfaces. To give an impression of the magnitude of the difficulty of achieving these obligations, a particular version of the TV Services sub-system had 85 *provided* interfaces (many of which are handlers for platform-generated notifications) and 141 *required* interfaces. Each of these interfaces could contain many individual functions.

2. There is a tension between the interests of the designer of a server component and tester of its clients. The designer may wish that some aspects of their interfaces,e.g. constraints on the order in which notifications are generated, are loosely specified, to allow flexibility in design choices to meet other non-functional requirements. In contrast, testers prefer maximal constraints on a client's *required* interfaces, so that only the smallest number of cases needs to be considered.

However, two further problems become apparent:

1. The nature of the faults that were being found during integration was not simple oversights, for which improvements to inspections, static analysis and testing would be an appropriate response. Often sensible, but incompatible, local decisions were made within each sub-system, resulting in system failures under obscure circumstances. Even if an improved test strategy were developed that triggered all such failures, it would still be necessary to identify the underlying faults and correct them in a way that gave some confidence that the sub-systems now interacted correctly. Therefore the problem is principally one of architecture and design, rather than testing. For example, if there are no constraints on the ordering of notifications delivered to a client, then the client's design must make it clear how this non-determinism will be accommodated. This is particularly important in a product-line context where different product instances may be based on different hardware and, in the TV example, different instances of the TV Platform, so that a point solution is not adequate.
2. Following an analysis of several developments in Philips, including members of the high-end TV product line, it became clear that there are a variety of classes of integration errors that cannot be detected by testing individual components or sub-systems. For products with internal concurrency and few hard real-time restrictions, the following classes of integration faults were identified:

   - Incompatibility between actual and formal parameter ranges
   - Inconsistent interpretation of parameter values
   - Inconsistent parameter ordering
   - Inconsistent use of shared global data
   - Unexpected state-event combinations
   - Unclear allocation of responsibilities between modules
   - Unexpected re-entrancy
   - Race conditions
   - Unprotected critical sections and deadlock

Detecting these classes of faults requires an integration testing activity whose aim is to check that components interact correctly *under all circumstances*, which may require much higher degrees of controllability and observability than would be possible in a complete system. The resulting testing obligations and the design constraints needed to be able to meet them economically have been discussed in [9] but, as before, what is principally required is a design approach that specifically aims to eliminate these classes of problems.

In the short term, these issues were addressed by extending the test strategy from one focused on testing interfaces against their specifications to one of mapping product-level scenarios down to lower-level sub-systems. This has led to the creation of successful products, but at the expense of greater coupling between sub-systems than was intended. This happens because:

- The mapping of test cases to lower-level sub-systems requires a detailed knowledge of the behavior of the higher sub-systems.
- The majority of testing is now performed on a partially integrated system with specific instances of the sub-systems and, as shown in figure 2, problems are resolved in the context of a particular product before the sub-systems stabilize and are used to derive the remaining members of the family.

This can result in unexpected constraints being introduced on the *required* interfaces of sub-systems, limiting the flexibility of the designers of new generations of servers that *provide* them. A design approach is required that avoids this, while satisfying all other constraints of the high-end TV development process.

3 Design for “Plug and Play”

An approach to design is required that results in sub-systems that:

- Can be expected to interact correctly under all circumstances, configurations and diversity settings.
- Meet all other non-functional requirements, such as performance and configurability.
- Can be developed on different sites with limited communication between them.
- Can be tested to detect implementation errors using methods and processes that can realistically be deployed in our organization.

Considering these, we are not particularly concerned with “design for analyzability” for functional correctness. The high-integrity systems community has identified generic rules that permit the behavior and performance of systems to be predicted or ana-lyzed[3]. However, given the loosely-coupled nature of multi-site product development and the need to have *apriori* confidence that a component can be reused in all instances of the product line, more specific design restrictions are necessary. It would be unacceptable, having focused on analyzability, for a new instance of the TV Platform to be designed, only for an analysis to reveal that a system property would not be maintained when it is composed with existing sub-systems. Not only is it likely that this would be found too late to meet time-to-market requirements, but guidance would still be required on how to resolve the problem.

We must therefore be able to answer the following questions:

- What are the properties that must be maintained for sub-systems to interact reliably?
- What are the rules that can be applied locally to ensure that these properties are maintained?
- How can the rules be packaged and deployed so that it is clear to an individual developer, with a very limited view of the system, which ones are relevant, without requiring architects to state this explicitly in each case?
- What are the obligations of each of the test phases to detect implementation errors?

These questions are addressed in the following sub-sections.

3.1 Identification of Crucial System Properties

The set of properties should be minimal, yet adequate. Their identification was bootstrapped through a root-cause analysis of approximately 900 problem reports from the product line, with the perspective of "how might this problem have been found earlier or eliminated entirely?". The problem reports were selected both from sub-systems that had proved to be difficult to integrate and from cases in which a new platform was introduced into the established product line. Although many issues were identified, from requirements to testing, this paper concentrates on those in which a design change would have avoided the problem. While the selection of problems proved to be very effective at identifying design issues, the focused approach means that it is difficult to draw statistical conclusions that would prioritize the improvements.

Initially, the need for a design change was identified from a testing perspective[9], prompted by cases in which it would be unreasonable to have expected a structured test case design to have found the problem. While this view provides great insight, it is not helpful in giving guidelines that can be applied from the earliest stages of design. Furthermore, a single problem report may be the result of a variety of distinct design issues or, conversely, it may not provide enough clues to recognize that it is the result of poor design rather than an oversight in a specific instance. There are substantial differences between the perspectives of the testers and architects and the mental process of making the transformation between them is akin to that of identifying a new design pattern, during which much disparate information is internalized.

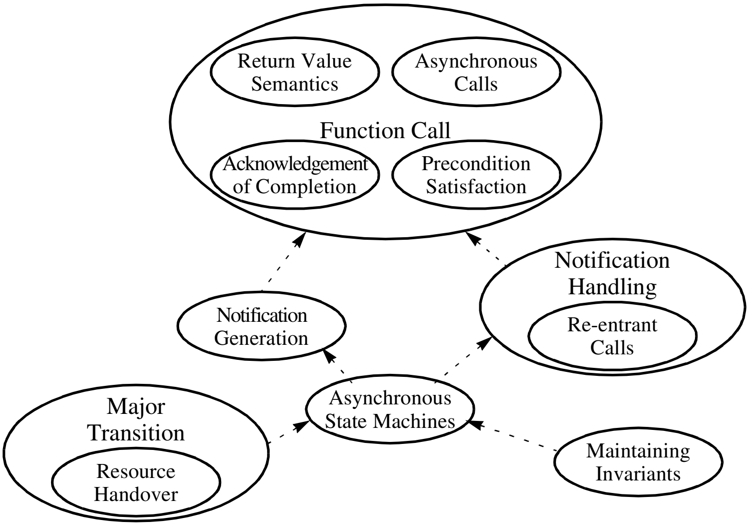


Fig. 3. Example "Interaction Contexts"

The result of this process is a 2D matrix of properties and “interaction contexts”. Interaction contexts, examples of which are shown in figure 3, provide the context within which an architectural rule is to be applied. “Interaction contexts” are therefore more abstract than Bachmann’s use of “patterns of interaction”[2] in relation to component-based software engineering, which would result from the application of the rules to particular interfaces. In figure 3, nested ellipses denote particular aspects of an interaction context, whereas the dependencies, represented by the dashed lines, indicate that a complex interaction context includes a simpler one. The crucial distinction is that the target of an included context can exist in its own right.

Table 1 shows some examples of properties and their interaction contexts, together with their most generic form. Subsequently, in relation to defining architectural rules, these properties will be designated “intents”. Properties are identified as follows:

- Following the analysis of a problem report, the interaction contexts involved in the problem are identified.
- The properties that should have held in each interaction context are identified.
- The generic properties are studied to determine whether the new property is a specialization of an existing generic property and whether there are other generic properties that could also be applicable in the current context.
- If no generic property exists for the new interaction context-specific one, it is abstracted and a generic property created.

Table 1. Example properties (*Intents*) and their specializations for specific interaction contexts

Interaction Contexts	**Intents**	
Generic	**Functional correctness should not depend on the relative priorities of threads**	**Variables must be initialized before they are used**
Notification Handling	The prevention of re-entrant callbacks should not depend on the server running at a higher priority than the client	Variables that will be read by a notification handler must be set before the handler executes.
Power-up	An initialisation function should not have to run at a high priority level to ensure that a component is fully initialized before other activities commence	Avoid cyclic dependencies between subsystems and initialize them in the correct order.

The explicit hierarchy has two advantages:

- It allows relevant properties to be identified without there being a corresponding problem report. The approach therefore gains predictive power.
- The generic properties can be phrased in a way that pre-empts dissent when considering the impact of rules on other non-functional requirements. It is difficult to argue against “pre-conditions should not be violated”, whereas the interaction context specializations may not appear so clear-cut, e.g. “the client must be able to ensure that no relevant changes are made to the system state while the pre-con-ditions are being met” is open to debate.

Table 2. Properties (*intents*) and the principal non-functional attributes that they impact

Non-Functional Attributes			**Intents**
Performance	Latency	Granularity	A component must never block the system long enough to compromise its real-time behavior
		Efficiency	A function should not be called repeatedly without performing a useful action
			Minimize the amount of processing carried out on the receipt of events with hard real-time deadlines
			Avoid introducing unpredictable delays when responding to events with hard real-time deadlines
Maintainability			Complex clients require a consistent programming model.
Reusability		Configurability /Diversity	Inter-feature interaction should not be hard-wired at a lower-level than it is defined
			Functional correctness should not depend on the relative priorities of threads
			It should be possible to compose components with compatible interfaces without information on their internal design decisions
Reliability	Design for 'Plug and Play'	Complexity	The state-space of a system should not be made unnecessarily large
		Predictability	Activities that can be initiated under several conditions should execute the correct number of times.
			State information should not be replicated
			It should be clear that pre-conditions can never be violated
			Policies of clients and servers must be matched
			Avoid re-entrant notifications unless strictly necessary.
			Designs should be insensitive to the order of completion of unconstrained activities
			Components should never act when the global system state is inappropriate
			Variables must be initialized before they are used
			Systems should be designed to be deadlock free
			The interaction between features on a shared resource should always be managed
	Testability	Cohesion	Maintain a separation of concerns
			Avoid replication of functionality
			Maximize cohesion

Table 2 shows the list of generic properties in relation to the principal non-function-al requirements that they affect. In practice, this relationship is not strictly hierarchical, and, as described in section 3.2, a graph provides a more comprehensive view.

3.2 Derivation of Architectural Rules

Having identified the properties that should be maintained, rules are required that will maintain them. In general, a rule identifies a set of policies, together with the conditions under which a particular policy should be used. Policies are proposed and then checked that they satisfy all properties, which are termed "intents", analogously to the form of design patterns, to indicate the purpose of the policies (a *policy* should satisfy an *intent* in an *interaction context*). The framework for this check is based upon the work of Gross and Yu[5], who showed the contribution (positive or negative) that design patterns make to non-functional requirements (NFRs) using an NFR softgoal graph. Their work is based upon Chung *et al*'s original development of the NFR softgoal interdependency graph[4], but with design patterns used as Chung *et al*'s "operationalization methods", outside the context of a particular application. It is this context-independence that we find most valuable, since architectural rules should be applicable to any feature supported by the product line. Policies are at a higher-level of abstraction than design patterns, and a qualitative record of the design rationale is required to guide and justify their selection. Each policy is assessed for its contribution to each of the intents and NFRs. The process of constructing the design argumentation and, in particular, making explicit the contribution of each policy to each intent or NFR, increases the insight into the issues to be addressed. In many cases we find that, by considering all the NFRs in this balanced way, the range of acceptable design choices is much smaller than had previously been thought.

Finally, architectural rules are defined, typically mandating that a specific policy be used, together with additional constraints to ensure that the conditions that it demands always hold. Alternatively, it might apply different policies, for example where different NFRs are important in the different layers of the architecture.

3.3 Packaging and Dissemination

Having followed the process described so far, the rules will have been accompanied by the specific problems that motivated them, the issues abstracted from the problems and the design rules that justify them. For conciseness, the rules are extracted and recast so that they are comprehensible in isolation, although the more elaborate description remains to provide justification and background information where necessary. Finally, many rules are expressed in terms of design patterns, either those that are well-known in the literature or ones that are more specific to the product line. The specific patterns address both the characteristics of Koala's static component binding, such as using the *Notification Multicaster* rather than the *Observer* pattern, and composition issues, such as the *Hierarchical Invariant Maintainer* pattern, which manages sharable servers, despite the changes in composition for different members of the product line. Figure 4 illustrates how these and subsequent activities are distributed throughout the lifecycle.

The sub-system specifications and design are created during the *system design* phase, which is when the sub-system architect is responsible for identifying the relevant rules. There has been some debate as to whether the consequences of rules should be completely transformed into patterns of interaction[2] in the interface specifications, or whether developers should be aware of the architectural rules throughout development and that the interface specifications should make reference to

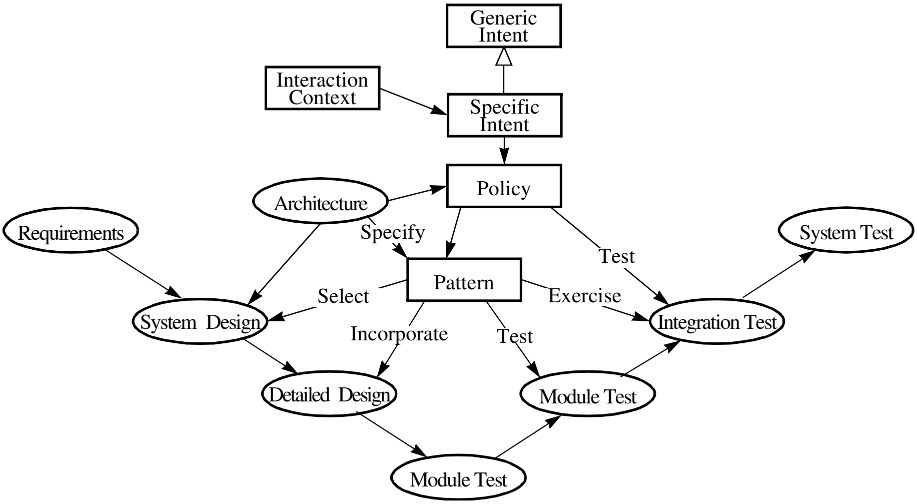


Fig. 4. Deployment of Design for Plug and Play through the development lifecycle

the relevant rules. Currently the product line interface specifications use intuitive semantics[7], but even the most rigorous form of interface specifications, based on the "Pre & Post" level of ISpecs[6], that are considered suitable for practical use in Philips, would not capture all aspects of the policies in isolation, e.g. rules for where an asynchronous call can be made in a notification delivery chain.

In the short term, the question is moot for this product line; the architectural rules must be available explicitly to be able to guide the specification and inspection of interfaces and the rules could not be made implicit in the specifications without rewriting them, running to thousands of documents.

Individual developers are responsible for implementing specific components and must be aware of the consequences of the rules within their scope. The rules naturally force a developer to consider more than just the components to which they have been assigned, which immediately reduces the sources of integration faults.

3.4 Testing

Of the classes of integration faults in section 2, the last five relate to the dynamic interaction between components, which may require an infeasible number of test cases to achieve test adequacy, even if the behavior of the components has been specified in enough detail. Many of the rules address these classes through the introduction of mechanisms that manage the interactions between components, transforming the integration testing problem into comprehensive local tests of the mechanism in each component and then simply exercising the composition with some typical scenarios, as indicated in figure 4. This shift in the balance of test responsibility to the core assets justifies the term "Design for Plug and Play" and goes some way to reducing the bottleneck of testing in product-line development. Following integration, it may be valuable to check that the policies are being honored, e.g. by using assertions during system testing.

Examples of mechanisms include *handshakes* or *numbered transactions* to eliminate race conditions, and *hierarchical invariant maintainers* for cases where a shareable server is controlled by multiple clients. Standard component testing, static analysis and inspection techniques can be used to test the mechanisms, which only have to be performed once for core assets.

The first four classes of integration faults in section 2 can reasonably be addressed by testing, although improvements to interface specifications[6] will both reduce the likelihood of their occurrence and increase the effectiveness of inspections.

4 Deployment

Establishing and packaging a set of architectural rules for a product line is a significant challenge, but actually re-engineering core assets according to the rules is more complex, given the continuous pressure on development teams to deliver new products, while adding new features to support digital TV standards and new display technologies. The principal obstacle to re-engineering is the cost of revalidating new sub-sys-tems, rather than the effort required to implement the changes, which would be relatively minor. Nevertheless, a new Applications sub-system has been created that conforms to the rules, particularly with respect to handling the *major transitions* (see figure 3) that occur when changing between modes in the TV. A *major transition* requires state changes in several unrelated, but potentially interfering, features that takes sufficient time that new system inputs cannot be deferred during the transition. The rules achieve a clear separation between the concerns of handling new user inputs, managing the interactions between features and managing the sub-transitions within one feature. The results have been promising, with many fewer faults found during integration than in previous versions. However, this must be viewed with caution, since any re-engineer-ing activity should deliver positive results. Opportunities are being sought to trial the rules at lower levels in the system but here the reusability of the TV Services sub-system has meant that there has been little need for it to be changed. An important aspect of the rules is that it is generally possible to introduce them incrementally, rather than having to upgrade all sub-systems simultaneously. Smaller new developments are being considered for a pilot, to confirm that the rules really address all the integration issues, before wide-scale deployment in as complex a product line as high-end TV.

5 Conclusions

The selection of different combinations of sub-system variants is one of the diversity mechanisms in Philips high-end TV product line. Consequently, it is vital that a sub-system performs correctly with any valid implementation of its *required* interfaces, rather than being tuned to a specific implementation. In the early days of the product line development it was assumed that reliable integration could be achieved by adequate testing of interfaces. Subsequently, the integration strategy used in practice tended to tune sub-systems to each other, which hampered the introduction of new sub-system instances. Having developed an improved understanding of the integration testing obligations, it became clear that the original ambitions of the product family could only be realized if more rules were introduced. These, *inter alia*, reduce the dependency of one sub-system on the state space of others and its sensitivity to the interleaving of function calls and notifications.

A root cause analysis of about 900 problem reports from the product line allowed rules to be identified that would have avoided many problems entirely and, by incorporating them in a matrix of *intents* and *interaction contexts*, to both generalize them and provide a structure that permits the relevant rules to be identified from the beginning of sub-system design. Relating these to a hierarchy of non-functional requirements ensures that other aspects, such as performance and reusability, are also given appropriate weight in formulating the rules.

The rules have been exercised when re-engineering the Applications sub-system. Their wider use in the product line depends upon there being a business case for re-en-gineering other sub-systems, but smaller product developments are being sought to demonstrate their effectiveness in reducing integration problems.

Acknowledgements. I would like to thank Nitin Koppalkar and M.R. Narasimhamurthy of Philips Research, Bangalore, for their contributions to the problem analysis that informed this work.

References

[1] "Product Line Hall of Fame", http://www.sei.cmu.edu/productlines/plp_hof.html.

[2] Felix Bachmann *et al*, "Technical Concepts of Component-Based Software Engineering", SEI Tech. Report CMU/SEI-2000-TR-008, 2000.

[3] Alan Burns *et al*, "Guide for the use of the Ada Ravenscar Profile in High Integrity Systems", University of York Technical Report YCS-2003-348, January, 2003.

[4] L. Chung *et al*, *Non-Functional Requirements In Software Engineering*, Kluwer Academic, 2000.

[5] Daniel Gross and Eric Yu, "From Non-Functional Requirements to Design Through Patterns", *Requirements Engineering*, **6**(1), pp.18-36, 2001.

[6] Hans Jonkers, "ISpec: Towards Practical and Sound Interface Specifications",W. Grieskamp *et al* (*eds.*) *IFM 2000*, LNCS **1945**, pp. 116–135, Springer-Verlag, 2000.

[7] Eivind Nordby and Martin Blom, "Semantic Integrity in CBD", Ivica Crnkovic and Magnus Larsson *(eds.) Building Reliable Component-Based Software Systems*, Artech House, 2002.

[8] Rob van Ommering, "The Koala Component Model", Ivica Crnkovic, Magnus Larsson *(eds.) Building Reliable Component-Based Software Systems*, Artech House, 2002.

[9] Tim Trew, "What Design Policies must Testers Demand from Product Line Architects?", *Proc. Int. Workshop on Software Product Line Testing*, 2004.

Design Verification for Product Line Development

Tomoji Kishi[1], Natsuko Noda[2], and Takuya Katayama[1]

[1] School of Information Science, JAIST-Japan Advanced Institute of Science and Technology,
1-1 Asahidai, Noumi-city, 923-1292 Ishikawa, Japan
{tkishi, katayama}@jaist.ac.jp
[2] NEC Corporation, Igarashi-Bldg, 2-11-5 Shibaura, Minato-ku, 108-8557 Tokyo, Japan
n-noda@cw.jp.nec.com

Abstract. Our society is becoming increasingly dependent on embedded software, and its reliability becomes more and more important. Although we can utilize powerful scientific methods such as model checking techniques to develop reliable embedded software, it is expensive to apply these methods to consumer embedded software development. In this paper, we propose an application of model checking techniques for design verification in product line development (PLD). We introduce reusable verification models in which we define variation points, and we show how to define traceability among feature models, design models and verification models. The reuse of verification models in PLD not only enables the systematic design verification of each product but also reduces the cost of applying model checking techniques.

1 Introduction

The recent advances in embedded and ubiquitous computing technologies increase the societal dependence on embedded software, and its reliability becomes more and more important. Until recently, the size and complexity of embedded software was relatively small, and the development style was implementation centric. However, in recent times, embedded software has become larger, and the development period has become shorter. Thereby, such conventional development style is becoming obsolete. Against this background, we began an industry-university joint project in 2003, supported by the Ministry of Education, Culture, Sports, Science and Technology, Japan, to develop a design environment for highly-reliable embedded software [10] utilizing model checking techniques [2].

Since most embedded systems reacts against events caused by physical phenomena, they have to handle every possible event occurrence and event sequence. Therefore, as compared to the design of business applications, the design of embedded systems requires a more exhaustive checking. Model checking techniques are promising techniques for such design verification; however, the application of such scientific techniques is expensive because it is time-consuming, and requires technical expertise. This makes it difficult to introduce these techniques in consumer embedded software development, such as in automobile and consumer-electronics fields.

In this paper, we introduce an application of model checking techniques in product line development (PLD) [11]. We observe that this has the following advantages.

H. Obbink and K. Pohl (Eds.): SPLC 2005, LNCS 3714, pp. 150 – 161, 2005.

Firstly, since it is common for embedded software developer to develop families of software, applying the techniques in PLD is reasonable. Secondly, in PLD, the design verification becomes important; because each time we develop a product, it is necessary to design the product based on the core asset and check whether the design satisfies the selected features. Thirdly, we can expect to reduce the cost of applying model checking techniques by reusing the verification model in PLD.

In section 2, we explain how to apply model checking techniques in design verification. In section 3, we propose techniques to define variation points in a verification model in order to accommodate the variability in a product family. In section 4, we show the method of organizing core assets in order to reuse the verification model systematically. In section 5, we introduce our support environment. In section 6, we evaluate our approach, and in section 7, we present some technical discussions.

2 Design Verification

In this section, we clarify the type of design verification that was examined and the method of applying model checking techniques to the verification.

2.1 Design Testing Based on Test Scenario

In this study, we examined a family of embedded software for a vehicle lighting system (VLS) that controls the interior lights of an automobile, based on the statuses of the door, locking, ignition, etc. Although the size of the software is not very large, hundreds of VLS products are developed every year; they have several common characteristics as well as different features depending on the light type (such as room light and foot light), vehicle type, grade, and target market. The manufacturer designs the system and subsequently order contractors to implement the system. Therefore, it is important to check the validity of the design in a short time period. Our objective is to support this type of design verification in PLD.

Among the many issues that have to be verified, the main issue in the project is to check the validity of the software design on based on test scenarios. A test scenario is a set of event sequences that are expected to make the target system move into a specific state; it can be defined as a quadruplet *(T, I, {S}, F)*, where *T* denotes target system, *I* denotes the initial state of *T*, *{S}* denotes a set of event sequences sent to *T* from external entities, and *F* denotes the final state of *T*. In other words, design verification based on test scenarios (we term it “design testing”) is an activity to check whether the target design model of state *I* moves into state *F* after receiving an event sequence included in *{S}*. It should be noted that, in this paper, how to prepare good test scenarios is outside the scope of this paper.

2.2 Applying Model Checking Techniques

One of the most common techniques to verify the design is reviewing. Although reviewing is a useful technique, it is not effective for exhaustive checking since it is performed manually. We can also verify the design by using tools such as design simulators and can actually execute the design model. However, we only check an event sequence one by one. In order to realize more exhaustive checking, we examine the application of model checking techniques for design testing.

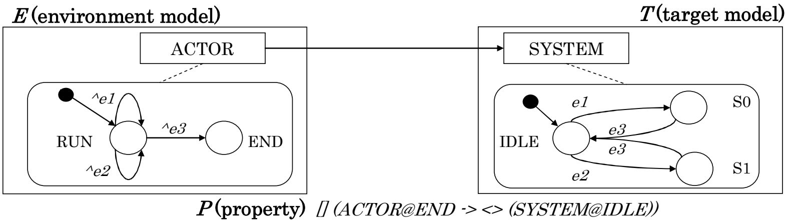


Fig. 1. Example of the Verification Model

A typical use of model checking techniques is to describe the target system and its environment as a state model, give some logical properties, and automatically check whether the given property holds. Based on this scheme, we develop a verification model *(T, E, P)* for each test scenario *(T, I, {S}, F).* Here, *T* denotes a target model similar to the one referred to in the test scenario, *E* denotes the environment model that sends event sequences included in *{S}* to *T*, and *P* denotes the property that expresses "if the target *T* in state *I* receives any event sequence included in *{S}*, eventually, it falls into state *F*." We can test the design model in terms of the test scenario by combining *T* and *E* and applying the model checker to verify whether the *P* holds.

Fig. 1 shows an example of the verification model. The right hand side shows the target model *T* and the left hand side shows the environment model *E*. The property *P* is expressed in our extended representation of LTL (Linear Temporal Logic) formula as "[] (ACTOR@END -> <> (SYSTEM@IDLE))" (whenever ACTOR reaches the state END, SYSTEM eventually reaches the state IDLE). We can check the model (combination of the target and environment models) along with the property by using a model checker. Thus, we can exhaustively check the test scenario because a model checker can handle generic test scenario expressed in regular expression (which is equivalent to state model).

3 Reusable Verification Model

In this section, we examine techniques to make the verification model reusable.

3.1 Context of Reuse

PLD comprises two phases—developing core assets for product families (domain engineering phase) and developing each product with the core assets (application engineering phase) [1, 3, 13]. We intend to apply our design testing to the application engineering phase. In other words, before actually implementing the product, we want to check whether its design correctly realizes the required features. Since products in a product family are similar, the test scenarios for each product are also similar. Therefore, similar to a reusable design model, we can develop a reusable verification model that can be applied to multiple products in a family.

Intuitively speaking, we define generic verification models, i.e., generic environment models and properties that can be applied to every product, and we reuse them throughout the PLD. If some parts of the environment model or properties differ among products, we define variation points at these points, further, we prepare variants for them. When we select a feature for a product, we identify the variants that correspond to the feature and apply the variants to the variation point in order to define the corresponding verification model.

3.2 Variation Points in a Verification Model

In order to examine reuse in PLD, we have classified the software design into the following three levels:

- Shared Architecture (SA) level: The component structure (configuration of components) that is shared by all products. This level corresponds to the frozen spot of product line architecture (PLA) and should be built and verified in the domain engineering phase. We generally need not re-verify its properties in the application engineering phase.
- Derived Architecture (DA) level: The component structure that may differ according to the product. In this case, derived implies that these structures are among the variations of PLA. Fig. 2 shows the PLA of VLS. "S-Fix" fixes sensor values from (generally multiple) "SENSOR," (generally multiple) "CTRL" decides the light control based on some sensor values, and "L-Ctrl" (light control) gathers the results and determines the actual light control. The number and type of "SENSOR" and "CTRL" may differ according to the product. "P-Setting" (parameter setting) is an optional component that stores parameter values. From this PLA, we can derive a variety of architectures. In the application engineering phase, we decide the concrete architecture for the product.
- Component (CO) level: Internal structure of each component. As each product has different component structure (in the DA level), each component can handle different events from different components. Further, each component can have different behavior depending on the product. In application engineering, we need to verify whether a design realizes the intended behavior.

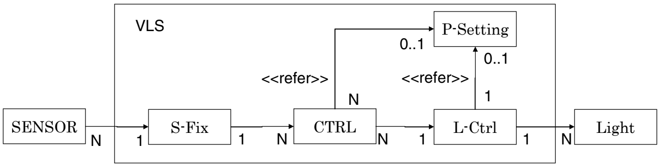


Fig. 2. Product Line Architecture (static structure) of VLS

Fig. 3 shows an example of a reusable verification model. The upper side shows a part of the design models of two products A and B. Product A has one sensor, "D-Sensor" (door sensor) and Product B has two sensors, "D-Sensor" and "G-Sensor" (gear position sensor). Since the input events are different, the state models for "S-Fix" and "D-ctrl" (door control) are different. Assume that our verification target is "S-Fix" and "D-Ctrl," and we want to verify the property "one of the doors is unlocked, and then 'all door lock' is issued, then system falls into some specific state". When we verify the design of product A, the environment model includes only "D-Sensor," and the state model for "D-Sensor" is developed so as to send the event sequences included in the above test scenario. On the other hand, when we verify the design of product B, the environment model includes not only "D-Sensor" but also "G-Sensor." In this case, the state model for "G-Sensor" sends arbitrary event sequences that may be shuffled with events from "D-Sensor."

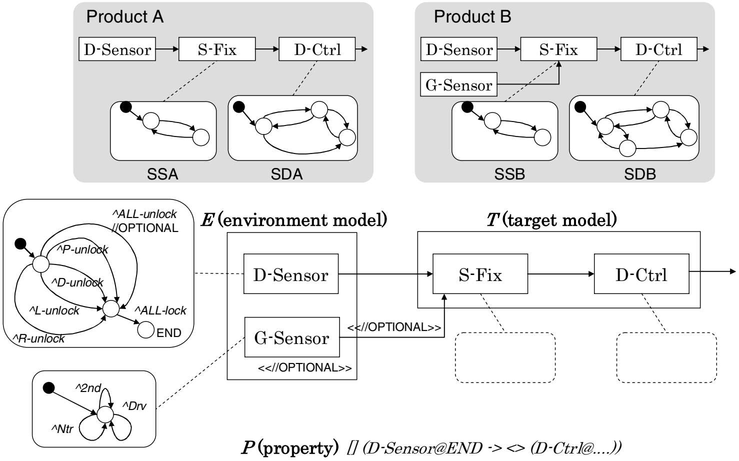


Fig. 3. Example of a Reusable Verification Model

The lower side of the figure shows a reusable verification model for both products. We do not use "G-Sensor" when we verify product A, but we use it when we verify product B. Therefore, "G-Sensor" in an environment model is defined as optional (denoted as "<<//OPTIONAL>>"). Further, each "D-Sensor" has a slightly different event set, and the state model of "D-Sensor" has optional transitions (denoted as "//OPTIONAL"). When we verify a product, we derive state models for target components from the design models ("SSA" and "SDA" for product A, and "SSB" and "SDB" for product B). In defining a reusable verification model for VLS, we used the following mechanisms:

- DA level: This level is related to the variations in components and connections defined in a static model.
  - Optional component
  - Optional connection

- CO level: This level is related to the variations in behavior of each component defined in the state model.
  - Alternative state model: prepare multiple state models for a component as variants. Strictly speaking, we define alternative components that have multiple components as variants; each has its own state model.
  - Optional transition
  - Alternative guard, conditions: prepare multiple guards/actions as variants.

We also define optional and alternative parts in the properties.

4 Organizing Core Assets

In this section, we show how to organize core assets and how to manage traceability among feature models, design models, and verification models.

4.1 Feature Model and Extended Design Model

In the feature model, we hierarchically depict the features of a product family. The features can be mandatory, optional, or alternative [8]. Fig. 4 shows a part of the feature model of VLS. In this case, "SENSOR" implies the abstraction of sensors, "CONTEXT" implies context judgment based on "SENSOR," and "PROCESS" implies functionality that is triggered or constrained by "CONTEXT" [9].

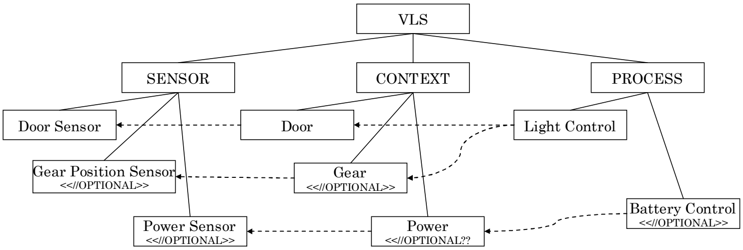


Fig. 4. Example of a VLS Feature Model

The design model is depicted in UML with some extensions in order to describe PLA; namely, we introduce an optional and alternative part into class and state diagrams. Fig. 5 shows a part of the design model of VLS. In this figure, optional components such as "B-Ctrl" (battery control) and "T-Ctrl" (timer control) are defined. "S-Fix" and "L-Ctrl" are defined as alternative components and have multiple variants. We can define a different state model for each variant and switch the behavior of alternative classes by selecting one of their variants. The notation of the state diagram is also extended so as to describe alternative transitions, as shown in Fig. 3.

The reusable verification model (target model *T* and environment model *E*) is depicted by using the same notation as the design model. The properties *P* are given by a textual description. Fig. 6 shows a part of the verification model of VLS. In thisverification model, sensors—such as "G-Sensor" and "D-Sensor"—belong to *E*, and other components belong to *T*. It should be noted that the configuration of target components is the same as the design model shown in Fig. 5, and the state models for these target components are derived from the design model. On the other hand, we define the state model of the environment components in order to send event sequences included in *{S}* of the corresponding test scenario.

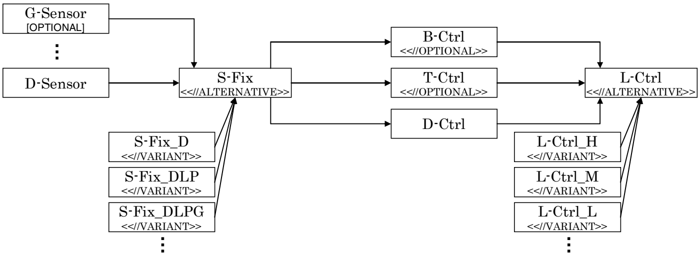


Fig. 5. Example of a VLS Design Model

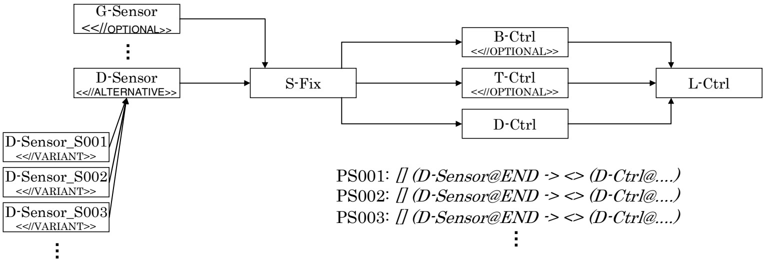


Fig. 6. Example of a VLS Verification Model

Since a target generally has multiple test scenarios, we prepare multiple verification models. Since the static structure of each verification model is generally the same, we can organize multiple verification models into one verification model utilizing the alternative notation. In Fig. 6, the environment component "D-sensor" is defined as an alternative component. Each variant (such as "D-Sensor_S001" and "D-Sensor_S002") corresponds to a different test scenario. Descriptions such as "PS001" and "PS002" are properties that correspond to a test scenario.

4.2 Traceability

We define the following links among the models explained in section 4.1 so as to systematically obtain the verification model for a specific test scenario (Fig. 7).

■ Product names and features: For each product in product families, define links between the product name and its features in a feature model. Using these links, we can identify the features of a product in a product family.

- Features and constituents of the design model and verification model: For each feature, define the links between the feature and components, connections, transitions/guards/actions in the design and verification models. Using these links, we can identify a design model for the product as well as a verification model with alternative parts (since a product generally has multiple test scenarios, the verification model for a product has multiple variants corresponding to them).
- Product and test scenario names: For each product, define links between the product and test scenario names. Using these links, we can identify test scenario names related to the product.
- Test scenario names and variants in the verification model: For each test scenario name, define links between it and the variants in the verification model. These links can be used to identify variants corresponding to the test scenario.

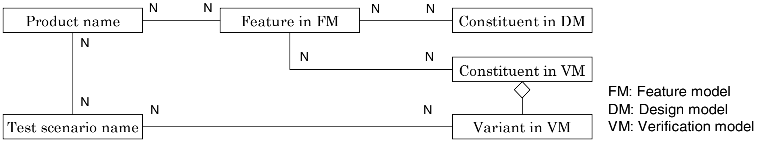


Fig. 7. Overview of Links among Models

5 Support Environment

In order to support the above design verification in PLD, we developed a support environment. This environment is developed on the Eclipse platform [4], and utilizes a UML plug-in [14] as the UML modeler and a SPIN model checker [6] as the model checking tool. The main features of the environment (Fig. 8) are as follows:

- Modeler: Define feature model, design model, and verification model along with their properties.
- Link Manager: Define links among models; further, identify the design model for a product and the corresponding verification model.
- Translator: Combine the target (defined in the design model) and verification models; subsequently, translate it into a format that can be understood by a model checking tool (Promela language for a SPIN model checker).
- Viewer: Show the verification result (counter example shown by the model checker) as a UML sequence diagram.

The translator merges the state models for target components defined in the design model with those of the verification model. For example, when we verify product A in Fig. 3, we assign “SSA” and “SDA” to the state models for “S-Fix” and “D-Ctrl,” respectively, in the target model *T*. We then translate it into Promela choosing the necessary information for design testing. A class diagram is translated into Promela using the rules in Table. 1

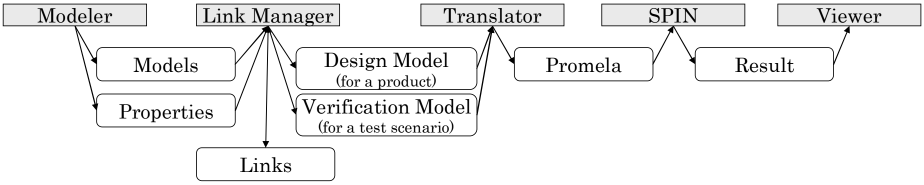


Fig. 8. Overview of the Support Environment

Table 1. Outline of Mapping Rules

Verification model	Promela
Class with stereotype <<process>>	Process that have the behavior (defined in the corresponding state model)
Attributes of class	Global variables
Association with stereotype <<channel>>	Channel exclusively used by two processes participating in the link
Association with stereotype <<shard_channel>>	Channel used by more than two processes

A state diagram is translated into the following Promela code segment, and it is embedded into the body of the corresponding process code.

```
<state_name>:
  <entry action>
  if
  :: <guard condition> ->
     <exit action>; goto <next_state_name>
  .....
  fi
```

Here, the state name is mapped onto the label, and each outgoing transition is mapped onto the selection construct (statement begin with "::") of an if-statement. This implies that each transition is selected non-deterministically, and the model checker checks every possible execution sequence.

Along with these mapping rules, we also extend the notation of LTL to enable the referring of UML identifiers in LTL; for example, we can refer to a class attribute as "<class name>[id].<attribute name>," execution of the first statement in a state of a class as "<class name>[id]@<state name>," execution of any statement in a state of a class as "<class name>[id]@@<state name>," etc. ("id" is an integer assigned to each instance of a class). In order to set the initial state *I* of the target, the environment model may be designed to send an event sequence to the target for initialization. However, this may complicate the environment model. Therefore, in our support environment, we developed a capability to directly set the value of each attribute and specify the initial state of each object.

Fig. 9 shows a snapshot of the support environment. The upper side shows the design model, the right side shows the translated Promela language, and the lower side shows the counter example presented in the sequence diagram.

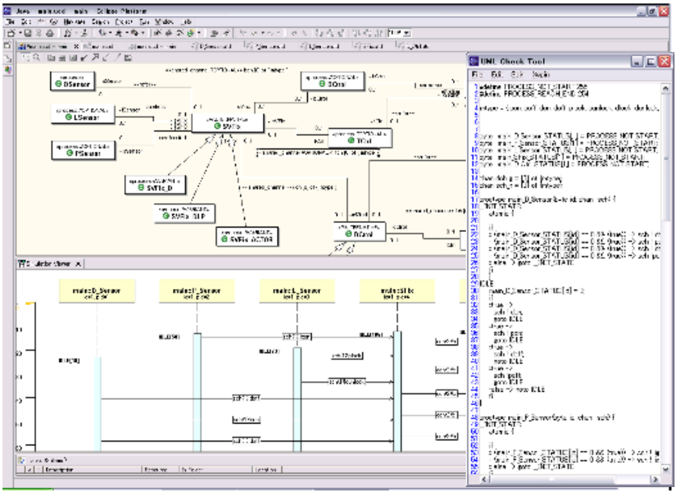


Fig. 9. Snapshot of the Support Environment

6 Evaluation

In this section, we evaluate our approach based on actual test cases obtained from the industry. Firstly, the extent to which our verification model is applicable to actual verification was evaluated; it was found to be as follows:

- By our design testing, we can verify 70% of the 113 test cases provided by the industry. We can partially check 7% of the test cases since they include real-time aspects. For example, suppose a test case—"A happens in 100 ms after B happens." In this test case, we can only check that "A happens after B happens"; however, we cannot check whether it happens in 100 ms.
- We can reduce 10% of the test cases since the verification model is defined in a general form (i.e., as a state model) and one state model in *E* can be used to check more than two test cases. Except for the initial states, 27% of the test cases are identical to other test cases; and they can be easily defined.

Secondly, we evaluated how well our verification model can be reused. We examined three types of products (say A, B, and C) and found the following:

- We can accommodate the variations in system configurations using the optional and alternative parts. Although each product has a different type and number of "SENSORS" and "CTRL," as explained in section 3.2, we can define a single reusable verification model by using optional and alternative components.
- Among the test cases that can be checked by our approach, 70% are reusable among three products; we can define a reusable verification model for them. Since 30% of the test cases are applicable only to product A, and we have to develop a verification model for product A alone.

7 Discussion

In this paper, we proposed design testing based on test scenarios utilizing model checking techniques. Since scenario-based testing is common in the development of reactive embedded systems, we believe that our approach is widely applicable.

We have examined the “light-weight” application of formal methods to design testing. We cannot rigorously prove the validity of the design; however, we can expect to check the validity more exhaustively as compared with conventional reviewing and ordinary testing. Applying model checking techniques to design testing based on test scenarios is not new, and there are similar approaches [12, 15]. However, the reuse of the verification model and organizing a reusable verification model along with the feature and design models are our contributions. PLD is a good application area for a formal approach because we can expect to reduce the cost by reusing the verification model throughout the development of the product family.

One of the problems of model checking techniques is state explosion. In order to avoid this, we have to adopt techniques such as design abstraction and assume guarantee techniques [5]. For design abstraction, we have defined a mapping rule from the design model to Promela. Although we prepared a problem-specific rule this time, we generally need multiple rules depending on the verification objectives. We could adopt some assume guarantee techniques in our scheme; however, we do not explicitly support it because the techniques still have various limits and constraints. Instead, we used conventional step-wise verification—verify a part of the design model by defining its environment as stubs in order to reduce the complexity.

In product line community, testing has become an important issue [7]. This is because even if products can be developed quickly, they cannot be released until they are efficiently tested. Although our design testing is intended to be used in the application engineering phase, other types of verification are required in the domain engineering phase, such as verification of the validity of the SA-level (shared architecture level) design. This is one topic for our future study.

8 Conclusion

In this paper, we examined the light-weight application of model checking techniques to design the testing for embedded software. In order to apply the design testing to PLD, we proposed a reusable verification model and a method of organizing them in core assets in order to enable their systematic reuse in the application engineering phase. We believe that PLD is one of the best application areas for formal methods, and our contribution is to show a framework for the application of formal methods in PLD. In future, we intend to increase the number of case studies, and enhance and refine the framework.

References

1. Bosch, J.: Design and Use of Software Architectures. Addison-Wesley, (2000)
2. Clarke, E., Grumberg, O., Peled, D.: Model Checking: MIT (1999)

3. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns, Addison-Wesley (2001)
4. http://www.eclipse.org/
5. Giannakopoulou, D., Pasareanu, C.S.: Assume-Guarantee Verification of Source Code with Design-Level Assumptions. 26th International Conference on Software Engineering (ICSE'04) (2004).
6. Holzmann, G.J.: The model checker SPIN. IEEE Trans. on Software Engineering **23 (5)** (1997) 279–295
7. Jamie, J. et al.: Test Case Management of Controls Product Line Points of Variability. International Workshop on Software Product Line Testing. (SPLiT 2004) (2004)
8. Kang, K. et al.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. CUM/SEI-90-TR-21 (1990)
9. Kishi, T., Noda, N.: Aspect-Oriented Context Modeling for Embedded Systems. Aspect-Oriented Requirements Engineering and Architecture Design (Early Aspects 2004) (2004)
10. Kishi, T. et al.: Project Report: High Reliable Object-Oriented Embedded Software Design. The 2nd IEEE Workshop on Software Technology for Embedded and Ubiquitous Computing Systems (WSTFEUS'04) (2004).
11. Kishi, T., Noda, N.: Design Testing for Product Line Development based on Test Scenarios. International Workshop on Software Product Line Testing, (SPLiT 2004), (2004)
12. Lilius, J., Paltor, I. P.: vUML: a Tool for Verifying UML Models. TUCS Technical Report No. 272 (1999)
13. Northrop, L.M.: SEI's Software Product Line Tenets. IEEE Software, **19(4)** (2002) 32–40,
14. http://www.eclipseuml.com/
15. Schäfer T., Knapp A., Merz, S.: Model Checking UML State Machines and Collaborations. Electronic Notes in Theoretical Computer Science **55**(3) (2001)

QFD-PPP: Product Line Portfolio Planning Using Quality Function Deployment

Andreas Helferich, Georg Herzwurm, and Sixten Schockert

Universität Stuttgart,
Chair of Information Systems II (Business Software),
Breitscheidstr. 2c, 70174 Stuttgart, Germany
{Helferich, Herzwurm, Schockert}@wi.uni-stuttgart.de

Abstract. In today's competitive business environment, it is extremely important to offer customers exactly the products they want. Software product lines have the potential to enable companies to offer a large variety of products while still being able to manage the complexity caused by this increased number of products. But offering a large range of variants does not necessarily mean increased profits, as many manufacturing companies had to notice in the early 1990ies. The task of Product Portfolio Planning is the development of a product portfolio that optimally satisfies customer demands and at the same time restricts the number of products offered. Quality Function Deployment (QFD) is a well-known and successfully used Quality Management method that can help companies to identify true customer needs and the features needed to fulfil these needs. This paper demonstrates how QFD can be used for Product Portfolio Planning, thus offering potentially great benefits.

1 Introduction

One of the main reasons for adopting Software Product Lines is the possibility of fast, economical and high quality development of new products (systems). Both time-to-market and maintenance effort are expected to decrease, while customer satisfaction is expected to increase since the software can be developed faster, in higher quality, and for more individual purposes [1]. But adopting a Software Product Line approach does not guarantee success: as many manufacturing companies learned in the 1990ies, offering too many products leads to substantial complexity costs, endangering profits [2]. A large part of these costs don't apply for software, since software is an intangible product. For example purchasing, handling or stocking raw materials, components or spare parts. But another part of the costs does apply for software: developing, deploying and maintaining assets, including testing, bug-fixing and upgrading systems once they are rolled-out. These tasks should not be understated. Especially in the domain of Information Systems, where one system hardly ever operates stand-alone but usually has to operate in an environment of varying combinations of hardware, operating systems, data bases, middleware, other software running on the same computer or on separate computers but exchanging data with each other. And the more components and/or products (as combinations of

H. Obbink and K. Pohl (Eds.): SPLC 2005, LNCS 3714, pp. 162 – 173, 2005.

components) a company offers, the more difficult this task will be. Software Product Lines with a well-defined architecture using commonality and variance well go a long way in reducing these problems, as do the right processes for configuration management, version management, requirements management and change management. Nonetheless, every additional product introduces some complexity and additional cost. Therefore, carefully planning and selecting the members of a product line is an important function.

But existing literature on Software Product Lines often treats the selection of products as input provided by the marketing department [3]. Unfortunately, research in both requirements engineering for software and product design for various kinds of products has shown that customers have huge difficulties in articulating their requirements. Therefore, we propose adapting the well-known Quality Management method Quality Function Deployment (QFD) for the use with Software Product Lines since this method has been successfully used to identify the true customer requirements in various industries, among them software [4].

Chapter two details why product portfolio planning for Software Product Lines is important. In chapter three, a brief introduction to QFD is given and the authors' new method for product portfolio planning using QFD is explained. Related work from the domains of Software Product Line Engineering and Quality Function Deployment is presented in chapter four, followed by the conclusions.

2 The Importance of Product Portfolio Planning

Product Portfolio Planning is a management activity closely associated with product development. Integrating information about technical innovations, market demand, cultural and legal developments, Product Portfolio Planning tries to develop a portfolio of products that optimally satisfies customer demands (thereby leading to increased sales) and at the same time restricts the number of products offered (thereby reducing costs and the risk of new products "cannibalising" old products' sales, i.e. customers buying the new product instead of an existing one). In an advanced stage, this includes planning for several product generations, taking into account technology S-curves and technology roadmaps [5].

For a (software) product line, product portfolio planning seeks to answer the following questions:

- Which products should be members of the product line?
- What technologies should members of the product line utilize?
- Which features/technologies should be common to all members of the product line?
- What should be the differences between members of the product line?
- In what direction should the product line and its members evolve?

From a business point of view, the answers are quite easy in theory: there should be as many different members of a product line as are necessary to satisfy the needs of the customers in the planned, profitable market segment. The common "core" consists of all features common to all members of the product line. The differences result directly from the different needs of different customers in this market segment. And the technology used is the one best satisfying customer needs (including the need "reasonable price").

In practice, none of these answers is easy, since customer needs are not easily identified and prioritized. The latter is necessary since some customer needs are conflicting, e.g. ease of use and a multitude of functions. Kano's *Attractive Quality Model* [6] provides some insight why even the customers themselves have problems stating their true needs. According to the model, customer needs can be classified into the three categories: *Must-be* or *Basic Attributes, One-dimensional* or *Performance Attributes,* and *Attractive* or *Exciting Attributes.* And according to Kano, only Performance attributes are voiced by the customer since he takes Basic Attributes for granted and Exciting Attributes are neither required nor expected by the customer. But nevertheless identifying and fulfilling the latter leads to great satisfaction and the willingness to pay a premium price. [7]. Finally, it is important to notice that customer expectations change over time and today's attractive attributes can be tomorrow's basic attributes [6].

Thus asking (potential) customers to fill out a questionnaire is not sufficient, rather it is important to get a deep understanding of customer needs and cross-check with technological opportunities [8]. Especially breakthrough innovations would never be developed if only explicit customer demands were taken into account since they result from exciting attributes.

Research on software requirements engineering has come to another conclusion: since software is immaterial in nature, customers have big difficulties expressing their expectations before using the final product [9].

Quality Function Deployment can be used to answer the questions that are part of Product Portfolio Planning and overcome the problems associated with identifying customer requirements for software, as will be shown in the following.

3 Product Portfolio Planning Using QFD

3.1 Quality Function Deployment

"QFD provides a systematic but more informal way of communication between customers and developers" [10] compared to traditional ways of formalizing and specifying product requirements. A project team consisting of customer representatives, developers/engineers and a moderator who is an expert in QFD works together during the whole QFD process. This is done in order to assure that the final product's features are not determined by the technically possible but by the fitness for use, i.e. the features the customers demand. The software developers and/or engineers assure that the features can be implemented and that technological breakthrough innovations are not ignored.

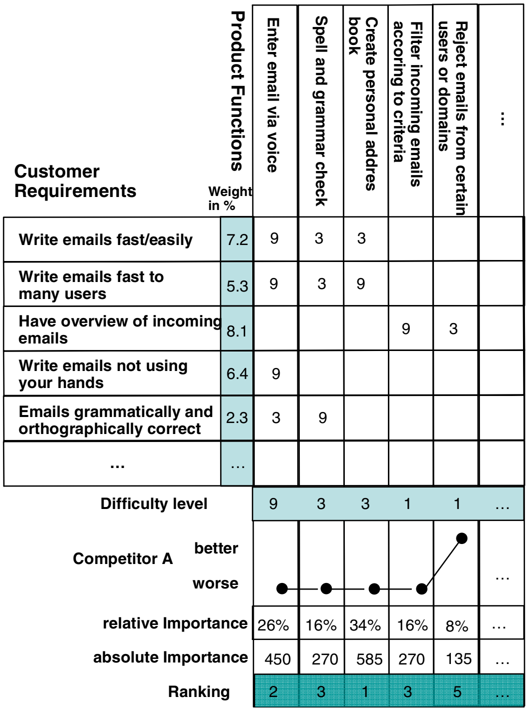
The best known instrument of QFD is the so-called House of Quality (HoQ). Generally speaking, the HoQ is the matrix which analyzes customer requirements in detail and translates them into the developers' language. The HoQ is the framework of most of the matrices used in QFD. For an in-depth description of QFD see [11].

QFD has been developed in the Japanese manufacturing industry [12], but can easily be adapted towards software development if two differences are considered: first, the software production process is basically a duplication process and implementation is largely determined by the system design, especially the system architecture. Therefore, the effort has to be directed mainly into the earlier stages. Secondly, "Software […] is valued not for what it is, but for what it does" [13]. Thus, the distinction between product function and quality element has to be made: a product function is a "functional characteristic feature of the product, usually not measurable (creates perceptible output)" [14], while a Quality Element is a "Non-functional characteristic feature of the product, possibly measurable during development and before delivery (does not create perceptible output)" [14]. The important first purpose of QFD in software engineering and the main focus of product planning is on setting prioritized development goals based on the most important customer requirements [14]. In planning software products the preference setting and focusing aspects of QFD by means of the HoQ are more important than the deployment by a matrix sequence. Applying QFD, however, takes more than filling out a HoQ matrix. A number of techniques (e. g. the Seven Management and Planning Tools and the Seven Quality Tools [14]) have to be combined in order to get all information that is necessary to form the matrices and to exhaust the potential of QFD as far as possible.

The entire QFD process is carried out by a QFD team with representatives of all departments (development, quality management, marketing, sales, service etc.) and is to be extended in several team meetings by the selected typical customer representatives. Substituting a customer survey, one of the first meetings tries to ascertain customer needs and to classify them in the Voice of the Customer Table. These requirements are structured using affinity- and tree diagrams and weighted (e. g. by pair-wise comparison or the Analytic Hierarchy Process [15]) by as many members of the customer groups as possible under control of the customer representatives. The weights of the different groups are then used to calculate the average weight by calculating the average of the weights assigned by the customer groups weighted with the importance of the groups.

If a new release of an existing product is developed, the customer representatives will evaluate them according to the level of satisfaction with the current fulfilment of the requirements (measured on a scale ranging from 1 indicating total dissatisfaction to 5 indicating perfect satisfaction). A (subjective) comparison with competitors at the requirements level is ineffective because customers cannot evaluate the competition's products as well. Thus, representatives of competing products' customers would have to be consulted for such a comparison to be effective.

The second major input is the Voice of the Engineer Table, compiled by the QFD team, among them particularly developers, that includes the potential product functions. The classic HoQ also uses measurable quality elements. These are derived from the requirements by the developers. The relationships between product functions and customer requirements in both prioritization matrices are identified together with the customer representatives. Analyzing the effects that one product function has on the other product functions leads to the roof of the HoQ [11]. Figure 1 displays an excerpt of a Software HoQ for an email-client including the tables of customer requirements and product functions.



Customer Requirements	Weight in %	Enter email via voice	Spell and grammar check	Create personal addres book	Filter incoming emails accoring to criteria	Reject emails from certain users or domains	...
Write emails fast/easily	7.2	9	3	3			
Write emails fast to many users	5.3	9	3	9			
Have overview of incoming emails	8.1				9	3	
Write emails not using your hands	6.4	9					
Emails grammatically and orthographically correct	2.3	3	9				
...	...						
Difficulty level		9	3	3	1	1	...
Competitor A better						●	
worse		●	●	●	●		...
relative Importance		26%	16%	34%	16%	8%	...
absolute Importance		450	270	585	270	135	...
Ranking		2	3	1	3	5	...

Fig. 1. Software-HoQ for an email client (adapted from [14])

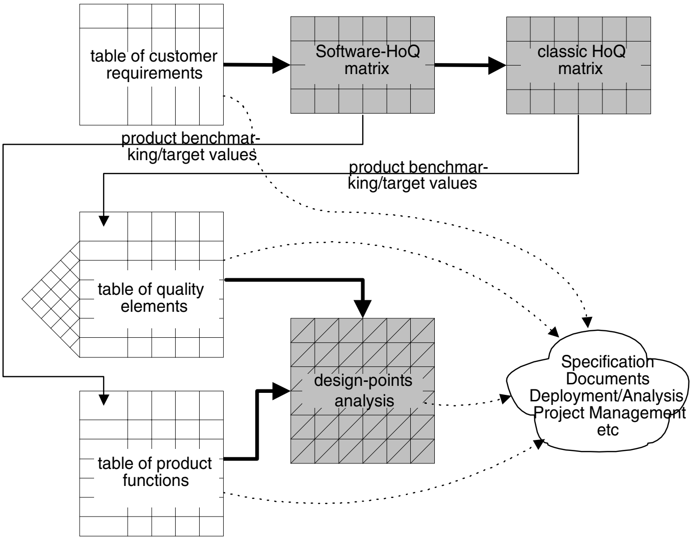


table of customer requirements
Software-HoQ matrix
classic HoQ matrix
product benchmar-king/target values
product benchmar-king/target values
table of quality elements
design-points analysis
table of product functions
Specification Documents Deployment/Analysis Project Management etc

Fig. 2. Herzwurm's and Schockert'S PriFo Software QFD model ([14], pg. 87)

Figure 2 gives an overview of the whole software development process using PriFo QFD. This approach has been used to develop the Calendar function in SAP R/3$^{©}$ [14]. A variation of PriFo QFD called Continuous QFD (C-QFD) using templates and iterative development cycles has been used for electronic and mobile business systems [16].

3.2 QFD-PPP

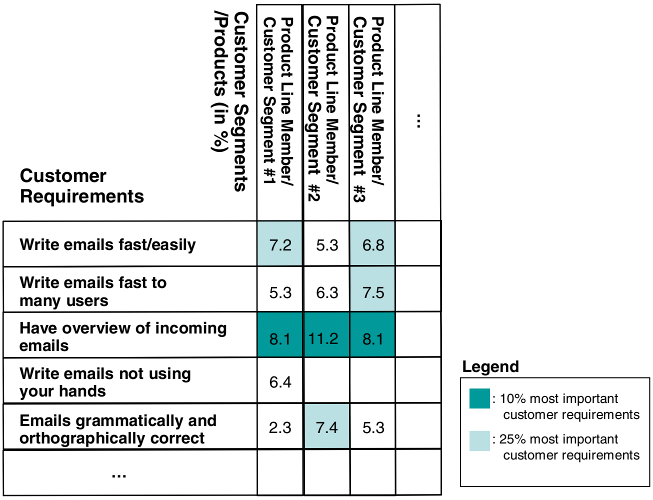
Our approach to Product Portfolio Planning makes extensive use of QFD while at the same time introducing two new matrices. First of all, the Voice of the Customer (VoC) is collected by asking existing and potential customers about the requirements they have for the product line. Once these answers are collected, they are analyzed and sorted before asking the customers to assign priorities to all requirements. Once these priorities are assigned, customer segments are derived based on these priorities using cluster analysis. Thus, unlike in PriFo QFD, there is no weighting of customer groups as this is only necessary to come up with common priorities. Another difference to PriFo QFD is the identification of customer groups not by attributes of the customer (e.g. job title or role description) but by statistical analysis.

The next step is to bring together developers, software architects and selected customers (based on the clusters identified) to build the Software House of Quality. Explicitly including the Voice of the Engineer in the form of product functions is important to identify exciting attributes according to the Kano model, i. e. software characteristics that customers themselves would not have come up with. Since a product function's level of fulfilling a customer requirement is independent from the weight assigned to the requirement, there is only one SW-HoQ for all the members of one product line. But since the weights of the customer requirements depend on the customer segments, the weight of the product functions does so either. The Software-HoQ in Figure 1 equals the Software-HoQ for one of the customer groups (including the weights), e.g. attorneys used to dictate letters who would therefore being able to dictate emails, too. The resulting matrix, including all customer requirements and customer segments, including the importance assigned to the requirements is shown in Figure 3.

As indicated in Figure 3, the members of the product line are identified using the simple rule *one member of the product line per customer segment*. Core and variable features are identified by comparing the weight of the product functions for the different customer segments. This is visualized in the second new matrix: product functions x members of the product line displayed in Figure 4.

The software developers and software architects perform the next step evaluating different software architectures and technologies taking into account necessary quality attributes and product functions. This is also done by using matrices (Classic HoQ for the quality attributes, Software HoQ for product functions), where the roof is intensively used to analyze the impact that different architectural or technological elements have on each other. The results of this analysis are used to decide on the software architecture and the technologies to be used for prototypes.

These prototypes are then presented to the customers, thereby demonstrating exciting features the software developers and software architects came up with and the proposed solutions to the requirements voiced by the customers. Showing all customers all prototypes, some of the customers will decide to include some features they previously hadn't assigned value to, maybe drop some features they requested.



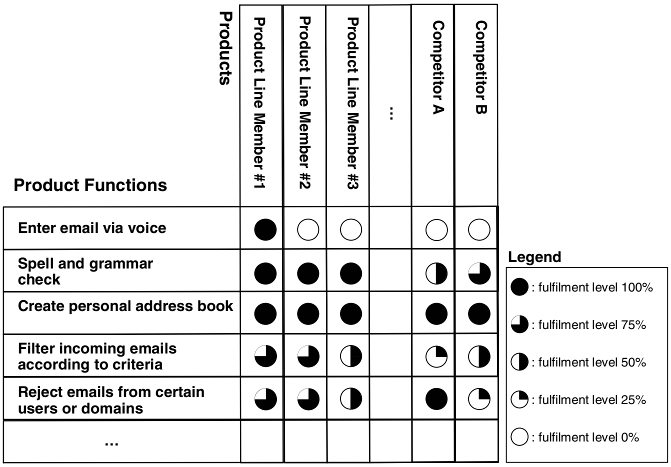
Customer Requirements \ Customer Segments /Products (in %)	Product Line Member/ Customer Segment #1	Product Line Member/ Customer Segment #2	Product Line Member/ Customer Segment #3	...
Write emails fast/easily	□ 7.2	5.3	□ 6.8	
Write emails fast to many users	5.3	6.3	□ 7.5	
Have overview of incoming emails	■ 8.1	■ 11.2	■ 8.1	
Write emails not using your hands	6.4			
Emails grammatically and orthographically correct	2.3	□ 7.4	5.3	
...				

Legend

■ : 10% most important customer requirements

□ : 25% most important customer requirements

Fig. 3. Matrix Customer Requirements x Customer Segments



Product Functions \ Products	Product Line Member #1	Product Line Member #2	Product Line Member #3	...	Competitor A	Competitor B
Enter email via voice	●	○	○		○	○
Spell and grammar check	●	●	●		◑	◕
Create personal address book	●	●	●		●	●
Filter incoming emails according to criteria	◕	◕	◑		◔	◑
Reject emails from certain users or domains	◕	◕	◑		●	◔
...						

Legend

● : fulfilment level 100%

◕ : fulfilment level 75%

◑ : fulfilment level 50%

◔ : fulfilment level 25%

○ : fulfilment level 0%

Fig. 4. Matrix Product Functions x Members of the Product Line

This discussion is based on the product functions, not the original customer requirements and their weights. Only when large changes are asked for the customer requirements will be re-evaluated.

The second new (matrix product functions x members of the product line) helps prioritizing the variants. Inputs are the expected costs for the product functions and the expected revenue a product will achieve. The second depends on the size of the potential market, the products currently available on the market and the customer satisfaction with these products and the advantage the member of the product line have over these products. Ulwick's so-called *opportunity algorithm* [17] or the algorithm used in [14] can be used as indicators here. Both algorithms use the importance of a feature and the customers' satisfaction with the current solutions provided by own and competitors' products to identify features where improvements provide a competitive advantage. A more detailed economic assessment is presented in [3] and [18]. Figure 5 gives an overview of this part of the process (for reasons of clarity, classic HoQ, design-point analysis and the integration with systems design and implementation are omitted).

Finally, derivation of new products for a Software Product Line and the evolution of the Software Product Lines and its members are facilitated, since the already existing matrices can be used as templates (a similar course of action for agile software development was proposed in [16]). Using the matrices as a starting point leads to reductions in both time-to-market and costs and helps achieving important goals associated with Software Product Lines.

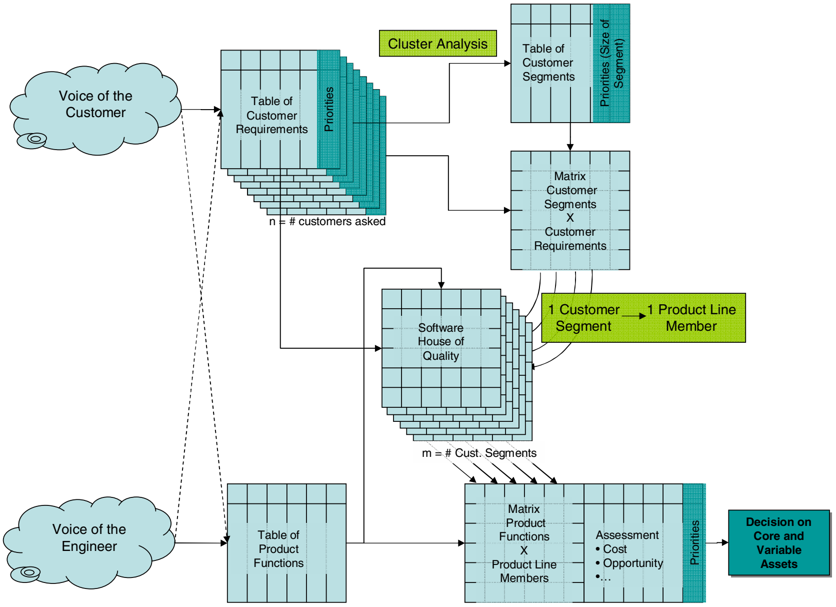

Cluster Analysis
Table of Customer Segments
Priorities (Size of Segment)
Voice of the Customer
Table of Customer Requirements
Priorities
n = # customers asked
Matrix Customer Segments X Customer Requirements
Software House of Quality
1 Customer Segment → 1 Product Line Member
m = # Cust. Segments
Voice of the Engineer
Table of Product Functions
Matrix Product Functions X Product Line Members
Assessment
• Cost
• Opportunity
•…
Priorities
Decision on Core and Variable Assets

Fig. 5. Overview of QFD-PPP (simplified)

4 Related Work

Related work can be classified into two categories: work from the Software Product Line Engineering domain and from the Quality Function Deployment domain. An important influence for QFD-PPP was PuLSE, a methodology developed at Fraunhofer IESE. Other Software Product Line approaches differ from PuLSE in the later stages, but their treatment of Product Portfolio Planning is similarly short, as [19] discovered while examining requirements engineering for product lines. PuLSE is briefly presented in 4.1.

Work from the QFD domain is related where product variants are taken into account. However, the literature on QFD for product variants is rather thin since QFD usually examines the development of one product and not a set of products. Nonetheless, it is possible to use QFD for product lines and existing approaches are presented in chapter 4.2.

4.1 PuLSE

PuLSE (Product Line Software Engineering) consists of several modules. Goal of PuLSE is "the conception and deployment of software product lines within a large variety of enterprise contexts" (cf. [20]). Product Portfolio Planning is considered part of Product Line Scoping which is defined as "the management activity that determines in which life-cycle (…) a certain functionality will be developed." [3]. Product Portfolio Scoping is one (and logically the first) of three kinds of scoping [3]: Product Portfolio Planning, Domain Scoping and Asset Scoping. The first deals with the definition of the products to be developed, i.e. definition which and how many products shall be developed and the functionality each of them shall have, but is explicitly treated as input [3]. But an activity called Product Line Mapping (PLM) is part of PuLSE. As Schmid points out, PLM is a technical activity, not a decision-making activity [3]. Nonetheless, important information is provided and analyzed during PLM: *genealogy charts* providing a quick overview of current and future members of the product line, and the so-called *product map*, providing a rather detailed view on the members of the product line, their features, competitor products, and models for analyzing the economic benefits of products or domains, thus aiding in prioritizing the development efforts. QFD-PPP basically adapts the product map and explicitly includes identifying customer segments and managerial decision-making. Thus an integration of QFD-PPP into PuLSE seems natural, thus integrating QFD-PPP in a well-documented and successfully applied methodology for the development of Software Product Lines.

4.2 QFD and Product Variation

There are a few examples in literature where QFD was used to define product variants. These will be presented in the following paragraphs, before explaining why these examples fall short of realizing the full potential of applying QFD for Software Product Lines.

Hoffmann and Berger [21] extend the House of Quality by using more than one target value per feature: they use specification classes (high, mid and low) for each product instead of one simple target value and indicate the evolution of the product using arrows (e.g. the lower class product starts with low value for feature F14, but the plan for the next generation is to improve F14 to medium value). Additionally, they include information on cost reduction potential and features offered by competitors. This approach is not suitable for a large number of products since it gets too complex. Also it is not clear how they distinguish between the needs of different customers or how they identify different customer groups.

Cheng et al. use QFD to derive a new product from an existing product platform as well as to develop a new product platform, and finally to differentiate common modules from variable module [22]. Their approach is primarily based on checking whether a certain feature is part of the core functionality or not, and close cooperation between Marketing, Sales and Engineering. While this approach stresses the need to cross-check customer input with technological input, identification of customer groups and their needs seems to depend on Marketing. Additionally, "real" (existing and potential) customers are not included in the cross-checking process. The results of their input are being filtered by Marketing and Sales.

Hunt and Walker [23] focus on what they call the *fuzzy front-end of strategy* i.e. the questions how to obtain a sustainable position in the market and which markets to operate in. They use QFD to gain a deep understanding of the marketplace, identify strategic outcomes (equivalent to customer requirements) and predictive metrics (equiv. to product functions) and identify what they call *natural segments*, i.e. customer segments that "share the same perceptions about outcomes, and more importantly who can be expected to prefer the same products or services…" [23]. Interesting about this method is the way they identify and use the *natural segments*: the identification is done using statistical clustering methods, to focus they concentrate on those outcomes that are at the same time important and where the customer satisfaction is currently rather low [17]. Positioning is then done by using the outcomes and calculated opportunity and taking into account competitors' positions. The link to identification of common and variable customer requirements is missing here, but the focus of this paper is strategy, not product development.

Fujita et al. [24] extend QFD with a so-called *variety table*, where the customer functions are further analyzed with regard to customer expectations for a high-class, a mid-market and a low-class model (small, medium and large refrigerators for the Japanese market). Thus, in the HoQ, the weights of the customer requirements are different according to the model, while the correlations between customer requirements and product functions are the same for all models. Product functions achieving a high score in fulfilling requirements that have no value for the low-class model but low scores in those requirements important for that model, are identified as variable requirements only needed for the high-class and maybe – depending on the importance of the respective requirements for the mid-market model – for the mid-market model. Additionally, they present a way to perform cost-worth analysis. This method simplifies the question of product portfolio by defining the models first (in the given example to three models, but theoretically, the number could be higher) and then assigning the necessary requirements to the models. But the underlying idea of the importance of a certain requirement depending on the customer (segment) is important.

5 Conclusions

It has been demonstrated how QFD-PPP can be used to identify different customer groups and their needs, to derive a product portfolio (i.e. members of a product line) systematically and derive common and variable product functions including exciting requirements that the customers would not have come up with. Thus, QFD-PPP has the potential for increased customer-orientation and at the same time higher profits since only products that are demanded by the customer and profitable are developed. Also some of the more operational challenges in Software Product Line Engineering can be tackled using QFD: von der Maßen et al. identify challenges in the categories *Organization and Management, Requirements Engineering, Product- vs. platform-specific* and *Architecture* [25]. Some of these problems, most notably "high communication overhead", "Discussions on design and not on requirements level" and "No explicit prioritization of requirements" can easily be solved using QFD (see. [14] for problems in Requirement Engineering and solutions provided by Software QFD).

Validation of this approach in industrial projects is still lacking, especially the integration into process models for Software Product Line Engineering. Also required is further research into the clustering algorithms to be used and into the integration of the QFD results towards later phases of the Software Product Line Engineering Process (for a method integrating QFD and object-oriented programming see [26]). As for Software Product Line Engineering in general, tool support is lacking.

References

1. Böllert, K. „Objektorientierte Entwicklung von Software-Produktlinien zur Serienfertigung von Software-Systemen“. Dissertation, TU Illmenau (2002).
2. Kaplan, R. S. "New Roles for Management Accountants". In: Journal of Cost Management, Fall 1995, pp. 6 – 13.
3. Schmid, K.: "Planning Software Reuse – A Disciplined Scoping Approach for Software Product Lines". Fraunhofer IRB, Stuttgart (2003).
4. Chan, L.-W. and M.-L.Wu. "Quality function deployment - A literature review." In: European Journal of Operational Research 143 (2002) 463–497
5. Ulrich, K.T. and S.D. Eppinger. Product design and development - 2nd ed.. Irwin McGraw-Hill, Boston (2000).
6. Kano, N., Seraku, N., Takahashi, F. and S.Tsuji. "Attractive quality and must-be quality." In: The best on quality, edited by John D. Hromi. Volume 7 of the BookSeries of the International Academy for Quality. Milwaukee:ASQC Quality Press (1986).
7. Sauerwein, E.; Bailom, F.; Matzler, K. and H. H.Hinterhuber. "The Kano Model: How to delight your customers". In: Preprints Volume I of the IX. International Working Seminar on Production Economics, Innsbruck/Igls/Austria (1996), pp. 313 -327.
8. Aasland, K.; Blankenburg, D.; and J. Reitan. „Customer and market input for product program development“. In: Proc. of the 6th Int. Symposium on QFD, Novi Sad, USA (2000).
9. Nuseibeh, B. and S.Easterbrook "Requirements Engineering: A Roadmap". In: Proc. of the Conference on The Future of Software Engineering, Limerick, Ireland (2000), pp. 35 – 46.

10. Herzwurm. G.; Schockert, S. and W. Pietsch: "QFD for Customer-Focused Requirements Engineering". In: Proceedings of the 11th IEEE International Requirements Engineering Conference, Monterey Bay, USA (2003), pp. 330-338.
11. Cohen, L.: "Quality Function Deployment", Addison-Wesley, Reading (1995).
12. Akao, J.: Quality Function Deployment: Integrating Customer Requirements into Product Design, Translated by Glenn H. Mazur and Japan Business Consultants, Ltd. Cambridge, Massachusetts (1990).
13. Zultner, R. E.: "Software quality function deployment – the North American experience". In: Software Quality Concern for people. Proceedings of the Fourth European Conference on Software Quality, Zürich (1994), pp. 143-158.
14. Herzwurm, G.; Schockert, S. and W. Mellis: Joint requirements engineering: QFD for rapid customer-focused software and Internet-development, Vieweg, Wiesbaden, Germany(2000).
15. Saaty, T. L.: Decision Making for Leaders: The Analytic Hierarchy Process for Decisions in a Complex World., 3rd edition, Pittsburgh, USA (1995).
16. Herzwurm, G.; Schockert, S.; Breidung, M. and U. Dowie: "Requirements Engineering for Mobile-Commerce Applications", in: In: Proceedings „M-Business 2002", Athens, Greece.
17. Ulwick, A.W.: "Turn Customer Input into Innovation", Harvard Business Review , 80(1) pp. 91-97 (2002).
18. Clements, P.C.; McGregor J.D. and S.G. Cohen: "The Structured Intuitive Model for Product Line Economics (SIMPLE)".Technical Report CMU/SEI-2005-TR-003 (2005)
19. Kuloor C. and A. Eberlein: "Requirements Engineering for Software Product Lines". In: Proceedings of the 15th International Conference on Software & Systems Engineering and their Applications (ICSSEA'02), Paris (2002).
20. Bayer, J. et al.: "PuLSE: A Methodology to Develop Software Product Lines". In: Proceedings of the 5th Symposium on Software Reusability, pages 122-131, 1999.
21. Hoffmann, J. and S. Berger: "Strategic Product Family Development by Extending the House of Quality". In: Proc. of the 6th Int. Symposium on QFD, Novi Sad, USA (2000).
22. Cheng, L.C.; Pfeilsticker, B.A. and F. de Aguiar Araujo: "An Application of QFD Method to Strengthen Product Development System of a Small Initiating Firm in Internet Mobile Technology". In: Proc. 8th Int. Symposium on QFD, Munich, Germany (2002), pp. 193-206.
23. Hunt, R.A. and M. Walker: Customer driven Strategy: Solving the Fuzy Front-End. In: Proceedings of the 9th Int. Symposium on QFD, Orlando, USA (2003), pp. 199-210.
24. Fujita, K.; Takagi, H. and T. Nakayama: "Assessment method for value distribution for product family deployment". In: Proc. Int. Conference on Engineering Design, Stockholm, Sweden (2003).
25. von der Maßen, T. et al.: Key challenges in Industrial Product Line Engineering. In (Adelsberger, H.H. et al., ed.): Multikonferenz Wirtschaftsinformatik 2004 Band 1, Akademische Verlagsgesellschaft Aka GmbH, Berlin (2004), pp.260-272.
26. Herzwurm. G.; Schockert, S. and S. Friebel,: "Quality Function Deployment object-oriented – a method for the combination of Quality Function Deployment and object-oriented modelling". In: Proceedings of the 10Th International Symposium on QFD, Monterrey, Mexico (2004).

Product-Line Architecture: New Issues for Evaluation

Leire Etxeberria and Goiuria Sagardui

Computer Science Department,
University of Mondragon,
Loramendi 4, 20500, Mondragon, Spain
{letxeberria, gsagardui}@eps.mondragon.edu

Abstract. In the product-line context, where a lack or mismatch in a quality attribute is potentially replicated among all products, product-line evaluation could detect problems before concrete products are developed. The life span of a software product-line architecture is much longer than the one of an ordinary software product and it serves as a basis for a set of related systems. Therefore, the product-line architecture should be adaptable to evolution as well as support a number of different products. All these characteristics set new requirements to the product-line architecture evaluation. This paper highlights the new issues that can arise when evaluating a product-line architecture versus evaluating a single-system architecture, including classifications of relevant attributes in product-line architecture evaluation, new evaluation moments and techniques. These issues are used as components of a framework to survey product-line architecture evaluation methods and metrics.

1 Introduction

The software architecture has a great influence on the system's final quality as it can inhibit or enable product's quality attributes. To be able to analyse the potential of an architecture to reach the required quality levels helps to find the problems early in the life cycle, when they are easier and cheaper to correct than in later stages such as implementation, testing or deployment. Besides, software architecture evaluation is helpful to improve the communication between stakeholders, improve documentation and prioritise quality goals, among others.

The evaluation of an architecture is defined as "the systematic examination of the extent to which an architecture fulfils requirements" [1]. The requirements can be functional or quality attributes but as architecture's influence on functional requirements is not so pronounced, almost all the evaluation methods focus on quality attributes. There are two broad categories of quality attributes [2][3]: **Observable via execution** or **operational** such as performance, security, availability, usability… and **not observable via execution** or **development attributes** such as modifiability, portability, reusability, integrability, testability…

In the case of product-line architectures (PLAs) the architecture assessment becomes crucial to ensure that the PLA is flexible enough to support different products

H. Obbink and K. Pohl (Eds.): SPLC 2005, LNCS 3714, pp. 174 – 185, 2005.

and to allow evolution. Another product-line specific characteristic is that there are two level of architectural abstraction where an evaluation can be performed (software product-line architecture and derived product architectures). To assess all the instances of the product-line may not be worthwhile due to the high cost. However, it is possible to shorten product architecture evaluations because the product architecture evaluation is a variation of the product-line architecture evaluation as the product architecture is a variation of the product-line architecture [4].

Organizational factors can also influence product-line architecture evaluations: a PLA involves more stakeholders than a single system because the scope is much larger that the one of a single-product architecture. Furthermore, stakeholders may be from different departments in different cities and even different countries.

The classical application of architecture evaluation occurs when the software architecture or product-line architecture has been specified but before implementation begins. However, architecture evaluation can be applied at any stage of an architecture's lifetime and particularly in product-line context new evaluation moments arise.

All these aspects of PLA must be considered when assessing and therefore they pose several challenges for existing evaluation approaches and techniques. In this paper a classification framework based on these aspects is proposed in order to classify product-line specific architecture evaluation methods.

The remaining of the paper is organized as follows. Section 2 introduces the attributes that are relevant in PLA evaluation. Then Section 3 introduces the evaluation moments in a product-line context, PLA evaluation techniques are shown in Section 4 and a classification of PLA evaluation methods and metrics in Section 5. And to conclude Related work in Section 6 and Conclusions in Section 7.

2 Relevant Attributes in Product-Line Architecture Evaluation

In a product-line there are two levels of architectural abstraction where it is necessary to perform an evaluation. Product-line or reference architecture[1] is the basis to assess family-specific aspects whereas concrete architecture provides the base to assess instance-specific aspects. With regard to **instance-specific** aspects, instance architectures should be evaluated to make sure they meet the specific behaviour and quality requirements of the product at hand [4].

With regard to **family-specific** aspects, the flexibility of the PLA should be evaluated to ensure it could serve as the basis for all the products of the family. It is also necessary to assess whether the PLA is able to address future requirements and products, that is, assessment of the modifiability and evolution of the PLA. The family-specific aspects ensure that the PLA addresses the required variation to get all the products as well as the variation that will require over time.

Attributes in a reference architecture can be classified in three different types: Product-line quality attributes, domain-relevant attributes and functional requirements or common behaviour (see Fig. 1).

[1] In the context of this paper, the terms reference architecture and product-line architecture are used interchangeably.

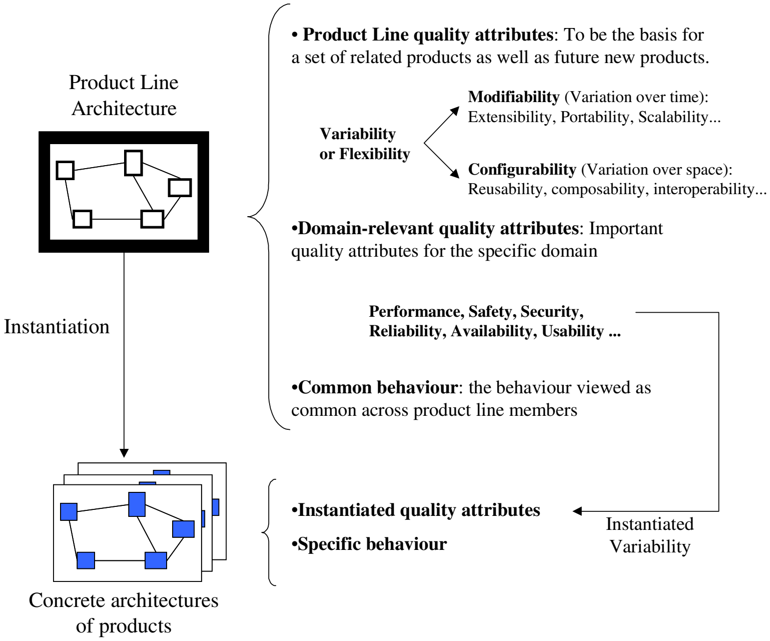

Product Line Architecture
• Product Line quality attributes: To be the basis for a set of related products as well as future new products.
Variability or Flexibility
Modifiability (Variation over time): Extensibility, Portability, Scalability...
Configurability (Variation over space): Reusability, composability, interoperability...
•Domain-relevant quality attributes: Important quality attributes for the specific domain
Performance, Safety, Security, Reliability, Availability, Usability ...
Instantiation
•Common behaviour: the behaviour viewed as common across product line members
•Instantiated quality attributes
•Specific behaviour
Instantiated Variability
Concrete architectures of products

Fig. 1. Classification of product-line requirements

Product-line quality attributes are those that are inherent or specific to product-lines to allow the architecture to be the basis for a set of related products as well as future new products. These attributes are the ones related to variability or flexibility. Assessing the variability of a PLA ensures that using the product-line architecture is possible to get all the functionality of the products in the envisioned scope. Variability [5], understood as modifiability (to allow variation or evolution over time) and configurability (variability in the product space) to get a set of related products.

Domain-relevant quality attributes (such as safety in safety-critical domain, performance in real-time domain, reliability in embedded systems, etc.) should be addressed in the PLA otherwise the implications or consequences can be very serious and difficult to fix. As different products in the domain can require different values in the attributes (not all products require the same level of security…), variability in the way the attribute is translated to the product is relevant for the assessment to assure that the realization of all the quality attributes for all the products in the product-line scope is possible with the product-line architecture.

Although many authors do no consider the **functional requirements** when evaluating software architectures, we reckon that in product-line architecture evaluation should be considered because a mismatch or error in a common behaviour may be reproduced in all the products of the line.

3 Evaluation Time

In traditional development, software architecture evaluation occurs usually during design. In a product-line context, the evaluation of the architecture can be useful in different moments (see Fig.2).

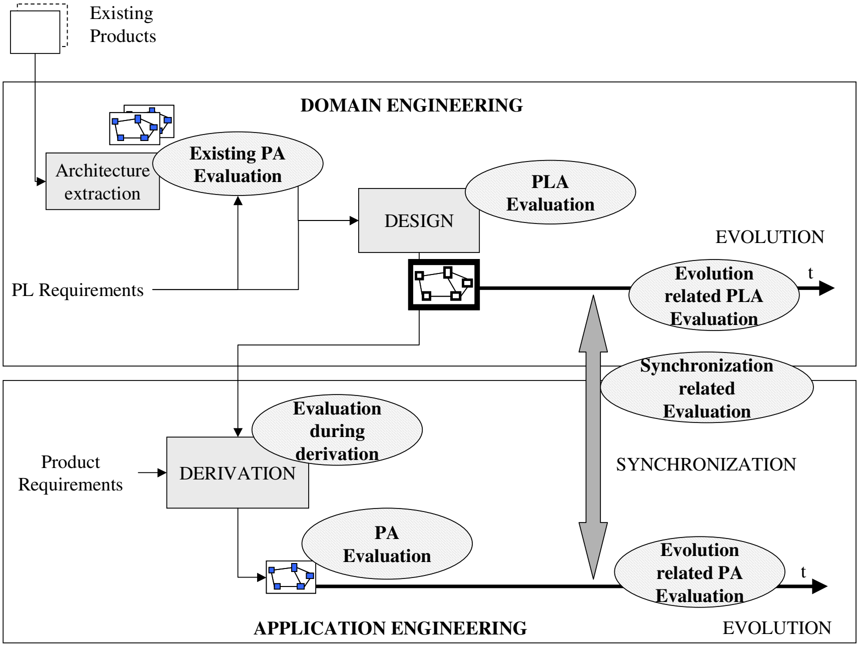


Fig. 2. Evaluation moments in a product-line context

Usually evaluation of architectures takes place *during architecture design*. In product-lines, this evaluation is replicated both at domain engineering and at application engineering. At domain engineering, evaluation (**PLA Evaluation)** assures reference architecture compliance to product-line quality attributes, domain-relevant quality attributes and common behaviour. It can be very useful to detect problematic issues and risks points or compare software architecture candidates to select the one that supports best the required quality attributes. At application engineering **(Product Architecture** or **PA Evaluation**), evaluation assures instantiated quality attributes and product specific behaviour, as well as architectural conformance to the reference architecture.

During evolution, it can be necessary or desirable to adapt the reference architecture to include new requirements (due to new products or new product requirements). Evaluation helps analysing the magnitude of the required architecture change (**Evolution related PLA Evaluation)** and thus deciding whether the requirements are considered at product-line level. Also during evolution, PLAs and PAs evolve from their initial design and the changes could provoke quality attributes not to be supported any longer. Evaluation in this case, is useful to assure that the architecture continues meeting its quality goals (**Evolution related PLA Evaluation** and **Evolution related PA Evaluation**). Evolution related architecture evaluation is much related to architecture recovering because sometimes an up-to-date architecture description is no longer available.

Specific to product-lines, three new evaluation moments arise: *before developing the reference architecture* (**Existing product architecture Evaluation**) in order to analyse and compare existing product architectures to use them as a basis for the product-line.

During product instantiation (**Evaluation during derivation**) to compare alternative variants that affect quality attributes. Product derivation consists on constructing individual products using a subset of the shared software artefacts [6] and during this process it is necessary to take architectural decisions that can affect the quality attributes of the product.

And *during architecture update* (**Synchronization related Evaluation**) to identify how the modifications and maintenance of a PLA affect the products already on the market and the opposite, how changes in the products affect PLA with the goal of maintaining the coherence between PLA and concrete architectures (Assess the impact of new requirements in products, detect variability that is not longer necessary…). This kind of evaluation is very related to evolution and derivation.

It is not cost-effective to evaluate the architecture in each of the moments. The architect must select the most appropriate moments to evaluate depending on the case. For instance in the case of an existing product-line, PLA evaluation during design or evaluation before developing are not applicable but evolution related PLA evaluation may be very interesting.

4 Product-Line Architecture Evaluation Techniques

Software architecture evaluation techniques are categorized in two groups [7]: Questioning techniques (qualitative evaluation) and Measuring techniques (quantitative evaluation). Questioning techniques include scenarios, questionnaires and checklists. Measuring techniques include simulations, prototypes, experiments and mathematical models (Metrics, RMA…). Questioning techniques can be used to evaluate any operational or development quality whereas measuring techniques address specific qualities, usually operational ones.

Product-line quality attributes are considered not operational or development attributes, so the evaluation is generally performed qualitatively. Most methods and experiences are based on scenarios. Scenarios concretise the quality attributes that are abstract into context dependent situations. This allows defining potential modifications to the product-line architecture and analysing the extensibility, portability, etc. Although scenarios are very used in product-line context, the only method that provides guidelines to adapt them to product-lines is D-SAAM [8]. This method reduces the gap between the reference architecture (abstract) and the scenarios (concrete), introducing two types of direct scenarios: concrete and floating. Concrete scenarios are scenarios that can be realized without changing the PLA and for which there are guidelines. Whereas floating scenarios can be realized by the derived products but there are not guidelines about how to do it.

Although qualitative evaluation is more frequent, there are also some metrics defined to assess product-line quality attributes: the service utilization metrics [9] which can be used to assess and improve product-line architectures and Rahman's metrics [10], a set of metrics for the structural assessment of product-line architectures, adapted from component based measures.

Evaluation of *domain relevant quality attributes* can be performed via scenarios but also some quantitative techniques (metrics, mathematical models, prototyping…) are available. In real-time domain, Alonso et al [11] use RMA models to assess the timing properties of new products of a family. In embedded system domain, Auerswald et al [12] present a method that performs qualitative as well as quantitative evaluation of reliability. Zhang et al [13] propose a method to capture and analyse the impact of variants on quality attributes using a Bayesian Belief Network (BBN). De Lange and Kang [14] propose a product-line architecture prototyping approach using PCs and networks to assess issues such as complexity, performance requirements...

Some communities have developed specific techniques and methods to assess their quality attributes: performance, safety, reliability… but these approaches are not specific for product-lines, so adaptation is needed.

For *functional requirement* evaluation other techniques are used: Model Checking, Theorem Proving, Proof Checking, Equivalence checking... Functional requirements or common behaviour can be analysed with automated tool support whereas product-line quality attributes are best supported by manual analysis techniques [15].

Due to the relevance of the variability in the product-line context, it is important to mention that there exist techniques for identifying and studying variation points, variants and dependences.

5 Classification of Architecture Evaluation Methods and Metrics

Previous sections have analysed the different aspects of product-line architecture evaluation. This section classifies architecture evaluation methods and approaches that are specific for product-lines according to these aspects. This classification can be very useful in order to select an appropriate evaluation method for the selected quality goals and a determined phase.

For evaluating product-line architectures *in design phase* there are different methods: FAAM (Family Architecture Assessment Method) [1] for evaluating information-system family's architectures, AQA (Architecture Quality Analysis) [16] for analysing product-line architectures, REDA[2] (Reliability Evaluation of Domain Architectures) [12] for analysing the reliability of a PLA and D-SAAM (Distributed SAAM) [8], a variant of SAAM for evaluating reference architectures. For evaluating *existing product-line architectures*: Gannod and Lutz [15] propose an approach that evaluates quality and functional requirements, Maccari [17] proposes a method to assess for evolution and Riva and Rosso [18] adapt Maccari's approach.

There are some methods that *assess variability*, one of the key aspects in product-lines, at architectural-level: SBA (Scenario-Based Architecting) [19] is a method for identifying and quantifying the potential benefits of the different architectural variability

[2] This abbreviated name is not original, it is used for convenience.

options. And at all layers of abstraction and not only at the software architecture: Wijnstra's approach [20] and COSVAM (The COVAMOF Software Variability Assessment Method) [21].

There are also methods oriented to evaluate *existing product architectures* to use them as basis for the product-line: SACAM (Software Architecture Comparison Analysis Method) [22] which is a method to compare architectures and Korhonen's approach [23] which analyse whether or not an architecture can be used as a basis for a product-line.

To evaluate *instantiated product architectures* there are two methods: TPA^2 (Timing Property Assessment) [11] which reuses the RMA models of individual components to derive the global RMA model of the system and Zhang et al's method [13] that is used during derivation to analyse the impact of variants on quality attributes using a Bayesian Belief Network (BBN).

There are also specific *metrics* defined for PLAs: service utilization metrics [9] and Rahman's metrics [10].

In the Table 1 these approaches are classified using previously identified aspects: attributes that they evaluate, evaluation phase when they can be used, evaluation techniques and some more general issues: process description, existing validation or case studies and relationship with other methods.

There are some very used architecture evaluation methods that can be also used to evaluate product-line architectures. SAAM (Software Architecture Analysis Method) [24] and its variants. And the successor of SAAM: ATAM (Architecture Trade-off Analysis Method) [24]. These methods are not product-line specific, they are used for evaluating single-product architectures but they are very adequate to address qualities that are *product-line quality attributes* such as maintainability and extensibility among others. ATAM has been used in product-line context [25][26] although there is not any special treatment in ATAM for product-line architectures; in these case studies the product-line particular aspects are addressed implicitly as quality requirements.

6 Related Work

There are some wider evaluation approaches such as [27] that defines a product-line evaluation framework with four dimensions: BAPO (Business, Architecture, Process, Organization) for determining the status of product-line engineering. Or FAE (Family Architecture Evaluation) method [28] that is used to benchmark product-line architectures. This approach not only considers quality attributes but other aspects such as the relation between architecture and business, context, domain knowledge, etc.

Evaluation process usually goes inside a more general process or method of design. There are quite a lot specific methods for designing product-line architectures but not all include a specific architecture evaluation phase in their method. Some of the methods that address evaluation are: QADA [16] (AQA is part of this method), QUASAR [29], QASAR [3], PuLSE-DSSA [30] and SEI's PL initiative [4].

Table 1. Classification of evaluation methods and metrics

Evaluation Method	**Goal**	**Attribute Types**	**Evaluation Phase**	**Evaluation Techniques**	**Process Description**	**Method's validation**	**Relation with other methods**
FAAM (Family Architecture Assessment Method) [1]	Stakeholder oriented assessment of information-system family's architectures	**PL qualities**: Interoperability, extensibility…	PLA evaluation	Scenarios, other techniques	Very detailed: Steps, guidelines, roles…	2 case studies in different domains	Extends SAAM
AQA (Architecture Quality Analysis) [16][31]	Analyse quality attributes of PLA	**PL qualities**: Reusability, modifiability… **Domain qualities**: Performance, reliability…	PLA evaluation, Evolution related PLA evaluation	Scenarios, knowledge base, customer value analysis	Well explained: Steps, guidelines…	2 Case studies in different domains	-
REDA (Reliability Evaluation of Domain Architectures) [12]	Evaluate PLAs to predict reliability	**Domain qualities**: Reliability	PLA evaluation	Failure cases, qualitative reliability model (QlRM), metrics…	Reasonable: Steps, techniques…	Case study in automotive control systems	-
D-SAAM (Distributed SAAM) [8]	Evaluate reference architectures reducing the organisational impact	**PL qualities**: Maintainability	PLA evaluation, Evolution related PLA evaluation	Scenarios	Well explained: Steps, guidelines, roles…	Applied on a copier systems PLA	Variant of SAAM
Gannod and Lutz's approach [15]	Analyse an existing PLA	**PL qualities**: Modifiability **Common behaviour**	Evolution related PLA evaluation	Scenarios and model checking	Reasonable: Steps, guidelines…	Applied on a telescopes PL	Includes a step similar to SAAM
Maccari's approach [17]	Assess the capability of a PLA to adapt to evolution	**PL qualities**: Evolution related ones: Scalability, modifiability…	Evolution related PLA evaluation	Scenarios	Briefly explained but illustrated through case studies	2 case studies in different domains	-
Riva and Rosso's approach [18]	Assess PLAs for evolution	**PL qualities**: Flexibility, modifiability	Evolution related PLA evaluation	Scenarios, experience-based analysis	Explained with a case study	Case study in a mobile terminals PLA	Adapts Maccari's approach
SBA (Scenario-Based Architecting) [19][32]	Identify and quantify the benefits of different variability options	**PL qualities**: Variability	PLA evaluation	Scenario-based quantitative analysis	Well explained: Steps, guidelines…	2 case studies of the medical domain	Uses SQUASH [33]

Table 1. (cont.) Classification of evaluation methods and metrics

Evaluation Method	Goal	Attribute Types	Evaluation Phase	Evaluation Techniques	Process Description	Method's validation	Relation with other methods
COSVAM (The COVAMOF Software Variability Assessment Method) [21]	Evaluate the variability of a PL in a evolution context	**PL qualities**: Variability	Evolution related PLA evaluation, Evaluation during derivation, Synchronization related evaluation	Product scenarios, expert-based analysis…	Well explained: Steps, guidelines…	Applied on an intelligent traffic systems PL	-
Wijnstra's approach [20]	Assess a PL for the way it deals with variation	**PL qualities**: Variability	Evolution related PLA evaluation	Study the gathered information	An overview	Case study in the medical domain	-
SACAM (Software Architecture Comparison Analysis Method) [22]	Compare candidate architectures (existing product architectures)	**PL qualities** **Domain qualities**	Existing product architecture evaluation	Scenarios, tactics and metrics	Detailed explanation: Steps, guidelines, participants…	An example to illustrate the method	-
Korhonen's approach [23]	Assess system adaptability to a product family	**PL qualities**: Adaptability, configurability… **Domain qualities**: Reliability, performance…	Existing product architecture evaluation	Scenarios	Explained with a case study	Applied on a case study of mobile machines	Loosely based on SAAM and ATAM
TPA (Timing Property Assessment) [11]	Analyse the timing properties of new products of a line	**Instantiated qualities**: Timing properties	PA evaluation	RMA	General description	A small example to illustrate the method	-
Zhang et al's approach [13][34]	Evaluate different architectural options of a PLA	**Instantiated qualities**: Scalability, Performance, security…	Evaluation during derivation	BBN (Bayesian Belief network)	Explained with examples	2 examples to illustrate the method	-
Service Utilization metrics [9]	Assess and improve PLAs	**PL qualities**: Structural soundness	PLA evaluation, Evolution related PLA evaluation, Evaluation during derivation	Metrics	Comprehensively explained	Case study in a digital library PLA	-
Rahman's metrics [10]	Measure the quality attributes of a PLA	**PL qualities**: Reusability, modularity	PLA evaluation, Evolution related PLA evaluation, Evaluation during derivation	Metrics	Reasonable	Case study in a library system	Include Service Utilization metrics

There are also architecture recovery or mining methods that include an architecture evaluation phase to analyse if the architecture can be the basis for a product-line: Pinzger et al's recovery method [35] which has an step to analyse and compare recovered architectures and MAP (Mining Architectures for Product lines) [36] that mines and evaluates product architectures.

Related to product-line architecture evaluation, Bass et al [37] compare the cost of variability decisions during architectural design. The cost is also a quality attribute but related to business attributes that are not explicitly discussed in this paper.

7 Conclusions

At product-line architecture level, evaluation becomes more important than in single systems because an error in the PLA can be spread into a lot of products. However, the evaluation is more difficult because the level of abstraction is higher.

Software product-line architecture evaluation is an emerging field where a more comprehensive investigation is necessary. This framework is an initial classification of product-line requirements, evaluation times, evaluation techniques and evaluation methods but it is open to new contributions. The classification of methods can be used to select the most appropriate method for each case depending on the quality attributes and the selected evaluation moment.

Among the surveyed methods, most of them focus on evaluating product-line quality attributes (flexibility) at product-line architecture level. While there are few methods to assess concrete architectures derived from the PLA that reuse assets. However, there are (no product-line specific) single-system architecture evaluation methods than can be used for derived product architectures and also for PLAs. Single-product architecture evaluation is quite a mature field where a lot of research, techniques and methods have been developed. This allows the possibility to adapt the methods and techniques used in single product architecture into product-line context.

Acknowledgement

This work was partially supported by The Basque Government, department of education, universities and research. Leire Etxeberria enjoys a doctoral grant of the researchers formation program.

References

1. Dolan, T.J.: Architecture Assessment of Information-Systems Families, Ph.D. Thesis, Department of Technology Management, Eindhoven University of Technology (2002)
2. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice, Addison Wesley (1998)
3. Bosch, J.: Design And Use of Software Architectures: Adopting and evolving a product-line approach, Addison-Wesley ACM Press (2000)
4. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns, Addison Wesley (2002)

5. Thiel, S., Hein, A.: Systematic Integration of Variability into Product Line Architecture Design, Chastek, G.J.(Ed.): Software Product Lines, Second International Conference, SPLC 2, Proceedings, LNCS 2379 Springer (2002) 130-153
6. Deelstra, S., Sinnema, M., Bosch, J.: Experiences in Software Product Families: Problems and Issues During Product Derivation, In Nord, R.L.(ed): Software Product lines, Third International Conference, SPLC 3, Proceedings, LNCS 3154 Springer (2004) 165-182
7. Abowd, G., Bass, L., Clements, P., Kazman, R., Northrop, L., Zaremski, A.: Recommended Best Industrial Practice for Software Architecture Evaluation, Technical Report, CMU/SEI-96-TR-025 (1997)
8. Graaf, B., Van Kijk, H., Van Deursen, A.: Evaluating an Embedded Software Reference Architecture –Industrial Experience Report, 9th European Conference on Software Maintenance and Reengineering (CSMR 2005), Proceedings. IEEE Computer Society (2005) 354-363
9. Van der Hoek, A., Dincel, E., Medvidovic, N.: Using Service Utilization Metrics to Assess and Improve Product Line Architectures, USC-CSE-2001-507 (2001)
10. Rahman, A.: Metrics for the Structural Assessment of Product Line Architecture, Master Thesis Software Engineering, Thesis nº: MSE-2004:24 (2004)
11. Alonso, A., García-Valls, M., de la Puente, J.A.: Assessment of Timing Properties of Family Products, Van der Linden, F.(Ed.): Development and Evolution of Software Architectures for Product Families, Second International ESPRIT ARES Workshop, Proceedings. LNCS 1429 Springer (1998) 161-169
12. Auerswald, M., Herrmann, M., Kowalewski, S., Schulte-Coerne, V.: Reliability-Oriented Product Line Engineering of Embedded Systems, Van der Linden, F.(Ed.): Software Product-Family Engineering, 4th International Workshop, PFE 2001, Revised Papers. LNCS 2290 Springer (2002) 83-100
13. Zhang, H., Jarzabek, S., Yang, B.: Quality Prediction and Assessment for Product Lines, Eder, J., Missikoff, M.(Eds.): Advanced Information Systems Engineering, 15th International Conference, CAiSE 2003, Proceedings. LNCS 2681 Springer (2003) 681-695
14. De Lange, F., Kang, J.: Architecture True Prototyping of Product Lines, In van Linden, F.(ed): Software Product-Family Engineering, 5th International Workshop, PFE 5, Revised Papers LNCS 3014 Springer (2004) 445-453
15. Gannod, G.C., Lutz, R.R.: An Approach to Architectural Analysis of Product Lines, ICSE, Proceedings of the 22nd International Conference on Software Engineering, ACM (2000) 548-557
16. Matinlassi, M., Niemelä, E., Dobrica, L.: Quality-driven architecture design and quality analysis method: A revolutionary initiation approach to a product line architecture, VTT Publications (2002)
17. Maccari, A.: Experineces in assessing product family software architecture for evolution, Proceedings of the 22rd International Conference on Software Engineering, ICSE 2002. ACM (2002) 585-592
18. Riva, C., Del Rosso, C.: Experiences with Software Product Family Evolution, 6th International Workshop on Principles of Software Evolution (IWPSE 2003), IEEE Computer Society (2003) 161-169
19. America, P., Hammer, D., Ionita, M.T., Obbink, H., Rommes, E.: Scenario-Based Decision Making for Architectural Variability in Product Families, In Nord, R.L.(ed): Software Product lines, Third International Conference, SPLC 3, Proceedings, LNCS 3154 Springer (2004) 284-303
20. Wijnstra, J.G.: Evolving a Product Family in a Changing Context, In van Linden, F.(ed): Software Product-Family Engineering, 5th International Workshop, PFE 5, Revised Papers LNCS 3014 Springer (2004) 111-128

21. Deelstra, S., Nijhuis, J., Bosch, J., Sinnema, M.: The COVAMOF Software Variability Assessment Method (COSVAM), 2nd Groningen Workshop on Software Variability Management (2004)
22. Stoermer, C., Bachmann, F., Verhoef, C.: SACAM: The Software Architecture Comparison Analysis Method, Technical Report, CMU/SEI-2003-TR-006 (2003)
23. Korhonen, M., Mikkonen, T.: Assessing Systems Adaptability to a Product Family, In Al-Ani, B., Arabnia, H.R., Mun, Y.(Eds.): Proceedings of the International Conference on Software Engineering Research and Practice, SERP '03, Volume 1. CSREA Press (2003) 135-141
24. Clements, P., Kazman, R., Klein, M.: Evaluating Software Architectures: Methods and Case Studies, Addison Wesley (2001)
25. Ferber, S., Heidl, P., Lutz, P.: Reviewing Product Line Architectures: Experience report of ATAM in an Automotive Context, Van der Linden, F.(Ed.): Software Product-Family Engineering, 4th International Workshop, PFE 2001, Revised Papers. LNCS 2290 Springer (2002) 364-382
26. Gallagher, B.P.: Using the Architecture Tradeoff Analysis Method to Evaluate a Reference Architecture: A Case Study, CMU/SEI Technical note (2000)
27. Van der Linden, F., Bosch, J., Kamsties, E., Känsälä, K., Obbink, H.: Software Product Family Evaluation, In Nord, R.L.(ed): Software Product lines, Third International Conference, SPLC 3, Proceedings, LNCS 3154 Springer (2004) 110-129
28. Niemelä, E., Matinlassi, M., Taulavuori, A.: Practical Evaluation of Software Product Family Architectures, In Nord, R.L.(ed): Software Product lines, Third International Conference, SPLC 3, Proceedings, LNCS 3154 Springer (2004) 130-145
29. Thiel, S.: On the definition of a Framework for an Architecting Process Supporting Product Family Development, Van der Linden, F.(Ed.): Software Product-Family Engineering, 4th International Workshop, PFE 2001, Revised Papers, LNCS 2290 Springer (2002) 125-142
30. Bayer, J., Flege, O., Gacek, C.: Creating Product Line Architectures, Van der Linden, F.(Ed.): Software Architectures for Product Families, International Workshop IW-SAPF-3, Proceedings, LNCS 1951 Springer (2000) 210-216
31. Dobrica, L., Niemelä, E.: A strategy for analysing product line architectures, VTT Publications (2000)
32. Ionita, M.T., America, P., Hammer, D.: A Method for Strategic Scenario-Based Architecting, Proceedings of the Hawai International Conference on System Sciences (HICSS-38), IEEE Computer Society (2005)
33. Svahnberg, M., Wohlin, C., Lundberg, L., Mattson, M.: A Quality-driven Decision Support Method for Identifying Software Architecture Candidates. International Journal of Software Engineering and Knowledge Engineering, Vol. 13, No.5 (2003)
34. Zhang, H., Jarzabek, S.: A Bayesian Network Approach To Rational Architectural Design, Accepted by IJSEKE (2005)
35. Pinzger, M., Gall, H., Girard, J.F., Knodel, J., Riva, C., Pasman, W., Broerse, C., Wijnstra, J.G.: Architecture Recovery for Product Families, In van Linden, F.(ed): Software Product-Family Engineering, 5th International Workshop, PFE 5, Revised Papers LNCS 3014 Springer (2004) 332-351
36. Stoermer, C., O'Brien, L.: MAP – Mining Architectures for Product Line Evalutions, Working IEEE / IFIP Conference on Software Architecture (WICSA 2001), IEEE Computer Society (2001) 35-44
37. Bass, L., Bachmann, F., Klein, M.: Making Variability Decisions during Architecture Design, In van Linden, F.(ed): Software Product-Family Engineering, 5th International Workshop, PFE 5, Revised Papers LNCS 3014 Springer (2004) 454-465

Strategies of Product Family Architecture Development

Eila Niemelä

VTT Technical Research Centre of Finland, Oulu, Finland
Eila.Niemela@vtt.fi

Abstract. Product family engineering (PFE) is successfully applied in different kinds of software intensive systems. As there are several ways to apply PFE, selecting an appropriate approach is a complex task. This paper introduces six ways to set the goal of PFE and eight strategies to achieve the goal. It also introduces steps how to evaluate which strategy provides the best fit for a company. The criteria for selecting a strategy have been derived from seventeen case studies, including nineteen product families, in the various contexts provided by small, medium size and large companies.

1 Introduction

The core idea in software PFE is to use as much as possible the same software assets in all family members. PFE is successfully applied in software intensive systems, especially in the development of embedded systems but also in pure software systems. The main reason in applying PFE is to get competitive advantage by shortening development time, decreasing development and maintenance costs and expanding markets and market share. The application of PFE enables even to deploy new products in few days or few weeks [1] achieving a reuse level from 70% up to 100%.

Several attempts have been made to explain what the preconditions are for a successful adoption of PFE. The context of product family adoption has been explored from the points of view of market, organization, and personnel [2]. The economics of PFE have also been considered [3], as well as the development and evolution of product family architectures (PFA) [4, 5]. Furthermore, a software family evaluation framework (FEF) has been introduced with four dimensions related to software engineering concerns: business, architecture, process and organization [6]. FEF is designed to be used for the benchmarking and assessment of product families. Therefore, a specific method for family architecture evaluation (FAE) was developed [7]. The method uses the architecture dimension of FEF for representing the maturity levels of PFAs. However, there is no method or guidance available on how to select an appropriate way of applying product family engineering.

The contribution of this paper is a Strategy Evaluation Framework (SEF) that provides a map of alternative PFAs and a method to select an appropriate PFE strategy. The focus is on the different ways of realizing family architectures and PFE strategies used in different business contexts. The SEF has been derived by analysing the data collected from twelve case studies documented in the literature and five cases studied by semi-structured interviews. Collected data was analysed and finally, a set of business aspects, important while making a decision which kind of PFA to apply,

H. Obbink and K. Pohl (Eds.): SPLC 2005, LNCS 3714, pp. 186 – 197, 2005.

were identified. The results are presented as an evaluation framework intended to be used for selecting a PFE strategy.

The structure of this paper is as follows. Section 2 presents how the study was carried out. Section 3 introduces the strategy evaluation framework. Section 4 compares the applied PFAs from the business points of view. Section 5 compares the findings to the related work. Concluding remarks close the paper.

2 Description of the study

Data for the analysis was collected in three phases. First, technical managers and architects of three small and medium size companies were interviewed in a semi-structural way. Architectural artefacts were also reviewed in two cases. As a result the FAE method was created [7]. Second, twelve case studies were collected from the literature. The analysis was based on the same comparison framework as used in the interviews. Third, two additional interviews were carried out in the semi-structured way in order to acquire new and fresh experiences from enterprise companies. In all phases, the FAE method and its comparison framework were used as tools for collecting and organizing textual data.

In summary, the collected data covered a wide variety of product families. Two of the 17 cases included more than one product family. Five interviewed companies with six product families targeted at embedded systems or embedded software. Eleven companies with 13 product families reported in literature developed embedded systems and devices. Five companies provided six pure software families. Three of them were, however, related to large embedded systems and were used only with them. Eight of the product families were applied in small or medium size (SME) companies. Eleven product families were applied in six large enterprises in different application fields or/and domains. The types of products varied from distributed information systems to command and control systems, from hardware related and digital signal processing software to user interface software and from large integrated products including mechanics, electronics and software to business supporting tools.

The maturity levels of FEF [6] and the architecture evolution framework [8] were used as a starting point for classifying the ways of defining PFAs, and each case was compared to them. The comparison resulted in the target PFAs defined in the next section.

Finally, a method for selecting a PFE strategy was defined based on the analysis results of the case studies. The aim was to find out the reasons why a company had decided to apply a particular PFA, how they realized PFA and why their selection had worked successfully, i.e. what were the critical factors for successful PFE adoption. Grounded Theory was applied to analyzing the cases [9]. The idea of Grounded Theory is to read and reread the collected data, and iteratively distil a set of concepts and their interrelationships. Thus, the SEF was created based on the critical enabling factors of PFE identified in the case studies. The author's earlier studies, e.g. [7, 10], and experiences with product families have certainly influenced on interpretation of the results.

3 Strategy Evaluation Framework

This section presents the PFAs identified in the case studies and the method designed for selecting an appropriate PFE strategy. PFA has a key role in the selection process because it is the core asset of PFE.

3.1 Alternative PFAs

The identified ways of implementing PFAs (**Fig. 1**) are presented as a combination of the maturity levels of FEF [6] [11] and the architecture evolution framework [8]. The arrows indicate possible transitions from one PFA to another. Different PFAs can also be applied concurrently.

FEF defines five maturity levels. The first level includes independent products that do not share any artefacts, i.e. requirements, features, architecture or components. Thus, this level is not considered as a PFE approach.

On the second level (standardized infrastructure in FEF), only external components are specified and used. This level is extended by a commonly used way of software reuse; new products are developed based on the earlier releases by using the cut, paste and change technique. In most cases, software is component based but not reusable as such, and software architecture is not defined or used as an asset. In [8], it has been suggested that consecutive releases use a stable architecture. Quite the contrary, it was found out that on this level architecture was not defined or used intentionally but rather components or pieces of software were reused by tailoring them for new products. Thus, this kind of reuse cannot be considered as PFE either.

On the third level (software platform in FEF), three different ways to apply PFE were identified; packaged services, internal platforms and platform products. FEF defines an internal platform including common features of all family members and a set of reusable components that realize the features. We also identified an approach, in which a company provided a platform product that was sold to customers as such or as part of a larger system. A similar finding is introduced in [8]. In addition, we identified a new approach, packaged services. This approach of family engineering was applied together with an internal platform or a platform product. Packaged services include additional products that are used during installation, configuration, training and maintenance. These service packages are also differentiated for several types of users; developers, customers and maintainers in a provider organization, customer organization and/or third parties. The approach 'platform customer' introduced in [8] corresponds to the use of Modified-Off-The-Shelf (MOTS) components in our model [12], and therefore it is not considered as a PFE approach.

On the fourth level (software product family in FEF), the family architecture and variation points are fully specified and managed. The PFA is the key artefact that enables systematic product derivation from a defined PFA. Two different ways used either together or separately were identified. PFAs were used as defining and managing variations when the amount of variation was reasonable or when variations were defined in two levels of degree; high-level variation was dealt with in architecture and low-level (fine-grain) variations as configurable features and/or components. A configurable features/components base without explicitly defined architecture was also used. In that case, architecture was integrated into the platform and the common component base.

On the fifth level in FEF, the product family architecture is enforced and product members are automatically generated. This level was identified only in two cases, which is why it can be assumed that this approach is suitable only for highly mature and stable product families. In summary, the most applicable PFE approaches seem to be

- Product family architecture,
- Configurable features or/and components base,
- Internal platform,
- Platform product, and
- Packaged services.

It appears beneficial to apply these approaches together, i.e. to make simultaneous use of, e.g., an internal platform, PFA, a configurable components base and packaged services.

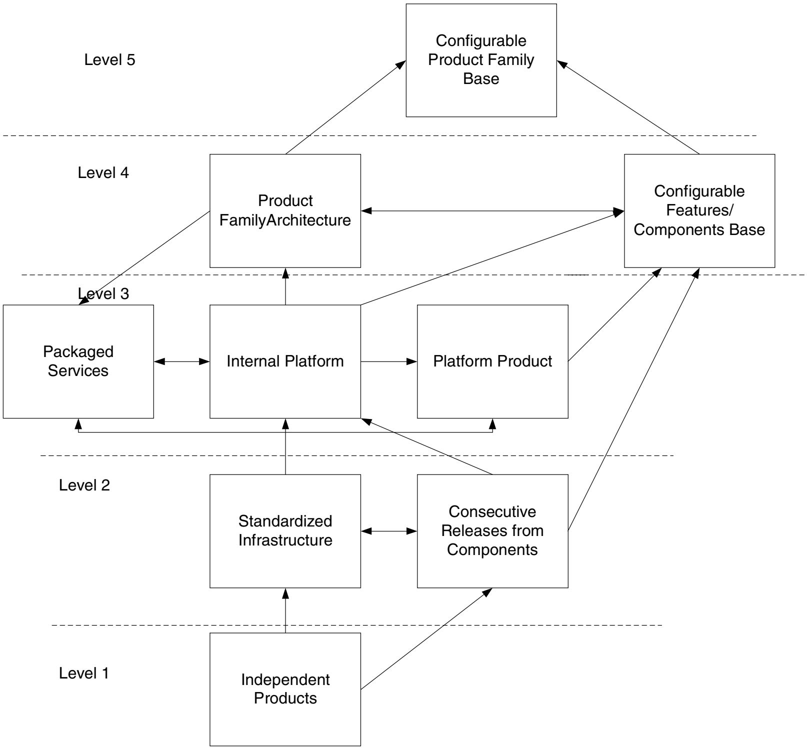

Level 5
Configurable Product Family Base
Level 4
Product FamilyArchitecture
Configurable Features/ Components Base
Level 3
Packaged Services
Internal Platform
Platform Product
Level 2
Standardized Infrastructure
Consecutive Releases from Components
Level 1
Independent Products

Fig. 1. Alternative ways to implement PFAs

3.2 Selecting a PFE Strategy

When determining which PFE approach to use, a company has to evaluate the quality of their products and what kind of reuse potential the products embody. This decision making process is divided into three phases. First, the assumptions for achieving benefits from PFE are evaluated. Second, the desired target PFA is defined, and last, the most appropriate PFE strategy is selected based on reasoning and estimations.

3.2.1 Estimate Benefits

In order to get added value from PFE, a company has to analyze what kind of benefits it can achieve from PFE adoption, i.e. what issues are important for the business of a company and its customers. The basic assumptions for successful PFE adoption are:

- Future customer needs are known or can be predicted. (Criterion: predictability)
- Short delivery times provide competitive advantage, for example, by extending market share or entering new emerging markets. (Criterion: cutting edge)
- Product costs have remarkable dependence on software development and maintenance. (Criterion: profitability)
- Products can be used in various ways, e.g. they are used as such and/or with other products. (Criterion: variability)
- High quality of products is essential in achieving customer satisfaction. (Criterion: customer satisfaction)

These assumptions can be used as the first evaluation criteria whether or not to apply PFE. If these assumptions are not valid, there is a risk that the benefits of applying PFE will not be achieved.

3.2.2 Set the Target PFA

The target PFA is defined through three steps:

- **Step 1.** Evaluate what is the status quo of the architecture of existing products by using the FAE method [7]. Key product(s) and key competence are evaluated against the quality criteria of most importance regarding company products and personnel know-how. Quality criteria of products are depending on the domain(s), used technologies and customers' needs. Quality of know-how is related to the importance of product qualities. The goal is to identify the strengths and weaknesses of a company in the context of PFE.
- **Step 2.** Identify the success factors and their importance (**Table 1**). The goal is to identify the main business factors why PFA is needed.
- **Step 3.** Estimate ROI (return on investment) for the different ways of implementing PFA considering their investment and potential. The estimation is made based on the economic models such as introduced in [3, 13].

3.2.3 Select the PFE Strategy

The PFE strategy that provides the best fit to the company's business strategy is selected using the criteria set for each strategy (**Table 2**). This analysis phase requires quantitative and qualitative measurements as an input for each criterion, e.g., the amount of variation points, personnel competence, and customer satisfaction. How these measurements are done falls beyond the scope of this paper. However, the comparative analysis presented in the next section introduce the aspects identified important in case studies as selecting a PFA strategy, and thus, they can be used as a guidance what aspects to look at.

Table 1. Success factors in PFA development

Success factor	Criteria	PFA alternative
Managing customer needs	Customer satisfaction Time-to-market	Configurable features/components base, configurable product family base
Key competence	Cutting edge	Internal platform, configurable features/components base
New opportunities	Cutting edge	Platform product, packaged services
Effective work organisation	Cost effectiveness	PFA
Evolution management	Cost effectiveness	PFA and configurable features/components base

4 Comparison of PFE Approaches

This section discusses how PFE adoption is influenced by business. The business aspects that seemed to have the greatest influence on PFE strategy selection, e.g. business fields, size and type of family members, etc. are discussed introducing their impact on the criteria used in setting the target PFA. The purpose of this analysis is to help while selecting an appropriate PFE strategy.

4.1 Business Fields

PFE provides the greatest advantage in the business fields where product families are complex from the technology, management and business points of view. In the case studies, 70% had to do with complex networked embedded systems, and three product families were pure software families. Thus, it can be concluded that PFE is appropriate when the complexity of software is high and long-term investments in PFE are likely to provide added value by improving the efficiency of product development and enhancing product quality.

Most of the companies applied the newest hardware and software technologies. The oldest product families (over 10 years) were mainly integrated systems using newest communication technologies and business expertise as drivers for entering emerging new markets. Shortening delivery times and decreasing costs were the advantages these companies were targeting at. Three main trends could be identified in the adoption of PFE. Firstly, PFE adoption for pure software families has just started and will expand in the future. Secondly, PFE is applied both to mass-market products and customized products. Thirdly, the use of platform products provided by a separated department or a 3rd party is increasing. These platforms used generic IT technologies for the integration of networked systems. This can also be regarded as a means of entering new markets by changing business role from a system supplier to a service supplier.

Table 2. PFE strategies

Strategy	Means	Criteria	Advantages	Disadvantages
Minimizing risks	internal platform	identified commonalities, high technological competence, the size of product family	separation of commonality and variability	substitutable generic IT technologies
Extending market share	platform product	high technological competence, openness, market share, organization model	commonalities separated; responsibilities defined; stability of software	ignoring vertical applications, emerging markets and family evolution
Maximizing end-user satisfaction	packaged services	familiarity with end-user needs	market segmentation; determination of responsibilities; different pricing; improved customer satisfaction	only supplement business; add-ons to the key products
Balancing cost and customer satisfaction	internal platform + configurable features /components base	separation of commonalities and variabilities, the amount of customization, quality of products	organisation based on separately managed commonalities and variabilities; key competences addressed in technology and domain knowledge	increased effort for evolution and knowledge management
Maximizing customer satisfaction	configurable product family base	quality of products, short delivery times	standard quality; cheap delivery and deployment; pricing based on quality and customer satisfaction	expensive to establish and maintain; long term investment
Balancing cost and potential	PFA	complexity and variability, market share, new markets	an extensible family with maintainable product variants, customer needs represented and managed, key competence areas addressed, entering to emerging markets possible	key product(s) and high-level knowledge in specific technology and domain areas required, long-term investment
Balancing cost, customer satisfaction and potential	PFA + configurable features/components base	complexity, variability, delivery times, market share, emerging new markets	improved customer satisfaction by flexibility, shortened delivery times, new emerging markets	time and cost consuming to establish; long-term investment
Maximizing potential	PFA + configurable features/components base + platform product + packaged services	complexity, variability, cost effectiveness, delivery times, customer satisfaction, market share, emerging new markets	separation of concerns: business by PFA, customer satisfaction by configuration and packaged services, and technology by a platform product	expensive to establish and maintain, long term investment, 'only for leaders'

The number of critical systems, such as power generation systems and medical systems, was high, especially in product families initiated a long time ago. Products embodying high quality requirements but not regarded as critical systems, e.g. telecommunication switching systems and different kinds of measuring systems, constituted the other main application field. Although the systems quality requirements were not explicitly defined in all cases, overall correctness and performance were the default requirements for all product families. Surprisingly enough, security did not appear to play too important a role in the product families. Most of the application fields were based on technology push. Application pull was visible only in banking service systems, embedded information systems, application-oriented integration platforms and special environments for business support services.

4.2 Size and Type of Product Family

A PFE approach can be applied to product families of different sizes. PFA initiation was typically based on one or a few key products that had been successful in terms of markets, quality and technology know-how. Over 90% of the cases applied an evolutionary PFE approach, starting small and extending the product family when deemed necessary, or gaining advantage by a transition, e.g., from the 'internal platform' approach to the 'configurable features and components base' approach. However, there was an exception, in which a configurable components base was established from scratch, applying the revolutionary PFE approach. Instead of the 'minimizing risk' strategy that was normally applied, a 'maximizing potential' strategy was used in that case. For using this approach, there were a number of preconditions that obviously had to be met. Firstly, application knowledge must have been tremendously high as most configuration parameters originate from the application field. Secondly, judging by the fact that separation of concerns was also used for configuration, it may be concluded that software and configuration techniques were well-known. Thirdly, the products included mainly in-house software – it is impossible or very difficult to adapt commercial software to variants. Lastly, the development organization has to be located at a single site.

Market segments were used as a starting point for scoping a product family (PF). Complementary products sold as integrated systems could also establish a product family. One observation was that product families established during the last 4-5 years seemed to be more market and business oriented than those established 10 years ago. This indicates that business orientation will impact more on PFE adoption in the future.

4.3 Maturity of PFA

The maturity of product families varied from 15 years to a couple of years. Most of the interviewed companies had renewed their product families during the last five years (originally established about 10 years ago). Thus, domain knowledge was stable, while technology knowledge required updating, i.e. in technology push mode affected the need for PFA recovery. In another case, a company owning a ten-year product family had identified a need for modernization in their software implementation technology, and since hardware and software dependencies were managed, the software technology platform could easily be tailored for new hardware. Thus, hardware did not directly push towards technology transfer. In summary, PFA can be considered to reach its optimum maturity at the age of 5-7 years. In order to benefit from PFA, it should be kept relatively stable for at least five years.

4.4 Importance of Software in Products

PFE is effective in software intensive systems. In the studied cases, the share of software could be calculated or estimated on the basis of the software development cost, which typically varied from 50 % to 80% of the total development costs. Of the total costs, the software costs were estimated lower (around 50%).

Maintenance costs turned out to vary according to types of configuration. In customized products (especially in configurable features and components base), maintenance was part of business, while only the system provider could configure the system (i.e. separate maintenance cost). In the case of software keys, customization was done remotely by a system provider (i.e. low or no maintenance cost). In the third case, customers did the configuration themselves according to the predefined rules (i.e. maintenance cost was part of the development cost). In the last case, maintenance was not managed but if improvements were needed, they were implemented during the development of a new release (i.e. part of development cost).

Customization was around 30% of the total software. A high customization degree turned out to push companies towards focusing on both PFA and configurable features/components base and investing 30% of their development cost in proprietary variability management tools. When fast product derivation and deployment was required, configuration support was a necessity. Configuration support also made it possible to outsource deployment work to 3rd parties while keeping control over product quality.

In-house software was important in PFs; most of the studied product families were based on proprietary software. However, there was a tendency among the case companies to transfer from proprietary software to 3rd party software (commercial off the shelf and open source), while at the same time restricting their use in a way that would enable the companies to take optimal advantage of the software without losing control over the product family. Open source software will be a future challenge in PFE.

4.5 Quality Requirements

Among business qualities, time-to-market (i.e. fast product derivation and deployment) and cost-effective development (i.e. increased reusability, flexibility and extensibility) were considered important. Standardization and technology push were deemed important as business drivers, related to qualities such as changeability, expandability and reliable functioning of applications. Price erosion was a new business driver, indicating that markets were changing from the technology push operation mode to the application pull mode. In that new situation, customer satisfaction plays an important role, product life cycles are shorter and price is becoming a remarkable competitive expedient.

External product qualities (visible while running a system) are as important as ever. Performance, reliability, interoperability, safety, availability, scalability and usability are still regarded as necessities that have to be provided by a PF. The importance of execution qualities may be lower if only the existing PFE approaches are looked at. However, many of the interviewed companies were moving towards service oriented software engineering, in which execution qualities become more important.

Internal qualities, i.e. reusability, testability, modifiability, maintainability, extensibility, are qualities related to the software development and maintenance. They are mostly of interest to software providers although also indirectly visible to customers and end-users. While the importance of the internal qualities was obvious, PFE is now focusing on proactive PFE with increased interest in flexibility and extensibility instead of modifiability and maintainability. Consequently, future product families will be even more complex and require sophisticated self-configuration, self-healing and self-organization mechanisms.

In the case of the embedded systems family, traditional hardware specific qualities, such as the ability to tolerate low temperature, vibrations and splashes, are still important in products intended for hard circumstances. Furthermore, low power consumption is relevant in mobile terminals and measuring systems used for environmental data collection.

4.6 Variability Management

The amount of variations depends on the number of sources producing differences in software. Variations were caused by new business manners, market segments, customer needs, standards, national regulations, diversity of environments, hardware, software platforms and implementation technologies, and differences in features, application data and component combinations.

The means of managing variability varied a lot; PFs applied software keys (license based), commercial options, separation of commonalities and product specific parts, property files, software configuration management, configuration parameters, tailoring rules and frameworks. However, there are two main extremes in variability management. Those product families that had a very long life cycle (20-30 years) and a long development time (20-30%/life cycle) employed software reuse as such or with configuration parameters. These companies manage variability in their normal release-based software development process by modifying and tailoring software just slightly for each release. The other extreme is represented by feature-based software configuration, which allows reactive PFE and to some extent proactive PFE. In both cases, technology changes were made relatively often. In the latter case, users' needs were the main reason for making changes to the software and thus feature-based variability management was applied. Features could be initiated in the development of a particular product and leveraged afterwards to the whole PF.

In summary, the means of variability management depends mostly on the time-to-market; i.e. how much time it will take to derive a product from a product family, and how fast the deployment of a product has to be. Product derivation was reported to last from a few days to some months. The deployment can take some hours or some months and still be fast enough. In a release-based product family, one or two new releases were introduced in a year. Thus, delivery time appears highly dependent on the business context.

5 Related Work

There are several evaluation frameworks that define the maturity levels of PFE. Bosch presented the first version of the maturity levels [14], from which the first two levels are still visible in Fig. 1, in FEF [6, 11] and in the evolution framework [8]. These two levels are not considered as PFE approaches in this study but the preceding phases of PFE.

Bosch defined the third level as 'platform'. We identified three ways of applying PFA on the third level; internal platform, platform product and packaged services. Nedstam dn Karlsson [8] have also identified 'internal platform' and their approach 'platform as product' is quite similar to our definition 'platform product'. The approach 'packages services' has not identified earlier. Software platform has also defined in FEF.

Level 4 differs the most. First, Bosch defined 'software product lines' and 'product population' between levels 3 and 4, and 'program of product lines' between levels 4 and 5. Second, in [8] level 4 is defined as 'software product line', although they also identified 'an unmanaged configurable product base' being a preceding phase of a product line. In the first version of FEF [6], level 4 was 'software product family', indicating that this level is the first level that really provides an explicitly defined family architecture. That is why we have called it 'product family architecture'. In the current version of FEF [11], level 4 is defined as 'product variants'. The different definitions indicate that all ways of applying PFA on level 4 is not identified yet. For example, in our study we identified the approach 'configurable features/components base' that could be applied as a separate means of managing product variations, or concurrently with the PFA.

The maturity level 5 is mainly defined as a configurable product base [6, 8, 14]. In [11], 'self-configurable products' is defined instead of our definition 'configurable product family base' but despite of different wording they have the same meaning.

Thus, there is a lot of related work done for defining maturity and evolution levels of PFAs and a decision framework based on process and organisation maturity [15], but any approach for making the selection based on business and architecture relations could not be found. Thus, the presented SEF is an initial concept, and it needs to be applied and validated in practice together with earlier developed PFE metrics, e.g. economic models [16].

6 Conclusions

Product family engineering can be applied in several ways. This paper introduced six approaches (i.e. internal platform, platform product, packages services, product family architecture, configurable features/components base, and configurable product family base) to adopting PFE and eight strategies from which a company can select the one that provides the desired benefits. In order to make the decision which strategy to select, a company has to evaluate their current status as regards business, architecture, process, and organization issues. The benefits achieved by a PFE approach are the main criteria when considering business issues, such as business field, size and type of product family, quality and variability management. After evaluating the current state of the product architectures, success factors are identified and their importance is ranked. Next, the benefits are estimated by pinpointing the identified six ways of implementing PFAs as regards investments and potential. Finally, the PFE strategy is selected by considering potential benefits, alternative strategies and the criteria set for the application of these strategies.

References

[1] M. Raatikainen, T. Soininen, T. Männistö, and A. Mattila, "A Case Study of Two Configurable Software Product Families," presented at the 5th Product Family Engineering workshop, Siena, Italy, 2003.

[2] S. Buhne, G. Chastek, T. Käkölä, P. Knauber, L. Northrop, and S. Thiel, "Exploring the Context of Product Line Adoption," presented at the 5th Product Family Engineering workshop, Sienna, Italy, 2003.

[3] K. Schmid and M. Verlage, "The Economic Impact of Product Line Adoption and Evolution," *IEEE Software*, vol. 19, pp. 50-57, 2002.

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

[5] J. Bosch, *Design and use of software architectures: adopting and evolving a product-line approach*. Harlow: Addison-Wesley, 2000.

[6] F. van der Linden, J. Bosch, E. Kamsties, K. Känsälä, L. Krzanik, and H. Obbink, "Software Product Family Evaluation," presented at the 5th Product Family Engineering workshop, Siena, Italy, 2003.

[7] E. Niemelä, M. Matinlassi, and A. Taulavuori, "Practical Evaluation of Software Product Family Arhitectures," presented at Third International Conference on Software Product Lines, Boston, USA, 2004.

[8] J. Nedstam and E.-A. Karlsson, "Experiences form Architecture Evolution," presented at Autralasian Architecture Workshop on Software and System Architectures, Melbourne, Australia, 2004.

[9] J. Corbin and A. Strauss, *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*: Sage Publications, 1998.

[10] E. Niemelä and T. Ihme, "Product Line Software Engineering of Embedded Systems," *ACM SIGSOFT Software Engineering Notes*, vol. 26 (3), pp. 118 - 125, 2001.

[11] F. van der Linden, J. Bosch, E. Kamsties, K. Känsälä, and H. Obbink, "Software Product Family Evaluation," presented at The Third International Conference on Software Product Lines, SPLC3, Boscton, USA, 2004.

[12] A. Taulavuori, Niemelä, E., Kallio, P., "Component documentation - a key issue in software product lines," *Information and Software Technology*, pp. 535-546, 2004.

[13] K. Schmid, "A Quantitative Model of the Value of Architecture in Product Line Adoption," presented at the 5th Product Family Engineering workshop, Sienna, Italy, 2003.

[14] J. Bosch, "Maturity and Evolution in Software Product Lines: Approaches, Artefacts and Organization," presented at Second International Conference on Software Product Lines, San Diego, USA, 2002.

[15] J. Bosch, "On the Development of Software Product-Family Components," presented at The 3rd International Conference on Software Product Lines, SPL3, Boston, USA, 2004.

[16] G. Böckle, P. Clements, J. D. McGregor, D. Muthig, and K. Schmid, "A cost model for software product lines.," presented at Fifth International Workshop on Product Family Engineering, Siena, Italy, 2003.

Defining Domain-Specific Modeling Languages to Automate Product Derivation: Collected Experiences

Juha-Pekka Tolvanen and Steven Kelly

MetaCase,
Ylistönmäentie 31,
FI-40500 Jyväskylä, Finland
{jpt, stevek}@metacase.com
http://www.metacase.com

Abstract. Domain-Specific Modeling offers a language-based approach to raise the level of abstraction in order to speed up development work and set variation space already at specification and design phase. In this paper we identify approaches that are applied for defining languages that enable automated variant derivation. This categorization is based on analyzing over 20 industrial cases of DSM language definition.

1 Introduction

Domain-Specific Modeling (DSM) can raise the level of abstraction beyond coding by specifying programs directly using domain concepts. The final products can then be generated from these high-level specifications. This automation is possible because the modeling language and generator only need to fit the requirements of one domain, often in only one company [8], [11].

This paper examines approaches applied for DSM language creation. Although there exists a body of work done on language development, most of this deals only with textual languages, and concentrates on their compilers rather than the languages. In general, such research has only looked at the initial creation of the languages (e.g. [1] [2]). Fewer studies (e.g. [9], [10]) have investigated the actual process of language creation, or of refinement and evolution of languages that are already in use. Moreover, the typical focus of a DSM language, providing models as input for generators, gives a special perspective to modeling language creation.

This paper identifies and categorizes approaches used for defining DSM languages. It is based on an analysis of cases that created DSM languages to support model-based software development and especially to automate product variant creation. Although all the DSM languages studied were implemented as metamodels and were not tied to customizing an available language, the approaches identified may also serve language creation that is based on extending available metamodels or using profiles for more lightweight language definition work.

In the next section we describe the cases and how they were analyzed in more detail. Section 3 describes the approaches identified by characterizing their main

H. Obbink and K. Pohl (Eds.): SPLC 2005, LNCS 3714, pp. 198 – 209, 2005.

focus and by giving a representative example[1] of a DSM in that category. Sections 4 and 5 evaluate the categorization and summarize the experiences gathered.

2 About the Studied DSM Cases

This study is based on data gathered from over 20 cases of DSM creation. The cases were chosen to cover different domains and modeling: from insurance products to microcontroller-based voice systems. Table 1 shows the cases, their problem domains and solution domains. The fourth column refers to the DSM creation approaches, which are discussed in more detail in Section 3. The cases are sorted by the fourth column for the benefit of the reader.

All the cases applied model-based development by creating models that then formed the input for code generation. Thus, DSM language creation was not only applying modeling to get a better understanding, support communication or have documentation, but for automating development with domain-specific generators. Actually, in most of the cases the generators aim to provide full code from the modelers' perspective. This means that no changes to the generated code were expected to be needed. In all the cases, the target platform (i.e. available components and generated output language) was already chosen before the DSM language creation started. With the exception of cases that generated XML, the final detailed structure and composition of the generated output was left open and in most cases new domain framework code was created. A domain framework provides a well-defined set of services for the generated code to interface to.

Many of these domains, and hence also their respective DSM languages, can be characterized as rather stable; some however were undergoing more frequent changes. Some languages have been used now for several years whereas some have only just been created. None of the languages were rebuilt during the DSM definition process, but rather maintained by updating the available language specification. All the language definitions were also purely metamodel-based: i.e. complete freedom was available when identifying the foundation for the language. In other words, none of the cases started language definition by extending UML concepts via profiles etc. The largest DSM languages have several individual modeling languages and over 580 language constructs, whereas the smallest are based on a single modeling language and less than 50 constructs. As a comparison, UML has 286 constructs according to the same meta-metamodel as the one applied in the analyzed cases.

The data on DSM development (also know as method construction rationale [9]) was gathered from interviews and discussions, mostly with the consultants or in-house developers who created the DSM languages, but also with domain engineers and those responsible for the solution architecture and tool support. All the languages were implemented with the same tool [5] and access to the language definitions (metamodels) was available for content analysis [7] while analyzing the cases.

[1] Due to confidentiality of industrial DSM cases, not all cases can be illustrated in detail.

Table 1. DSM cases by domain and generation target

Case ID	Problem domain	Solution domain/ generation target	Creation approach(es)
1	Telecom services	Configuration scripts	1
2	Insurance products	J2EE	1
3	Business processes	Rule engine language	1
4	Industrial automation	3 GL	1, (2)
5	Platform installation	XML	1, (2)
6	Medical device configuration	XML	1, (2)
7	Machine control	3 GL	1, 2
8	IP telephony	CPL	2, (1)
9	Geographic Information System	3 GL, propriety rule language, data structures	2
10	SIM card profiles	Configuration scripts and parameters	2
11	Phone switch services	CPL, Voice XML, 3 GL	2, (3)
12	eCommerce marketplaces	J2EE, XML	2, (3)
13	SIM card applications	3 GL	3
14	Applications in microcontroller	8-bit assembler	3
15	Household appliance features	3 GL	3
16	Smartphone UI applications	Scripting language	3
17	ERP configuration	3 GL	3, 4
18	ERP configuration	3 GL	3, 4
19	Handheld device applications	3 GL	3, 4
20	Phone UI applications	C	4, (3)
21	Phone UI applications	C++	4, (3)
22	Phone UI applications	C	4, (3)
23	Phone UI applications	C++	4, (3)

3 DSM Definition Approach Categorization

Analysis of the metamodels revealed that the languages differed greatly with regard to their concepts, rules and underlying computational model (see samples in Fig. 1, 2 and 3). The collected data indicates that the driving factor for language construct identification was based on at least four approaches:

1. Domain expert's or developer's concepts
2. Generation output
3. Look and feel of the system built
4. Variability space

This list of approaches is not complete (being based on a rather limited set of cases), nor are the approaches completely orthogonal to each other. Actually, many of the cases applied more than one construct identification approach. In the following subsections we describe these approaches in more detail and discuss how the languages' constructs were identified and defined. We also attempt to describe the process of language creation (identification, definition, validation, testing), and discuss the need for a domain framework to ease the task of code generation.

3.1 Domain Expert's or Developer's Concepts

One class of DSM definitions seemed to be based on concepts applied by domain experts and developers of the models (cases 1–8 as listed in Table 1). Fig. 1 shows a sample DSM of this class (case 2). All the modeling concepts are related to insurance products: an insurance expert draws models like this to define different insurance products, and then the generators produce the required insurance data and code for a J2EE website.

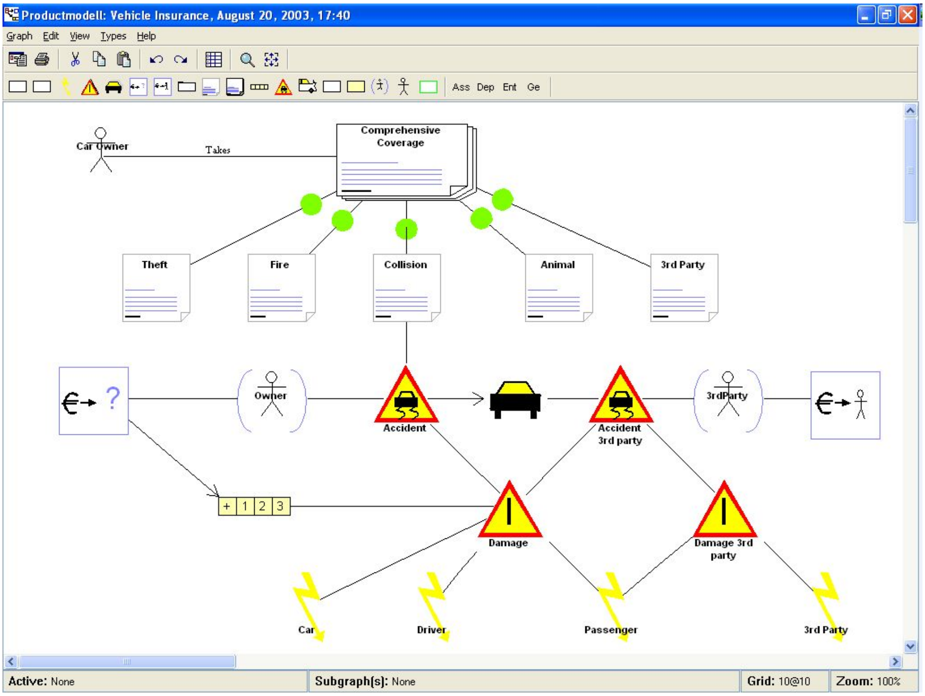


Fig. 1. DSM example: modeling insurance products

This type of language raises the level of abstraction far beyond programming concepts. Because of this, the generated output could easily be changed to some other implementation language. Similarly, users of these languages did not need to have a software development background, although in most cases they had. The computational models behind these languages were fairly simple and consistent over the cases analyzed: all were based on describing static structures or various kind of flows, their conditions and order. Code was usually produced by listing each model instance separately, along with its properties and relationships to other model elements. The code generation was guided by the relationship types, e.g. code for composite structures and flow-based ordering was generated differently.

Languages based on domain experts' concepts were considered easy to define: for an expert to exist, the domain must already have established semantics. Many of the modeling concepts could be derived directly from the domain model, as could some constraints. Constraints specifically related to modeling often needed to be refined, or even created from scratch, together with the domain experts. This process was rather easy as testing of the language could easily be carried out by the domain experts themselves. If the modelers were not themselves software developers, language visualization (e.g. the visual appearance of the notation), ease of use and user-friendliness were emphasized.

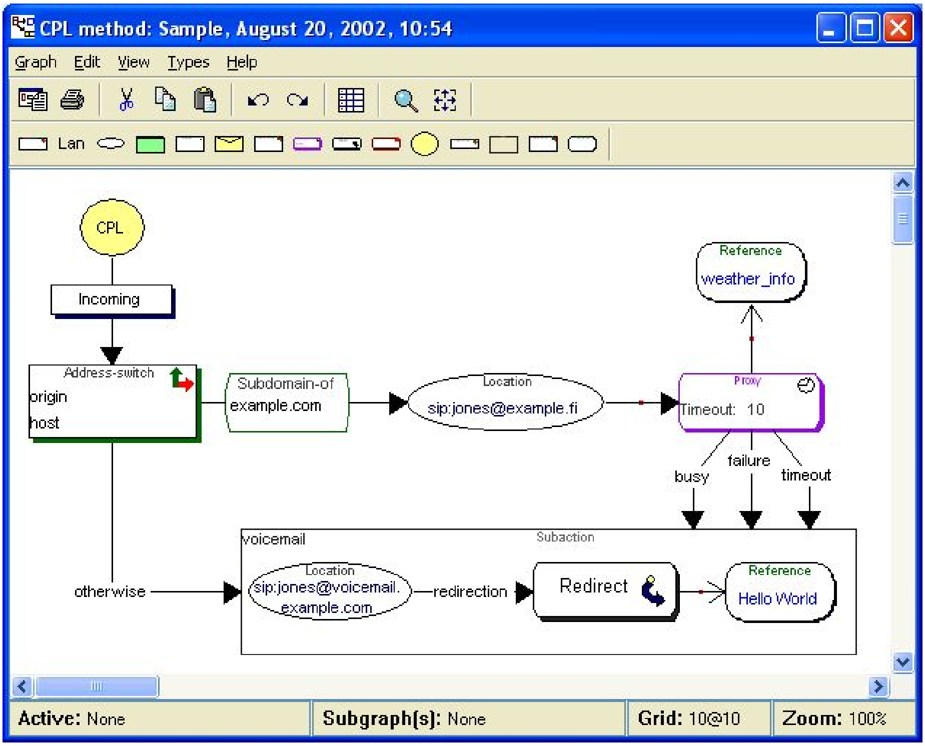
3.2 Generation Output

One class of DSM definitions was driven by the required code structure: modeling languages concepts were derived in a straightforward way from the code constructs (cases 7–12). An example of this kind of DSM is the Call Processing Language (CPL) [4], used to describe and control Internet telephony services (cases 8 and 11). The required XML output forms a structure and concepts for the modeling language (see Fig. 2).

DSM concepts to describe static parts like parameters and data structures, or the core elements and attributes in CPL and XML above, were quick and easy to define. The real difficulty was in finding appropriate concepts for modeling the behavioral parts and logic based on domain rules. This was achieved when the underlying platform provided services the models could be mapped to. This is often called analyzing the variability space (see Section 3.4). Once defined, the services and modules of the platform could even be applied directly as modeling concepts, or by having general interface concepts that allowed the modeler to choose or name the required platform service.

If a domain could not be defined or an existing architecture was not available, languages tended to use modeling only for the general static structures. The rest was done with textual specifications – often directly with programming concepts that do not provide domain-specific support.

A similar class of modeling languages are those originating from coding concepts, such as UML, schema design languages and various code visualization add-ons in IDE environments. Having models and code at substantially the same level of



Active: None
Subgraph(s): None
Grid: 10@10
Zoom: 100%

Fig. 2. DSM example: Call Processing

abstraction typically also raises the need for reverse engineering. This is similar to a class of tools, Microsoft's Whitehorse, Rational's XDE, Borland's TogetherJ, that aim to offer transparency between the use of models and textual specifications.

Such a close mapping to programming concepts did not raise the level of abstraction much, and offered only minor productivity improvements. Typical benefits were better guidance for the design and early error prevention or detection. Using the CPL/XML as an example, designs could be considered valid and well-formed already at the design stage. In that way it was far more difficult to design Internet telephone services that were erroneous or internally inconsistent: something that was all too easy in hand-written CPL/XML.

3.3 Look and Feel of the System Built

Products whose design can be understood by seeing, touching or by hearing often led to languages that applied end-user product concepts as modeling constructs (cases 11–23). Fig. 3 gives an example of a language whose concepts are largely based on the widgets that Series 60 and Symbian-based smartphones [6] offer for UI application development (case 16). The behavioral logic of the application is also described mostly based on the widgets' behavior and the actions provided by the actual product.

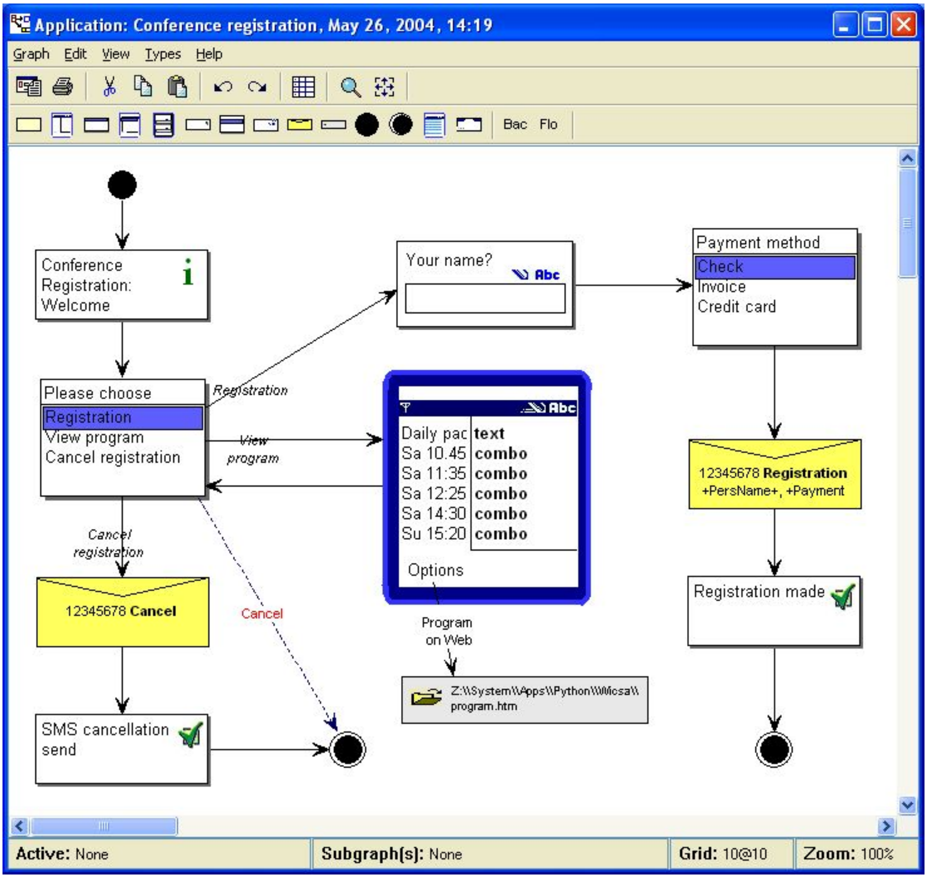


Fig. 3. DSM example: Smartphone UI applications

The generator produces each widget and code calling the services of the phone platform. Some framework code was created for dispatching and for multi-view management (different tabs in the pane). By using domain-specific information, much modeling work could be saved: for instance, the normal behavior of the Cancel key is to return to the previous widget. Relationships for Cancel transitions thus need not normally be drawn, but can be automatically generated; only where Cancel behaves differently need an explicit relationship be drawn.

Identification, definition and testing of the language constructs were considered easier in this approach than any other language construct identification approach. Therefore, language creation could often be carried out by external consultants with only a little help from domain experts. Although the language definition was relatively straightforward, the main challenges seemed to be in relating other types of modeling elements and constraints to those constructs originating from the look and feel. If the look and feel constructs were sufficiently rich to also cover functionality, the level of abstraction of modeling was raised substantially beyond programming.

In many cases, the look and feel based cases had an existing framework, product platform or API, which formed a reasonably solid foundation for the key modeling language concepts. The APIs varied in their levels, from very low-level APIs near the code, to very abstract operations and commands. The simpler generators usually produced the code as a function per widget or similar state, with the end of the function calling the next function based on user input. Tail recursion was used to reduce stack depth where necessary. More complex generators produced state-based code, either in-line or as state transition tables. None of the languages based on look and feel required frequent reverse engineering, but some called for importing libraries as model elements. Usually only interfaces were required for these libraries, but in at least one case components with their implementation (i.e. whitebox) were needed. Generators targeting other implementation languages were not defined, although that was considered possible to achieve.

3.4 Setting the Variability Space

The final language definition approach was based on expressing variability (cases 17–23). Such cases were typical in product families, where languages were applied for variant design. Typically, the variability space was captured in the language concepts, and the modelers' role was to concentrate on the issues which differ between the products. All the cases that were based on describing variability had a platform that provided the common services the generated code interfaced with. This interfacing was typically based on calling the services of the platform, but there were also cases where generators produced the component code.

Languages describing variability were among the most difficult DSMs to create. The main reason was the difficulty to predict the future variants. This called for flexible language definitions that were possible to extend once new kinds of variations arose. Languages for pure static variability (often for configuration) were found relatively easy to create, however. The difficulty lay in behavioral variability and coming up with a language that supported building almost any new feature based on the common services of the platform. The success of the language creation was dependent on the product expert's knowledge, vision to predict the future, and insight to lay down a common product architecture. Therefore, the role of external consultants to support DSM creation was often smaller than with other approaches. In the best cases, though, the external consultant's experience of DSMs and generators complemented the expert's experience in the domain and its code. This normally required a consultant who was himself an experienced software developer (although not in that domain), and an expert who was not too bound to a low-level view of code.

In these cases language constructs were explored using domain analysis to identify commonalities and variabilities among the products to be built using model-based code generators. For example, Weiss and Lai [12] present a method to detect commonality and variability of both static and dynamic nature. Each variation point will be specified with variation parameters. By setting parameters for variation it offers a clear starting point for language concepts, like proposing data types and their variation space as well as constraints for combining variability. Feature modeling [3] was not applied to explore variability as it was found to operate at a level too general to identify DSM concepts. Feature models do not capture the dependencies and constraints that are required to define modeling constructs. Among the studied cases, product architecture served better to find product concepts and their interrelations.

A product family platform and its supporting framework also have a notable influence on the modeling language concepts and constraints. Commonalities were usually hidden into the generator or framework in addition to complex issues which can be solved in an automated generator. In many cases there were several different computational models used to support all the required views of the systems. For example, in embedded product families, it was common to follow the state machines with domain specific extensions to best describe the system's behavior and interactions.

The level to which abstraction was raised was dependent on the nature of variability. As would be expected, cases where the variability could be predicted reasonably well showed higher levels of abstraction than those where future variability could not be pinned down. A common solution for these latter cases was to make the modeling language and generators easy to extend, allowing the level of abstraction to be raised substantially now, and making it possible to maintain that level in the future.

4 Evaluation of the Categorization and DSM Definition Approaches

After having categorized the cases according to which of the four approaches were used, we noticed that each case had used only one or two approaches. Further, where there were two approaches, only certain pairs of approaches seemed to occur. Of all 16 possible pairs made up of a primary approach and a secondary approach, only 5 were actually found in the data. This prompted us to re-order the categories into the order now shown (previously generation output was last), so that each case used one approach and its successor or predecessor.

Cases performed mainly by the customer mostly occur early in the list. Conversely, those cases which had been performed by more experienced DSM practitioners tended to come later in the list. The order of approaches thus probably reflects an order of increasing DSM maturity.

Some cases were found to resemble others from the language point of view, although the product domain and generated code were different (e.g. the cases of ERP configuration and eCommerce marketplace).

Approach 1, domain expert's concepts, seems to provide little insight. In some cases it simply means that somebody else identified the concepts, and we thus lack the information of which of the other approaches they used. In the three cases where the customer was not mainly responsible for the concept identification, the DSM project has not progressed beyond an initial proof of concept. These cases thus probably reflect domains that are immature, and where the DSM consultants lacked previous experience that would have enabled them to raise the maturity in that domain.

In approach 2, generation output, there were significant differences between those cases whose generation output was itself an established domain-specific language, and those where the output was a generic language or an ad hoc or format such as a configuration file. Those cases worked best where the output was an established domain-specific language, because the domain was more mature and the company in question already had a mature implementation framework, either their own or from a third party. In both CPL cases, the companies wanted their own additions to the languages, further improving the domain specificity.

When the output is in a generic programming language, it would often be better apply an approach other than generation output, to truly raise the level of abstraction. When the output is to an immature format, it would often be better to analyze the domain further to improve its understanding and the output format, rather than build a direct mapping to the existing shaky foundation.

Approach 3, look and feel, can be regarded as the first of the four approaches that consistently yields true DSM solutions. It is thus a valid approach to apply in new DSM projects, whenever the end product makes it possible. It was also the most commonly applied approach, found in 13 out of 23 cases.

Approach 4, variability space, was only found in combination with approach 3. The cases where it was the primary approach, 20–23, were all in the domain of phone UI applications, generating C or C++ (case 16 was a simpler domain, a subset of these). These cases are certainly among the most complex, and this partly accounts for the similar solutions. A second major factor is that experience with previous similar cases had provided a proven kind of solution for this domain. Whilst each language was created from scratch, the knowledge of previous cases from this domain certainly influenced the way the cases were approached. The resulting DSM languages and in particular generators differed substantially, reflecting the different needs of the domains, customers and frameworks.

The use of the variability space approach in the radically different domain of ERP configuration (17 & 18) shows that this approach is not restricted to state-based embedded UIs. Perhaps the most likely explanation for this clustering of cases is that this approach requires the most experience from the language creators, and yet also offers the most power. In particular, the combination of the almost naïve end-user view of the look and feel approach with the deep internal understanding of the domain required by the variability space approach seems to yield the best solutions, particularly in the most complex cases. When used together, the look and feel approach tended to identify the basic concepts, and the variability space approach helped define relationships and what properties or attributes each concept should have.

5 Conclusion

In this paper we have examined approaches to identifying concepts for DSM languages, based on experiences collected from over 20 real-world cases. The cases show that there is no single way to build DSM languages: more than one language creation approach was applied in the majority of cases. In the cases studied, we identified four different approaches used by the domain expert, expert developer or DSM consultant.

Of the four approaches in our categorization, the first relied on the domain expert's intuition or previous analysis to identify concepts. This approach is essential in that it emphasizes the role of the expert, but forms a weak point of the categorization in that the experts themselves must normally have applied one of the other approaches. The second approach identifies concepts from the required generation output, and can only be recommended where that output is already a domain-specific language. The third and fourth approaches, end product look and feel and variability space, seem to be the best overall, although not applicable in every case. Using them together seemed particularly effective in raising the level of abstraction and speeding up development.

Defining a language for development work is often claimed to be a difficult task: this may certainly be true when building a language for everyone. The task seems to become considerably easier when the language need only work for one problem domain in one company. According to the cases analyzed the main difficulties are found in behavioral aspects and in predicting future variability. Almost all cases with both these difficulties required experienced DSM consultants, and all used more than one approach to identify concepts.

In all cases, DSM had a clear productivity influence due to its higher level of abstraction: it required less modeling work, which could often be carried out by personnel with little or no programming experience. The increase in productivity is not surprising, considering that research shows the best programmers consistently outperform average programmers by up to an order of magnitude. DSM embeds the domain knowledge and code skill of the expert developer into a tool, enabling all developers to achieve higher levels of productivity.

This paper targets automated derivation of software products based on design specifications. It examines and analysis experiences from practice of how DSM language creators identify and define modeling constructs. More research work is needed to better understand the DSM creation process, and to disseminate the skills to a wider audience. Particularly welcome would be empirical studies that cover more cases from various domains, and using different metamodeling facilities. As DSM use grows, research methods other than field and case studies would also be welcome, for example surveys and experiments.

References

1. Cleaveland, J. C., Building application generators, *IEEE Software*, July (1988)
2. Deursen van, A., Klint, P., Little languages: Little maintenance? *Journal of Software Maintenance*, 10:75-92 (1988)
3. Kyo, C., 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 (1990)
4. Lennox, J., et al., CPL: A Language for User Control of Internet Telephony Services. Internet Engineering Task Force, IPTEL WG, April (2004)
5. MetaCase, MetaEdit+ Method Workbench 4.0 User's Guide, www.metacase.com (2004)
6. Nokia Series 60 SDK documentation, version 2.0, 2 (www.forum.nokia.com/) (2004)
7. Patton, M., Qualitative Evaluation and Research Methods, Newbury Park, Sage, 2nd edition (1990)
8. Pohjonen, R., Kelly, S., Domain-Specific Modeling, *Dr. Dobb's Journal*, August (2002)

9. Rossi, M., Lyytinen, K., Ramesh, B., Tolvanen, J.-P., Managing Evolutionary Method Engineering by Method Rationale, Journal of the Association for Information Systems (AIS), (5) 9 article 12, (2004)
10. Sprinkle, J., Karsai, G., A domain-specific visual language for domain model evolution, Journal of Visual Languages and Computing, Vol 15 (3-4), Elsevier (2004)
11. Tolvanen, J.-P., Kelly, S., Domain-Specific Modeling (in German: domänenspezifische Modellierung) *ObjektSpektrum*, 4, July/August (2004)
12. Weiss, D., Lai, C.T.R., Software Product-line Engineering, Addison Wesley (1999)

Supporting Production Strategies as Refinements of the Production Process

Oscar Díaz, Salvador Trujillo, and Felipe I. Anfurrutia

ONEKIN Group, University of the Basque Country,
PO Box: 649,
20009 San Sebastián Spain
Phone: + 34 943 018 064
{oscar.diaz, struji, felipe.anfurrutia}@ehu.es

Abstract. The promotion of a clear separation between artifact construction and artifact assembling is one of the hallmarks of software product lines. This work rests on the assumption that the mechanisms for producing products considerably quicker, cheaper or at a higher quality, rest not only on the artifacts but on the assembling process itself. This leads to promoting production processes as "first-class artifacts", and as such, liable to vary to accommodate distinct features. Production process variability and its role to support either production features or production strategies are analyzed. As prove of concept, the *AHEAD Tool Suite* is used to support a sample application where features require variations on the production process.

1 Introduction

1.1 Problem Statement

Software Product Lines (SPLs) are defined as "*a set of software-intensive systems, sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way*" [7]. In this paper, we focus on "the prescribed manner" in which products are manufactured: **the production plan.**

A production plan is "*a description of how core assets are to be used to develop a product in a product line*" [6]. Among the distinct concerns involved in a production plan, this paper focuses on the **production process** which specifies how to use the production plan to build the end product [6]. As stated in [14] "*product production has not received the attention that software architecture or programming languages have*". It is often so tightly coupled to the techniques used to create the product pieces that both are indistinguishable. For example, integrated development environments (e.g. *JDeveloper*) make it seamless by automatically creating a build script for the project or system under development so that the programmer can be unaware of the process that leads to the end product.

Indeed, production plans have been traditionally considered as mere scripts, and left to the programmers that built the other artifacts. In a traditional setting, build scripts are often kludged together for that, built by people who would rather be writing source

H. Obbink and K. Pohl (Eds.): SPLC 2005, LNCS 3714, pp. 210–221, 2005.

code than developing a process. Such scripts are notorious for their poor or misleading documentation [9], which was thought to be consumed by other core-asset developers.

SPLs change this situation by explicitly distinguishing between core-asset developers and product developers where the latter are involved in intertwining the core assets to obtain the end product. This distinction not only reinforces a separation of concerns between programming and assembling, but explains the preponderant and strategic role that production plans have in SPLs. That is, there is a growing evidence that the mechanisms for producing products considerably quicker, cheaper or at a higher quality, rest not only on the components but on the assembling process itself. Despite this observation, most approaches just support a textual description of the production plan [5], where variability or requirements specific to the production plan are almost overlooked.

1.2 Our Contribution

Based on these observations, we strive to turn production processes into *"first-class artifacts*". Specifically, the main contribution of this paper rests on observing how the explicit and separate specification of the product process permits to account for variations at both the product and process level. To this end, the paper distinguishes between "product features" and "build-process features". By "product features" we meant those that characterize the product as such, whereas "build-process features" refer to variations on the associated process. Hence, two end products can share the same "product features" but being produced along distinct process standards.

We attempt to show some evidence of how this process variability impacts both the modifiability (i.e. variability along time) and the configurability (i.e. variability in the product space) of SPLs. To this end, these ideas are supported for *AHEAD* [3], a methodology for SPLs based on step-wise refinement. So far, the companion tool suite, *AHEAD TS,* (1) hides the production process into the integrated development environment, and (2) excludes build scripts from refinement. Hence, the upgrades include, (1) an explicit representation of the production process that AHEAD implicitly conducts, and (2) a refinement operator for production processes. Production processes are specified using *Ant* [12], a popular script language in the Java world.

The rest of the paper is structured as follows. Section 2 outlines the AHEAD methodology. Section 3 introduces the running example. Process refinement at work is the subject of section 4 and 5. Finally, conclusions are given.

2 A Brief on the AHEAD Methodology

Step-wise refinement (SWR) is a paradigm for developing a complex program from a simple program by incrementally adding details [11]. *GenVoca* is a design methodology for creating application families and architecturally extensible software, i.e., software that is customizable via module additions [2]. It follows traditional SWR with one major difference: instead of composing thousands of microscopic program refinements, *GenVoca* scales refinements so that each adds a whole **feature** to a program, being a feature a "*product characteristic that is used in distinguishing programs within a fam-*

ily of related programs" [3][1]. Hence, a final program (i.e. a product) is characterized as a sequence of refinements (i.e. features) applied to the core artifacts. This permits the authors to conceptualize the production process as a mathematical equation where core assets are mapped into constants, and refinements are functions that add features to the artifacts.

This approach is supported by the ***AHEAD Tool Suite*** *(AHEAD TS)* [3] where refinements to realize a feature are packaged into a **layer**. Broadly speaking, the base layer comprises the core artifacts, where lower layers provides the refinements that permit enhancing the core artifacts with a specific feature. The base layer is then, the root of the refinement hierarchy[2].

Layer composition implies the composition of the namesake artifacts found in each layer. Implementation wise, a layer is a directory. Hence, feature composition is directory composition. Artifact composition depends on the nature of the artifact. Hence, the composition operator is polymorphic. *Java* files, *HTML* files, *Ant* files will each have their own unique implementation of the composition operator.

For the perspective of the production process, it is most important to distinguish between:

- the intra-layer production process, which specifies the production process for the set of artifacts included within a layer or upper layers (from which artifacts are "inherited"). This is specified as *Ant* files in *AHEAD TS*. This would correspond to the "*product-build process*" in Chastek's terminology [6].
- the inter-layer production process, which specifies how layers are intertwined to obtain the end product. This is hard-coded in *AHEAD TS*. This is referred to as "*product-specific plan*" in Chastek's parlance [6].

Unfortunately, *AHEAD TS* does not consider yet *XML* artifacts. Since the production process is an *XML* document [3], production processes are not refined as such. Lower layers always override the *build.xml* file of upper layers so that the *build.xml* of the leaf layer is the only one that remains.

This implies that leaf layers should be aware of how to assemble the whole set of artifacts down in the refinement hierarchy. This could be a main stumbling block to achieve loose coupling among layers, and leads to increasing complex *build.xml* files as you go down in the layer hierarchy.

Turning production processes into first-class artifacts makes production processes liable to be refined as any other artifact. This permits to account for both "product features" and "process features". By "product features" we meant those that characterize the product as such, whereas "process features" refer to variations on the associated process.

[1] Other definitions of features include "*a logical unit of behavior that is specified by a set of functional and quality requirements*" [4] or "*a recognizable characteristic of a system relevant to any stakeholder*" [10].

[2] Design rules checking is also introduced to specify feature dependencies (e.g. selection of feature F1 disables feature F2) [1].

[3] *AHEAD TS* names it *ModelExplorer.xml*, but it plays the same role than *build.xml* in traditional Java projects.

It is worth noting that "product features" commonly impact the intra-layer production process (i.e. the process adds a new artifact to build the end product). By contrast, "process features" influence the inter-layer production process (i.e. the process that indicates how layers are intertwined). Again, this distinction reinforces the separation of concerns between asset developers and product developers.

Different upgrades were conducted into *AHEAD TS* to accommodate variability into the production process, namely

- intra-layer production processes are currently specified as *ANT* files [4]. Since *AHEAD TS* does not consider yet *XML* artifacts, and *ANT* files are *XML* document, production processes are not refined as such. Lower layers always override the *build.xml* file of upper layers so that the *build.xml* of the leaf layer is the only one that remains. This implies that leaf layers should be aware of how to assemble the whole set of artifacts down into the refinement hierarchy. This could be a main stumbling block to achieve loose coupling among layers, and leads to increasing complex *build.xml* files as you go down in the layer hierarchy. To overcome this situation, the *refinement* operator has been extended to handle *ANT* files.
- the inter-layer production process is hard-coded into the *AHEAD TS*. This production process is made explicit, and hence, subject to refinement.

Next sections illustrate the advantage of bringing refinement to the process realm through a running example.

3 The Sample Problem: *WebCalculator*

Batory et al. uses a Java-based calculator to illustrate how *AHEAD* can nicely accommodate the refinement process whereby features are gradually added to the core assets till the end product is obtained. In their example, refinements affect artifacts other than the product process [3].

We have used a similar domain but in a Web setting, and where variations mainly affect the production process. *WebCalculator* is a *J2EE* Web applications [16] which has been developed using Apache Struts[5]. A Web application refers to an aggregate of artifacts, namely

- *Java* class files (action classes), needed libraries, and resource files,
- *JSP* pages and their helper Java classes,
- Static documents: images, *HTML* pages, and so on,
- Web deployment descriptors, configuration files, and tag libraries.

Web applications (also known as *Web Modules*) are packaged into a *Web ARchive* (WAR) which follows a directory structure defined in *Java Servlet Specification* [8]. Mostly, this structure corresponds with *public_html* content which is shown on the left of figure 1.

[4] *AHEAD TS* names it *ModelExplorer.xml*, but it plays the same role than *build.xml* in traditional Java projects.

[5] http://struts.apache.org/

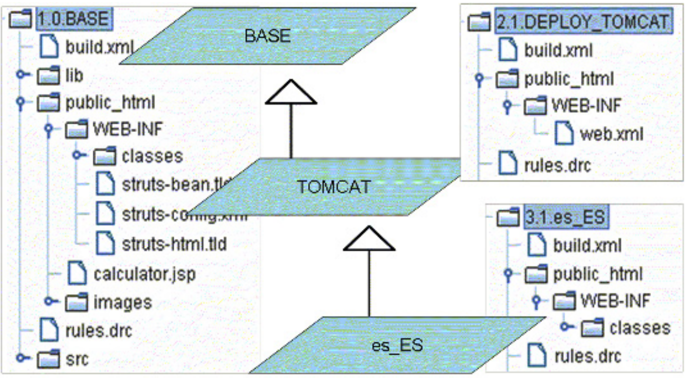


Fig. 1. Refinement layers: each layer accounts for a "product feature"

Broadly speaking, a layer comprises the set of artifacts that realize a given feature. This might include a process. Being in a Java setting, *Ant* is used to specify this process; the so-called, *build.xml* file. [15].

Ant is a Java-based tool for scripting build processes. Scripts are specified using XML syntax: <*project*> is the root element whose main child is <*target*>. A target describes a unit of enactment in the production process. This unit can be an aggregate of atomic tasks such as *compile, copy, mkdir* and the like.

The process itself (i.e. the control flow between targets) is described through a target's attribute: *"depends"*. A target is enacted as long as the target it depends to, has already been enacted. This provides a backward-style description of the process flow. Data sharing between targets is achieved through the external file directory.

Figure 2[6] shows a snippet of the specification of the production process for the *base* layer. According with this figure, the production plan includes the following steps:

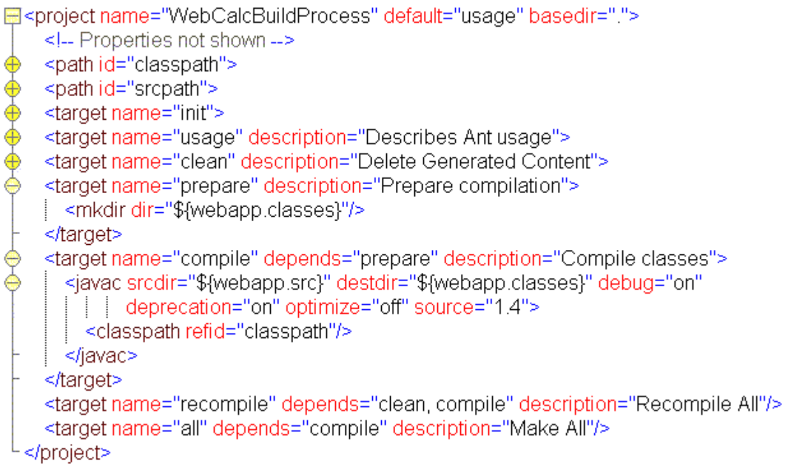
1. compile *Java* classes, and get the byte code,
2. package artifacts (classes, libraries, pages, resources, etcetera) into a *WAR* file,
3. deploy web application into a container.

The use of *Ant* for specifying product processes is not new. After all, *Java* programmers have been using *Ant* as a scripting language for years. However, instead of burying it into the integrated development environment, we make it explicit as any other artifact. This allows for refinements.

4 Intra-layer Production Process Refinement

Previous section describes the *base* layer of *WebCalculator*. This *base* layer might then be refined to account for distinct "product-features". The example introduces two features which imply a refinement in the production process, namely

[6] Space limitations prevent us for given the full *build.xml* files. Some targets are collapsed and variables are defined in properties files.



```
<project name="WebCalcBuildProcess" default="usage" basedir=".">
  <!-- Properties not shown -->
  <path id="classpath">
  <path id="srcpath">
  <target name="init">
  <target name="usage" description="Describes Ant usage">
  <target name="clean" description="Delete Generated Content">
  <target name="prepare" description="Prepare compilation">
    <mkdir dir="${webapp.classes}"/>
  </target>
  <target name="compile" depends="prepare" description="Compile classes">
    <javac srcdir="${webapp.src}" destdir="${webapp.classes}" debug="on"
        deprecation="on" optimize="off" source="1.4">
      <classpath refid="classpath"/>
    </javac>
  </target>
  <target name="recompile" depends="clean, compile" description="Recompile All"/>
  <target name="all" depends="compile" description="Make All"/>
</project>
```

Fig. 2. Product process: base artifact

- the container feature. The variants include *Tomcat* [7] and *JBoss*[8]. By default, there is no *base* web container [9],
- the locales feature. The alternatives are *EN* (i.e. English), *es_ES* (Spanish at Spain), and *eu_ES* (Basque at Spain). The base locale is *EN*.

Figure 1 shows one possible layer composition, which equation is *"es_ES(Tomcat(base))"*. The *base* layer contains the base artifacts, whereas the other layers contain either refinements on existing artifacts or new artifacts. The important point to notice is that both *Tomcat* and *es_ES* features imply the refinement of the product process. That is, deploying *WebCalculator* in *Tomcat* requires to refine the *build.xml* accordingly.

AHEAD does not provide a way to refine *XML* artifacts. However, Batory et al. state the Principle of Uniformity whereby *"when introducing a new artifact* (type), *the tasks to be done are (1) to support inheritance relationships between instances and (2) to implement a refinement operation that realizes mixin inheritance*"[3].

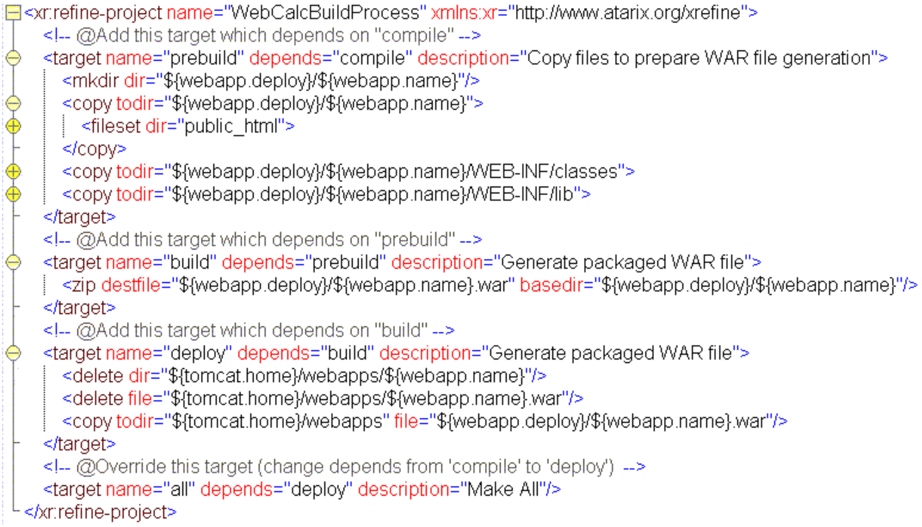
This principle is realized for *build.xml* artifacts as follows. Inheritance is supported by building on the uniqueness of the <*target*> name within a given <*project*>. Basically, the project maps to the notion of class, and the target corresponds to a method. This permits to re-interpret inheritance for *Ant* artifacts by introducing the following tags:

1. <*xr:refine-project*> which denotes a project refinement (a kind of "is_a"),
2. <*xr:super-target/*> which is the counterpart of the *"super"* constructor found in object-oriented programming languages

[7] http://jakarta.apache.org/tomcat/

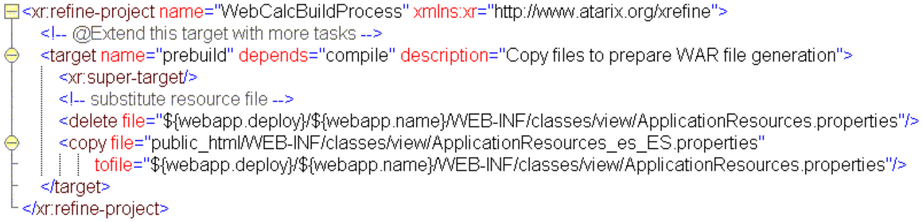
[8] http://www.jboss.org/

[9] A design rule can be used here to ensure that the final product will have a container.



```
<xr:refine-project name="WebCalcBuildProcess" xmlns:xr="http://www.atarix.org/xrefine">
  <!-- @Add this target which depends on "compile" -->
  <target name="prebuild" depends="compile" description="Copy files to prepare WAR file generation">
    <mkdir dir="${webapp.deploy}/${webapp.name}"/>
    <copy todir="${webapp.deploy}/${webapp.name}">
      <fileset dir="public_html">
    </copy>
    <copy todir="${webapp.deploy}/${webapp.name}/WEB-INF/classes">
    <copy todir="${webapp.deploy}/${webapp.name}/WEB-INF/lib">
  </target>
  <!-- @Add this target which depends on "prebuild" -->
  <target name="build" depends="prebuild" description="Generate packaged WAR file">
    <zip destfile="${webapp.deploy}/${webapp.name}.war" basedir="${webapp.deploy}/${webapp.name}"/>
  </target>
  <!-- @Add this target which depends on "build" -->
  <target name="deploy" depends="build" description="Generate packaged WAR file">
    <delete dir="${tomcat.home}/webapps/${webapp.name}"/>
    <delete file="${tomcat.home}/webapps/${webapp.name}.war"/>
    <copy todir="${tomcat.home}/webapps" file="${webapp.deploy}/${webapp.name}.war"/>
  </target>
  <!-- @Override this target (change depends from 'compile' to 'deploy') -->
  <target name="all" depends="deploy" description="Make All"/>
</xr:refine-project>
```

Fig. 3. Product process: refinement for feature *Tomcat*



```
<xr:refine-project name="WebCalcBuildProcess" xmlns:xr="http://www.atarix.org/xrefine">
  <!-- @Extend this target with more tasks -->
  <target name="prebuild" depends="compile" description="Copy files to prepare WAR file generation">
    <xr:super-target/>
    <!-- substitute resource file -->
    <delete file="${webapp.deploy}/${webapp.name}/WEB-INF/classes/view/ApplicationResources.properties"/>
    <copy file="public_html/WEB-INF/classes/view/ApplicationResources_es_ES.properties"
        tofile="${webapp.deploy}/${webapp.name}/WEB-INF/classes/view/ApplicationResources.properties"/>
  </target>
</xr:refine-project>
```

Fig. 4. Product process: refinement for feature *es_ES*

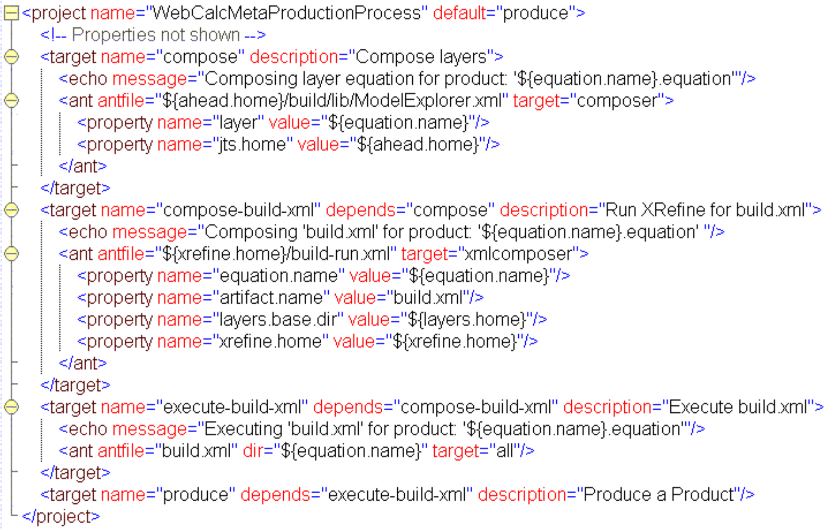
Hence a <*refine-project*> can refine a <*project*> by introducing a new <*target*>, extending a previously existing <*target*> (calling <*super-target*>) or overriding a <*target*> (by introducing this target with new content).

An example is given for the *Tomcat* feature (see figure 3). Feature *Tomcat* permits to deploy *WebCalculator* in the namesake container. This requires the refinement of the *build.xml* artifact found in the base layer, as follows:

- a new <*target*> is added to prepare *WAR* building (prebuild),
- a new <*target*> is added to build *WAR* (build) specific for *Tomcat*,
- a new <*target*> is added to deploy it into the *Tomcat* container,
- target <*target name= "all"*> is overridden.

Likewise, feature *es_ES* overrides the English locale of the *base* layer to the Spanish locale. The counterpart refinement is shown in figure 4. It includes extending <*target name= "prebuild"*> to copy appropriate resource files. Here the <*xr:super-target/*> constructor is used[10].

[10] It is worth noticing that the *es_ES* refinement requires the container been already selected. This implies a design rule to regulate how layers are intertwined.



```
<project name="WebCalcMetaProductionProcess" default="produce">
  <!-- Properties not shown -->
  <target name="compose" description="Compose layers">
    <echo message="Composing layer equation for product: '${equation.name}.equation'"/>
    <ant antfile="${ahead.home}/build/lib/ModelExplorer.xml" target="composer">
      <property name="layer" value="${equation.name}"/>
      <property name="jts.home" value="${ahead.home}"/>
    </ant>
  </target>
  <target name="compose-build-xml" depends="compose" description="Run XRefine for build.xml">
    <echo message="Composing 'build.xml' for product: '${equation.name}.equation' "/>
    <ant antfile="${xrefine.home}/build-run.xml" target="xmlcomposer">
      <property name="equation.name" value="${equation.name}"/>
      <property name="artifact.name" value="build.xml"/>
      <property name="layers.base.dir" value="${layers.home}"/>
      <property name="xrefine.home" value="${xrefine.home}"/>
    </ant>
  </target>
  <target name="execute-build-xml" depends="compose-build-xml" description="Execute build.xml">
    <echo message="Executing 'build.xml' for product: '${equation.name}.equation'"/>
    <ant antfile="build.xml" dir="${equation.name}" target="all"/>
  </target>
  <target name="produce" depends="execute-build-xml" description="Produce a Product"/>
</project>
```

Fig. 5. Layer-composition product Process: *base* artifact

Both examples illustrate how refinements have been realized for *Ant* artifacts. Implementation wise, the composition operator for *Ant* is implemented using *XSLT* and *XUpdate* [13]. This operator can be integrated within *AHEAD TS* so that when *build.xml* artifacts are found, the composition process is governed by the *Ant* plug-in.

5 Inter-layer Production Process Refinement

Previous section focuses on *Ant* artifacts found within a layer. These artifacts describe the "*product-build process*" within a layer. By contrast, this section focuses on layer-composition processes that state how layers themselves should be composed. This comprises the steps of the methodology being used. For AHEAD, these steps include:

1. feature selection. Output: a feature equation (e.g. *"es_ES(Tomcat(base))")*.
2. feature composition (i.e. layer composition in Batory's parlance). Output: collective of artifacts that support an end product.
3. enactment of the *build.xml* associated with the end product. Output: end product ready to be used.

Figure 5 illustrates the targets that realize previous steps *(*the *equation.name* property holds the feature equation):

- *compose*, which calls the *AHEAD TS* composer,
- *compose-build-xml*, which supports the composition operator for *build.xml* artifact that *AHEAD TS* lacks,
- *execute-build-xml,* which runs the *Ant* script supporting the production process of the end product,
- *produce*, which performs the whole production.

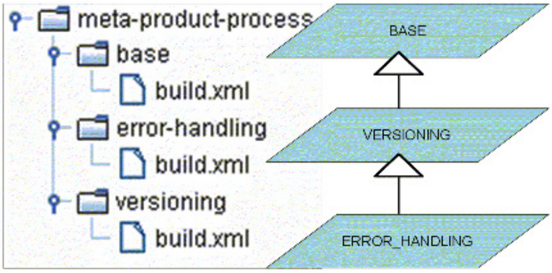


Fig. 6. Refinement layers: each layer accounts for a “process feature”

The enactment of this layer-composition script leads to an end product that exhibits the features of the input equation. *AHEAD TS* hard-codes this script.

However, this work rests on the assumption that the mechanisms for producing products considerably quicker, cheaper or at a higher quality, rest not only on the artifacts but on the assembling process itself. From this viewpoint, the inter-layer production process can accommodate important production strategies that affect the process rather than the characteristics of the final product. These strategies can affect the product costs, increase product quality, or improve the production process.

Based on this observation, the previous *base* layer might be refined to account for distinct “process-features”. The example introduces two features which imply a refinement in this layer-composition process, namely

- the *version* feature. Consider that security reasons recommend to version each new delivery of an end product. This implies that artifacts that conform the end product, should have appropriate backups.
- the *errorHandling* feature. Errors can rise during the production process. How these errors are handled is not a characteristic of the product but depends on managerial strategies. Hence, the base process can be customized to support distinct strategies depending on the availability of resources or the quality requirements of the customer.

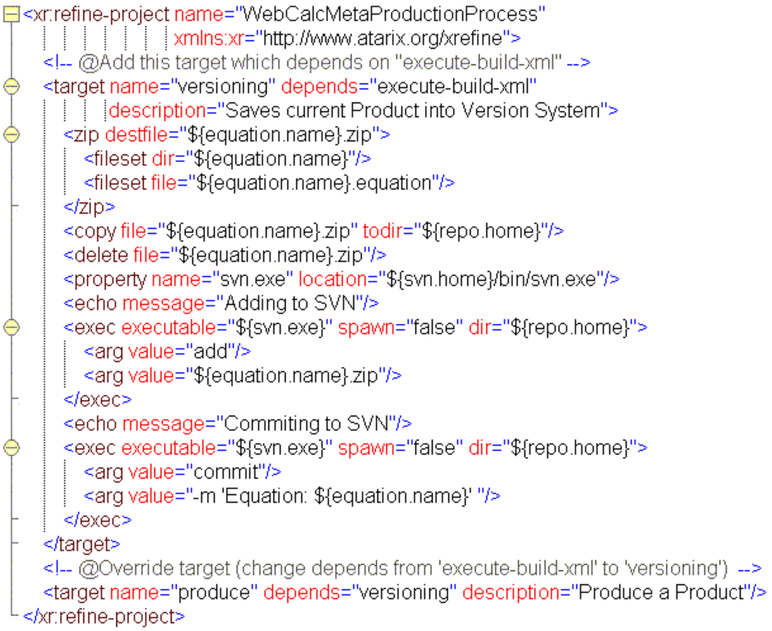
Figure 7 shows how the *base* process can now be refined to account for the *version* feature, namely:

- a new <*target*> is added to back up artifacts into the versioning system. For this purpose, *Subversion* is used[11],
- target <*target name= “produce”*> is overridden.

The equation *versioning(base)* leads to a *“product-specific plan”* that supports the naive security policy of the organization. As further experience is gained, and stringent demands are placed, more sophisticated plans can be defined.

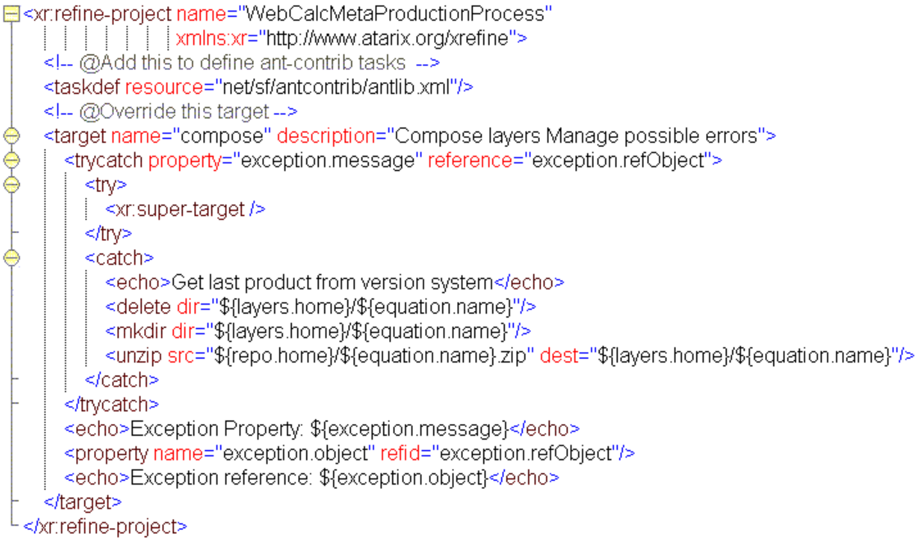
Likewise, figure 5 shows the *“substitution_eh”* policy for error handling:

[11] http://subversion.tigris.org/



```
<xr:refine-project name="WebCalcMetaProductionProcess"
                   xmlns:xr="http://www.atarix.org/xrefine">
  <!-- @Add this target which depends on "execute-build-xml" -->
  <target name="versioning" depends="execute-build-xml"
          description="Saves current Product into Version System">
    <zip destfile="${equation.name}.zip">
      <fileset dir="${equation.name}"/>
      <fileset file="${equation.name}.equation"/>
    </zip>
    <copy file="${equation.name}.zip" todir="${repo.home}"/>
    <delete file="${equation.name}.zip"/>
    <property name="svn.exe" location="${svn.home}/bin/svn.exe"/>
    <echo message="Adding to SVN"/>
    <exec executable="${svn.exe}" spawn="false" dir="${repo.home}">
      <arg value="add"/>
      <arg value="${equation.name}.zip"/>
    </exec>
    <echo message="Commiting to SVN"/>
    <exec executable="${svn.exe}" spawn="false" dir="${repo.home}">
      <arg value="commit"/>
      <arg value="-m 'Equation: ${equation.name}' "/>
    </exec>
  </target>
  <!-- @Override target (change depends from 'execute-build-xml' to 'versioning') -->
  <target name="produce" depends="versioning" description="Produce a Product"/>
</xr:refine-project>
```

Fig. 7. Layer-composition process: refinement for feature *versioning*



```
<xr:refine-project name="WebCalcMetaProductionProcess"
                   xmlns:xr="http://www.atarix.org/xrefine">
  <!-- @Add this to define ant-contrib tasks -->
  <taskdef resource="net/sf/antcontrib/antlib.xml"/>
  <!-- @Override this target -->
  <target name="compose" description="Compose layers Manage possible errors">
    <trycatch property="exception.message" reference="exception.refObject">
      <try>
        <xr:super-target />
      </try>
      <catch>
        <echo>Get last product from version system</echo>
        <delete dir="${layers.home}/${equation.name}"/>
        <mkdir dir="${layers.home}/${equation.name}"/>
        <unzip src="${repo.home}/${equation.name}.zip" dest="${layers.home}/${equation.name}"/>
      </catch>
    </trycatch>
    <echo>Exception Property: ${exception.message}</echo>
    <property name="exception.object" refid="exception.refObject"/>
    <echo>Exception reference: ${exception.object}</echo>
  </target>
</xr:refine-project>
```

Fig. 8. Layer-composition process: refinement for feature *errorHandling*

- a new <*taskdef/*> is added in order to extend *Ant* targets with try&catch routines[12].
- target <*target name= "compose"*> is overridden in order to handle possible errors. The base task *"compose"* is monitored so that when an error occurs, the last error-free version of the artifacts outputted by *"compose"* are taken. This policy might be applicable under stringent time demands or if debugging programmers are on shortage.

The "process feature" equation (*"substitution_eh(version(base)))"*) then strives to reflect the managerial and strategic decision that govern the production plan. Making theses strategies explicit facilitates knowledge sharing among the organization, facilitates customization, eases evolution, and permits to manage resources for product production in the same way as the product itself.

The latter is shown for the *version* and *errorHandling* features: a design rule is needed to state that the *substituion_eh* policy requires the *version* feature to be in place. The *version* feature in turn requires a new artifact, namely, *subversion*. It is a well-known fact among programmers of complex systems, that setting the appropriate environment is a key factor for efficient and effective throughput. SPLs are complex systems, and SPL techniques should be used not only to manage the artifacts of the product itself, but also those artifacts that comprise the environment/framework where these products are built. These include a large number of artifacts such a compilers, debugger, monitors or backup systems. Making explicit the layer-composition process facilitates this endeavour.

6 Conclusions

The clear separation between artifact construction and artifact assembling is one of the hallmarks of software product lines. However, little attention has been devoted to the assembling process itself, and how this process might realize important process strategies.

This work strives to illustrate the benefits of handling production processes as "*first-class artifacts*", namely:

- it permits to focus on how the product is produced rather than at what the product does. Programmers and assemblers can wide their minds to ascertain how features might affect the process itself so that scripting is no longer seen as a byproduct of source code writing,
- it extends variability to the production process itself.

Using *Ant* for process specification, and *AHEAD* as the SPL methodology, this work illustrates this approach for a sample application. Our next steps include to increase the evidence of the benefit of the approach by addressing more complex problems, and to investigate on the impact that distinct SPL quality measures have into the production process.

[12] This is achieved using Ant-Contrib at http://ant-contrib.sourceforge.net/

Acknowledgments. This work was partially supported by the Spanish Science and Education Ministry (MEC) under contract TIC2002-01442. Salvador Trujillo enjoys a doctoral grant for the MEC.

References

1. D. Batory and B.J. Geraci. Composition Validation and Subjectivity in Genvoca Generators. IEEE Transactions on Software Engineering, 23(2):67.82, February 1997.
2. D. Batory and S. O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. ACM Transactions on Software Engineering and Methodology, 1(4):355.398, October 1992.
3. D. Batory, J.Neal Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. IEEE Transactions on Software Engineering, 30(6):355.371, June 2004.
4. J. Bosch. Design & Use of Software Architectures - Adopting and Evolving a Product Line Approach. Addison-Wesley, 2000.
5. G. Chastek, P. Donohoe, and J.D. McGregor. Product Line Production Planning for the Home Integration System Example. Technical report, CMU/SEI, September 2002. CMU/SEI-2002-TN-029.
6. G. Chastek and J.D. McGregor. Guidelines for Developing a Product Line Production Plan. Technical report, CMU/SEI, June 2002. CMU/SEI-2002-TR-06.
7. P. Clements and L.M. Northrop. Software Product Lines - Practices and Patterns. Addison-Wesley, 2001.
8. D. Coward and Y. Yoshida. JSR 154, Java Servlet 2.4 Specication, 2003. http://www.jcp.org/en/jsr/detail?id=154.
9. J. Creasman. Enhance Ant with XSL Transformations, 2003. http://www- 128.ibm.com/developerworks/xml/library/x-antxsl/.
10. K. Czarnecki and U. Eisenecker. Generative Programming. Addison-Wesley, 2000.
11. E.W. Dijkstra. A Discipline of Programming. Prentice Hall, 1976.
12. Apache Software Foundation. Apache Ant. http://www.ant.apache.org/.
13. A. Laux and L. Martin. XUpdate - XML Update Language. http://xmldborg. sourceforge.net/xupdate.
14. J.D. McGregor. Product Production. Journal Object Technology, 3(10):89.98, November/December 2004.
15. N. Serrano and I. Ciordia. Ant: Automating the Process of Building Applications. IEEE Software, 21(6):89.91, November/December 2004.
16. I. Singh, B. Stearns, and M. Johnson. Designing Enterprise Applications with the J2EE Platform. Addison-Wesley, 2002.

Using Variation Propagation for Model-Driven Management of a System Family[*]

Patrick Tessier[1], Sébastien Gérard[1], François Terrier[1], and Jean-Marc Geib[2]

[1] CEA/List Saclay, F-91191 Gif sur Yvette Cedex, France
{Patrick.Tessier, Sebastien.Gerard, Francois.Terrier}@cea.fr
[2] LIFL, Laboratoire d'informatique Fondamentale de Lille,
Université des Sciences et Technologies de Lille,
59655 Villeneuve d'Ascq Cedex
Jean-Marc.Geib@lifl.fr

Abstract. A system family model (SFM) contains a set of common elements and a set of variable elements known as variation points. Variability modeling is a source of numerous problems: how to express variations, how to ensure the consistency of various views and avoid conflicts. Does the SFM cover all the desired systems? To obtain a specific system, known as "derivation", also known as a product, it is necessary to choose certain variation points from among those included in the SFM model by using a feature model (built during application domain analysis) or a decision model (after SF modelling). The SyF approach presented in this article proposes the "variation point propagation" concept as a means for achieving consistency and dealing with potential conflicts between variations. Under this approach, a decision model, generated from the SFM alone, then enables system family management: analyze coverage of the SF application domain, automate the derivation.

1 Introduction

The model-driven approach to system development involves the following key phases (Fig. 1. Process used to build a product):

- *Modeling the system to user requirements*. This phase is iterative and progressive. The system model becomes gradually more detailed and complies, at each successive increment, with the expressed requirements. For the approach described in this article, the language used for modeling is UML.
- *Code generation and compilation*, which provide the final application from its model.

A solution for shortening "time-to-market" for systems sharing certain features is to apply the development principles associated with the system family concept [1, 2]. The idea is to build a single model that factorizes parts of models common to all

[*] This work has been partially supported by the Families European project. Programme, ITEA project ip 02009.

H. Obbink and K. Pohl (Eds.): SPLC 2005, LNCS 3714, pp. 222 – 233, 2005.

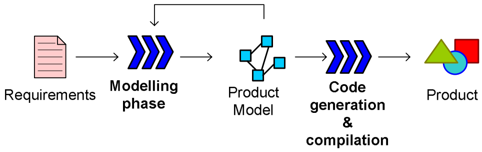


Fig. 1. Process used to build a product

systems in a same family and expresses system differences through inclusion of variable elements. This generic model is then known as the SFM or "system family model".

The system family design process usually calls for the following phases (Fig. 2. System family design process) :

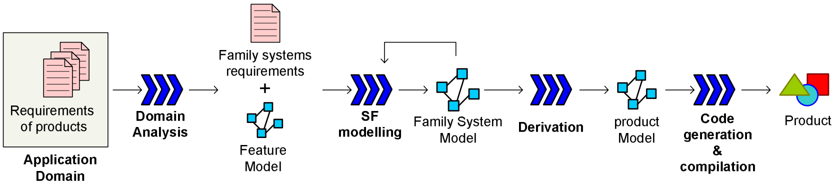


Fig. 2. System family design process

- *Domain analysis:* the application domain covers all requirements set for the products. This initial phase, derived from the FODA approach [3], is intended to classify system family requirements and produce a feature diagram. The feature diagram is a tree structure in which each node corresponds to a feature defined as "a prominent or distinctive user-visible aspect, quality or characteristic of a software system or systems" [3]. The characteristics of this tree can be optional, mandatory or linked by an "xor"-type relationship.
- *SF Modeling:* The second phase consists of building the system family model on the basis of these requirements. The SFM contains both variable elements and elements common to all of the desired products. Such modeling is progressive and takes place by increments.
- *Derivation*: Phase three calls for obtaining an application model from the SFM, which serves as its "pattern". To do so, the designer must make a set of choices with regard to the variation points contained in the SFM.
- *Code generation & compilation:* Phase four includes code generation based on the application model derived from the SFM, then compilation of this code to obtain the final product.

For system family designers, joint use of UML and a model-driven approach holds out the promise of enhanced productivity and better quality development, through possible automation of some stages. However, these advantages go hand in hand with certain constraints, which include need for:

- *expressing variability in a model* – this implies use of an expressive language that is sophisticated enough to describe every possible variation in a system family model. There may also be dependency relationships between variations included in the requirements. It must therefore also be possible to place interdependency constraints on the relevant variation points.
- *ensuring variation point consistency between SFM views* – in accordance with the separation of concerns principle, a model is often described via several complementary and interacting views: class diagrams, sequence diagrams, state diagrams, etc. Such views are in fact dependent on each other. For example, an event triggering a state machine transition may be linked to an operation call. If that element becomes variable in one view, there may also be impact on the other views in the model. In a multiview environment such as UML, it is necessary to provide for variability consistency management.
- *validating full coverage by the SFM of SF requirements* – in the system family development context, all possible systems must be defined for a given family. This requirement is described by the feature diagram. The difficulty consists of ensuring that once the SFM has been fully described, it will allow perfect derivation of all products defined in the feature diagram.
- *deriving a system from its SFM*. The question is how to derive from the SFM a well-formed system, and where a variation has been specified, how to modify the model to account for its impact.

There are numerous approaches that implement processes like the one shown in Fig. 2. System family design process. In the FODA-type approaches from which the feature model concept originates [3-6], there is no special formalism for expressing variability. These approaches rely essentially on the feature diagram for management of system family variations. Their main challenge is to devise an SFM that complies with the feature diagram [7] while ensuring consistency between variation points.

Unlike the above, the Kobra approach [8] provides a special formalism for expressing variability. It also defines a set of rules to keep SFM views consistent. Its strong point is facilitating derivation. To do so, it relies on use of a decision model. This model is devised as a treelike check list. The list asks a series of questions about decisions on SFM options and gives possible answers, which may in turn refer to new decisions. Each choice adds to the decision model a set of instructions for modifying the SFM so that the desired product can be obtained. If, for example, an operation is marked "optional", a choice not including that operation will result in instructions to delete it. To obtain a given product, the designer merely follows the relevant decision model path and applies the corresponding instructions to the SFM. The only drawback of this approach is that the SF designer is responsible for building the decision model, with a resulting risk of inconsistencies between this model and SFM variation points. There is no means for verifying overall consistency.

The Triskell team approach described in [9-11] is characterized by use of a feature diagram and a decision model. The decision model is a tree whose nodes are classes linked to each other by inheritance relationships. Each class is then responsible for generating a specific system. The classes are designed to "own" the operations applicable to an SFM for obtaining a particular system. This approach is also characterized by use of OCL rules to ensure consistency between model views.

However, it provides no means for ensuring that the SFM includes all models of systems belonging to a same family.

One way of affording consistency in variations from one SFM view to another under the approaches studied here is to set appropriate rules. None of these approaches can, however, guarantee full coverage by the SFM of the systems to be produced. The derivation stage remains a stumbling block. In FODA-type approaches, this problem is transferred to the SFM, which must ensure compliance with the feature model. The approaches described in [8-11] call for derivation to take place via the decision model. The designer nevertheless remains responsible for creating the decision model and must manually indicate the impact of variations on the SFM.

The following table summarizes the positions of these different approaches with regard to the four main difficulties inherent in developing system families like the one selected for our study.

	FODA-type approach	**Kobra**	**Triskell approach**
Expression of variability in the SFM	no special formalism	yes	yes
Consistency of variation points in the SFM	no	Set of textual rules	Set of OCL rules
Affording full SFM coverage of product models belonging to a given family	no	no	no
SFM derivation	Feature model	Decision model built by the designer	Decision model built by the designer

Fig. 3. Comparison summary

Analysis of this table shows that a set of textual or OCL rules can be used to ensure variation point consistency. None of the approaches studied can, however, validate SFM coverage of all products in a given family. Note that, while SFM derivation can take place via a decision model, the latter must always be built manually.

The SyF approach presented here therefore proposes the following solutions:

- for expressing variability in the SFM – use of a structure known as a variation group, to supplement variation points for the purpose of simplifying the expression of constraints on said points.
- for achieving variation point consistency in the SFM – a variation propagation mechanism.
- for verifying SFM coverage of family product models – a mechanism to automatically generate decision models from an SFM.
- for derivation from the SFM – a mechanism to generate products from an SFM and the decision tree.

The following pages are divided into several sections. Sections 2 and 3 cover the first two aspects described above, i.e. expressing variability and ensuring consistency. Section 4 addresses the two remaining aspects by describing the decision model concept and its utilization.

2 Expressing Variability in the SFM

An SFM consists of elements common to a given set of systems and elements that vary from one system to another. SFM description thus implies devising concepts to express variability within the model. The purpose of this section is to propose means to do so. A simple case study based on a family of watches is used to illustrate the proposals.

2.1 "Watch" Case Study

In subsequent paragraphs, a family of watches is used to illustrate the various mechanisms and applications of interest. The products included in this family can perform the following functions: display current time, display "dualTime", i.e. also show current time for another time zone, trigger an alarm (buzzer or vibration), act as a "heart rate controller" by displaying the rate of heartbeat.

Fig. 4. Feature diagram of the "watch" system family shows the diagram of features that depicts combinations of the different functionalities listed above and describes the products to be included in the system family.

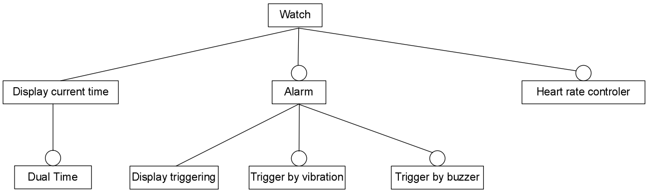


Fig. 4. Feature diagram of the "watch" system family

2.2 Variation Points and Variation Groups

An SFM factorizes several product models in one. It is therefore made up of common elements and variable elements, the latter being also known as variation points. Model elements not marked as variation points are implicitly considered common to all systems.

Specifying that a model element is variable is not sufficient to describe an SFM. This is because variabilities not only serve to locally specify a model, they also constrain one another. To fill the gap left here, several approaches propose mechanisms to add constraints between variations: in one approach [12], a dependency stereotyped *"requires"* is used to declare that a variation point requires another variation point; in another approach [13], OCL constraints are used to add variation point dependencies that likewise are a powerful mechanism for constraining variation points.

In order to support this concept of constraints between variation points, we have introduced the concept of variation group as depicted in Fig. 5. SyF metamodel for variability modeling. A variation group contains a set of variation points and constrains all owned elements such as sets of OCL constraints.

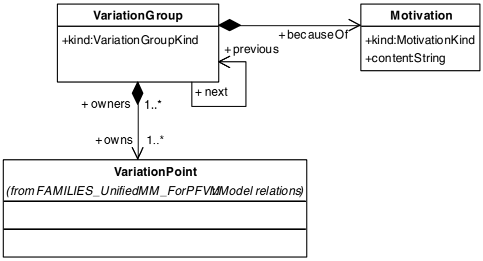


Fig. 5. SyF metamodel for variability modeling

A variation group may be any one of the types defined in the *VariationGroupKind* enumeration:

- **And** – all variation points are contained in the system model.
- **Alternative** – only one variation point is contained in the system model.
- **Optional** – each variation point may or may not be contained in the product model. Variation points of this kind of group are never constrained and remain independent.
- **OneAmongSeveral** – at least one variation point is contained in the system model. It is then possible for several variation points of this type to be contained in the system model.
- **Implication** – a variation point implies the existence of another variation point in the system model.

The variation group is always associated with a Motivation element. This model element is used to document the reason for a variation group. A *Motivation* class contains both of the following attributes:

- *kind* – defines the format of the comment: Natural language or OCL.
- *comment* – describes the motivation by type of language used.

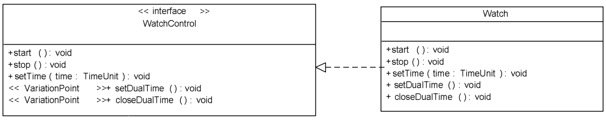

<< interface >>
WatchControl
+start ( ) : void
+stop ( ) : void
+setTime (time : TimeUnit) : void
<< VariationPoint >>+ setDualTime ( ) : void
<< VariationPoint >>+ closeDualTime ( ) : void
Watch
+start ( ) : void
+stop ( ) : void
+setTime (time : TimeUnit) : void
+ setDualTime ( ) : void
+ closeDualTime ( ) : void

Fig. 6. Two variation points for the DualTime feature

For the watch family, the DualTime mode is an example of a variable feature. In the SFM, the DualTime mode involves two operations *setDualTime* and *closeDualTime*, which are defined in the *WatchControl* interface (Fig. 8. Propagation of two variation points for the DualTime feature). The operations *setDualTime* and *closeDualTime* are therefore each given a "variation Point" stereotype (Fig. 6).

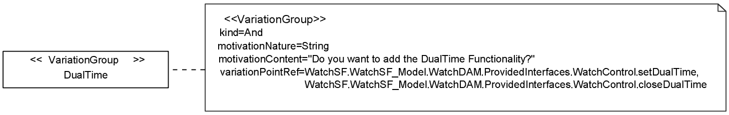

<< VariationGroup >>
DualTime
<<VariationGroup>>
kind=And
motivationNature=String
motivationContent="Do you want to add the DualTime Functionality?"
variationPointRef=WatchSF.WatchSF_Model.WatchDAM.ProvidedInterfaces.WatchControl.setDualTime,
WatchSF.WatchSF_Model.WatchDAM.ProvidedInterfaces.WatchControl.closeDualTime

Fig. 7. Example of variation group utilization

Both *setDualTime* and *closeDualTime* cover the same requirement – DualTime mode – and are thus linked to each another. Either the watch has a DualTime functionality and, if so, the WatchControl interface has both operations; or the watch does not offer this mode, and neither of the two operations should appear in the corresponding watch model. Both points of variation must therefore be constrained by a relation “And”.

2.3 Variation Transitivity

The SyF solution for achieving consistency between the views contained in a model and determining variation point impact in the SFM, is to propagate the variabilities specified by the user through a model. This causes new variation points, called "propagated points" or "propagated variation groups" to appear.

Transitivity mecanisms are classified into three categories:

- Variation transitivity patterns applying to the structural model: in this scheme, a variation point placed on an interface affects all the classes that implement it.
- Variation transitivity patterns applying from structural model to behavior model: in this scheme, a variation point placed on a class operation affects the transitions of the state machine associated with that class.
- Variation transitivity patterns applying to the behavior model: here a variation point on a machine state can affect outgoing and incoming transitions.

For behavior model purposes, the mechanisms of analysis developed by the SyF approach enable derivation of a protocol state machine without causing malformations. This step is based on use of formal techniques originating from graph theories to calculate state machines derived from a protocol machine with variability. For reasons of space, it cannot be described in greater detail here.

2.4 Application to the Watch Family

By applying the rules of variability propagation described in section 2.3, the DualTime variation group can be defined as described in Fig. 7. Example of variation group utilization. The relationship linking the two variation points is the "And" type. This constraint ensures that the two variation points are selected together, not independently from one another.

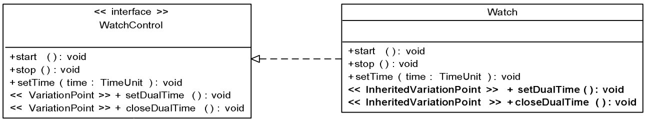

<< interface >>
WatchControl
+start ( ) : void
+stop ( ) : void
+ setTime (time : TimeUnit) : void
<< VariationPoint >> + setDualTime ( ) : void
<< VariationPoint >> + closeDualTime ( ) : void
Watch
+start ( ) : void
+stop ( ) : void
+ setTime (time : TimeUnit) : void
<< InheritedVariationPoint >> + setDualTime ( ) : void
<< InheritedVariationPoint >> + closeDualTime ( ) : void

Fig. 8. Propagation of two variation points for the DualTime feature

The *Watch* class *setDualTime* and *closeDualTime* operations then become variation points by transitivity (Fig. 8).

Under the SyF approach, two new variation groups are created to specify constraints relating to the impact of variation points (Fig. 9). The relationship between these new variation points and those at the origin of propagation is the "and" type. This constraint ensures that variation points will only appear together in a model.

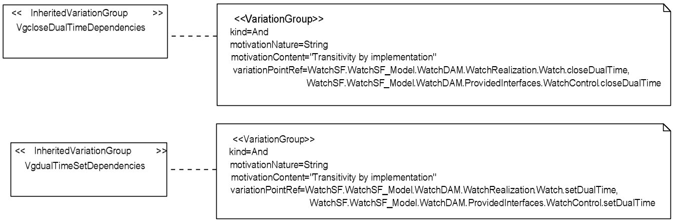

<< InheritedVariationGroup >>
VgcloseDualTimeDependencies
<<VariationGroup>>
kind=And
motivationNature=String
motivationContent="Transitivity by implementation"
variationPointRef=WatchSF.WatchSF_Model.WatchDAM.WatchRealization.Watch.closeDualTime,
WatchSF.WatchSF_Model.WatchDAM.ProvidedInterfaces.WatchControl.closeDualTime
<< InheritedVariationGroup >>
VgdualTimeSetDependencies
<<VariationGroup>>
kind=And
motivationNature=String
motivationContent="Transitivity by implementation"
variationPointRef=WatchSF.WatchSF_Model.WatchDAM.WatchRealization.Watch.setDualTime,
WatchSF.WatchSF_Model.WatchDAM.ProvidedInterfaces.WatchControl.setDualTime

Fig. 9. Example of a variation group appearing by transitivity

An SFM contains a set of variation points and a set of constraints. The variation points are placed and constrained by means of variation groups to ensure compliance with constraints induced by the feature diagram derived from system family requirements. The points are then propagated through the system family. This causes new variation points, called "propagated points," and new variation groups to appear.

Propagation ensures the consistency of various system views and enables identification of all elements impacted by a variation point.

Not only do the constraints introduced by variation groups guarantee the consistency of all variation points, they also induce a side effect for calculation of the different possible derivations. Product derivation consists of choosing a set of variation points that comply with all the constraints imposed by the variation groups. As a result, calculation of possible derivations involves calculating possible sets of variation points complying with these constraints.

3 Management of SFs with a Decision Model

A decision model is built to ensure appropriate SFM coverage and derive products from the SFM. This section gives a quick presentation of the metamodel used to describe the decision model, and explains how it serves as an SFM validation and derivation tool.

3.1 Language Concepts Required to Describe Decision Models

The decision model is intended as an aid to deriving an SM from an SFM (Fig. 10. Metamodel of concepts for describing decision models). It is a tree consisting of derivation paths, and each path is a sequence of possible choices related to the variation points and variation groups specified in the SFM. Each derivation path leads to a well-formed model of a possible product from the family of interest.

The decision model contains sets of decisions. For each decision, the designer must choose a resolution. This resolution implies a set of effects on the system family model. Such effects can be described using model transformation language (e.g. a QVT-compliant language).

The model-driven development (MDD) process calls for model transformation to be aimed at facilitating transition from an SFM to an SM. For this reason, the concept of *Effect* is defined and attached to any decision of a decision model. An effect may be the *TransformationEffect* type, which is in fact the model transformation required to account for a choice; and, in this process, the decision model, together with scripts for executing model transformation, are automatically generated.

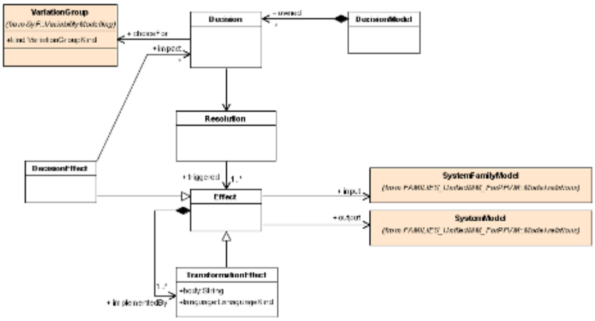


Fig. 10. Metamodel of concepts for describing decision models

3.2 Decision Modeling

By selecting a set of variation points, the user can choose a specific system. Decision modeling then consists of calculating all the sets of variation points that comply with all the constraints derived from the variation group. Such calculations can be long and complex and therefore require use of Quine McCluskey-type [8] constraints. Each set of solutions obtained results in a set of choices for the variation points. This in turn provides the resolutions needed to build a product. Each group of variations then becomes a potential element of decision.

To build the decision model, it then suffices to treat the sets of choices one by one. When the path is identical, no new decision nodes are created.

After adding all the sets of decisions and choices for all products in the decision model, the designer must identify any unnecessary decision nodes. An unnecessary node is one associated with a single choice. All such nodes are then eliminated. The effects of the associated choices are, however, preserved.

In the example used here as an illustration, the decision model is built from the set of systems whose decision sets are classified in the order defined by the designer. To obtain a new decision model, the user simply changes the order of the decisions.

The number of decision models depends on the number of decisions to be taken, i.e. the number of variation groups existing in the SFM.

For a number m of variation groups, there are $m!$ possible combinations. The computing time required to build all the decision models is long, and this is not necessarily beneficial to the designer.

To reduce the number of decision models linked to an SFM, the designer can therefore supply a partial or total order for the variation groups. To do so, he uses the "previous" and "next" properties of the VariationGroup stereotype.

Creation of the decision model facilitates efficient management and utilization of the SF model.

4 Use of the Decision Model

The decision model obtained by the process described above is a tree structure in which each branch corresponds to a specific system. This tree may take various forms. Its decision nodes may be binary or n-ary, depending on the number of possible choices for a decision. Fig. 11. Decision model for the watch family shows a decision model for a complex case allowing creation of 20 products.
Its decision nodes are organized horizontally and its products vertically.

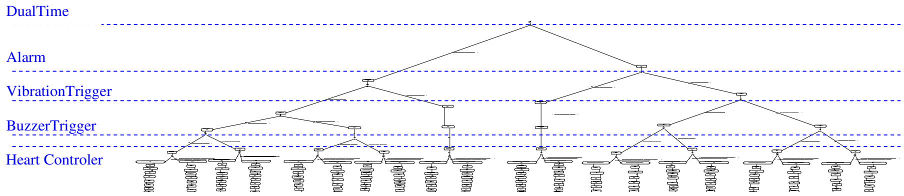


Fig. 11. Decision model for the watch family

4.1 Verifying Coverage with Respect to Application Domain

Firstly, the decision model verifies that the products covered by the SFM can be implemented. Where this is not the case, it is possible to identify any choices that are not possible and modify the constraints placed on the variations.

Comparison of the decision model with the feature model also systematically compares the truly feasible systems with those considered so at the time of application domain analysis. This approach therefore compares an SFM as it is (Decision model) with the requirement described by the feature diagram.

This comparison may identify numerous cases of:

- failure to produce the required systems: In such cases, the decision model does not allow production of the systems identified in the feature model.
- production of too many systems. The decision model then shows that systems not considered feasible at the time of analysis can in fact be implemented.

In both cases, the decision that causes deviation from the feature model can be easily identified. Since a decision corresponds to a group of variations, it is easy to identify the variations causing the problem and to modify either the variation points or the constraints included in the group of variations not complying with the feature diagram requirements.

This decision model can be used at any stage in SF modeling and thus guides the designer through said modeling stages by verifying compliance with user requirements at each stage.

The decision model therefore not only serves as a method for stage by stage SFM analysis, but can also support a tool that derives SF models for a specific system mode.

4.2 Derivation

The purpose of derivation is to choose a decision model path and transform it as necessary in the SF model.

To do so, the Effect element of the metamodel used to describe the decision model contains relevant model transformation instructions: impact of variation points in the SFM.

The principle for obtaining a specific derivation model is to collect all "Effect"-type elements from the path, execute their lists of instructions, then clean the model of all variability expression elements (variation groups, stereotypes, etc.).

For this purpose, an "SFManager" class is created in the SFM at the time of decision model creation. This class contains as many operations as systems that can be implemented. Each of the methods contains the model transformation codes for decision effects, along with a code for cleaning the SFM. The SFManager class also contains the operation 'run()' in which the designer writes the method call for the desired product.

5 Conclusion

System family approaches are applied to produce systems that have common features. The models built under these methodologies contain both common and variable elements. Several difficulties arise in the process of integrating variable elements. They relate to: *expressing variability in a system family model, ensuring consistency between SFM views, validating correct SFM coverage of requirements expressed and deriving an SFM.*

The SyF approach proposes mechanisms for managing these problems. By affording variation point transitivity throughout the model, it can ensure consistency and identify the impact of the variation points. The decision model can also verify coverage of the application domain and help the user to automate the derivation process.An SyF tool has been developed as a plugin for the Poseidon SDE[1]. Its tools are coded in MTL for model manipulation and generate MTL code for derivation purposes.

The objective of future research will be to refine variation management by adding management functionality to the activity and sequence diagrams.

References

1. Clements, P.C.: Software Product Lines – Basic Concepts and Research Challenges. in International Colloquium of the Sonderforschungsbereich 501. Tagungszentrum Betzenberg, Kaiserslautern (2003).
2. Clements, P. and L.M. Northrop : Software Product Lines: Practices and Patterns. Addison Wesley, Boston (2001).

[1]. http://www.gentleware.com

3. Kang, K.C., et al.: Feature-Oriented Domain Analysis (FODA). Carnegie Mellon University (1990).
4. Griss, M.L.: Implementing product line features with Component Reuse. In 6th International Conference on Software Reuse, Vienna, Austria (2000).
5. Svahnberg, M., J.v. Gurp, and J. Bosch: On the Notion of Variability in Software Product Lines. In IEEE/IFIP Conference on Software Architecture (WICSA 2001). (2001).
6. Riebisch, M., et al : Extending Feature Diagrams with UML Multiplicities. in 6th Conference on Integrated Design & Process Technology; Pasadena, California, USA (2002).
7. Chastek, G., et al: Product Line Analysis: A Practical Introduction. 2001. SEI Carnegie Mellon University (2001).
8. Atkinson, C., J. Bayer, and D. Muthig: Component-Based Product Line Development : The KobrA Approach. in the First Software Product Line Conference, Boston (2000).
9. Monestel, L., T. Ziadi, and J.-M. Jézéquel. Product Line Engineering : Product Derivation. Model Driven Architecture and Product Line Engineering, associated to the SPLC2 conference, San Diego (2002).
10. Ziadi, T., L. Hélouët, and J.-M. Jézéquel: Modeling behavior in product-lines. International Workshop on Requirements Engineering for Product Lines, Essen/Germany (2002).
11. Ziadi, T., J.-M. Jézéquel, and F. Fondement: Product line derivation with uml. in Software Variability Management Workshop. University of Groningen Departement of Mathematics and Computing Science, (2003)
12. Clauß, M: Modeling variability with UML. Net.ObjectDays.. Erfurt, Germany. (2001)
13. Ziadi, T., L. Hélouët, and J.-M. Jézéquel: Towards a UML Profile for Software Product Lines. International Workshop on Product Family Engineering. Seana / Italy, Springer-Verlag (2003).

Author Index