Rafael Capilla
Jan Bosch
Kyo-Chul Kang *Editors*

Systems and Software Variability Management

Concepts, Tools and Experiences

Springer

Systems and Software Variability Management

Rafael Capilla • Jan Bosch • Kyo-Chul Kang
Editors

Systems and Software Variability Management

Concepts, Tools and Experiences

Springer

Editors
Rafael Capilla
Rey Juan Carlos University
Madrid
Spain

Jan Bosch
Chalmers University of Technology
Gothenburg
Sweden

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

ISBN 978-3-642-36582-9 ISBN 978-3-642-36583-6 (eBook)
DOI 10.1007/978-3-642-36583-6
Springer Heidelberg New York Dordrecht London

Library of Congress Control Number: 2013940739

ACM Computing Classification (1998): D.2, K.6

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

There are several books on product line engineering, but most of these books either introduce specific product line techniques or include brief summaries of industrial cases or researches. From these sources, it is difficult to gain a comprehensive understanding of the various dimensions and aspects of software variability that product line engineering practitioners and researchers must understand. The book aims to address this gap by providing a comprehensive reference on the notion of variability modeling in the context of software product line engineering, to give an overview of the techniques proposed for variability modeling, and to give a general perspective on software variability management. We believe that practitioners as well as researchers and computer science students will gain a new insight into software, software engineering, and variability in product line engineering.

The most important attribute of software is the "softness" of software, i.e., software that is easy (cost-effective) to modify and adapt to evolving requirements or changing operating environments, easy to port on different hardware or software platforms, and easy to reuse for development of similar applications. The "softness" of software cannot be attained without engineering it into software. In order to "embed" softness into software, we need to understand the "space" of the commonality and variability of a family of related systems (i.e., a product line) and its evolution, and then organize and codify the knowledge gathered as a commonality and variability model. With this understanding, we can engineer software applying various design principles and embedding variation points that can later be bound with variants.

Once the initial variability of a software product line has been established and implemented, the focus shifts towards evolution of the provided variability in response to changes in the variability required from the software product line. Over time, new products are added to a product line, old products are removed, and the functionality that used to be highly differentiating and only used in the high-end products of the product line commodities is included in all products, removing the need for variation points. Thus, during evolution, the focus of variability management is as much on removing unnecessary variability as it is on adding new variation points in response to new needs. The focus on removing variability is important as the complexity of hundreds or thousands of variation points can easily become unwieldy.

One aspect contributing to the complexity of software variability management is the dependencies between variation points and between the variants available at each variation point. Where industrial software product lines frequently hold well over 1,000 variation points, the total number of variants is even larger. The variants cannot be freely selected independent of all other selected variants, but instead there are dependencies that need to be respected. This leads to a situation where the number of variation points is over a thousand, the number of variants a multiple of that, and the number of dependencies between variants and between variation points is of a similar or larger number. This explains the importance of intentional management of software variability: even though the inherent complexity of variability is already quite high, the total complexity easily becomes unmanageable if not kept under control.

A final factor increasing the importance of software variability is concerned with later binding of software variation points. Traditional pre-deployment configuration of products allows for testing of the specific configuration as a safety net for avoiding inconsistent configurations. However, run-time dynamicity is increasing in importance for virtually all software-intensive systems. In those situations, testing of the resulting configuration after a run-time change is complicated and often only the most basic of system integrity is verified. The trend towards late binding is indicative of the importance of software variability management even outside the traditional area of software product lines and is now becoming important for large software products that have significant installation time, startup time, and run-time configuration taking place. Consequently, proper management of variability avoiding inconsistent configurations at run-time through the use of first-class models is particularly important.

Above, we have raised four reasons to stress the importance of software variability management, i.e., modeling multiple products in a product line, evolution, complexity, and the shift towards later, even run-time, binding of variation points. As is illustrated in the industrial experiences part of the book, there are no theoretical problems without any bearing on industrial practice, but rather challenges originating in industrial practice that the software engineering research community has responded to. In our experience, mature software product lines may have ten thousand variation points or more and the number of legal configurations of products in the product line may number in the millions. Also, in our writing of the book and the interaction with contributing authors, we are increasingly becoming convinced that software variability management is evolving into a field of its own, rather than a subfield of software product lines. In all systems where configuration and run-time dynamism are important, software variability management offers a powerful toolbox to deal with the resulting complexity, independent of the system being part of a software product line or not.

From a software engineering research perspective, software variability management represents a complex, multifaceted problem that intersects with several traditional topics, including, among several others, software configuration management, run-time dynamism, domain specific languages, modeling, and software architecture. The field has borrowed techniques from these traditional fields, but in return also contributes back with new insights, approaches, and techniques.

The book is organized in four main parts which guide the reader into the various aspects and dimensions on software variability. Each chapter briefly summarizes "*What you will learn in this chapter*"; so expert and non-expert? Readers can easily locate what topics they will find, but it also describes areas of practice for the applicability of the concepts explained.

In Part I, we intentionally drive the reader to the major topics on software variability modeling, but as we do not have a specific chapter for variability management, the chapters included in this part should be seen as different sides of the management perspective. First, we introduce the paradigm of software product line engineering in Chap. 1, where Product Line Engineering is compared with traditional Software Engineering and the role of software variability management is highlighted for the current practice of product lines. We then explore various dimensions of commonality and variability (C&V) in Chaps. 2 and 3, separating C&V modeling into problem and solution space modeling, and constraints specification. Managing traceability between various C&V models and the notion of variability in time and space is also discussed. The dimension of feature binding time, the implications of deciding a specific binding time, and its importance for the software development life cycle are discussed in Chap. 4, which also provides a renewed taxonomy of different binding times. Chapters 5 and 6 describe ways to implement and configure software variability. In Chap. 5 we outline from a high-level perspective various mechanisms for implementing software variability, and how variability implementation mechanisms affect the architecture, components, and code levels. We did not go into the specific implementation details as many of the mechanisms described in the chapter depend on the programming language selected. Once product line variability is embedded into the product line asset (code) using various mechanisms, we should be able to configure products from the asset. Chap. 6 focuses on processes of product derivation activities for pre- and post-deployment times, with special mention of the configuration tasks of software products at run-time and reconfiguration activities. Because of the complexity of C&V models and complex interrelationships among them, visualizing the relationships between modeling elements is useful and enhances understandability and maintainability. Techniques for visualization are discussed in Chap. 7. Finally, we conclude this part of the book in Chap. 8 with a description on how different life-cycle products are related to each other in terms of variability when feature models are considered a first-class artifact for any product line engineering process.

Part II of the book describes an overview of research and commercial tools, from Chaps. 9–12. Three research tools, COVAMOF, PLUM, and FaMa, address different aspects of variability management as they provide automatic support for managing, configuring, and testing feature models with other related software artifacts. The commercial tool pure::variants is a variability management suite that evolves from the original FODA feature model to support the problem and solution spaces for describing variant configurations.

Part III shows the most practical viewpoint of the book as we collect three different industry cases on how variability is managed in real industry projects. Chapter 13 provides the view of Philips Healthcare Systems where product line engineering is used extensively to manage the complexity and the diversity of the Philips systems that rely on C&V and configuration properties of the Philips Software Product Line (SPL). In Chap. 14 Toshiba researchers use a product line strategy to describe the variability in power plants software for managing an automatic control system where complex rules model the relationships between events, conditions, and actions. Chapter 15 from BigLever Software focuses on Second Generation Product Line Engineering (2GPLE) activities and tools and applied to an industrial case at General Motors. In this chapter we can discover the differences between traditional SPLE activities (first generation) and those suggested for a second generation (2GPLE), where variability is described and managed consitently and traceable across the full engineering life cycle and configuration management is simplified.

Part IV concludes the book and encompasses six different chapters focused on emerging topics about software variability that, currently, are under research. The diversity of topics include dynamic software product lines, variability in autonomic computing and web services, the relationship and role of variability in service-oriented product lines, the impact and use of design decisions in conjunction with variability models, and finally, how variability is realized using aspect orientation. We believe that there are more interesting research topics that can be discussed with more detail, but this part of the book provides and suggests the readers current and future trends where variability can be applied to manage the diversity of products in different types of systems or how other software engineering techniques can be also applied with variability models and vice versa.

As authors and editors, we feel that the book presents an important contribution both to the industrial practice of software product lines and software engineering more broadly and to the software engineering research community. We have strived to capture the current state of the art and state of practice in the chapters and to indicate important, open research challenges as well as pitfalls for industrial practitioners to be aware of. We hope that the book can serve as a platform for the community of researchers and practitioners in software variability management, allowing the community to develop the next set of solutions, techniques, and methods to address this complicated and yet fascinating field in software engineering.

Madrid, Spain — Rafael Capilla
Gothenburg, Sweden — Jan Bosch
Pohang, Republic of Korea — Kyo-Chul Kang
January 2013

Contents

List of Reviewers

We would like to thank the valuable comments and suggestions made by the external reviewers, many of them coauthors of this book.

Paris Avgeriou, University of Groningen, Groningen, The Netherlands
Felix Bachman, Software Engineering Institute, Pittsburgh, PA, USA
Jan Bosch, Chalmers University of Technology, Gothenburg, Sweden
Rafael Capilla, Universidad Rey Juan Carlos, Móstoles, Madrid, Spain
Carlos Cetina, Universidad de San Jorge, Villanueva de Gállego, Zaragoza, Spain
Sybren Deelstra, Océ Technologies B.V., Venlo, The Netherlands
Juan Carlos Dueñas, Universidad Politécnica de Madrid, Madrid, Spain
Matthias Galster, University of Canterbury, Christchurch, New Zealand
Patrick Healy, University of Limerick, Limerick, Ireland
Mike Hinchey, Lero, Limerick, Ireland
Kyo Chul Kang, Pohang University of Science and Technology, Pohang, Republic of Korea
Gerald Kontoya, Lancaster University, Lancaster, UK
Frank van der Linden, Philips, Eindhoven, The Netherlands
Jason Mansell, Tecnalia, Derio, Bizkaia, Spain
Vicente Pelechano, Universidad Politécnica de Valencia, Valencia, Spain
Rick Rabiser, Johannes Kepler Universität Linz, Linz, Austria
Mark-Oliver Reiser, Technische Universität Berlin, Berlin, Germany
Steffen Thiel, Furtwangen University of Applied Sciences, Furtwangen, Germany
Salvador Trujillo, Ikerlan, Arrasate-Mondragón, Gipuzkoa, Spain

List of Contributors

Paris Avgeriou Department of Computer Science, University of Groningen, Groningen, The Netherlands

David Benavides University of Seville, Seville, Spain

Danilo Beuche pure-systems GmbH, Magdeburg, Germany

Jan Bosch Chalmers University of Technology, Gothenburg, Sweden

Goetz Botterweck Lero-The Irish Software Engineering Research Centre, University of Limerick, Limerick, Ireland

Rafael Capilla Rey Juan Carlos University, Móstoles, Madrid, Spain

Ciaran Cawley Dublin Institute of Technology, Dublin, Ireland

Carlos Cetina Universidad de San Jorge, Villanueva de Gállego, Zaragoza, Spain

Paul Clements BigLever Software, Austin, TX, USA

Sybren Deelstra Océ Technologies B.V., Venlo, The Netherlands

Rick Flores General Motors, Detroit, MI, USA

Makoto Fujii Fuchu Complex, TOSHIBA Corporation, Fuchu-shi, Tokyo, Japan

Matthias Galster Department of Computer Science and Software Engineering, University of Canterbury, Christchurch, New Zealand

Sven Hallsteinsen SINTEF ICT, Trondheim, Norway

Mike Hinchey Lero-The Irish Software Engineering Research Centre, University of Limerick, Limerick, Ireland

Kyo Chul Kang Pohang University of Science and Technology (POSTECH), Pohang, Republic of Korea

Gerald Kotonya School of Computing and Communications, Lancaster University, Lancaster, UK

Charles Krueger BigLever Software, Austin, TX, USA

Hyesun Lee Pohang University of Science and Technology (POSTECH), Pohang, Republic of Korea

Jaejoon Lee School of Computing and Communications, Lancaster University, Lancaster, UK

Kwanwoo Lee Department of Information Systems Engineering, Hansung University, Seongbuk-gu, Seoul, Republic of Korea

Frank van der Linden Philips Healthcare, Eindhoven, The Netherlands

Cristina López Fundación TECNALIA Research & Innovation, Derio, Bizkaia, Spain

Jason Mansell Fundación TECNALIA Research & Innovation, Derio, Bizkaia, Spain

Yoshihiro Matsumoto The ASTEM Research Institute of Kyoto, Shimogyo-ku, Kyoto City, Kyoto, Japan

Masami Okamoto Fuchu Complex, TOSHIBA Corporation, Fuchu-shi, Tokyo, Japan

Sooyong Park Sogang University, Mapo-gu, Seoul, South Republic of Korea

Vicente Pelechano Universidad Politécnica de Valencia, Valencia, Spain

Antonio Ruiz-Cortés University of Seville, Seville, Spain

Klaus Schmid University of Hildesheim, Hildesheim, Germany

Sergio Segura University of Seville, Seville, Spain

Marco Sinnema Q-Free ASA, Beilen, The Netherlands

Steffen Thiel Furtwangen University of Applied Sciences, Furtwangen, Germany

Pablo Trinidad University of Seville, Seville, Spain

Part I
Variability Management

Chapter 1
Software Product Line Engineering

Jan Bosch

What you will learn in this chapter
- *Software product lines*

1 Introduction

The competitive landscape of software-intensive companies is changing and intensifying rapidly. The size and complexity of systems is increasing while the speed of innovation is accelerating at the same time. In addition, the balance of power is increasingly shifting to the customer and the ability of the customer to demand products that specifically address that segment or, as an extreme case, customer-specific adaptations to the products. This has lead to a situation where many companies are stuck in a fire-fighting mode where the cost of developing new products increases constantly due to increased size and complexity while, on the other hand, the number of products and customer-specific adaptations required increases constantly. This puts an unwieldy strain on the R&D organization, and over time, this causes the competitive position of the company to deteriorate, as it is unable to innovate in its product portfolio and processes due to the singular focus on short-term deliverables for customers.

The forces described above are of course not unique or new in their generic form. However, over the last decade, many industries have reached a threshold where just improving existing R&D practices no longer allows the company to maintain its competitive position. As it becomes increasingly obvious that just "working harder" is not going to deliver the desired results, organizations reach a point where new

J. Bosch (✉)
Chalmers University of Technology, Gothenburg, Sweden
e-mail: jan@janbosch.com

R. Capilla et al. (eds.), *Systems and Software Variability Management*,
DOI 10.1007/978-3-642-36583-6_1,

ways need to be found to instead work smarter, i.e., find new ways of working where the diverse demands on the company can be met more easily.

Over the last decade, many companies have found the notion of software product lines to provide a set of work practices that allows them to drastically increase the amount of R&D resources that focused on highly differentiating functionality and, consequently, decreasing the investment in commoditized functionality. Software product lines allow a family of products to share a common core, the platform, while allowing for product-specific functionality being built on top of the platform.

Successful introduction of a software product line provides a significant opportunity for a company to improve its competitive position, but of course it is no panacea. In several companies that we studied, the initially successful adoption of a software product line eroded over time, and the benefits started to decrease and in some cases turned into liabilities. When studying this development at several companies, we identified that the lack of systematic software variability management was the root cause for most of the identified problems. This insight lead to the book that you are now reading: systematically managing the required and provided variability in a software product line is critical to maintain the competitive advantage that software product lines provide. In addition, it prepares the company and its product line for the next stage in its evolution, such as the introduction of a software ecosystem around a successful product line [1].

The remainder of this chapter is organized as follows. The next section is concerned with introducing the basic concepts that constitute a software product line. Then, we present an overview of the key challenges that we see in companies that initially successfully deployed software product lines develop over time. The subsequent section then analyses these challenges and introduces the notion of software variability in the context of software product lines. Finally, Sect. 5 provides a summary of the discussion in this chapter and presents an outlook of the remainder of the book.

2 Software Product Lines

Software product lines are concerned with sharing common functionality within a family of products. Earlier approaches to software reuse had a tendency to focus only on the technology aspects of reusing software assets and occasionally included some process aspects. The key success factor of software product lines is that it addresses business, architecture, process and organizational aspects of effectively sharing software assets within a portfolio of products. This is sometimes referred to as the BAPO model [4].

In the sections below, we introduce the software product line concept using the BAPO model.

2.1 Business and Strategy

The decision to introduce a new software product line is, obviously, a strategic business decision. The company, either proactively and offensively or reactively as a defensive strategy, adopts a product line approach. Although every company, and consequently its strategy, is unique, one can identify a number of common reasons for deciding on the introduction of software product line. The exact implementation is based on the key drivers in the industry segment that the company is active in. Below we present four typical strategic goals that an organization may have to introduce a product line:

- *Product portfolio diversity*: The first and perhaps most common reason for introducing a software product line is to be able to offer a much broader and richer product portfolio against the same R&D investment. Especially in the case where the market currently served by a small number of independently developed products can be much more effectively monetized by offering more products serving smaller customer segments more accurately, the introduction of the product line allows for effective sharing of functionality needed by all products while allowing for product-specific functionality being built on top to serve the specific market segment.
- *Common user experience for products in the portfolio*: A quite common, but less publicized, alternative reason for introducing a software product line is to share one major subsystem between the products in the portfolio. A typical example is to share the UI framework and the generic menu structure and use cases between the products. This allows for a common look and feel across the product portfolio allowing for higher productivity of users that use different devices from the same manufacturer.
- *More customizable customer products*: In some cases, the company is eager to present the market with only one or a small number of products. However, in order to serve the needs of all customers, the product needs to contain significant amounts of variability. In this case, the company can internally use a software product line approach and use automated configuration as a mechanism for providing each customer with a possibly unique configuration and version of the product.
- *Higher-quality products due to reliable shared core*: Especially in markets where product quality is a major issue, creating the products from shared components and based on a common architecture is a cost-effective mechanism for increasing quality. As the architecture and shared components are used in several products and configurations, their quality increases over time.

Although the above items provide the strategic arguments for introducing a software product line, there are some advantages that are often ignored in the initial decision process, but then recognized as important once the product line is in place.

- *Improvements become available for all products at once*: For new requirements that need to be implemented in all products, one of the major advantages of

software product lines is that once the new requirement is implemented in the shared assets, it automatically becomes available for all products in the product line. Especially in industries where regulation, protocols, or other cross cutting requirements are common, this can provide a significant advantage.

- *Improved productivity due to specialization of teams*: Assuming R&D in the organization is organized using component teams, the adoption of a software product line can result in improved productivity as teams associated with a shared component build up domain knowledge to an earlier unachievable level.
- *Low opportunity cost of new product experiments*: Software engineering often focuses on efficiency, but of course the key success factor for any organization is its ability to innovate. Successful innovation systems allow for promising ideas to surface and then test these ideas with customers against the lowest possible R&D investment. In the case of a software product line, the cost associated with creating a new (prototype) product for testing with customers is much lower as most of the required functionality is already available in the shared assets. This will cause the company to run more new product experiments, resulting, over time, in a more innovative and faster growing company.

Although engineers tend to focus on the efficiency aspects of software product lines, the above clearly illustrates the relevance of the technology from a business and strategy perspective. Both from the perspective of creating a competitive differentiating position for the company as well as becoming more innovative, software product lines, when deployed well, can provide a significant advantage.

2.2 Architecture and Technology

The second aspect of software product lines is the architecture and technology choices underlying the software product line. There are several dimensions to be considered, but the first is the scope of the shared assets in the product line versus the amount of functionality covered in product-specific code. As illustrated in Fig. 1.1, we can identify a typical evolution path for a product line from this perspective. The model uses four stages:

- *Standardized infrastructure*: Starting from a set of independent products, the first step for an organization is to standardize on the software acquired externally. Typically, these software components are infrastructural in nature. Standardizing the infrastructure and having each product build on the same set of components can achieve significant benefits.
- *Platform*: The second stage is the formation of a platform on top of the infrastructure. An overloaded term, the platform refers here to a layer of functionality that is common to all products within the product line. In this stage, software variability often is a limited concern, as all products need the same functionality.
- *Software product line*: Once the value of sharing software between products is established, there will be a tendency to put more functionality in the shared

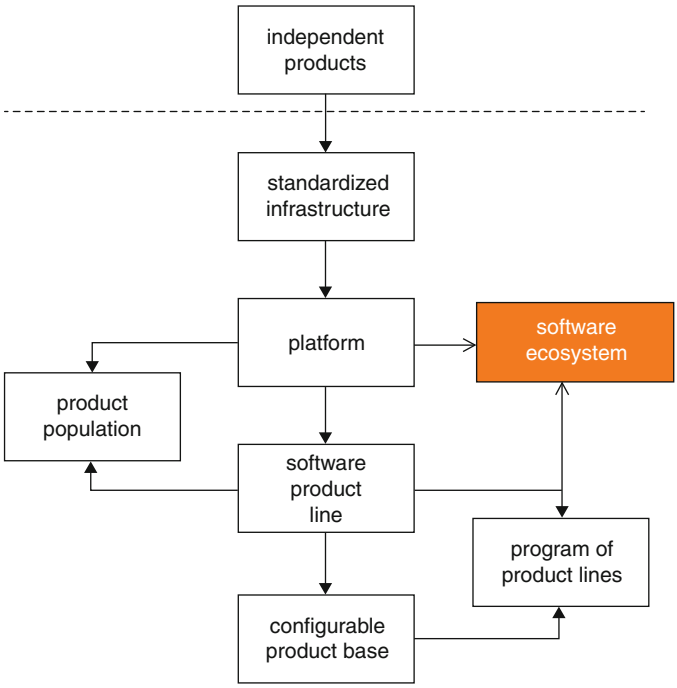


Fig. 1.1 Evolution of a software product line

components. At this stage, also functionality that is used by a subset of the products is put in the product line. Consequently, software variability management, the topic of this book, starts to become a significant concern.

- *Configurable product base*: The most advanced stage is where a complete set of products can be automatically derived from the common asset base. There is no need anymore for product-specific development teams, as the products can be derived automatically.

There are additional paths outlined in Fig. 1.1 on how a product line might evolve. These are, however, not discussed in the chapter. Instead, we refer to [2] for more details.

2.3 Process and Tools

Similar to the business and architecture approaches evolving through different levels, also an organization's approach to the development process tends to evolve over time. Although there are top-down, metric-centric approaches to software process, such as CMMI [3], a more fruitful way to describe process is by focusing on the way individual teams can release their software to the product line.

The release process drives virtually all earlier steps in the process and will, to a large extent, decide on the amount of coupling and, consequently, the coordination required between different teams involved in the software product line. We identify five levels of decoupling between teams in the software product line organization that we describe below.

2.3.1 Integration-Centric Development

Many, if not most, software development companies apply an integration-centric approach, in which the organization relies on the integration phase of the software development lifecycle. During the early stages of the lifecycle, there is allocation of requirements to the components. During the development phase, teams associated with each component implement the requirements allocated to the component. When the development of the components making up the system is finalized, the development enters the integration phase, in which the components are integrated into the overall system and system-level testing takes place. During this stage, typically, many integration problems are found that need to be resolved by the component teams.

If the component teams have not tested their components together during the development phase, this phase may also uncover large numbers of problems that require analysis, allocation to component teams, coordination between teams, and requiring continuous retesting of all functionality as fixing one problem may introduce others.

In response to the challenges discussed above, component teams often resort to sharing versions of their software even though it is under development. Although this offers a means of simplifying the integration phase, the challenge is that the untested nature of the components being shared between component teams causes significant inefficiency that could have been avoided if only more mature software assets would be shared. One approach discussed frequently in this context is continuous integration, but in our experience this often addresses the symptoms but not the root causes of the lack of decoupling.

Although most organizations employing this approach utilize techniques like continuous integration and inter-team sharing of code that is under development, the process tends to be organized around the integration phase. This often means a significant peak in terms of work hours and overtime during the weeks or sometimes months leading up to the next release of the product line and the products that are part of it.

A challenge that often occurs in this context is lockstep evolution. When the system or platform can only evolve in a lockstep fashion, this is often caused by evolution of one asset having unpredictable effects on other, dependent assets. In the worst case, with the increasing amount of functionality in the assets, the cycle time at which the whole system is able to iterate may easily lengthen to the point where the product or platform turns from a competitive advantage to a liability. The root cause of the problem is the selection of interface techniques that do not

sufficiently decouple components from each other. APIs may expose the internal design of the component or be too detailed that causes many change scenarios to require changes to the API as well.

Although the integration-oriented approach has its disadvantages, as discussed above, it is the approach of choice when two preconditions are met. First, if conditions exist that require a *very deep integration between the components* of a system or a family of systems, e.g., due to severe resource constraints or challenging quality requirements, the integration-oriented approach is, de facto, the only viable option. Second, if the *release cycle* of a system or family of systems is *long*, e.g., 12–18 months, the amount of calendar time associated with the integration phase is acceptable.

2.3.2 Release Groupings

In this approach, the development organization aims to break the system into groups of components that are pre-integrated, i.e., a release group, whereas the composition of the different release groups is performed using high decoupling techniques such as SOA-style (service-oriented-architecture) interfaces. At the level of a release group, the integration-centric approach is applied, whereas at the inter-release group level coordination of development is achieved using periodic releases of all release groups in the stack.

The process is now also different between the release groupings, but the same as the previously discussed approach within the release grouping. The decoupling allows the release groupings to be composed, with relatively few issues. This is often achieved by more upfront work to design and publish the interface of each release group before the start of the development cycle.

In some of the cases that we studied, the organization failed to realize that processes needed to vary between and inside release groupings. This lead to several consequences, including features that cross release groupings tend to be underspecified before the start of development and need to be "worked out" during the development by close interaction between the involved teams. This defeats the purpose of release groupings and causes significant inefficiency in development.

The release grouping approach is particularly useful in situations where teams responsible for different subsets of components are *geographically dispersed*. Aligning release groupings with location is, in that case, an effective approach to decreasing the inefficiencies associated with coordination over sites and time zones. A second context is where the architecture covers a number of application domains that require *high integration within the application domain, but much less integration between application domains*. For instance, a system consisting of video processing and video storage functionality may require high integration between the video processing components, but a relatively simple interface between the storage on processing parts of the system. In this case, making each domain a release grouping is a good design decision.

2.3.3 Release Trains

In release trains, the decoupling is extended from groups of components to every component in the system. All interfaces between components are decoupled to the extent possible, and each component team can by and large work independently during each iteration. The key coordination mechanism between the teams is an engineering heartbeat that is common for the whole R&D organization. With each iteration, e.g., every month, a release train leaves with the latest releases of all production-quality components on the train. If a team is not able to finalize development and validation of its component, the release management team does not accept the component. Once the release team has collected all components that passed the component quality gates, the next step is to build all the integrations for the software product line. For those components that did not pass the component quality gates, the last validated version is used. The integration validation phase has two stages. During the first stage, each new release of each component is validated in a configuration consisting of the last verified versions of all other components. Component that do not pass this stage are excluded from the train. During the second stage, the new versions of all components that passed the first stage are integrated with the last verified versions of all other components, and integration testing is performed for each of the configurations that are part of the product family. In the case where integration problems are found during this stage, the components at fault are removed from the release train. The release train approach concludes each iteration with a validated configuration of components, even though in the process a subset of the planned features may have been withdrawn due to integration issues between components. The release train approach provides an excellent mechanism for organizational decoupling by providing a heartbeat to the engineering system that allows teams to synchronize on a frequent basis while working independently during the iterations.

The key process challenges are the predevelopment cycle work around interface specification and content commitment and the process around the acceptance or rejection of components at the end of the cycle. In addition, especially when the organization uses agile development approaches, sequencing the development of new features such that dependent, higher level features are developed in the cycle following the release of lower level features allows for significantly fewer ripple effects when components are rejected.

The release train approach allows team to work independently from each other during the development of the next release, but it still requires all teams to release at the same point in time. The process of testing the new version of components consists of two stages. First, each new version of a component is tested in the context of the released versions of all other components. This verifies backward compatibility. In the second stage, the new versions of all components are brought together to verify the newly released functionality across component boundaries.

The release train approach is particularly suited for organizations that are required to deliver *a continuous stream of new functionality in their products or*

platform, either because new products are released with a high frequency or because existing products are released or upgraded frequently with new functionality. The organization has a business benefit from frequent releases of new functionality. Companies that provide web services provide a typical example of the latter category. Customers expect a continuous introduction of new functionality in their web services and expect a rapid turnaround on requests for new functionality. The release train approach does require a relatively *mature development organization and infrastructure*. For instance, the amount and complexity of validation and testing that is required demands a high degree of test automation. In addition, interface management and requirement allocation processes need to be mature in order to achieve sufficient decoupling, backward compatibility, and independent deployment of components.

2.3.4 Independent Deployment

The independent deployment approach assumes an organizational maturity that does not require an engineering heartbeat (a heartbeat in the engineering system allows teams to synchronize on a frequent basis while working independently during iterations) including all the processes surrounding a release train. In this approach, each team is free to release new versions of their component at their own iteration speed. The only requirement is that the component provides backward compatibility for all components dependent on it. In addition, the teams develop and commit to roadmaps and plans. The lack of an organization-wide heartbeat does not free any team from the obligation to keep their promises. However, the validation of a component before being released is more complicated in this model as any component team, at any point in time, may decide to release its latest version.

The perception in the organization easily becomes that there no longer is an inter-team process for development as any team can develop and release at their leisure. In practice, this is caused because the process is no longer a straightjacket but provides more guardrails within which development takes place. The cultural aspects of the software development organization, especially commitment culture and never allowing deviations from backward compatibility requirements, need to be deeply engrained and enforced appropriately.

As the process does not enforce joint releasing of components, any component team can release at their own frequency and time. This requires an even higher degree of automation and coverage of the testing framework in order to guarantee the continued functioning of the overall system.

The independent deployment approach is particularly useful in cases where *different layers of the stack have very different "natural" iteration frequencies*. Typically, lower layers of the stack that are abstracting external infrastructure iterate at a significantly lower frequency. This is both because the release frequency of the external components typically is low, e.g., one or two releases per year, and because the functionality captured in those lower layers often is quite stable and

evolves more slowly. The higher layers of the software stack, including the product-specific software, tend to iterate much more.

The key factor in the successful application of the independent deployment approach is the *maturity of the development organization*. The processes surrounding road mapping, planning, interface management and, especially, verification and validation need to be mature and well supported by tools in order for the model to be effective.

2.3.5 Open Ecosystem

The final approach discussed is an approach in which inter-organizational collaboration is strived after. Successful software product lines are likely to become platforms for external parties that aim to build their own solutions on top of the platform provided by the organization. Although this can, and should, be considered as a sign of success, the software product line typically has not been designed as a development platform, and providing access to external parties without jeopardizing the qualities of the products in the product line is typically less than trivial. Even if the product line architecture has been well prepared for acting as a platform, the problem is that external developers often demand deeper access to the platform than the product line organization feels comfortable to provide.

The typical approach to address this is often twofold. First, external parties that require deep access to the platform are certified before access is given. Second, any software developed by the certified external parties needs to get validated in the context of the current version of the platform before being deployed and made accessible to customers.

Although the aforementioned approach works fine in the traditional model, modern software platforms increasingly rely on their community of users to provide solutions for market niches that the platform organization itself is unable to provide. The traditional certification approach is infeasible in this context, especially as the typical case will contain no financial incentive for the community contributor and the hurdles for offering contributions should be as low as possible. Consequently, a mechanism needs to be put in place that allows software to exist within the platform but to be sandboxed to an extent that minimizes or removes the risk of the community-offered software affecting the core problem to any significant extent.

The open ecosystem development model allows unconstrained releasing of components in the ecosystem not only by the organization owning the platform but also by certified third parties as well prosumers and other community members providing new functionality. Although few examples of this approach exist, it is clear that a successful application of this approach requires run-time, automated solutions for maintaining system integrity for all different configurations in which the ecosystem is used.

As the ecosystem participants are independent organizations, no common process approach can be enforced, except for gateways, such as security validation of external applications. However, each limitation put in place causes hurdles for

external developers that inhibit success of the ecosystem, so one has to be very careful to rely on such mechanisms.

The open ecosystem model is a natural evolution from the release train and independent deployment models when the organization decides to *open up the software product line to external parties*, either in response to demands by these parties or as a strategic direction taken by the company in order to *drive adoption by its customers*.

The key in this model, however, is the ability to provide proper architectural decoupling between the various parts of the ecosystem without losing integrity from a customer perspective. In certain architectures and domains, the demand for deep integration is such that, at this point in the evolution of the domain, achieving sufficient decoupling is impossible, either because quality attributes cannot be met or because the user experience becomes unacceptable in response to dynamic, run-time composition of functionality.

Two areas where this approach is less desirable are concerned with the platform *maturity* and the *business model*. Although the pull to open up any software product line that enjoys its initial success in the market place, the product line architecture typically goes through significant refactoring that can't be hidden from the products in the product line or the external parties developing on top of the platform defined by the architecture. Consequently, any dependents on the product line architecture are going to experience significant binary breaks and changes to the platform interface. Finally, the transition from a product to a platform company easily causes conflicts in the business models associated with both approaches. If the company is not sufficiently financially established or the platform approach not *deeply ingrained in the business strategy*, adopting the open ecosystem approach fails due to internal organizational conflicts and mismatches.

2.4 Organization

The final dimension that we discuss in this section is how to organize around the work of building the software product line and the products that it includes. Although there are many different ways to organize, we present four standard models of organizing development that cover and address most of the cases that we have encountered in the industry. For each model, we present the applicability, the advantages, and the disadvantages.

2.4.1 Development Department

The development department model imposes no permanent organizational structure on the architects and engineers that are involved in the software product line. All staff members can, in principle, be assigned to work with any type of asset within the family. Typically, work is organized in projects that dynamically organize staff

members in temporary networks. These projects can be categorized into domain engineering projects and product (or system) engineering projects. In the former, the goal of the project is the development of a new reusable asset or a new version of it, e.g., a software component. The goal is explicitly not a system or product that can be delivered to internal or external customers of the development department. The product engineering projects are concerned with developing a system, either a new or a new version, that can be delivered to a customer. Occasionally, extensions to the reusable assets are required to fulfill the system requirements that are more generally applicable than just the system under development. In that case, the result of the product engineering project may be a new version of one or more of the reusable assets, in addition to the deliverable system.

The development department model has, as most things in life, a number of advantages and disadvantages. The primary advantage is simplicity and ease of communication. Since all staff members are working within the same organizational context, come in contact with all parts of the system family and have contact with the customers, the product line can be developed and evolved in a very efficient manner with little organizational and administrative overhead. A second advantage is that, assuming that a positive attitude towards reuse-based software development exists within the department, it is possible to adopt a software product line approach without changing the existing organization, which may simplify the adoption process.

The primary disadvantage of this approach is that it is not scalable. When the organization expands and reaches, e.g., around 30 staff members, it is necessary to reorganize and to create specialized units. A second disadvantage is that typically within organizations, staff members are, depending on the local culture, more interested in either domain engineering or system engineering, i.e., it has higher status in the informal organization to work with a particular type of engineering. The danger is that the lower status type of engineering is not performed appropriately. This may lead to highly general and flexible reusable components, but systems that do not fulfill the required quality levels, or vice versa.

Summarizing, this approach has the following characteristics:

- *Applicability*: Smaller R&D organizations (less than 30 members) have a strong project focus, rather than a product focus.
- *Advantages*: The approach excels in simplicity and easy of communication.
- *Disadvantages*: The main limitation of this approach is the lack of scalability. In addition, the organization tends to prioritize product engineering or domain engineering.

2.4.2 Business Units

The second organizational model that we discuss is organized around business units. Each business unit is responsible for the development and evolution of one or a few products in the software product line. The reusable assets in the product line

are shared by the business units. The evolution of shared assets is generally performed in a distributed manner, i.e., each business unit can extend the functionality in the shared assets, test it and make the newer version available to the other business units. The initial development of shared assets is generally performed through domain engineering projects. The project team consists of members from all or most business units. Generally, the business units most interested in the creation of, e.g., a new software component, put the largest amount of effort in the domain engineering project, but all business units share, in principle, the responsibility for all common assets.

Depending on the number and size of the business units and the ratio of shared versus system-specific functionality in each system, we have identified three levels of maturity, especially with respect to the evolution of the shared assets.

Unconstrained Model

In the unconstrained model, any business unit can extend the functionality of any shared component and make it available as a new version in the shared asset base. The business unit that performed the extension is also responsible for verifying that, where relevant, all existing functionality is untouched and that the new functionality performs according to specification.

A typical problem that companies using this model suffer from is that, typically, software components are extended with too system-specific functionality. Either the functionality has not been generalized sufficiently or the functionality should have been implemented as system-specific code, but for internal reasons, e.g., implementation efficiency or system performance, the business unit decided to implement the functionality as part of the shared component.

These problems normally lead to the erosion or degradation of the component, i.e., it becomes, over time, harder and less cost-effective to use the shared component, rather than developing a system-specific version of the functionality. As we discussed in [2], some companies have performed component reengineering projects in which a team consisting of members from the business units using the component reengineers the component and improves its quality attributes to acceptable levels. Failure to reengineer when necessary may lead to the situation where the product line exists on paper, but where the business units develop and maintain system-specific versions of all or most components in the product line, which invalidates all advantages of a software product line approach, while maintaining some of the disadvantages.

Asset Responsibles

Especially when the problems discussed above manifest themselves in increasing frequency and severity, the first step to address these problems is to introduce asset responsibles. An asset responsible has the obligation to verify that the evolution of

the asset is performed according to the best interest of the organization as a whole, rather than optimal from the perspective of a single business unit. The asset responsible is explicitly not responsible for the implementation of new requirements. This task is still performed by the business unit that requires the additional functionality. However, all evolution should occur with the asset responsible's consent, and before the new version of the asset is made generally accessible, the asset responsible will verify through regression testing and other means that the other business units are at least not negatively affected by the evolution. Preferably, new requirements are implemented in such a fashion that even other business units can benefit from them. The asset responsible is often selected from the business unit that makes most extensive and advanced use of the component.

Although the asset responsible model, in theory at least, should avoid the problems associated with the unconstrained model, in practice it often remains hard for the asset responsible to control the evolution. One reason is that time-to-market requirements for business units often are prioritized by higher management, which may force the asset responsible to accept extensions and changes that do not fulfill the goals, e.g., too system-specific. A second reason is that, since the asset responsible does not perform the evolution him or herself, it is not always trivial to verify that the new requirements were implemented as agreed upon with the business unit. The result of this is that components still erode over time, although generally at a lower pace than with the unconstrained model.

Mixed Responsibility

Often, with increasing size of the system family, number of staff, and business units, some point is reached where the organization still is unwilling to adopt the next model, i.e., domain engineering units, but wants to assign the responsibility for performing the evolution assets to a particular unit. In that case, the mixed responsibility model may be applied. In this model, each business unit is assigned the responsibility for one or more assets, in addition to the product(s) the unit is responsible for. The responsibility for a particular asset is generally assigned to the business unit that makes the most extensive and advanced use of the component. Consequently, most requests for changes and extensions will originate from within the business unit, which simplifies the management of asset evolution. The other business units have, in this model, no longer the authority to implement changes in the shared component. Instead, they need to issue requests to the business unit responsible for the component whenever an extension or change is required.

The main advantage of this approach is the increased control over the evolution process. However, two potential disadvantages exist. First, since the responsibility for implementing changes in the shared asset is not always located at the business unit that needs those changes, there are bound to be delays in the development of systems that could have been avoided in the approaches described earlier. Second, each business unit has to divide its efforts between developing the next version of

its product(s) and the evolution of the component(s) it is responsible for. Especially when other business units have change requests, these may conflict with the ongoing activities within the business unit and the unit may prioritize its own goals over the goals of other business units. In addition, the business unit may extend the components it is responsible for in ways that are optimized for its own purposes, rather than for the organization as a whole. These developments may lead to conflicts between the business units and, in the worst case, the abolishment of the product line approach.

Conflicts

The way the software product line came into existence is, in our experience, an important factor in the success or failure of a family. If the business units already exist and develop their systems independently and, at some point, the software product line approach is adopted because of management decisions, conflicts between the business units are rather likely because giving up freedom that one had up to that point in time is generally hard. If the business units exist, but the product line gradually evolves because of bottom-up, informal cooperation between staff in different business units, this is an excellent ground to build a product line upon. However, the danger exists that when cooperation is changed from optional to obligatory, tensions and conflicts appear anyhow. Finally, in some companies, business units appear through an organic growth of the company. When expanding the set of systems developed and maintained by the company, at some point, a reorganization into business units is necessary. However, since the staff in those units earlier worked together and used the same assets, both the product line and cooperation over business units develop naturally, and this culture often remains present long after the reorganization, especially when it is nurtured by management. Finally, conflicts and tensions between business units must be resolved by management early and proactively since they imply considerable risk for the success of the product line.

The advantage of this model is that it allows for effective sharing of assets, i.e., software architectures and components, between a number of organizational units. The sharing is effective in terms of access to the assets, but in particular the evolution of assets (especially true for the unconstrained and the asset responsible approaches). In addition, the approach scales considerably better than the development department model, e.g., up to 100 engineers in the general case.

The main disadvantage is that, due to the natural focus of the business units on systems (or products), there is no entity or explicit incentive to focus on the shared assets. This is the underlying cause for the erosion of the architecture and components in the system family. The timely and reliable evolution of the shared assets relies on the organizational culture and the commitment and responsibility felt by the individuals working with the assets.

Summarizing, this approach has the following characteristics:

- *Applicability*: The approach works well for mid-sized R&D departments, e.g., up to 100 engineers.
- *Advantages*: The approach allows for effective sharing of software assets, especially the cost of evolving assets. Also, it offers much better scalability than the previous approach.
- *Disadvantages*: The main limitation of this approach is the lack of attention to domain assets and the consequent higher rate of erosion.

2.4.3 Domain Engineering Unit

The third organizational model for software product lines is concerned with separating the development and evolution of shared assets from the development of concrete systems. The former is performed by a so-called domain engineering unit, whereas the latter is performed by product engineering units.

The domain engineering unit model is typically applicable for larger organizations, but requires considerable amounts of communication between the product engineering units that are in frequent contact with the users of their products and the domain engineering unit that has no direct contact with customers, but needs a good understanding of the requirements that the product engineering units have. Thus, one can identify flows in two directions, i.e., the requirements flow from the product engineering units towards the domain engineering unit and the new versions of assets, i.e., the software architecture and the components of system family, are distributed by the domain engineering unit to the product engineering units.

The domain engineering unit model exists in two alternatives, i.e., an approach where only a single domain engineering unit exists and, secondly, an approach where multiple domain engineering units exist. In the first case, the responsibility for the development and evolution of all shared assets, i.e., the software architecture and the components, is assigned to a single organizational unit. This unit is the sole contact point for the product engineering units that construct their products based on the shared assets.

The second alternative employs multiple domain engineering units, i.e., one unit responsible for the design and evolution of the software architecture for the product line and, for each architectural component (or set of related components), a component engineering unit that manages the design and evolution of the components. Finally, the product engineering units are, also in this alternative, concerned with the development of products based on the assets. The main difference between the first and second alternatives is that in the latter, the level of specialization is even higher and that product engineering units need to interact with multiple domain engineering units.

Despite the skepticism in especially smaller organizations, the domain engineering unit model has a number of important advantages. First, as mentioned, it

removes the need for *n*-to-*n* communication between the business units and reduces it to 1-to-*n* communication. Second, whereas business units may extend components with too product-specific extensions, the domain engineering unit is responsible for evolving the components such that the requirements of all systems in the product line are satisfied. In addition, conflicts can be resolved in a more objective and compromise-oriented fashion. Finally, the domain engineering unit approach scales up to much larger numbers of software engineering staff than the aforementioned approaches.

Obviously, the model has some associated disadvantages as well. The foremost is the difficulty of managing the requirements flow towards the domain engineering unit, the balancing of conflicting requirements from different product engineering units and the subsequent implementation of the selected requirements in the next version of the assets. This causes delays in the implementation of new features in the shared assets, which, in turn, delays the time-to-market of products. This may be a major disadvantage of the domain engineering unit model since time-to-market is the primary goal of many software development organizations. To address this, the organization may allow product engineering units to, at least temporarily, create their own versions of shared assets by extending the existing version with product-specific features. This allows the product engineering unit to improve its time-to-market while it does not expose the other product engineering units to immature and instable components. The intention is generally to incorporate the product-specific extensions, in a generalized form, into the next shared version of the component.

Summarizing, this approach has the following characteristics:

- *Applicability*: The domain engineering unit approach allows for significant scalability up to hundreds of software engineers.
- *Advantages*: The advantages of the model are threefold. First, it reduces the *n*-to-*n* communication in the previous model to 1-to-*n*. Second, it guarantees proper attention both to domain and product engineering. Finally, it offers excellent scalability.
- *Disadvantages*: Managing evolution and exchanging information between domain and product engineering is inherently more complex. This can cause slower time-to-market of new features.

2.4.4 Hierarchical Domain Engineering Units

There is an upper boundary on the size of an effective domain engineering unit model. However, generally even before the maximum staff member size is reached, often already for technical reasons, an additional level has been introduced in the software product line. This additional layer contains one or more specialized product lines that, depending on their size and complexity, can either be managed using the business unit model or may actually require a domain engineering unit.

In the case that a specialized product line requires a domain engineering unit, we have, in fact, instantiated the hierarchical domain engineering unit model that is the

topic of this section. This model is only suitable for a large or very large organization that has an extensive family of products. If, during the design or evolution of the product line, it becomes necessary to organize the product line in a hierarchical manner and a considerable number of staff members is involved in the product line, then it may be necessary to create specialized domain engineering units that develop and evolve the reusable assets for a subset of the products in the family.

The reusable product line assets at the top level are frequently referred to as a platform and not necessarily identified as part of the product line. We believe, however, that it is relevant to explicitly identify and benefit from the hierarchical nature of these assets. Traditionally, platforms are considered as means to provide shared functionality, but without imposing any architectural constraints. In practice, however, a platform does impose constraints, and when considering the platform as the top-level product line asset set, this is made more explicit and the designers of specialized product lines and family members will derive from the software architecture rather than design it.

The advantages of this model include its ability to encompass large, complex product lines and organize large numbers of engineers. None of the organizational models discussed earlier scales up to the hundreds of software engineers that can be organized using this model.

The disadvantages include the considerable overhead that the approach implies and the difficulty of achieving agile reactions to changed market requirements. Typically, a delicate balance needs to be found between allowing product engineering units to act independent, including the temporary creation of product-specific versions of product line components, versus capitalizing on the commonalities between products and requiring product engineering units to use shared versions of components.

Summarizing, this approach has the following characteristics:

- *Applicability*: The hierarchical domain unit model scales up to many hundreds, potentially thousands of engineers.
- *Advantages*: The approach allows for the management of very large product families with very complex behavior and huge development departments.
- *Disadvantages*: The inherent consequence of the approach is that there is significant organizational overhead and associated cost.

3 Key Longitudinal Challenges

As discussed in the introduction and business section, successful product lines have an enormous impact on the revenue, profitability, and brand of the organization. The success of the SPL approach caused two interesting patterns in the companies that we have studied in our research. First, over time the scope of the SPL was extended significantly from an initial small set of products to cover a much broader set of products that cover a much broader set of functional and quality requirements.

The reason for this was that the success in the market place allows and almost demands from the company to significantly increase the scope of its product portfolio as well as serve customer segments with unique requirements that it could not have served without a SPL.

Second, as the SPL approach turned out so successful, earlier unrelated products or product families, originating from other parts of the business or acquired through mergers, were added to the original SPL, despite the lack of architectural alignment. The reason for bringing earlier unrelated products under the umbrella of the SPL was typically related to the overlap in domain functionality.

There were several consequences of these patterns, but one major factor was that the amount of staff working in the software product line grew significantly, up to an order of magnitude. Despite this increase, the fundamental approach to software development in each of the case study companies was not adjusted to the new scale of operations. Instead, there was strong implicit belief that the approach itself was a core element of the success of the initiative, even when it became blindingly obvious that the approach was very inefficient.

Although several problems can be identified, in this chapter we focus on three key problems that occur in successful software product lines over extended periods of time:

1. *Coordination overhead*: With increasing scope of the software product line and the significant increase in the number of people working on it, the cost of coordinating the efforts of teams, individuals, product derivation efforts, roadmapping, integration of the platform, etc. increases exponentially to the point that most staff spends most of its time coordinating through meetings, email exchange, or other mechanisms and has rapidly decreasing amounts of time dedicated to adding value to the products and shared assets in the software product line.
2. *Slow release cycles*: In each of the case study companies, the shared part of the software product line, i.e., the platform was integrated and released periodically to the product teams that derived new or evolved existing products built on top of the platform. With the increasing size and complexity of the overall SPL as well as the platform, the cost of verifying all functional and quality requirements became very high, and as a consequence, the release frequency tended to decrease and slow to a point that it was below the "speed of the market". This caused the benefit of the SPL to turn into a liability as the SPL caused delays in product releases.
3. *High system-level error density*: One of the main reasons for slow release cycles was that many errors were only found during the integration stage as the complexity of the platform and the SPL as a whole had reached a point where teams and their architects were unable to predict the implications of their design decisions and extensions on the overall system. Hence, the negative implications were found late in the development cycle and could cause time-consuming rework in the various components. This caused the integration stage to be long and painful, putting a strong pressure on the organization to perform integration

as infrequent as possible, reinforcing the aforementioned problem of slow release cycles.

After carefully studying SPLs at a variety of companies over close to two decades, our conclusion is that one can identify a single predominant root cause for the discussed problems: *lack of effective software variability management*. Over time, the number of variation points, the number of variants, and the dependencies between all these increase to a point where the original business benefits of the software product line erode to a level where the competitive advantage is no longer present.

Although this may, at first sight, seem a technical issue, the business strategy, the development process, the organization of R&D, and the culture of the R&D teams all contributed to an evolution path where the number of software variability dependencies between software assets increased, causing an increasing number of points where teams, responsible for these assets, need to coordinate causing the described problems as well as other concerns.

In the next section, we focus more on software variability and provide an initial overview as a preparation for the rest of the book.

4 Software Variability

Over the last few decades, the software systems that we have used and built have required and exhibited increasing variability, i.e., the ability of a software artifact to vary its behavior at some point in its life cycle. We can identify two underlying forces that drive this development. First, we see that variability in systems has moved from mechanics and hardware to the software. Second, because of the cost of reversing design decisions once these are taken, software engineers typically try to delay such decisions to the latest phase in a system's life cycle that is economically viable. An example of the first trend is car engine controllers. Most car manufacturers now offer engines with different characteristics for a particular car model. A new development is that frequently these engines are the same from a mechanical perspective and differ only in the software of the car engine controller. Thus, whereas previously the variation between different engine models was incorporated through the mechanics and hardware, due to economies of scale that exist for these artifacts, car developers have moved the variation to the software.

The second trend, i.e., delayed design decisions, can be illustrated through software product families and the increasing configurability of software products. Over the last decade, many organizations have identified a conflict in their software development. On the one hand, the amount of software necessary for individual products is constantly increasing. On the other hand, there is a constant pressure to increase the number of software products put out on the market in order to better service the various market segments. For many organizations, the only feasible way forward has been to exploit the commonality between different products and to

implement the differences between the products as variability in the software artifacts. The product family architecture and shared product family components must be designed in such a way that the different products can be supported, whether the products require replaced components, extensions to the architecture or particular configurations of the software components. Additionally, the software product family must also incorporate variability to support likely future changes in requirements and future generations of products. This means that when designing the commonalities of a software product line, not all decisions can be taken. Instead, design decisions are left open and determined at a later stage, e.g., when constructing a particular product or during run-time of a particular product. This is achieved through variability.

As this book is solely concerned with software variability management, it is important to provide a definition of the term: software variability is the ability of a software system or artifact to be efficiently extended, changed, customized, or configured for use in a particular context. In the remainder of the book, we provide more elaborate definitions of variability and introduce a wealth of techniques that enable software developers to improve variability of software artifacts as well as manage this variability over time.

As discussed earlier in this chapter, it is not a trivial task to effectively manage variability in a software product family. We also see that engineers are seeking variation mechanisms beyond those shipped with their development tools or that are not supported by used software systems. The adoption of mechanisms such as aspect-oriented programming and the popularity of generative and reflective techniques in programming communities such as Java and .Net are evidence of this.

Essentially, by supporting variability, design decisions are pushed to a later stage in the development. Rather than making specific design choices, the design choice is made to allow for variability at a later stage. For example, by allowing users to choose between different plug-ins for a media player, the media player designers can avoid hardwiring the playback feature to a particular playback functionality (by enabling the system to use plug-ins). Thus they can support new file formats after the media player has been shipped to the end user.

Many factors influence the choices of how design decisions can be delayed. Influencing factors include, for example, the type of software entity for which variability is required, how long the design decision can be delayed, the cost of delaying a design decision and the intended run-time environment. Another factor to consider is that variability does not need to be represented only in the architecture (i.e., the fundamental organization of a system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution) or the source code of a system. It can also be represented as procedures during the development process, making use of various tools outside of the actual system being built.

As the remainder of the book focuses on software variability management, we have only provided a high-level introduction into the notion of software variability. The next chapter introduces the subject in much more detail.

5 Summary

The competitive landscape for companies building software-intensive systems is changing and the pressure to outperform competitors is intensifying. The speed at which innovations need to be pushed to market is increasing while the size and complexity of the products is increasing as well. The resulting tension makes it clear to most organizations that just "working harder" is not going to address the issues and instead the organization needs to change its ways of working rather fundamentally. Several companies have adopted the notion of software product lines as an innovation that addresses the seemingly conflicting forces. Those that were successful in the transition process typically reaped significant benefits in terms of establishing or expanding market leadership, accelerated revenue growth, and increased profitability.

Over time, however, successful product lines often start to suffer from several problems, including expanding coordination overhead, slowing release cycles, and increasing system-level error density. Based on our analysis we identified lacking software variability management as the key underlying root cause. Over time, the increasing number of variation points, variants, and dependencies between these causes a web of complexity that causes a gradual reduction in competitiveness of the organization that over time removes the advantages provided by the software product line approach.

The key solution to addressing the aforementioned concerns is to significantly improve and professionalize the way software variability is managed, both from a problem domain and from a solution domain perspective. This has business, architectural, process, and sometimes even organizational implications. The objective of this book is to provide a perspective on software variability management that allows organizations to better understand their situation and to provide a set of concrete techniques to address the concerns that surface in their R&D organizations.

References

1. Bosch, J.: From software product lines to software ecosystems. In: Proceedings of the 13th International Software Product Line Conference (SPLC 2009), August 2009
2. Bosch, J.: Maturity and evolution in software product lines: approaches, artefacts and organization. In: Proceedings of the Second Software Product Line Conference (SPLC2), pp. 257–271, August 2002
3. Chrissis, M.B., Konrad, M., Shrum, S.: CMMI for Development: Guidelines for Process Integration and Product Improvement. Addison Wesley, Boston, MA (2011)
4. van der Linden, F., Bosch, J., Kamsties, E., Kansala, K., Obbink, H.: Software product family evaluation. In: Proceedings of the Third Conference Software Product Line Conference (SPLC 2004). LNCS, vol. 3154, pp. 110–129. Springer, September 2004

Chapter 2
Variability Modeling

Kyo C. Kang and Hyesun Lee

What you will learn in this chapter

- *The different aspects and viewpoints of variability modeling one needs to consider in software product line engineering*
- *How these different viewpoints are interrelated to each other*
- *Variability modeling techniques*

1 Introduction

The aim of this chapter is to provide a comprehensive description of the notion of variability modeling in the context of software product line engineering and to give an overview of the techniques proposed for variability modeling.

Since its first introduction in 1990, feature modeling [1] has been the most popular technique to model commonality and variability (C&V) of products of a product line. Commonalities and variabilities are modeled from the perspective of *product features*, "stakeholder visible characteristics of products" in a product line that are of stakeholders' concern. For example, the fund transfer feature of a banking system may be of interest to customers, i.e., a service feature, but how the fund transfer happens may not be of interest to customers as long as it is done securely. However, it will be an important concern for the designer of the system and, when there are alternative ways, it is the responsibility of the designer to choose the right one for the target system.

The original feature model, FODA [1], is a simple model with features that are organized using "consists of" and "generalization/specialization" relationships

K.C. Kang (✉) • H. Lee
Pohang University of Science and Technology (POSTECH), Pohang, Republic of Korea
e-mail: kck@postech.ac.kr; compial@postech.ac.kr

R. Capilla et al. (eds.), *Systems and Software Variability Management*,
DOI 10.1007/978-3-642-36583-6_2,

using the AND/OR graph. Features are typed as mandatory, alternative, or optional features to represent C&V. Attributes of a feature may also be documented.

As it has gained a wide acceptance both by practitioners and researchers, this rather simple model was extended by many researchers introducing new modeling primitives such as feature cardinality and XOR relationships. Various research activities followed such as formal analysis of feature model, feature configuration, generative programming, etc. Also, there are a wide variety of product lines FODA and its extensions have been applied to, and it has been reported that C&V models tend to become complex as the size of product line increases. This complexity of a model is highly correlated with the complexity of the problem domain that is modeled. However, it has been noticed that many different types of C&V information, such as product goals as well as functional and design features, are all integrated into a single model which makes a C&V model even more complex.

In this section, we explore various dimensions of C&V in product line engineering. We separate C&V modeling into problem and solution space modeling. Problem space modeling is further refined to product goal, usage context, and quality attribute C&V modeling. Also, solution space modeling is refined to capability/service, operating environment, and design feature C&V modeling. Relationships/traceability between these models is managed separately from these models.

2 Concepts

The most important attribute of software is the "softness" of software, i.e., software that is easy (cost effective) to modify and adapt to evolving requirements or changing operating environments, easy to port on different hardware or software platforms, and easy to reuse for development of similar applications. Softness of software cannot be attained without engineering it into software. To embed softness into software, there have been many software engineering principles and concepts proposed, such as information hiding, program families, modularity, design patterns, etc.

In order to apply these design principles and concepts, however, we need to understand the commonality and variability (C&V) of the product line, i.e., a family of products. We need to explore the "space" of C&V of the products in a product line and potential evolution ("time"-dependent variability) of these products in the future, and then organize and codify the knowledge gathered as a C&V model. With this understanding of C&V, we can engineer software applying various design principles and embedding variation points that can later be bound with variants. For example, design decisions (design features) that can change may be encapsulated into software components applying the information hiding principle, and each changeable decision (alternative design features) can be implemented as a variant.

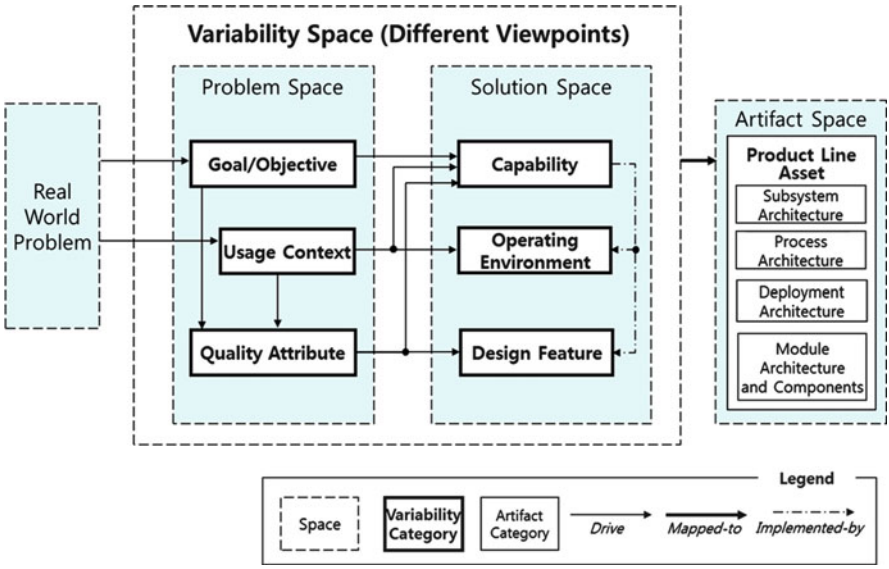


Fig. 2.1 Variability modeling space

In the following section, we explore different dimensions of variability of a software product line.

3 Commonality and Variability Modeling: The Scope

C&V of a product line can be modeled in many different ways based on different viewpoints (i.e., separation of different concerns). Largely, we can separate the problem space from the solution space[1] (see Fig. 2.1). For the problem space, user goals and objectives, required quality attributes, and product usage contexts are typically modeled in product line engineering. Within the solution space, C&V is typically modeled for the functional dimension (i.e., capabilities, services), the operating environmental dimension (e.g., operating systems, platform software, etc.), and the design dimension (e.g., domain technologies). C&V explored and modeled for these dimensions are materialized as software architectures, components, variation points, and variants in the artifact space. Implementation mechanisms such as inheritance, template, framework, macro, and generator may be used to implement variation points and variants.

[1] The terms "problem" and "solution" are relative. A solution for one may be a problem for others to solve. Requirements, which are considered "problems" to solve by designers, are "solutions" to real-world problems. One may view features in the "solution" space as problems for asset development in the artifact space.

The goals and objectives modeled for a product line defines "problems" at their highest level of abstraction to be addressed by the products of a product line, and therefore, they drive derivation of capabilities and quality attributes, which in turn may trigger derivation of other capabilities in the solution space. For example, the goal of moving passengers between floors safely will require elevator "capabilities" such as cabin moving, call handling, and door operation in addition to the obstacle detection for safety, a quality attribute. Techniques for implementing capability features are modeled as design features, each of which has associated quality attributes. For example, different obstacle detection devices may have different performance characteristics.

Typically, products used in different usage contexts require different capabilities and/or different quality attributes. For example, elevators in a hospital require a higher quality floor leveling feature than those in an office building to let wheel chairs and other medical equipment rolled in and out of an elevator easily. A flash memory for USB drivers needs a higher frequency data update than those built into a camera, for example, as they may be pulled out anytime. It should be noted that what derives decisions on quality requirements, operating environmental elements (e.g., devices, software platforms used), and design techniques to use is not just required capabilities but often the context in which the product is used. Analyzing and understanding different product usage contexts are very important for successful product line engineering.

What is important in the variability modeling is that:

- There are different market segments or user communities who may have different goals and/or different product usage context
- Different goals or usage contexts may require different quality attributes or capabilities
- Same capabilities may be implemented in different ways (design decisions), which may have different quality characteristics

In variability modeling, we explore these different dimensions and model relationships between modeling elements as shown in Fig. 2.1. In the following section, we review techniques for variability modeling.

4 Modeling Techniques

4.1 Feature Modeling

Since feature modeling [1] was first introduced two decades ago, it has been widely accepted by the software reuse and the software product line engineering (SPLE) communities as a means for modeling C&V of a product line, i.e., a family of products. This is because features are abstract concepts effectively supporting communication among diverse stakeholders of a product line, and therefore, it is

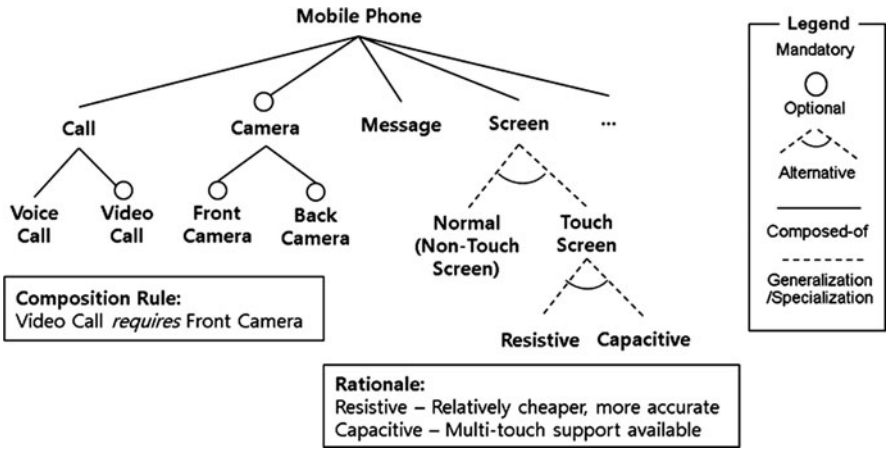


Fig. 2.2 A FODA feature model of a mobile phone product line

natural and intuitive for people to express commonality and variability (C&V) of product lines in terms of features. Also, it has been recognized that the C&V information codified by a feature model is most critical for developing reusable software assets.

In practice, many feature-based approaches to SPLE use features as units of:

- Capability that is delivered to customers
- Requirement containers, i.e., units of requirement specification
- Product configuration and configuration management
- Development and delivery to customers
- Parameterization for reusable assets, i.e., parameters for instantiating reusable assets
- Product management for different market segments

Furthermore, future products are typically discussed and described in terms of features gathered from market surveys, individual customers, research labs, or technology roadmaps.

The original feature model has very simple modeling primitives: structural relationships (composition, generalization/specialization), alternativeness, optionality, and mutual dependencies (inclusion, exclusion). Textual description and attributes of a feature may be defined. Also, the rationale for selection of an optional or alternative feature may be added as a textual description. Figure 2.2 shows an example of FODA feature model. This feature model describes a product line for mobile phones. In Fig. 2.2, *Video Call*, *Camera*, *Front Camera*, and *Back Camera* features are optionally selectable features. *Resistive* and *Capacitive* features are alternatives and can be thought of as specializations of general *Touch Screen* feature. As can be seen in the composition rule in Fig. 2.2, *Front Camera* feature must be selected when *Video Call* is selected. Selection of alternative features, *Resistive* and *Capacitive*, is made based on rationales shown in Fig. 2.2.

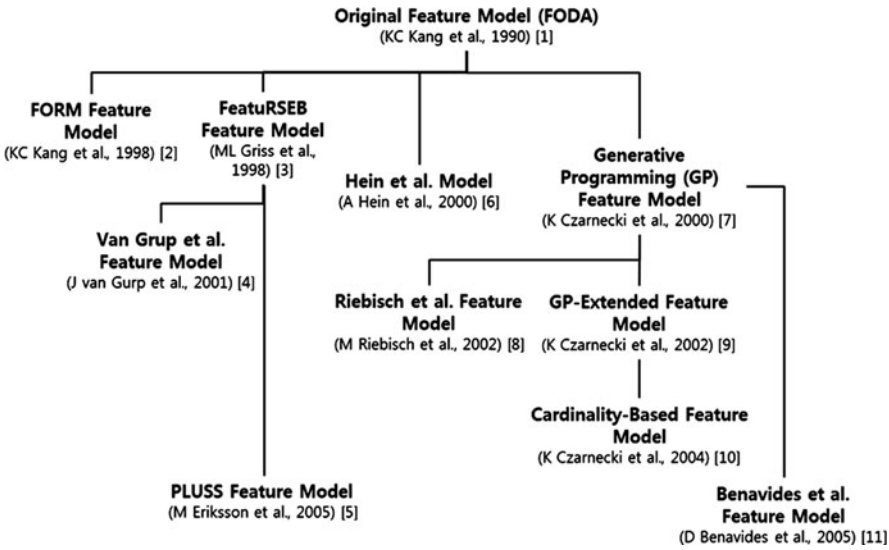


Fig. 2.3 Feature modeling approaches

For example, in case of buying a mobile phone, if a customer only concerns touch accuracy, then s/he may want to select *Resistive* rather than *Capacitive*.

After the introduction of FODA, many researchers have extended the feature model by adding new concepts for their researches [2–20], thereby resulting many variations (see Fig. 2.3) and extensions are still continuing. For instance, FODA was extended in [2] by introducing different viewpoints and grouping features into capability features modeling C&V of functions and services provided by the products, operating environment features modeling C&V of the environments in which these products are deployed and interface with, and domain technology and implementation techniques modeling important design decisions. A new relationship type "implemented by" was introduced to connect capability features (the functional dimension) with domain technology and implementation technique features (the design dimension) that may be used to implement capability features. Griss [3], Gurp [4], and Eriksson [5] made notational changes to the feature model and also provided notations for expressing dependencies and feature binding time. Hein [6] provided a UML-based modeling language. Czarnecki [7, 9, 10], Riebisch [8], and Benavides [11] refined the alternative relationship of FODA to XOR and OR relationships and also added the concept of cardinality allowing multiple selection of a feature. Attributes of features are also included in the feature model. Table 2.1 shows a summary of extensions made by each feature modeling approach.

As we examined the applications of these feature modeling approaches, we noticed that a feature model was often used to model different "concerns" of a product line in one model without delineating them. These concerns include the following: missions or business goals that need to be achieved by a product line,

Table 2.1 Summary of feature modeling approaches

Feature modeling approach	Extensions
FORM Feature Model [2]	• Introducing different viewpoints: capability, operating environment, domain technology, and implementation technology • Introducing a new relationship type *implemented by*
FeatuRSEB Feature Model [3]	• Making notational change: alternative features → variation point feature and variant features • Providing constraint (e.g., *require*) notation • Providing binding time notation: reuse-time and use-time binding
Van Gurp et al. Feature Model [4]	• Introducing *external* features • Refining the generalization/specialization relationship to OR-specialization and XOR-specialization relationships • Providing binding time notation: compile-time, link-time, and run-time binding
PLUSS Feature Model [5]	• Making notational changes: – A group of alternative features → single adaptors – A group of optional features → multiple adaptors • Providing constraint notation
Hein et al. Feature Model [6]	• Providing UML-based modeling language • Introducing secondary structure for constraint (e.g., *require*) dependencies
Generative Programming (GP) Feature Model [7]	• Refining the alternative relationship to XOR and OR relationships
Riebisch et al. Feature Model [8]	• Introducing the concept of feature group and group cardinality • Providing constraint notation
GP-Extended Feature Model [9]	• Introducing the concept of feature cardinality
Cardinality-Based Feature Model [10]	• Introducing the concept of feature cardinality, feature group, and group cardinality • Introducing a new relationship type *feature diagram reference*
Benavides et al. Feature Model [11]	• Including feature attributes in the feature model

functional capabilities provided by a product line, required nonfunctional properties (quality attributes), operating environments in which products are deployed, major design decisions to realize functional capabilities and achieve quality attributes, and rationales for configuring features for a certain usage context. These concerns may be classified as shown in Fig. 2.1.

This coexistence of multiple viewpoints[2] in a single model naturally leads to the following problems:

[2] For the same object, we can observe it from different angle, i.e., viewpoint, and extract different information. For example, an orthopedic doctor's view of human will be different from that of an internist.

- Analyzing, understanding, and defining the relationships between different viewpoints are a big burden to product line analyst
- Relationships between different viewpoints are not always explicitly defined
- A feature model with multiple concerns tends to become very complex, making it hard to comprehend and maintain
- The boundary between the problem space (features to capture the context of a product line) and the solution space (features to capture the services and design decisions of a product line) are not clearly distinguished
- Optimal configuration of products considering quality attributes is difficult

There is a need for a holistic approach [21] to feature modeling to alleviate these difficulties by first exploring the feature space to identify different concerns and divide it into subspaces based on different concerns and then to examine how they are related to each other, enabling product line analysts to examine a broad spectrum of concerns of a product line while focusing on specific concerns separately. By delineating these concerns as distinct viewpoints, analysis of a product line becomes thorough and systematic. This means that a product line analyst can concentrate on a specific modeling space with a clearly defined viewpoint (i.e., concern) at a time and then analyze and model relationships between different concerns later. An example of this holistic approach is shown in Sect. 5.

4.2 Decision Modeling

The decision modeling technique for modeling variability was introduced by [22].

A decision model consists of:

- Domain-related questions to be answered in developing products
- The set of possible answers/decisions to each question
- References to the affected artifacts and variation points, or references to the affected decisions
- Descriptions of the effect on the assets for each decision, or descriptions of the effects on the answer sets of the affected decisions

The decision modeling technique relates domain questions to other related domain questions and then, ultimately, to domain solutions which are variation points and/or variants. It focuses on capturing decisions to be made in configuring products. The feature modeling, however, focuses on exploring, understanding, and modeling the feature space (i.e., domain "questions"-problems and their solutions) of a domain in terms of commonalities, variabilities, and relationships among them. The rationale for each choice may be provided as textual description. Both modeling techniques may be used to configure products of a product line.

5 Variability Modeling: An Example

In this section, we further explore various dimensions of variability modeling explained in Sect. 3 using an Elevator Control System (ECS) product line as example [23, 24]. We will also see how these different dimensions are related to each other. The feature modeling technique is used in the exploration.

5.1 Problem Space Exploration

The problem space includes features for goals/objectives, usage contexts, and quality attributes of a product line as shown in Fig. 2.1. These features present the concrete context of a product line, i.e., external forces that drive selection of specific design decisions, i.e., architectures, algorithms, or implementation techniques; these problem features are important to understand real-world problems[3] that the product line should address. That is, the problem space captures the information of:

- Why is the product line required in the market?
- When is a certain product configuration used?
- What are the expected qualities of a specific product or the product line?

The answers to these questions should be captured in an exploitable form so that we can establish clear traceability, not starting from functional product features, but from real-world problems.

The problem space can be divided into three sub categories: goal/objective, usage context, and quality attribute features. The goal/objective features represent what a system should achieve in order to solve real-world problems. For example, in the ECS product line, the real-world problem is as follows: as multistory buildings are introduced and the number of floors increases, moving objects between floors becomes difficult. In order to solve this *real-world problem*, the goal/objective of ECS may be: "Move objects between different floors of a building in an *efficient* way." It is important to clearly define the goal as it implies the scope of the product line. The above goal, for instance, can also be achieved by an escalator. If it is not the intension and if we want to include only elevators, the goal should be refined as: "Move objects between different floors of a building *vertically* in an efficient way *using a cage with doors*" (see Fig. 2.4a). Through such refinement iterations, product line analysts, market analysts, and developers can establish an explicit boundary of a product line and can share a common understanding about the ultimate goal of the product line.

[3] In this chapter, we did not cover modeling real-world problems but focused on "external factors" derived from real-world problems that influence configuration of features in the solution space.

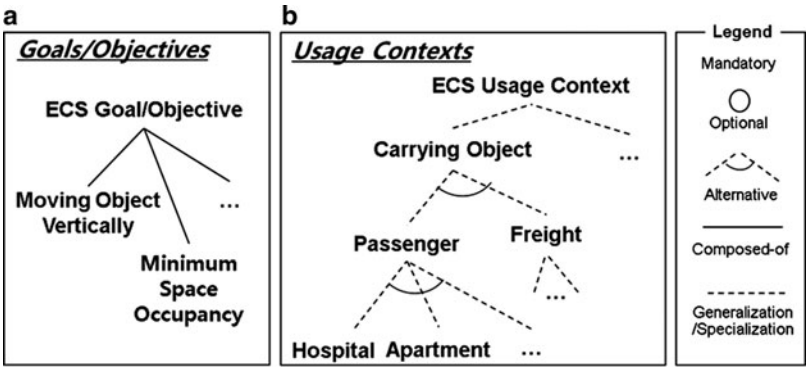


Fig. 2.4 (**a**) A goal/objective feature model and (**b**) a usage context feature model of the ECS product line

The next category is usage context, which represents a set of circumstances where a system is operated in. According to [25], usage contexts are any contextual setting in which a product is deployed and used. We follow this definition and it includes features about physical environments, user profiles, social or legal issues, business concerns, etc. For example, depending on the types of objects carried by an elevator, the usage context of ECS can be either a passenger elevator or a freight elevator (as shown in Fig. 2.4b).

The last category is about quality attributes: goal/objective and usage context features determine quality attribute features. Quality attribute features represent nonfunctional requirements that a system should satisfy while meeting its functional requirements. For example, for a passenger elevator, *Safety* and *Usability* features are important, while, for a freight elevator, "car call cancelation" feature may not be used for safety because of the weight of the load and the momentum of the elevator. Figure 2.5 shows an example of quality attribute feature model.

We need to explore C&V along these dimensions, which essentially derive decisions on required capabilities (functions) and various design choices.

In the following section, we discuss the solution space feature.

5.2 Solution Space Feature

The solution space captures functional, operational, and technical features that should be implemented for a product line. Most feature modeling approaches in the literature starts analyzing features that belong to this space, which can be classified into four categories (i.e., capability, operating environment, domain technology, implementation technique) according to FODA. It should be also noted that the term "solution" does not mean design artifacts in the space; features in this space are "solution decisions" for the problem space features, and these

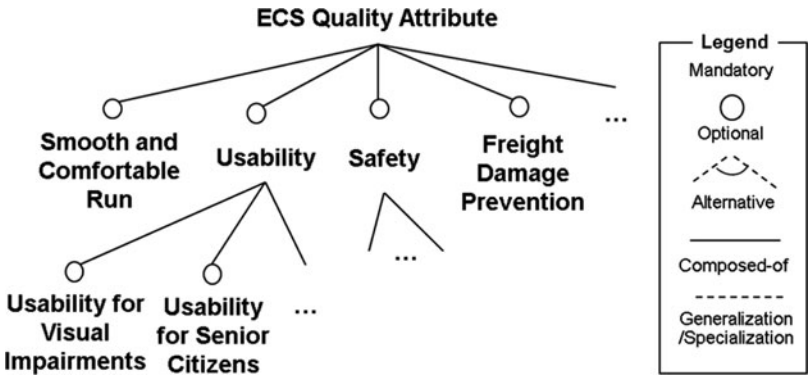


Fig. 2.5 A quality attribute feature model of the ECS product line

solution decisions must be implemented as product line assets (e.g., components) (The Artifact Space in Fig. 2.1). Figure 2.6 shows an example of solution space features.

Firstly, the capability features represent end-user visible characteristics of system such as service, operation, and function. For example, *Speed*, *Capacity*, *Hall Call Handling*, and *Motor Control* in Fig. 2.6a are capability features of the ECS product line. Secondly, the operating environment feature model captures C&V of target environments where products are deployed and operated in/on. For example, *RTLinux*, *VxWorks*, and *WindowsCE* in Fig. 2.6b are various real-time operating systems of the ECS product line. There are various sensors for detecting weight and leveling an elevator with building floors. Finally, design features represent design decisions such as domain technologies and implementation techniques. For example, in Fig. 2.6c, domain-specific algorithms such as *Motor Control Method* and *Weight Detection Method* are design decisions that are only meaningful in the ECS product line. Communication methods such as *TCP* and *UDP* represent concrete implementation techniques for a product line but they are more general and can be used in other product lines.

In the following section, we describe the relationships between these different viewpoints.

5.3 Dependencies Between Different Variability Viewpoints

In the variability modeling discussed in this section, features in the problem space *drive* decisions on features in the solution space. This means that the problem space features set clear contexts for identifying the solution space features and, thus, establishing explicit mapping between features in the two spaces. To model these spaces, we identified four activities and their relationships as depicted in Fig. 2.7.

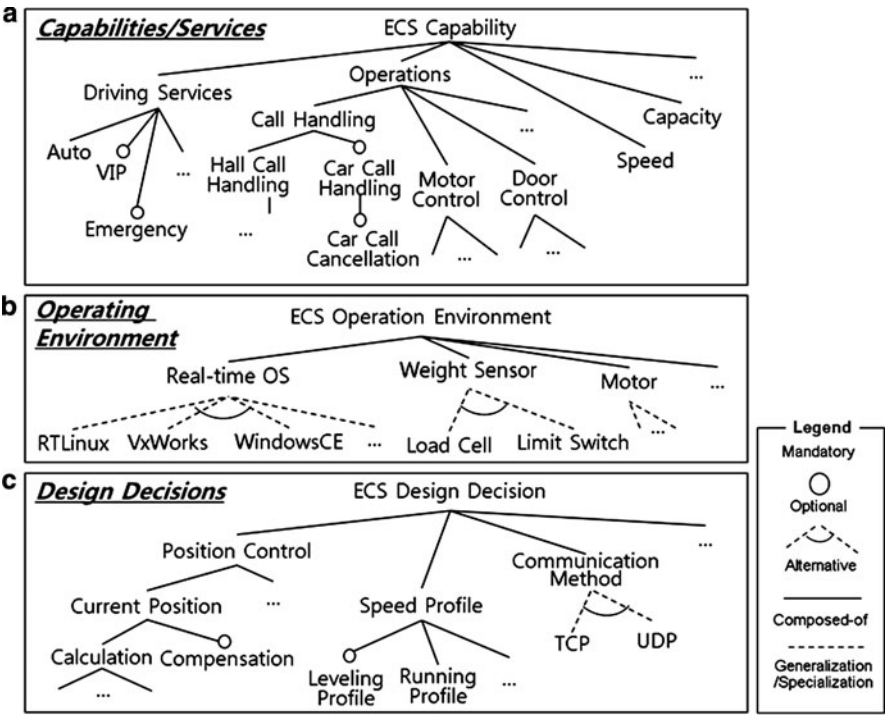


Fig. 2.6 A solution space feature model of the ECS product line

These activities are iterative and the arrows in Fig. 2.7 show data flow, i.e., use of work products at each activity. Each activity is briefly described below.

Organizing goal/objective features and usage context features from real-world problems of a product line initiates the modeling process. Goal/objective features specify the boundary of the product line and usage context features set specific contexts for the product line. The organized goal/objective features and usage context features are used as inputs to other activities.

In quality attribute feature modeling, quality requirements needed to achieve goals/objectives under various usage contexts are identified and organized into a quality attribute feature model. For example, the "safety" quality requirement of the ECS product line is to achieve the goal/objective of moving passengers safely in passenger elevators, and the "freight damage prevention" quality requirement is a goal set for freight elevators.

The problem space features (i.e., goal/objective, usage context, and quality attribute features) are used as primary inputs for the solution space feature modeling activity. Functional requirements that support the goal/objective under various usage contexts are identified as capability features. For example, the *Motor Control* capability feature is defined to satisfy the goal/objective of carrying objects between floors. The identified capability features may be refined further, and relevant domain technology and implementation features are identified considering

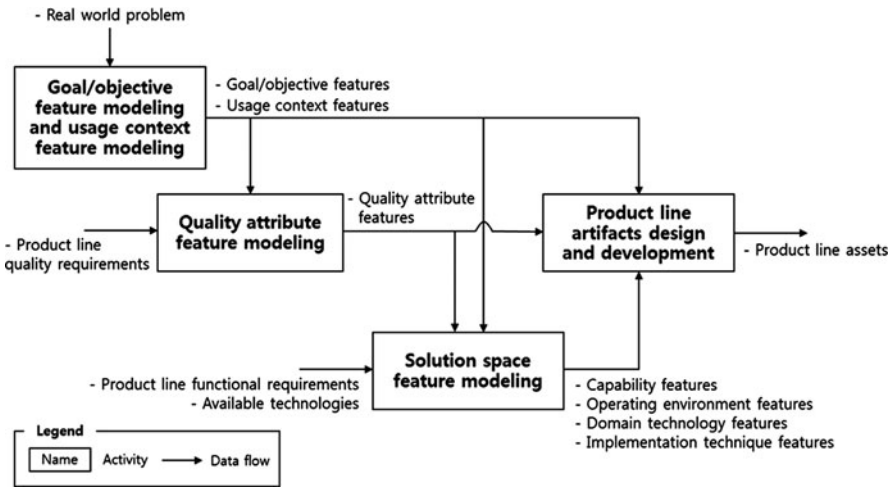


Fig. 2.7 Variability modeling process

goals and quality features and modeled in a solution space feature model. For example, leveling profile techniques that support the "smooth and comfortable run" quality attribute are identified as domain technology features.

In the product line artifacts design and development activity, the identified solution space features are implemented as product line artifacts including product line architectures, objects, and code modules. Variabilities captured as optional/alternative features in the solution space are embedded into the product line artifacts using various variability realization techniques (e.g., macro, aspects, etc.) [26–28].

In this section, we have examined the scope of variability. We will explore the temporal variability of product line software in the next section.

6 Feature Binding Time: Variability in Temporal Dimension

So far, we have seen C&Vs in the spatial dimension only, i.e., what features are common and what can vary. However, we should also explore C&Vs in the temporal dimension, i.e., when variability occurs, which is generally known as feature binding time. Generally, feature binding time has been looked at from the software development lifecycle viewpoint [7, 29], in which the focus has been given to the phase of the lifecycle at which a feature is incorporated into a product. In product line engineering, however, there exists another dimension that we have to consider, which we call *feature binding state* [12]. A feature may be *included* in the asset or a product at any product line lifecycle phase, but their *availability* for use can be determined at the time of inclusion or at any time after inclusion by enabling or disabling the included feature. Activation of the available features may

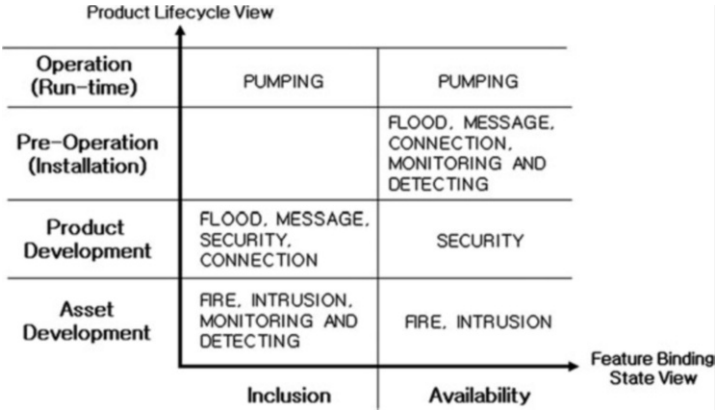


Fig. 2.8 Feature binding time analysis

be controlled to avoid a feature interaction problem.[4] Thus, feature binding time analysis with an additional viewpoint on *feature binding state* (which includes *inclusion* and *availability* states) provides a more precise framework for feature binding analysis.

For the purpose of temporal variability analysis, we can simplify the product line lifecycle into four phases: asset development, product development, pre-operation, and operation (run-time), shown as the vertical axis in Fig. 2.8. The horizontal axis shows binding states. The example in Fig. 2.8 shows that both *FIRE* and *INTRUSION* features are included in assets, and they are available for use as soon as the assets are included in a product. However, *FLOOD* and *MESSAGE* features are included during the product development time as product-specific features, but their availability is determined at installation time. The *PUMPING* feature is included and becomes available at operation time (i.e., run-time binding).

7 Discussion

After the FODA method [1] was published, there have been various efforts to introduce different viewpoints for feature modeling based on their own experiences [2, 10, 12–20]. These extensions include structural, configuration, binding, operational dependency, and traceability viewpoints. For the structural viewpoint [2, 10, 13, 14, 17–19], extended feature specification, feature relationships, and feature categories [10, 13, 16–18] added strict or recommended constraints into a feature model for helping product feature configuration in the configuration viewpoint.

[4] The problem of unexpected side effects when a feature is added to a set of features is generally known as the feature interaction problem.

Lee and Kang [12] extended a feature model by introducing feature binding unit (i.e., groups of features bound together) with binding time and techniques. Fey et al. [14–16, 19, 20] identified various operational dependencies between features, such as activation dependency, modification dependency, etc. Kang et al. [2] defined implementation relationship (i.e., a feature is necessary to implement another feature) to model traceability between functional and design features. These extensions, however, are limited to solution space modeling. Kang et al. [21] extended the scope of feature modeling further to cover problem space modeling.

In FODA [1], it is stated that issues and decisions must be incorporated into a feature model in order to provide the rationales for choosing options and selecting among alternatives. However, how issues and decisions are modeled and how they are related to (solution space) features was not explained. Kang et al. [21] modeled issues and decisions as problem space features and explicitly captured the relationships between problem space features and solution space features. These relationships are used in product feature configuration.

In FOPLE [30], marketing and production plan (MPP) is introduced as rationales for identifying and selecting product features. MPP can include goal/objective features and usage context features (e.g., user profile and cultural/legal constraints of MPP are similar to usage context features). In FOPLE, it is stated that MPP provides quality attributes for architecture design and refinement. However, they do not discuss how MPP provides different quality attributes and how quality attributes affect selection of product features. In this chapter, we explicitly explain relationships among usage context features, quality attribute features, and product features.

Some researchers [31, 32] added a quality attribute viewpoint into feature model and associated quality attributes with solution space features. Yu et al. [31] proposed a goal model to capture stakeholder goals that may represent quality attributes and associate goals to features. Thurimella et al. [32] suggested issue-based variability model that combines rationale-based unified software engineering model [33], and orthogonal variability model [34]. In their model, quality attributes can be modeled as criteria for selecting product features. However, Yu et al. and Thurimella et al. did not discuss how product-specific quality attributes are identified. In [21], Kang et al. discussed how product-specific quality features are identified from product usage context features and product quality requirements.

Some researchers [25, 35] proposed usage context viewpoint into feature model and associate usage contexts with solution space features. Hartmann and Trew [35] introduced a context variability model and define dependencies (i.e., requires, excludes, and sets cardinality) between a context variability model and a feature model. Lee and Kang [25] proposed usage context variability model and quality attribute variability model and defined relationships among usage contexts, quality attributes, and product features; selection of variant usage contexts eliminates choices of variant quality attributes and those of variant product features, and selection of variant quality attributes eliminates choices of variant product features, which is similar to modeling discussed in this section. Kang et al. [21] adopted usage context analysis introduced in these papers [25, 35], but, unlike these papers,

they clearly defined boundaries and relationships between the problem space, solution space, and artifact space.

Czarnecki et al. [10] suggested the concept of staged configuration, a process of specifying a family member in stages where each stage eliminates configuration choices, which can reduce the complexity of feature selection. Czarnecki et al. [36] extended this idea and introduce multi-level configuration, a form of staged configuration where the choices available to each stage are represented by separate feature models. In [36], it is stated that the criteria (e.g., geographical area or market segment) used to distinguish between the multiple product lines can be captured in a level-0 feature model, which is similar to usage context features discussed in this section. Their approaches [10, 36] are in the context of software supply chains [37] (i.e., each configuration stage is performed by different stakeholders in a software supply chain). Kang et al. [21] suggested a product feature configuration process that facilitates quality-based product configuration.

8 Summary and Outlook

This section introduces a holistic feature modeling method that enables product line analysts to capture complex concerns of a product line into different viewpoints and to decide product configuration systematically. Coexistence of multiple viewpoints in a single model without delineating them resulted in a highly complex and unmanageable feature model. The key idea in this section is the explicit separation of problem space features from solution space features. The approach also provides multiple viewpoints for each space so that a product line analyst can concentrate on a specific modeling space with clearly defined viewpoints at a time and do not need to consider other concerns. Relationships between these different viewpoints are explicitly modeled and used in making configuration decisions.

In this chapter, we explored explicit connections between goals/objectives, product usage contexts, quality attributes, and functional and design features. We also explored feature binding time issues. We expect to see more formal treatments of these subjects in a near future.

References

1. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (FODA) feasibility quality attributes and study. Technical report, CMU/SEI-90-TR-21, November 1990
2. Kang, K.C., Kim, S., Lee, J., Kim, K., Shin, E., Huh, M.: FORM: a feature-oriented reuse method with domain-specific reference architectures. Ann. Softw. Eng. **5**, 143–168 (1998)
3. Griss, M.L., Favaro, J., d'Alessandro, M.: Integrating feature modeling with the RSEB. In: 5th International Conference on Software Reuse, pp. 76–85 (1998)

4. van Gurp, J., Bosch, J., Svahnberg, M.: On the notion of variability in software product lines. In: Working IEEE/IFIP Conference on Software Architecture, pp. 45–54 (2001)
5. Eriksson, M., Börstler, J., Borg, K.: The PLUSS approach – domain modeling with features, use cases and use case realizations. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, pp. 33–44. Springer, Heidelberg (2005)
6. Hein, A., Schlick, M., Vinga-Martins, R.: Applying feature models in industrial settings. In: 1st International Software Product Line Conference, pp. 47–70 (2000)
7. Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley, Reading, MA (2000)
8. Riebisch, M., Böllert, K., Streitferdt, D., Philippow, I.: Extending feature diagrams with UML multiplicities. In: 6th World Conference on Integrated Design & Process Technology, Pasadena, CA, USA, 23–27 June 2002
9. Czarnecki, K., Bednasch, T., Unger, P., Eisenecker, U.: Generative programming for embedded software: an industrial experience report. In: Batory, D., Consel, C., Taha, W. (eds.) GPCE 2002. LNCS, vol. 2487, pp. 156–172. Springer, Heidelberg (2002)
10. Czarnecki, K., Helsen, S., Eisenecker, U.: Staged configuration using feature models. In: Nord, R.L. (ed.) SPLC 2004. LNCS, vol. 3154, pp. 266–283. Springer, Heidelberg (2004)
11. Benavides, D., Trinidad, P., Ruiz-Cortés, A., Pastor, O., Falcao e Cunha, J.: Automated reasoning on feature models. In: CAiSE 2005. LNCS, vol. 3520, pp. 491–503. Springer, Heidelberg (2005)
12. Lee, J., Kang, K.C.: Feature binding analysis for product line component development. In: van der Linden, F. (ed.) PFE 2003. LNCS, vol. 3014, pp. 250–260. Springer, Heidelberg (2004)
13. Capilla, R., Dueñas, J.C.: Modeling variability with features in distributed architectures. In: van der Linden, F. (ed.) PFE-4 2001. LNCS, vol. 2290, pp. 319–329. Springer, Heidelberg (2002)
14. Fey, D., Fajta, R., Boros, A.: Feature modeling: a meta-model to enhance usability and usefulness. In: Chastek, G. (ed.) SPLC2 2002. LNCS, vol. 2379, pp. 198–216. Springer, Berlin (2002)
15. Lee, Y., Yang, C., Zhu, C., Zhao, W.: An approach to managing feature dependencies for product releasing in software product lines. In: Morisio, M. (ed.) ICSR 2006. LNCS, vol. 4039, pp. 127–141. Springer, Heidelberg (2006)
16. Zhang, W., Mei, H., Zhao, H.: A Feature-oriented approach to modeling requirements dependencies. In: 13th IEEE International Conference on Requirements Engineering, pp. 273–284 (2005)
17. Streitferdt, D., Riebisch, M., Philippow, I.: Details of formalized relations in feature models using OCL. In: 10th IEEE International Conference on Engineering of Computer-Based Systems, pp. 45–54 (2003)
18. Ye, H., Liu, H.: Approach to modelling feature variability and dependencies in software product lines. IEEE Proc. Softw. **152**(3), 101–109 (2005)
19. Ferber, S., Haag, J., Savolainen, J.: Feature interaction and dependencies: modeling features for reengineering a legacy product line. In: Chastek, G. (ed.) SPLC2 2002. LNCS, vol. 2379, pp. 235–256. Springer, Berlin (2002)
20. Lee, K., Kang, K.C.: Feature dependency analysis for product line component design. In: Bosch, J., Krueger, C. (eds.) ICSR 2004. LNCS, vol. 3107, pp. 69–85. Springer, Heidelberg (2004)
21. Kang, K., Lee, J., Lee, H.: A holistic approach to feature modeling. September 2011 (unpublished)
22. Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R., Muthig, D., Peach, B., Wust, J., Zettel, J.: Component-Based Product Line Engineering with UML. Addison-Wesley, London (2002)
23. Lee, K., Kang, K.C., Chae, W., Choi, B.W.: Feature-based approach to object-oriented engineering of applications for reuse. Softw. Pract. Exp. **30**(9), 1025–1046 (2000)

24. Kang, K.C., Donohoe, P., Koh, E., Lee, J., Lee, K.: Using a marketing and product plan as a key design driver for product line asset development. In: Chastek, G. (ed.) SPLC2 2002. LNCS, vol. 2379, pp. 366–382. Springer, Berlin (2002)
25. Lee, K., Kang, K.C.: Usage context as key driver for feature selection. In: Bosch, J., Lee, J. (eds.) SPLC 2010. LNCS, vol. 6287, pp. 32–46. Springer, Heidelberg (2010)
26. Svahnberg, M., Gurp, J., Bosch, J.: A taxonomy of variability realization techniques. Softw. Pract. Exp. **35**(8), 705–754 (2005)
27. Anastasopoulos, M., Gacek, C.: Implementing product line variabilities. In: Symposium on Software Reusability: Putting Software Reuse in Context, pp. 109–117 (2001)
28. Bachmann, F., Clements, P.C.: Variability in software product lines. Technical report, CMU/SEI-2005-TR-012, September 2005
29. Bosch, J., Florijn, G., Greefhorst, D., Kuusela, J., Obbink, J.H., Pohl, K.: Variability issues in software product lines. In: van der Linden, F. (ed.) PFE-4 2001. LNCS, vol. 2290, pp. 13–21. Springer, Heidelberg (2002)
30. Kang, K.C., Lee, J., Donohoe, P.: Feature-oriented product line engineering. IEEE Trans. Softw. Eng. **19**(4), 58–65 (2002)
31. Yu, Y., Lapouchnian, A., Leite, J.C.S.P., Mylopoulos, J.: Configuring features with stakeholder goals. In: 2008 ACM Symposium on Applied Computing, pp. 645–649 (2008)
32. Thurimella, A.K., Bruegge, B., Creighton, O.: Identifying and exploiting the similarities between rationale management and variability management. In: 12th International Software Product Line Conference, pp. 99–108 (2008)
33. Wolf, T.: Rationale-based unified software engineering model. Dissertation, Technische Universität München (2007)
34. Pohl, K., Böckle, G., van der Linder, F.: Software Product Line Engineering Foundations, Principles, and Techniques. Springer, Berlin (2005)
35. Hartmann, H., Trew, T.: Using feature diagrams with context variability to model multiple product lines for software supply chains. In: 12th International Product Line Conference, pp. 12–21 (2008)
36. Czarnecki, K., Helsen, S., Eisenecker, U.: Staged configuration through specialization and multi-level configuration of feature models. Softw. Process Improv. Pract. **10**(2), 143–169 (2005)
37. Greenfiled, J., Short, K.: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. Wiley, Indianapolis, IN (2004)

Chapter 3
Variability Scope

Rafael Capilla

What you will learn in this chapter

- *The importance of binding system options*
- *The notion of variability in space*
- *Variability constraints and dependencies*
- *Automation of variability scoping techniques*

1 Introduction

A fundamental aspect of variability modelling and for software product line engineering refers to the scope of the product portfolio that is to know the number and type of the products to be produced. As software variability concerns with multiple product development and multiple product configurations, there is a need to delimit the scope of the products and determine the size of the domain in our product line. Scoping identifies what products are "in" our product line and relies on a set of allowed options described in the variability model to determine the list of feasible products that can be built. Therefore, software engineers must define which design choices and combinations of them will be valid for a given market segment.

There are many reasons (e.g. economic, business, technical) for delimiting the scope of the SPL products and thereby the scope of the variability model. Each reason must justify why a number of available choices must be out of the selection and product configuration activities, as the high number of combinations in large variability models, often belonging to industrial product lines, makes unmanageable and unfeasible the development and maintenance of a large set of software products. Consequently, delimiting the number and type of products must be driven

R. Capilla (✉)
Rey Juan Carlos University, Móstoles, Madrid, Spain
e-mail: rafael.capilla@urjc.es

R. Capilla et al. (eds.), *Systems and Software Variability Management*,
DOI 10.1007/978-3-642-36583-6_3,

by the scope of valid options defined in the architecture. Such restrictions are often based in a set constraint and dependency rules defined for the software artefacts and used to prune the number and type of products we can develop. In this chapter we will deal with notion of variability in space and with the reasons and technical solutions used for bounding the design choices used to keep the SPL products under control.

2 Scoping Activities

The need SPLs focus on specific market segments motivates domain scoping activities. Therefore, *domain scoping* is considered one of the first SPL activities used to delimit the number and type of products that will be inside the product line. Scoping activities narrow the domain of the product portfolio for the success of the SPL from a business and economic perspective. As a result, the scope of the variable options is also delimited by rules and constraints aimed to reduce the number and type of allowed products.

As discussed in [1], product portfolio analysis results are key to evaluating and establishing the type of products we want to engineer. As the product line evolves, the product portfolio may grow or change, and the variability implemented in the architecture must be flexible enough to support new variations in a controlled manner. Scoping activities also encompass the identification of requirements that are common to all products and those ones that make the difference between SPL products. Such activities will have a great impact on commonality and variability analysis to identify the variable parts in the architecture and with reusability of components and products in mind.

Moreover, *market analysis* activities are also carried as an early step before launching the product line in order to determine the product portfolio and to encompass which assets and products will be part of the product line. Therefore, product variants are defined and modelled on the basis of scoping activities and driven by economic and business reasons that keep the product line competitive.

Otherwise, the flexibility of variability models aimed to support a broad number of products in the product line scope often relies on more technical activities and current SPL capabilities, such as extensibility of variability models to support evolution and product configuration and derivation tasks. Bosch [2] mentions three different forms of scoping:

(i) *Domain scoping* aims at defining the boundaries of the domain where artefacts and products will be used. Domain analysis techniques are often used to delimit the scope of domain products and to derive the products from domain models (see also *Design Space Models* for product line scoping [3]).
(ii) *Product scoping* defines the products that will be engineered, often under a product line approach.
(iii) *Asset scoping* focuses on the identification of those reusable assets that will be employed in the construction of the software products.

These forms of scoping are used to constraint the number and type of options of variability models in order to make them more manageable. The scoping activity is fundamental for the product line strategy and economic benefits depend on how well the scope is chosen (e.g. a large scope may waste the investment of assets while a narrow scope may lead to not supporting reuse across all relevant products) [4].

Clements [5] states the importance of product line scope as a crucial activity for bounding the limits of the product line and define what's "in" and what's "out." Proactive approaches attempt to delimit the full scope of products when a product line is launched from scratch, while reactive approaches deal more with the scope of new products as the product line evolves and when new requirements appear. Scoping is sometimes considered a fuzzy activity during variability modelling and product line start-up, but several reasons motivate its importance in a product line context.

2.1 Reasons for Scoping

We can think in many reasons to enact scoping activities, but most of them may fall into the following categories:

- *Economic*: As not all the products can be built, there is a strong need to reduce the number and type of the assets and products because of economic reasons. Sometimes a product is technically feasible but difficult to sell and hence, it should not be included in the product line. For instance, an expensive product supporting a large number of configurable options that many of them will never be used. However, the case of a software product supporting only one single variation could be included in a product line if it shares a large number of assets. In other cases, a company can produce hundreds of products using a highly customizable variability model but building and maintaining such huge number of products will be highly costly (e.g. due to an excessive number of software development hours). Consequently, only those configurable assets and products that are worthy of value must be considered within the scope and a balance between the cost supporting a large number of configurable options (i.e. more products may lead to a broader scope) and a given pricing scheme must be achieved for each particular customization strategy.
- *Business*/*Strategic*/*Commercial*: Many times the variability model can support the development of a certain number of worthy configurable products, but business, strategic or commercial reasons may suggest to, for instance, delay its development. During scoping activities we do not restrict the scope of the variants for those products that will not be engineered in a certain period of time. Rather, we support such variations as part of the scope of the product line model but we decide later if certain variants (often known as *internal variability*) will be available in a new version of the product because the market demands new features (e.g. activate a new feature in the software of a mobile phone that remains hidden or unavailable in previous models of the SPL).

- *Technical*: Delimiting the scope of products or assets is necessary for maintenance reasons, as huge variability models are difficult to maintain and manage and may also increase product derivation activities. We use constraints not because the current technologies cannot support an infinite number of combinations but because of technical and other business reasons. Some systems that exhibit a large number of dependencies between their assets increase maintenance effort (e.g. the dependency network between packages in Linux kernels) and something similar may happen with variability models. Therefore, it is desirable to keep the number of dependencies and constraints under control and use tools for automating these tasks.
- *Cultural/Political*: Sometimes different configurable options are driven by cultural factors such as the language of use in different countries, which may lead to supporting a variety of languages in the GUI menu options of the product, while the functionality of the software remains the same. Delivering a software product in only a certain number of countries (e.g. due to political or military reasons) is another form to delimit the scope of the variants.

2.2 Variability in Space

Once the product line assets and products are well scoped, we can say that the pair variants and variation points defined in the variability model are ready to be used in product configuration and derivation activities in order to produce the reusable assets and the products in a given domain. The number and type of configurable products are determined by the design options defined in the architecture and implemented in the code assets. We refer to this as *variability in space*, where product line artefacts and releases are engineered and configured from the same variability model and belonging to a given domain.

Definition 3.1. Variability in space

Variability in space represents the set of products, releases and reusable assets that can be derived and configured from a concrete variability model in a given timeframe.

2.3 Notation for Binding Time

Variability in space provides the necessary ability to produce multiple products through variant selection and takes advantage against single-system development when several products and configurations must be engineered and put on time in the market. As mentioned in [6], "*feature declarations model the scope of variation in the production line*," and the adoption of software mass customization must support the complete scope of products on a predictable horizon. Also, depending on how

flexible and extensible the variability model has been designed to support evolution, new requirements should not be a problem if new design options and constraints can be easily added without changing the structure of the variability model.

In addition, the scope of the variability model is not only limited by the configurable options available but also by the constraint and dependency rules that will determine which products are allowed or within the scope. Such constraints must be described and implemented as part of the variability model, such as we explain in next sections.

3 Variability Scope

Product line scoping in its different forms have a direct impact on bounding variability. Because huge variability models applied in industrial product lines offer a large number of possible combinations, the feasibility to build only a subset of these products must rely on the limits established in the variability model to support a reduced number of allowed products.

3.1 The Graphical Limits of FODA

Variability models often use FODA trees to provide a graphical representation of the system features and how these interrelate with each other. A FODA tree describes the system features in terms of mandatory, alternative and optional variants which are also related using the notion of variation point. This hierarchy forms a tree where the root node represents the type of products we want to build.

One weak aspect of FODA trees is how constraints between features, used to delimit the variability in space, can be represented graphically. Also, representing variation points to relate variants located in different parts of the FODA tree can complicate the visualisation capabilities of the variability model, in particular in large feature models. In FODA, it is commonly accepted to draw a direct line associating two features to describe that there is a relationship between them, which can be either a constraint or a dependency rule, but constraints and dependencies are often managed separately from the graphical representation of the feature tree. With FODA, structural dependencies are modelled graphically and configuration constraints among optional and alternative features are specified separately to reduce the complexity of the graphical representation. Both of them must share the same name space. The same happens when we want to relate two or more variants and group these under a common variation point. A circle or dotted line surrounding the variants in the variation point is often used, but the logical formula describing such relationship must be written out of the FODA tree.

Figure 3.1 shows an example of a feature tree where variation points are surrounded by a dotted line and relationships between features are described

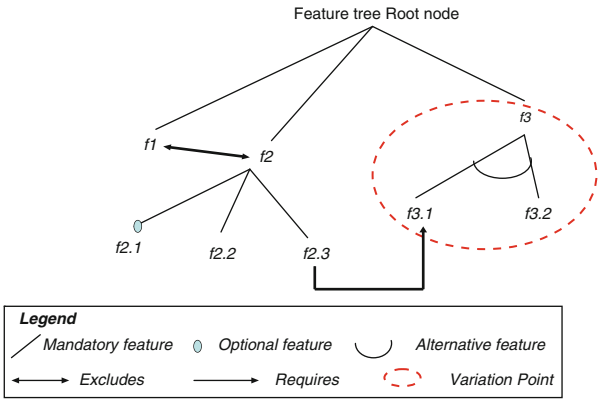


Fig. 3.1 A FODA tree example annotated with different types of relationships between features

using a solid line, but as mentioned, all this information must be described in textual form apart from the graphical representation. Therefore, FODA trees are simple and useful techniques to visualise the entire or a subset of the variability model, but the rules and constraints that define the limits of the allowed products must be defined and managed in a textual form. The existence of hundreds of features often makes hard the proper visualisation of all the potential constraints used to delimit the variability implemented in the product line products. For instance, in Fig. 3.1 we show three sample types of relationships that can be used to define the scope of the variability model, such as the following:

- Feature *f1* **excludes** feature *f2*.
- Feature *f2.3* has one **requires** relationship with feature *f3.1*. For instance, a feature cannot be activated if another feature has not been activated first. This can be seen as a special case of, the "requires" dependency.
- A **variation point** VP_x is defined to encapsulate and relate the alternative features *f3.1* and *f3.2* using, for instance an OR logical connector and having feature *f3* as parent of the relationship (e.g. $VP_{f3} = \{f3.1 \text{ OR } f3.2\}$).

Non-graphical representation techniques like matrixes can be also used to describe the dependencies and constraints of features. In addition, languages supporting rules and constraints constitute an interesting alternative as they can be processed automatically by software.

3.2 Variation Points

A variation point defines a relationship between features of a feature model and represents an area of a software system affected by variability. Variation points are used to relate two or several features located in the feature tree, and from the same parent or from different ones. Variation points encompass set of variants and other variation points that are represented by a logical formula that uses logical

connectors (e.g. OR, AND, XOR) to relate features. As not all the possible combinations are valid, the scope defined for each variation points is restricted by system constraints that limit the scope of products in space.

Variation points provide a flexible way to play with the scope of system features by grouping them as related functionality, often implemented as subsystems. When a variation point relates distinct functional parts of a system, the resultant area has a broader scope and the variability implemented in related system functional parts can be managed as a whole by means of such variation point. For instance, the variability implemented in the architecture that manages the electronics of a car can be used to describe the variations of both the Navigation subsystem while other features describe the variability implemented in the Multimedia subsystem (i.e. radio, DVD). Both subsystems can be integrated under one variation point representing an integrated multimedia system which can be also managed using a common control centre (e.g. the BMW's iDrive system consists of a button that manages all functions of the vehicle control system).

Variation points are often represented in feature trees as circles or boxes surrounding the variants included in the variation point, but because this technique may distort the representation of the feature model, variation points are better described separately in text notation or grouped in tables. The distortion of feature trees when using variation points can be reduced if we group subsystems or related functionality from the same parent, as we can avoid crosscutting lines across the feature tree. In huge variability models, it is rather difficult to avoid the existence of variation points relating distant features located in the tree or belonging to different parents, as in other case this may lead to a reorganisation of the whole variability model.

Just to give an example, Fig. 3.2 shows an example of variation points belonging to the same and to different parents. In the first case (left side of the figure), a variation point is defined and comprises three different variants. In the second case (right side of the figure), a variation point is defined to relate two distant features containing the variants defined in the feature tree but belonging to two different parents and depicted using a dotted circle line, as FODA lacks an explicit notation to describe such cases. In both situations the variation point defines the scope of certain functionality or related system features but this is managed differently. In the second case the scope seems to be broader than the first case because the functionality encompassed in the variation point shown in the right side of the figure encompasses functionality that belongs to separated or different part of the software product.

Because the scalability of the graphical representation of feature models is sometimes limited to describe and/or visualise hundreds of variation points, we need machine-processable techniques to solve this problem. However, most of FODA implementations are machine processable, as described in the Appendix of the FODA report, which includes a method of textual specification and also the extensibility of the model.

From an architecture point of view, variation points can be annotated as UML text notes and stereotypes in UML diagrams as no specific notation or neutral

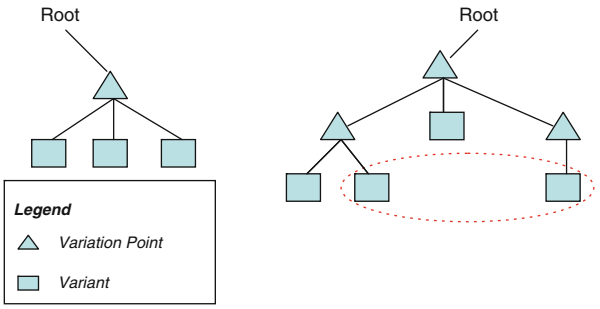


Fig. 3.2 Variation points belonging to the same and different parents

standard format exists to describe a variation point in the software architecture. This lack is common to all UML modelling tools and specific tooling has to be used to describe the variability of systems, in particular for industrial product lines where hundreds of variants and variation points need to be defined. Hence, the constraint and dependency rules used for delimiting the scope of the variability model can be hardly represented in the architecture and specific variability modelling and management tools are required.

3.3 Variability Constraints

In a feature-oriented approach, features are usually not independent each other, and the number and type of allowed products that can be technically and economically produced in a product line is often restricted using *requires* and *excludes* constraints (i.e. a kind of dependency). These variability constraints describe additional relationships between product features that can be hardly represented in the feature tree. Such dependencies can be applied either between variants and variation points in order to restrict the number of feasible product variations and thereby the number of product configurations.

- *Requires* dependency: It is used to represent that a variant V_x or a variation point VP_x needs another variant V_y or variation point VP_y. A requires dependency means that when a feature is selected the other must be present in the same product.
- *Excludes* dependency: It is used to represent that variant V_x or a variation point VP_x excludes another variant V_y or variation point VP_y. That is, an excludes dependency means that two features cannot be present in the same product.

In FODA, an arrow between two variants or variation points labelled with "requires" or "excludes" is enough to describe graphically such relationships, but the high number of such constraints in large variability models makes that all these rules must be processed automatically depending on the language used. Simple *if–then* constructs are enough to describe these dependencies, but constraint

programming constitutes another alternative to describe the dependencies between features.

Example 3.1. Requires and excludes dependencies using *if–then*

A feature fy *is required* if a feature fx is present

IF (fx) THEN fy

A feature fy *is excluded* if a feature fx is present

IF (fx) THEN NOT fy

A feature fz (e.g. a variant) *is required* if the variation point represented by features fx AND fy is present

IF (fx AND fy) THEN fz

In addition, the *requires* and *excludes* dependencies can be defined statically when product options are bounded before runtime or dynamically when such dependencies define a runtime condition during product execution or as part of a runtime reconfiguration process.

Example 3.2. Static and dynamic "requires" and "excludes" constraints

Static: During a software installation procedure, a software package requires another package before it is installed. Hence, a static requires dependency is defined and resolved.

Dynamic: During system execution, the software of an elevator checks the maximum allowed weight before the user can press the button of a given floor. In this case, a dynamic excludes dependency is realised at runtime when the maximum weight is exceeded.

Variability models delimit the solution space using *requires* and *excludes* dependencies to constraint the diversity of products, but these dependencies often complicate the variability model due to a high number of interrelated relationships between variants and variation points. As a consequence, the variability that is coded in a given subsystem or product becomes less reusable and difficult to decouple when the product options have dependencies to other system features.

3.4 Operational Dependencies

Feature dependencies have many implications in the development of product line assets and products as these are used to delimit the scope of the structural variability. However, other dependencies are possible. As mentioned in [7], *operational dependencies* represent implicit or explicit relationships between features that happen during the operation of the system. This kind of dependencies can be considered as different forms of requires and excludes dependencies but associated to runtime properties rather than to those defined statically in the feature model. Therefore, operational dependencies delimit the scope of execution features instead of the scope of the number and type of products; however, they can be used to configure products with different execution capabilities.

Based on a previous work [7], we describe the following six operational dependencies[1]:

- *Usage dependency*: It represents a feature that depends on other features for the correct system functioning. For instance, the location of certain services in a mobile phone depends on the correct functioning of the GPS system feature.
- *Modification dependency*: The behaviour of a feature might be modified by another feature while it is in activation. For instance, the feature that activates the Anti-lock braking system (ABS) in the car depends on the features controlling the sensors of the wheel, and the ABS feature works differently based on the information received from the sensors.
- *Activation dependency*: The activation of a feature depends of another feature, and it can be classified into the following four categories:
  - *Exclusive-activation dependency*: This dependency refers to features that cannot be active at the same time.
  - *Subordinate-activation dependency*: It represents a feature that can be active while another feature is also active.
  - *Concurrent-activation dependency*: Two or more features that are subordinated to an active parent feature must be also active at the same time (i.e. concurrently).
  - *Sequential-activation dependency*: Some subordinators of a parent feature must be active in sequence, and the parent feature will be active after the completion of the sequence.

The complexity of modern software systems may lead to many expected and unexpected situations where the status and operation mode of a system may change and more operational dependencies may arise to deal with new situations when the environment changes.

4 Automating Variability Scoping Checking

In large variability models, where hundreds of features are required, the number of constraints and dependencies may become unmanageable and hence, automatic mechanisms are necessary (1) to check that the right products will be produced, and (2) to ensure the compatibility between hundreds of constraints and dependency rules.

The automatic analysis of feature models can be used to check the scope of the product line products and their different configurations based on the provided variability in order to ensure the compatibility of hundreds of product constraints.

[1] In this chapter we will consider operational dependencies as part of a previous work of one of the co-authors of this book rather than a mere reference to the related work.

Table 3.1 Mapping features to propositional logic and CSP notations

Feature relationship	Propositional logic	CSP

OR
P = (X OR Y)
	$P \leftrightarrow (X \wedge Y)$	If (P > 0)
		Sum (X, Y)
		Else
		X = 0, Y = 0

Excludes X *excludes* Y	$\neg (X \wedge Y)$	If (X > 0) Y = 0
Requires X *requires* Y	$A \rightarrow B$	If (X > 0) Y > 0

As nicely described in [8], there are different techniques that can be used to check the consistency of dependencies in feature model. In this chapter we summarise two representative techniques used to automate the analysis of feature models.

- *Propositional logic*: It uses a propositional formula consisting of a set of primitive variables related by logical connectors aimed to constraint the values of the variables. A feature model can be mapped as a propositional formula and then use SAT solvers[2] to determine the satisfiability of the formula expressed using first-order logic. The formula can be specified in Conjunctive Normal Form (CNF) and uses three logical symbols, as connectors (i.e. $\neg$, $\wedge$, $\vee$) that are used by most SAT solvers. Features are mapped to variables in the propositional formula and the relationships between features are described using several formulas and including constraints.
- *Constraint programming*: Is a programming paradigm where relations between variables are stated in the form of constraints. These constraints can be described using Constraint Satisfaction Problems (CSPs) (e.g. A or B is true) where the values for the variables are found and all constraints are satisfied. Conversely to propositional formulas, a CSP solver can deal with numerical values in addition to Boolean ones. Feature models can be mapped as CSP variables with values TRUE or FALSE, while the relationships between features are defined as constraints. A description of the usage of CSP solvers in the automated analysis of feature models can be found in [9].

Table 3.1 shows an example on how constraints and dependencies of a feature model can be expressed in propositional formulas and CSP.

5 Areas of Practice

Product line scoping is a key activity for the success of the product line. In the early stages of the SPL phases, domain scoping is sometimes perceived a fuzzy task and difficult to carry out. Hence, one first area of practice is to define clearly the scoping

[2] A SAT solver is a software that takes as input a propositional formula and determines if the formula is satisfiable, that is there is a variable assignment that evaluates the formula to true. Input formulas are often specified in Conjunctive Normal Form (CNF) notation [8].

activities in the SPL approach used, not only at the process level but also in variability modelling tasks. Some well-known SPL approaches like PuLSE [10] have a domain scoping phase (i.e. PuLSE-Eco) used to identify the scope of the product line and determine the product line members. Other approaches (Software Engineering Institute's Framework for Software Product Line Practice 5.0[3]) combine economic and business reasons to establish SPL scoping activities with more technical activities focused on production constraints.

Closer to variability management techniques, staged configuration of feature models are used to iteratively select features in order to reduce the variability in the feature model [11]. This technical activity can be seen as a way to reduce the scope of the final products during product derivation. In other cases, new features can be added to enhance the functionality of a given product (e.g. the calculator product line incrementally adds new functionality by adding features). Using this approach, errors can be detected easily on each stage.

Another area of practice concerns with the evolution of the current asset and product features. If variability models become too rigid to expand the scope for new product line members, a reorganisation of the structural variability is needed, maybe because the variability model is unable to support runtime changes. In more flexible approaches, where runtime variability is supported, existing features can be modified or new ones added affecting the scope of the product line.

6 Summary

Product line scoping is an important and challenging area to determine the allowed product configurations that will belong to the product line. As discussed in the chapter, there are several reasons (e.g. economic, business, technical, etc.) that justify the need for product line scoping activities at various levels of abstraction such as domain or product scoping.

We have described the notion of variability in space to refer to the number and type of products to be produced from a given variability model and limit the scope of the products in the product line. FODA and their successors (i.e. extended notations of the original FODA) or constraint programming techniques are of common use to describe variability models and to determine the valid product configurations.

Finally, other forms of dependencies between features highlight these relationships from a runtime perspective rather than from the structural point of view, as many systems that use context information requires additional capabilities to adapt themselves to a new environment, and variability that is managed at execution time play an important role. Hence, these new dependencies must be

[3] http://www.sei.cmu.edu/productlines/frame_report/index.html.

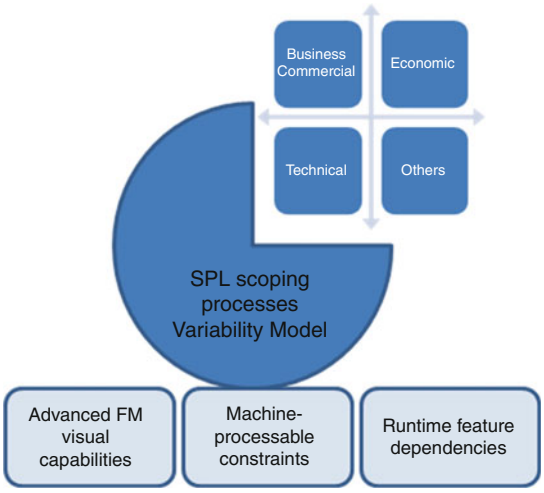


Fig. 3.3 Drivers and techniques for SPL variability scoping activities

able to address those runtime concerns among features that exploit runtime conditions.

Figure 3.3 summarises the main reasons for SPL scoping activities and the related techniques used to delimit the scope of variability models.

7 Outlook

Well-defined product line and variability scoping techniques are still needed. However, feature models are widely used to describe the variability of software systems, but other representation forms are required to accomplish the interrelationships between hundreds of features. Hence, new ways to represent large variability models and filtering techniques to describe a subset or a subsystem containing variability are welcome. Moreover, the scalability of feature models must be managed efficiently to expand or reduce the scope of the product line variants and hence facilitate the evolution of the product line. New trends and techniques in runtime variability models will help to support the dynamicity of systems and ecosystems and represent dynamic relationships between features that the structural variability cannot describe.

References

1. Pohl, K., Böckle, G., Van der Linden, F.: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer, Heidelberg (2005)

2. Bosch, J.: Design and Use of Software Architectures: Adopting and Evolving a Product Line Approach. Addison-Wesley, Reading, MA (2000)
3. Tekinerdogan, B., Aksit, M.: Managing variability in product line scoping using design space models. In: Software Variability Management Workshop, Ankara, Turkey, pp. 5–12. University of Twentepp (2003)
4. Schmid, K.: A comprehensive product line scoping approach and its validation. In: ICSE'02, pp. 593–610. ACM DL (2002)
5. Clements, P.: On the importance of product line scope. In: Software Product-Family Engineering, 4th International Workshop, PFE 2001, Bilbao, Spain. LNCS, vol. 2290, pp. 70–78. Springer (2001)
6. Krueger, C.: Easing the transition to software mass customization. In: Software Product-Family Engineering, 4th International Workshop, PFE 2001, Bilbao, Spain. LNCS, vol. 2290, pp. 282–293. Springer (2001)
7. Lee, K., Kang, K.C.: Feature dependency analysis for product line component design. In: ICSR 2004. LNCS, vol. 3107, pp. 69–85. Springer (2004)
8. Benavides, D., Segura, S., Ruiz-Cortés, A.: Automated analysis of feature models 20 years later: a literature review. Inf. Syst. **35**(6), 615–636 (2010)
9. Benavides, D., Segura, S., Trinidad, P., Ruiz-Cortés, A.: Using Java CSP Solvers in the automated analysis of feature models. In: GTTSE 2005. LNCS, vol. 4143, pp. 399–408. Springer (2005)
10. Bayer, J., Flege, O., Knauber, P., Laqua, R., Muthig, D., Schmid, K., Widen, T., DeBaud, J.-M.: PuLSE: a methodology to develop software product lines. In: SSR 1999, pp. 122–131 (1999)
11. Czarnecki, K., Helsen, S., Eisenecker, U.: Staged configuration using feature models. In: SPLC 2004. LNCS, vol. 3154, pp. 266–283. Springer (2004)

Chapter 4
Binding Time and Evolution

Rafael Capilla and Jan Bosch

What you will learn in this chapter

- *The notion of variability in time and binding time*
- *What a Feature Binding Unit is*
- *How binding time affects the evolution of products and architecture*
- *Open variability models*

1 Introduction

Software variability, as a powerful mechanism that enables the construction of different artifacts from a common architecture, enables the realization of variation points and variants at different times or stages. The moment in which the variability is bound to concrete design choices provides a flexible way to delay our design decisions to later stages during the software development process. Because supporting the evolution of variability models is critical for the success of the product line, we introduce in this chapter the notion of binding time. However, for a variety of reasons, different artifacts may require different times to realize their design options, and the ability in which the product line core assets and products match to concrete values at different binding times increases the flexibility of the product line architecture configurable options.

The realization of the variability may affect both to the developer side and to the client side as well, because the binding time happens at different stages of the

R. Capilla (✉)
Rey Juan Carlos University, Móstoles, Madrid, Spain
e-mail: rafael.capilla@urjc.es

J. Bosch
Chalmers University of Technology, Gothenburg, Sweden
e-mail: jan@janbosch.com

R. Capilla et al. (eds.), *Systems and Software Variability Management*,
DOI 10.1007/978-3-642-36583-6_4,

development process, from design time to runtime. Hence, the evolution of product line products using variability techniques is better supported with the selection of different binding times. Variability plus binding time mechanisms are used to avoid rigid architectural approaches that are difficult to evolve, and runtime binding offers, for instance, quick adaptation of systems to new context conditions without changing the architecture.

2 Variability in Time

In previous chapters, we learned that *variability in space* refers to the number and type of products that are built under the scope of a particular software product line. Such variations must be concretized at a given time to allow each particular product configuration, so the architect, developer, or customer knows when the variability will be realized to their concrete values.

As not all software systems have the same needs, *variability in time* [4] allows you to select when the system must be configured and to delay your design decisions to later stages in the software development process. It also refers to the different versions of an artifact that are valid at different times [13]. Thus, variability in time provides a powerful mechanism that software engineers use to delay or advance the decisions implemented in the software architecture and adapt their components and products at different stages in the software life cycle. The flexibility gained to configure the software at different times increase developers and customers' satisfaction and reduces further product configuration effort.

2.1 Binding Time

The notion of variability in time is often known in software product line engineering as *binding time*. Binding time can be understood as a property of variation points to delay the design decisions to a later stage, as new requirements or different context conditions may require concretize the variability at any time after design time.

Definition 4.1. Binding time

Is an attribute of variation points and/or variability technique used to delay the architectural design decisions to later stages in the software development process?

2.2 Binding Time in the Software Development Process

Different kind of systems may require an adaptation to different context conditions, and not all the software systems pose the same capabilities to react or change their own configuration or system properties. Therefore, different binding times are needed to respond to different adaptation demands. There is a wide variety of

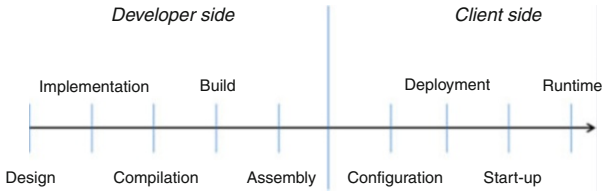


Fig. 4.1 Different binding times where variability can be realized

binding time choices to decide when system features can or must be configured and according to the adaptation level desired. Some of the possible binding times happen in the developer side, while others occur in the client side. Figure 4.1 shows the diversity of binding times, from design time to runtime. As we illustrate in the figure, part of the variability can be defined and realized in the developer side to configure the design choices earlier, while in other cases variability matches to concrete values much later, often in the client side, such as what Fig. 4.1 shows [5].

At *design time*, all variants and variations points are defined in the software architecture or in a complementary feature tree or table. During *implementation time*, the variability described in the architecture must be implemented in the software components (e.g., core assets in a product line) by means of a variety of programming techniques (e.g., parameters, class hierarchy, etc.). In other cases, binding time may happen at *compilation and build time*, where different software components can be selected according to different needs to produce a different version of a software package (e.g., a recent version of a math package that has to be installed in a Linux system or a new version of the Linux kernel that has to be reconfigured and afterwards built linking all their modules). Finally, the *assembly time* of products from the same or different suppliers may lead to bind and integrate products at the end of a product line. Variability can be used to decide on the selection of the products that will form part of a specific version.

In addition, variability can increase the flexibility of the configurable options if these can be resolved more in advance, often in the client side, as we do not need to ask developers to modify the variability model to support certain changes. In this case, binding variants at *configuration time* introduce a degree of freedom that let customers to configure a software product before execution. For instance, the installation of a new operating system requires configuring the date, language, and other O.S. features at the client side. Also, some systems need certain configuration operations when the system is deployed (*deployment time*). This could be the case of distributed systems, which need to configure IP addresses and server nodes, when they are deployed. Configuring variants at start-up is one of the most common used runtime binding. For instance, an operating system already preloaded in the machine needs to configure certain system parameters and user preferences before first start-up. Finally, the most flexible binding time occurs during the execution of the system (*runtime*), which can be reconfigured itself with low or minimal human intervention, and variability is dynamically bounded to the values according to different context conditions.

2.3 Feature Binding Units

In a software product line, not all system features need to be activated at the same time. Therefore, the need to accommodate the variability to different situations means that not all of the features will be switched on/off concurrently. Different binding times can be defined for the products and software components to allow designers to activate or deactivate a concrete feature or group of them in different moments also because one or more feature may pose one or several binding times.

Example 4.1. Activation of group of features

The variability implemented in an intelligent home system (IHS) covers a wide range of areas from lightning to security. If variability is used to model the activation and deactivation of IHS features, under normal circumstances, it may happen that the heating system can be activated at runtime according to different temperature conditions, while changing the password access of the door entrance could be done only at configuration time. In this scenario, the same system exhibit different binding times for different but related group of features.

During feature modeling, we need to define which features must be activated at a given time and, hence, define the binding time for them. It is easier for implementation purposes to agree the same binding time for a related group of features, preferably those that can be implemented in a single software component or class. Some authors [10] use the term feature binding unit (FBU) as "*to identify and bind service features that represent a major functionality of a system that can be added or removed as a service unit*".

The example on Fig. 4.2 represents part of the feature model of a Virtual Office of the Future (VOF) system described in [9], where the red circles represent an FBU for a group of related features with the same binding time. Pre- and post-conditions determine static or dynamic binding of features to reconfigure the system according different user needs. In dynamic scenarios, *require* and *excludes* rules relate dependent features and help also to model when a certain feature must be activated or not.

Modeling feature binding as units has several advantages, among which we can mention the following:

- Model groups of related features help to understand better how variants and variation points are activated at the same time.
- Facilitates the understanding of the dependencies between features and the order in which they are activated or tracked.
- Makes easier the implementation of the variability in the system which can be also associated to hardware components.
- Facilitates consistency and feature management.

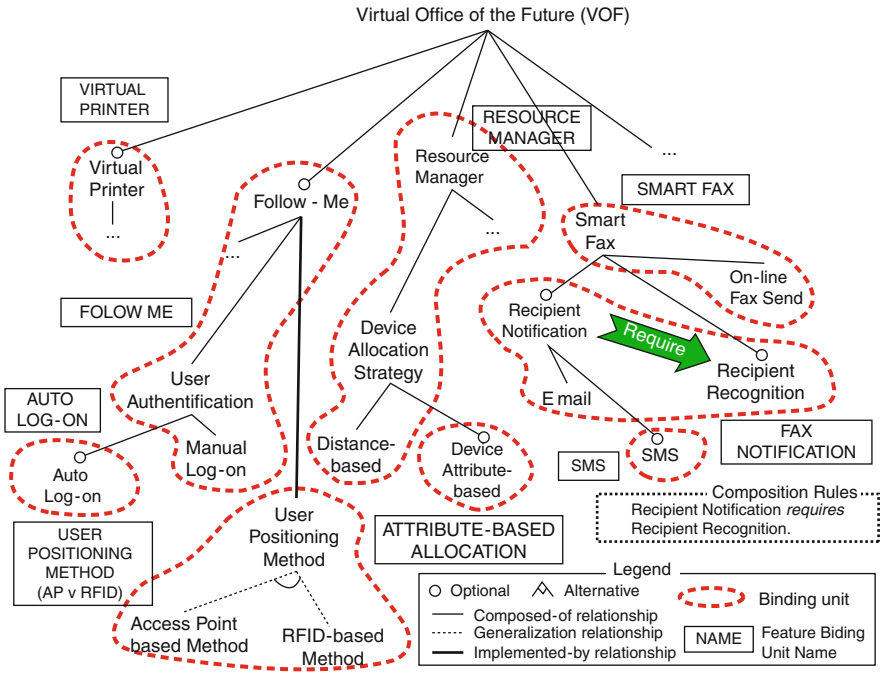


Fig. 4.2 A feature model and binding units of a virtual office system [9]

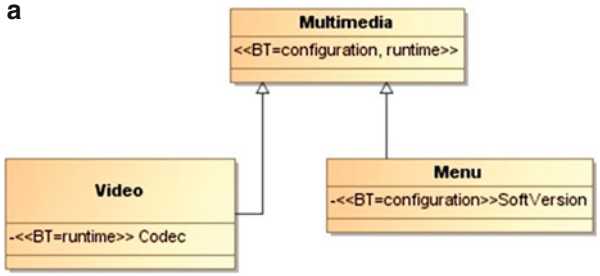
2.4 Notation for Binding Time

Conversely to variation points and variants, where several authors have proposed graphical and text notations to represent the variability in the architecture and feature models [13, 14], binding time lacks explicit notation. In feature trees, the binding time never appears, and only the FBUs attempt to describe graphically a common binding time for a set of features. There are two simple ways to describe the binding time of in the variability model.

(a) One way is to use a tabular form where the binding time appears in a single column and is specified using plain text for a set of features, software component, or architectural element.
(b) The second way is to annotate it graphically in the feature model or UML design, and indicate when the variability occurs.

A tabular representation has the advantage to relate easily the binding time with the provided variability and offer good precision to know when a variant or variation point bind to concrete options. Including the binding time in UML descriptions, like in [12], offers a quick view of the binding time in software components or classes, but worst to decipher at finer grain levels, also because a certain variant may pose more than binding times.

a



b

VP	Variant	Parameter/ Attribute	BT	Meaning	Value space	Default
VP1	Variant3	P3: TempSensor	Runtime	Detects the room temperature	[16-40]	23
---	Variant4	P4: Lights	Implemt. Config.		High Medium Low	Medium

Fig. 4.3 (**a**) Graphical representation of the binding time. (**b**) Tabular representation of the binding time

As variants and variation points are stereotyped in UML models with <<variant>> and <<variation point>>, respectively, and have well-known acronyms like "V" and "VP", we advocate to use the acronym "BT" for the binding time that is usually stereotyped with <<BT>> or <BindingTime>>. We prefer to avoid the inclusion of binding times in feature trees as these are mostly used to describe the structural variability, often at design time, rather than times where variability is bound. Figure 4.3a, b shows an example on how the binding time is shown using a graphical and tabular description.

Sometimes it is difficult to know if the binding time property must be attached to an entire class, software component or to smaller elements like class attributes or parameters. In the case of an entire class containing a portion of the system variability which has the same FBU, we can use one single binding time for such class or software component. In those cases where different variants have different binding times, a tabular description seems more suitable than UML diagrams to describe the binding time of the features. In Fig. 4.3, the UML description shows that the multimedia class has two different binding times that affect the entire class. In addition, the subclasses define two different binding times for two attributes, one of these binds the *Codec* attribute at runtime while the other binds the *SoftVersion* attribute at configuration time. Stereotyping the binding type (<<BT>>) in UML diagrams gives some initial indication, but the tabular representation constitutes a more detailed alternative to indicate when each feature binds to its values as well as other complementary information.

2.5 Binding Time Implementation Mechanisms

Binding time and variability implementation mechanisms are closely related. Depending on the moment in which variability is realized, we will need to use a different implementation technique. In the following subsections, we discuss different techniques to realize variability at different times.

2.5.1 Pre-compilation Time

Before compilation time, variants and variation points can be described in a UML diagram using stereotypes, tagged values, notes, or even OCL constraints to define the pre- and post-conditions. As UML designs are static artifacts, they can only describe which features or classes will change when the software architecture is configured as product architecture. As mentioned in [13], to bind the values of variants before compilation, certain techniques can be used like generative programming [1] or model-driven architecture [9], which attempt to automate code generation and give values for the variants selected to the available parameters. Also, aspect-oriented programming [8] is another pre-compilation technique that weaves different aspects or cross-cutting features in a program, and such aspects may include their own variability (please refer to Chap. 14).

2.5.2 Compilation Time

Variability can be used to define conditional compilation of program modules, where common compilation directives are separated from conditional compilation sections. Compiler flags and code sections are examples where variability is bounded at compilation time. The `#ifdef` directive is often used to define the variations during compilation to expand the macros for each compilation option, as code sections are included or excluded when the program compiles. Also, nested `#ifdef` sentences may increase the complexity of the dependencies during compilation. For instance, a Linux program that requires other Linux packages to be installed may generate a complex dependency network based on different options that are bound at compilation time, and all these options described using `#ifdef` directives must have the same binding time. Other directives such as `#include` can be used to define different options or files that will be included during compilation. Users must be aware about side effects between incompatible compilation options or when macros are expanded, similarly to incompatible variants or constraints rules defined in feature models, as compilation may stop when a variant that concretizes its value does not find the source (e.g., a file required which is not present).

Example 4.2. Binding at compilation time with `#idfef`

```
#ifdef unix
   #include <unistd.h>
   unlink(file);
#else
   remove(file);
#endif
```

2.5.3 Link and Assembly Time

Software products can bind the variable options at multiple stages along the life cycle, and hence, each variation point can be bound at a specific time. At build time, the variation points and variants realize the dependencies between modules prior to execution, and files are linked into an executable artifact. Usually, a Makefile provides the sequence needed to link their files or static libraries. Variability offers different alternatives to link the files by means of different linkage parameters. Similar to the example given for compilation time, a Linux kernel that must be recompiled and linked may use different linkage options that can be used by means of parameters to select the binary files and define the order in which the files must be linked. As Makefile generators are of common use, much of the variability that can be introduced is set up with specific configuration programs.

Hence, Makefiles can be constructed automatically and based on environmental variables (e.g., using a Configurator utility) before the made program is invoked. Macros and flags defined as variants work equally or similarly when a program is compiled. Resolving the variability at link time offers a flexible way to replace modules before execution time, as the names and versions of the binaries can be changed combining several Makefiles with other text configuration files to set the link options to concrete values.

A similar form of binding time where products are integrated is known as assembly time (e.g., a set of products are integrated at the end of a Software Product Line). Assembly can be seen as another way to link binary modules, such as software composition (i.e., composability), or even use generative approaches where a script is executed to produce a final executable file or to incrementally add new functionality by adding new features to an existing basic configuration. During assembly time, the software engineer defines which functionality will be added or removed into a concrete product version.

2.5.4 Configuration and Deployment Time

Configuring the variants after build and assembly time can be enacted in the developer side or in the client side. During configuration time, those variable options of a software system are configured before execution, during the first

start-up or on every start-up. A program can read a configuration text file with the concrete values without human intervention and set the values of the variants with the right configuration values. Tools can be used to configure the software that is going to be deployed (e.g., a new operating system uses an administration management tool to set up certain parameters), or automatic configuration scripts can be employed to upload the values of system features. For instance, we can use XML-based scripts to add new features to a software and then instantiate the variants to configure a new version of the product.

At deployment time (i.e., we assumed a software piece which has been already configured), some variants may still need a value before its execution. This is, for instance, the case of a distributed application that needs to set up a different IP address for the server or host where it will be installed. In this case, a script reads from the network interface or from the operating system network files the IP assigned to the destination host, and sets the variant with the right IP address.

2.5.5 Start-Up and Runtime

Binding the variants at runtime is the most flexible way to set up the variants and variation points. There are several forms where variants can be bound during execution and according to different needs. One form is to use dynamic files or libraries where one or several variants realize their values at runtime. For instance, as mentioned in [3], the Apache server supports dynamic loading of modules that can be linked statically at build time or dynamically at start-up. Sometimes, the system needs to be restarted to assume the new configuration. In other cases, the software locates a new version of or new functionality that has to be downloaded and installed but no restart is needed. The variability implemented in the system must support variants to locate the new software or library of the right or most recent version, and such values will be bound during the execution of the system. The order to load a new configuration is critical for the success of new software installation to avoid incompatible configurations. The values of these variants must be checked periodically to warn the user about the need to perform new operations in the system.

Pure runtime binding usually affects those systems that need to readapt themselves to a new context environment (e.g., autonomic computing, ubiquitous systems, self-adaptive, and self-healing systems, etc.). In this case, the values of the variants supporting certain context information are monitored periodically as they may change during runtime. Therefore, in case, a new configuration is required; the middleware enacts the corresponding procedure to set the variants with new values and perform certain reconfiguration operations that in most cases happen without human intervention. If binding happens at runtime, we recommend the use of binding time units to facilitate the implementation issues. Some additional effort must be done to provide purely runtime binding, but this extra effort is necessary for certain critical systems to provide unattended configuration facilities.

3 Multiple Binding Times

In complex software products, use of variability techniques in more than one binding time is possible as different functional parts of the systems may bind their variable options at different times. Also, the adaptation capabilities of certain systems demand runtime binding of their configurable options and this situation states the need for a transition between different binding times. For instance, an autonomous system may reconfigure some system options at runtime and certain software modules may go through a reconfiguration process and change its operational mode dynamically. Therefore, it is necessary to describe the possible transitions between different binding modes. As described in [15], capabilities and dynamic rebinding of multiple binding times are necessary to many of today's embedded system families that demand runtime adaptation and autonomous decision-making when context conditions change. In the era of post-deployment, current mobile and service-based systems may need to rebind to software services dynamically or be reconfigured at runtime.

Like in [15], Table 4.1 shows the transition between multiple binding times, from static to dynamic. The transition column shows the possible outcomes when multiple binding times occur. For a predominant binding time, we describe the possible binding times that can be supported simultaneously.

As showed in the table of Fig. 4.3b, a certain feature may pose more than one binding times which can be selected indistinctly according to different product derivation needs or to increase the flexibility of the product line. Features may cross from one binding time to another, but depending on how separate the two binding times are, more implementation effort will be required. For instance, it does not have much sense for a feature which realizes their values at design time to bind these at runtime. A feature or a variant which realizes their values at both and configuration and deployment times is more feasible as both binding times are much closer and they can be managed with (semi)automatic procedures. The transition between binding times using automatic mechanisms will require more implementation effort in those case closer to runtime binding modes, from configuration to pure runtime binding.

4 Open and Closed Variability Models

Evolution of variability models is directly related to the evolution of systems and their architecture. Variability models cannot be seen as purely static descriptions of systems' variations, as they might change during the evolution of the system. Variability cannot be used to predict unexpected changes in the future, but some flexibility can be introduced to modify the structural variability for certain types of systems that would require new variants or change the existing ones. Hence, the modification of variants and variation points that affect the structural variability is related to the concepts of *open* and *closed* variability models. Closed variability

Table 4.1 Transition between multiple static and dynamic binding times

Binding time	Static/dynamic binding	Configurability	Single/multiple binding	Binding in the developer and customer side	Transition for multiple binding times
Design	S	N/A	SI	D	N/A
Compilation	S	Low	SI	D	N/A
Build/assembly	S	Low	SI	D	N/A
Programming	S	Medium	SI	D	N/A
Configuration (Cf)	S/D	Medium/high	SI/MU	D/C	Cf→Dpl Cf→RT
Deploy (Dpl) and redeploy (Rdpl)	S/D	Medium/high	SI/MU	D/C	Dpl→Rdpl Rdpl→Cf
Runtime (RT) (start-up)	D	High	SI/MU	C	Dpl→RT Rdpl→RT RT→Rdpl RT→Cf
Pure runtime (PRT) (operational mode)	D	Very high	SI/MU	C	RT→PRT PRT→Cf Cf→PRT PRT→PRT

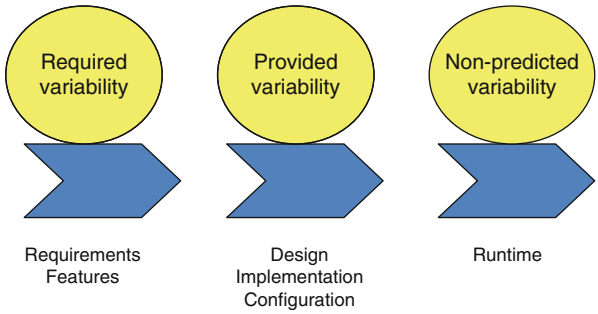


Fig. 4.4 Evolution of variability from design to runtime binding

models do not allow the modification of their variants and variation points until a redesign task is carried out. By contrary, open variability models allow adding variation points with new variants at runtime. However, introducing new variation points becomes more complicated as it requires some kind of human intervention to redesign the variability model and, maybe, introduce new constraints.

As Lehman said in his software evolution law [11] "*Variability has to undergo continual and timely change, or a product family will risk losing the ability to effectively exploit the similarities of its members*". Hence, the extensibility of variability models is intrinsically related to open variability models able to change the relationship of system's features during the execution of the system. If a feature model is modified during runtime, it must be redesigned and redrawn to reflect the new changes and dependencies between variants and variation points.

The evolution of variability models attempt somehow to anticipate to future requirements or to new situations that may occur at runtime. Hence, if a feature model is modified (e.g., due to a new variant), the conflicts and constraints between features must be resolved before the new variant can be used in order to lead to a new feature model consistent with the current state of the system.

In this context, the *required variability* of a software system understood as the *provided variability* evolves to initially *non-predicted variability* that could be modified over time (Fig. 4.4). Software engineers attempt to increase the flexibility and evolvability of feature models by pushing the binding time to runtime modes and foresee beforehand where new variants and variation points can be needed.

5 Evolution of the Structural Variability

The evolution of open variability models is intimately related to the changes performed over the structural variability, such as what we discuss in the next subsections.

5.1 Modification of Variants

Binding the variants and variation points at runtime is the most flexible form of variability. The inclusion, removal, and modification of variants at runtime are not trivial operations, but runtime variability becomes necessary for certain type of systems. Predicting if a variant which has not been included initially in the feature model can be needed later and requires additional implementation effort to manage dynamic changes of the structural variability model without, when possible, human intervention. We can find the following three situations.

Adding a new variant implies to know the place where the new variant will be added in the feature model. We foresee two possibilities. If the variant will not be part of an existing variation point, we can simply add it to the feature model and also indicate if the variant will be optional or mandatory, as an alternative configuration is often defined as part of a variation point. If the new variant will be part of a variation point, we need to redefine the logical formula that connects the new variant with the existing elements in the variation point. In addition, we may need to check existing constraints rules before the variant can be added to avoid incompatible configurations.

Removing an existing variant at runtime requires first to check if that variant will be no longer needed by all product configurations. If the variant is classified as optional, it shouldn't be a problem to remove it. In the case the variant belongs to a variation point, we will need to redefine the logical formula that relates the variants with other features for that variation point. Additionally, if the variant being removed has *require* or *exclude* constraint rules with other features, we will need to revisit all the rules and modify them accordingly to the new situation.

Changing a variant may lead to three different situations. When the allowed values of a variant vary (e.g., a new value or range, an existing value drops from the list, etc.), we only need to replace the values with the new ones (e.g., using a configuration file or parameters list). Another situation happens when a variant is moved to a different location in the feature model. This case can be treated as a removal operation of the variant and followed by an addition of the variant removed to a different place. In the case one variant replaces another, the constraint rules and compatibility type checks must be enacted, as the new variant does not change the dependencies and the current structural shape of the feature tree.

Moreover, it might be necessary to carry out some additional type checking when new variants are added or replaced, as using variants from different types in the same variation point may cause conflicts. Imagine that a new feature belonging to the car multimedia system is engaged with the features that describe the electronic control of the fuel system. Hence, we must define additional type checking for features of different functional areas complementary to the basic types (e.g., string, Boolean, numerical). Part of this process can be done automatically, but deciding which logical operator must connect a new variant with others is a manual task.

5.2 Modification of Variation Points

Modifying a variation point at runtime is harder than changing a variant, as it requires the modification of the logical formula that relates other variants and variation points.

Adding a variation point cannot be completely automated as the software engineer has to decide how new the variants and/or variation points will be connected using logical connectors that will define a new relationship between features. Additionally, we will check the compatibility of the types between the elements that will form the new variation point and check if new constraint rules introduced with the new variation point are in conflict with existing ones.

Removing a variation point implies that the logical formula connecting their underlying elements disappears. Hence, we need to check recursively if all the underlying elements for that variant are no longer needed as well as the existing constraints rules for where each of the removed features participated in. Removing a variation point may not imply that one of the underlying features should be also removed, as this can be still needed in the feature model and must be then relocated in a different place (e.g., due to a *require* rule for that feature).

Changing a variation point may imply a big reorganization of the feature tree, where variants and variation points can be moved individually to different locations. If we move a whole variation point with its underlying elements, we can consider this as a removal operation followed by an addition of the variation point in a different place. In case we move single elements like variants, you should refer to the discussion in Sect. 5.1. Moving an entire variation point as a whole does not require type checking as the existing compatibility between all the elements is kept the same as before the change is made. Only some additional constraint rule checking can be needed to define new dependencies in the refactored feature tree.

The modification of variation points is much more complex than changing variants in the feature model, as some manual tasks have to be done hampering full automation operations during runtime. Certain design decisions, such as selecting a new place in the feature tree to locate a variation point or defining new logical formulas, are hard to automate. Also, complementary type checking to basic types in features and constraint rule checking can be automated using compatibility lists. The modification of the variability model at runtime is not easy, but at least some automation can be possible to redesign the entire feature model and redraw it at runtime to reflect the changes made.

However, depending of the implementation technique used, we may allow the selection of more than one binding times and also decide how these changes can be managed: manually, semiautomatically, or automatically.

6 Areas of Practice

Planning the evolution of a product line implies not only the evolution of their products and multiple versions but also how variability models can scale up, evolving their feature models to incorporate new features or changing the existing variability model to adapt it to new requirements.

The definition of evolution scenarios to support the inclusion and modification of features is crucial, such as the case of a MobileMedia product line [2]. The approach described in [6] discusses the case of a Multimedia Home Platform (MHP) which requires late binding as such applications are characterized by constant domain changes and rapid customization is often required. Hence, designing runtime variation points is the solution proposed by the authors and based on a pattern language for building, manipulating, and managing domain-specific runtime variation points efficiently. However, some drawbacks may arise, such as degraded performance, increased memory consumption, and higher runtime complexity. The notion of meta-variability as a superseding variability model able to manage creation, removal, and modification of variants and variation that points on the fly seems crucial, in particular for Dynamic Software Product Lines (DSPL). This approach is discussed in [7], where a meta-variability model is used to support long-lived evolution of product line products. In the aforementioned approach, an AGV-automated transportation system (ATS) in DaimlerChrysler shows how variation points and variants can be changed during runtime to anticipate changes in the variability model.

Activating and deactivating features at runtime is another area of research, which concerns more with product configuration rather than with the evolution of feature models. Hence, those systems depending on different context conditions can activate or deactivate their system features and often during system execution, but such changes cannot be considered part of the evolution of variability models.

7 Summary

The notion of variability in time complements the notion of variability in space for producing multiple versions of products as it introduces the time condition. Hence, variability in time is implemented through different binding times that make possible to configure your software products at different stages of the development process. Such mechanism facilitates the evolution of both architecture and products because it provides high flexibility to realize the variability of feature models at different stages and hence to allow the necessary reaction to changes in the software requirements or when context conditions change at runtime.

The most flexible binding happens during system execution but the additional implementation effort required to change the binding time of a feature at runtime is not suitable and affordable for all type of systems. Critical systems that require real-time requirements are very dependent of context conditions, and they are the most

suitable candidates to include explicit support for binding their features at more than one time.

Conversely to static variability models, a clear enhancement of structural variability is to support their own evolution under critical conditions that would require open variability models, where new features can be added, removed, or changed at runtime. In addition, such structural changes require depicting automatically feature models to reflect the changes made and to provide mechanisms to check and avoid incompatible configuration when the variability model is changed. Current limitations from recent research show that modifying variation points is harder than changing variants, as some manual intervention is required.

8 Outlook

New trends attempting to provide better ways to manage the evolution of variability models will require more automation efforts to manage open variability models and, in particular, to deal with changes in variation point. Variability models are becoming more and more capable to support dynamic changes for certain types of systems in order to enhance the flexibility of purely static future models.

Also, in order to facilitate the task of software engineers to decide which binding times are more suitable, a categorization of application types and their functional modules that are associated to possible binding times would be helpful to decide which binding times are more suitable.

Finally, we need to clarify how much binding times are needed and desired for each particular application, module, or variant and how often these should change without decreasing performance and without increasing runtime complexity. All these issues are good candidates to be explored for future work in order to improve the evolution of variability related issues.

References

1. Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley, Reading, MA (2000)
2. Días, M., Tizzei, L., Rubira, C., García, A., Lee, J.: Leveraging aspect connectors to improve stability of product-line variability. In: 4th International Workshop on Modelling Variability of Software-intensive Systems (VaMoS 2010), Essen, Germany, pp. 21–28 (2010)
3. Dolstra, E., Florijn, G., de Jong, M., Visser, E.: Capturing timeline variability: with transparent configuration environments. In: Proceedings of the International Workshop on Software Variability Management (ICSE'03), Portland, OR, USA, pp. 47–52 (2003)
4. Elsner, C., Botterweck, G., Lohmann, D., Schröder-Prekshat, W.: Variability in time – product line variability and evolution revisited. In: 4th International Workshop on Modelling Variability of Software-intensive Systems (VaMoS 2010), Essen, Germany, pp. 131–137 (2010)
5. Fritsch, C., Lehn, A., Strohm, T., Bosch, R.: Evaluating variability implementation mechanisms. In: Proceedings of International Workshop on Product Line Engineering (PLEES), pp. 59–64 (2002)

6. Goedicke, M., Köllmann, C., Zdun, U.: Designing runtime variation points in product line architectures: three cases. Sci. Comput. Program. **53**(3), 353–380 (2004)
7. Helleboogh, A., Weyns, D., Schmid, K., Holvoet, T., Scheltfhout, K., van Betsbrugge, W.: Adding variants on-the-fly: modeling meta-variability in dynamic software product lines. In: Proceedings of 3rd International Workshop on Dynamic Software Product Lines (DSPL 2009), San Francisco, CA, USA (2009)
8. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: ECOOP 1997, pp. 220–242 (1997)
9. Kleppe, A., Warrmer, J., Bast, W.: MDA Explained: The Model Driven Architecture: Practice and Promise. Addisson-Wesley, Reading, MA (2003)
10. Lee, J., Muthig, D.: Feature-oriented analysis and specification of dynamic product reconfiguration. In: ICSR 2008. LNCS, vol. 5030, pp. 154–165. Springer, Heidelberg (2008)
11. Lehman, M.M., Ramil, J.F., Wernick, P.D., Perry, D.E., Turski, W.M.: Metrics and laws of software evolution – the nineties view. In: Proceedings of the Fourth International Software Metrics Symposium, Albuquerque, NM, USA (1997)
12. Myllymäki, T.: Variability management in software product lines. Tampere University of Technology. Software Systems Laboratory, ARCHIMEDES (2001)
13. Pohl, K., Böckle, G., Van der Linden, F.: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer, Berlin (2005)
14. Robak, S.: Feature modeling notations for system families. In: International Workshop on Software Variability Management (SVM) in International Conference on Software Engineering (ICSE'03), Portland, OR, USA, pp. 58–62 (2003)
15. Bosch, J., Capilla, R.: Dynamic variability in software-intensive embedded system families. IEEE Comput. **45**(10), 28–35 (2012)

Chapter 5
Variability Implementation

Jan Bosch and Rafael Capilla

What you will learn in this chapter

- *Mechanisms to implement software variability*

1 Introduction

Software variability is modeled, reasoned about, and discussed in many organizations, but at some point, it needs to be realized in the software of a system or product line. The subject of this chapter is to discuss the realization of variability in a software system or software product line.

The realization of a variation point can be achieved by a variety of technologies and approaches. Selecting the optimal approach is driven by two factors. The first is the abstraction level at which the variation point is explored, ranging from the architecture to the code level. The second is the stage in the life cycle at which the variation point is bound, whether the binding is permanent as well as the stages during which variants can be added to the variation point.

Choosing the right realization mechanism is of significant importance for two reasons [1]. The first is that it often is difficult to change the selected mechanism once it has been chosen. The reason for this is that variants are written to operate with a specific mechanism. In addition, frequently, variants are written by other organizational units or even other organizations altogether, as in the case of software ecosystems [2], which complicates changing the selected mechanism.

J. Bosch (✉)
Chalmers University of Technology, Gothenburg, Sweden
e-mail: jan@janbosch.com

R. Capilla
Rey Juan Carlos University, Móstoles, Madrid, Spain
e-mail: rafael.capilla@urjc.es

R. Capilla et al. (eds.), *Systems and Software Variability Management*,
DOI 10.1007/978-3-642-36583-6_5,

The second reason is that over time, many variation points tend to be bound at later and later times in the software development life cycle. A rigid realization mechanism that complicates this process will cause tension in the organization and inefficiencies in development.

Consequently, it is important to focus attention on variability realization. The remainder of this chapter is organized as follows. The next section provides a conceptual context of software variability management using the software life cycle by discussing the software variability realization implications in the different stages. The subsequent section discusses the abstraction levels at which variability can be captured. This is followed by the main part of the chapter where we present the different variability realization mechanisms. The chapter is closed by a discussion of relative advantages and disadvantages of different mechanisms and a conclusion.

2 Introducing, Selecting, and Binding Variants

Software variability can be discussed at several levels of abstraction, but at some point it needs to be implemented in the software system. For this, we need to have a good understanding of the software variability life cycle. This life cycle is obviously related to the overall software development life cycle. Although one can have different perspectives on the software development life cycle, in this chapter we consider the following stages:

- *Requirement specification*. During this stage, the team aims to maximize the clarity of what is to be built. There may be explicit requirements for variability, but equally often decisions are taken as part of the requirement specification process that reduces the required variability.
- *Architecture design*. The top-level breakdown of the system into its main components is the stage where the first variation points can be, and often are, introduced.
- *Detailed design*. Once the overall breakdown of the system is agreed and in place, the focus can shift to the design of the individual components. At this level, additional variation points can be introduced.
- *Software development*. Especially more narrowly defined variation points in the system are implemented using code-level variation points.
- *Compilation*. The compilation stage is often where the first variation points are bound to variants.
- *Linking*. During linking, especially higher-level variation points are often bound to specific variants. Most bindings during compilation and linking are permanent and cannot be changed in later stages.
- *Installation/configuration*. Assuming the software system is installed and configured at the customer, binding of variation points takes place during installation in response to settings selected by the customer installing the product.

- *Start-up*. During system start-up, several variation points can be bound to variants. Often, configuration files are used that are read during system start-up to bind certain variants to the remaining variation points.
- *Run-time*. Finally, the variation points that are not permanently bound in earlier stages can be bound and rebound during run-time. During installation and especially start-up and run-time, the binding of variants to variation points is often not permanent and can be rebound during at run-time.

During the software life cycle, a variation point evolves through a number of phases. The first is the *introduction* of the variation point at a specific stage in the life cycle. Frequently, this is in the earlier stages, but there are techniques that allow for the late introduction of variation points in the system. The second stage is the *addition of one or more variants* to the variation point. These variants capture the differences in behavior that are required from the system. The third stage is the *binding of a variant* to the variation point. At this point in the life cycle, the variant bound to the variation point can still be rebound. The final stage, though not reached by all variation points, is the *permanent binding* of the variant to the variation point. A variation point is bound permanently in a life cycle phase if in all subsequent phases it cannot be rebound to a different variant. At this point, the variation point, for all purposes, has been removed from the system at that phase in the life cycle.

One aspect of variation points is them being open or closed. At a certain phase in the software development life cycle, if variants can be added to a variation point, it is considered to be *open*. Many variation points will, in a later phase, become closed, meaning that the set of available variants can no longer be extended. This is largely orthogonal to the binding of a variant to a variation point. For instance, in an internet browser, a codec variation point can be bound to a particular variant, but the user can still add new codecs (variants) to the browser.

The coding effort for implementing binding times of features to support dynamic changes (e.g., system features that can be activated dynamically) can be reduced if we adopt flexible approaches like the one described in [3], where code-level idioms based on aspect-oriented languages can be used to avoid duplicate code for static and dynamic binding and enhance maintainability as well.

There are more complicated cases that we will not discuss in this chapter, including the reduction of the set of variants during progressive stages in the life cycle due to constraining dependencies as well as cases where variation points are permanently bound because of dependencies on other variation points and variants where their selection limits the set of alternatives to one. As discussed in earlier chapters, variation points and variants have dependencies on other variation points and variants. As the designer or customer configures the system, choosing a variant for one variation point will limit the set of possible variants for other variation points. Occasionally, this can lead to situations where a variation point has no remaining variants (e.g., the variability included in dead code will have no effect on the selection and realization of those variants). This, however, does not necessarily lead to an illegal configuration as the system configuration may not need the functionality provided at the variation point.

3 Variability Abstraction Levels

Depending on the size of the functionality that is to be variable, different variability abstraction levels can be identified at which reasoning about and realization of software variability can take place. We identify the following three levels:

- *Architecture*. At the architecture level, the primary mechanism for variability is the replacement of top-level components with other implementations of these components or the binding of optional components depending on the context in which the system is deployed.
- *Component*. At the component level, variability is often more pervasive and complex and often this is the main level at which variability is modeled. This is more concerned with extension points, superimposition[1] of code, wrapping, and other mechanisms that adjust the behavior of components.
- *Code*. At the code level, there is a large set of variability mechanisms available. The main concern, however, is that the code-level mechanisms can be applied for normal algorithmic implementation as well as for managing variation points.

Appreciating the differences between variation points at different levels of abstraction is quite important as each level brings its own advantages and disadvantages. Selecting the right level should be driven by the variability that is specified in the requirement specification, the expected evolution of the variation point, and the binding time of the variant to the variation point. In addition, specific trace mechanisms should be defined to track the changes from one abstraction level to another and vice versa, as managing the variations in one level (e.g., the variability defined in the architecture does not mandate how this will be implemented) is radically different from another level (e.g., different implementation mechanism can be used for coding variability at the code level) and the modification of the structural variability (i.e., the variability defined in a feature model representing the variants and variation points to describe system features) impacts the lower levels or configurations files supporting allowed options.

4 Variability Realization Mechanisms

There are several techniques to implement the variability which is described in feature models and each of these techniques is used in different stages of the life cycle and is driven by the time when variants will be bounded (i.e., variability realization). Basic variability enabling mechanisms are described in 1, 4, 5, such as inheritance, parameterization, conditional compilation directives, dynamic

[1] Superimposition of code is a black box component adaption technique that allows one to impose predefined but configurable types of functionality on a reusable component. Using superimposition, additional behavior is wrapped around existing behavior.

libraries, etc., but all these ways to implement variability in code are sometimes driven or limited by the language, framework, or technology used.

To provide a perspective driven by software variability management needs, we focus the discussion of variability realization mechanisms based on an earlier work by one of the authors [6]. In Table 5.1 below, we present an overview of techniques at different levels of abstraction and with binding times in different stages of the software development life cycle.

4.1 Binary Component Replacement

Intent. The intent of binary component replacement during linking is to permanently bind a specific component implementation. This allows the system to be bound to specific components needed for a particular configuration of the overall system. "Replacement" refers to a binary component that is specifically added for a concrete product or configuration instance.

Solution. The binding to binary libraries can be done at compilation and linking times prior to deployment. If linking is realized at run-time, the variability must manage this binding internally to the system assuming all libraries are available.

Example. Dynamic libraries such as Apache modules can be uploaded and bound at run-time when needed, whereas Linux kernel modules are linked before deployment when the kernel is recompiled.

Implications. This variability realization technique is easy to manage and to implement with few consequences to the system, as security is an aspect well covered in this case. By contrary, the unavailability of run-time libraries or incompatibility problems with existing version may cause severe problems.

4.2 Binary Component Selection

Intent. The intent of binary component selection is similar to selecting one component among a set of existing alternatives, and the binding time for selecting a component goes from installation to post-deployment time.

Solution. Dynamic components, libraries, and files are selected and bound among several. The alternatives can be bound more statically at installation time while they become more dynamic from start-up to post-deployment time, and variability is often realized externally to the binaries.

Example. Like in the previous case, any dynamic library or configuration file aimed to update the current system configuration or functionality fits under this category. In this case, the variability is managed externally to the component but some variants or system features can be defined in specific configuration files that can be uploaded dynamically.

Table 5.1 Variability realization mechanisms

Abstraction level	Binding time				
	Compilation	Linking	Installation/configuration	Start-up	Run-time
Architecture	N/A	Binary component replacement	Binary component selection	Binary component selection	Binary component selection
Component	Variant component specialization Optional component selection Code fragment superimposition	Optional component selection Code fragment superimposition	Optional component selection Variant component implement.	Optional component selection Variant component implement.	Optional component selection Code fragment superimposition Run-time variant component specialization Variant component implementation
Code	Condition on constant Code fragment superimposition	N/A	N/A	Condition on variable	Condition on variable

Implications. The implications are similar like in the previous case, but incompatibility problems of system features in system configuration or binary files may arise if these have not been pre-checked before. For instance, an older version of binary file is selected and such instance is incompatible with the existing version of the system or application running.

4.3 Variant Component Specialization

Intent. The intent of variant component specialization is to adjust a component implementation to the product architecture when the provided interfaces of a component implementation representing a variant feature vary. Specialization assumes a context-specific extension that is then developed for an individual product/configuration instance.

Solution. Separating the interfacing parts into different classes facilitates the interaction between components as we can decide what variant of the interfaced component to include in the product architecture. The variability in this case is bound externally but variants are realized at system design.

Example. A software using an enhanced security detection mechanism is only used in certain cases under a set of predefined conditions.

Implications. Several implementations must coexist that can be selected dynamically, sometimes at start-up time or at run-time.

4.4 Optional Component Selection

Intent. The intent of optional component selection is to include or exclude a particular component implementation, often selected from a set of existing alternatives.

Solution. System features are included or excluded as we separate the optional behavior in a different class or component. The binding time for an optional functionality goes from compilation to post-deployment time, as system features can be added or modified at any time. Binding is done externally by configuration management tools or by the compiler.

Example. A smart home system that adds or removes optional functionality for different customers and at a different cost (e.g., the system can use different security access methods). A basic package configured at compilation/linking time can be modified later by, for instance, adding a new module at configuration time.

Implications. Decoupling optional behavior is not always easy and depends on how the structural variability is defined and implemented in the system and the dependencies among the variants.

4.5 Code Fragment Superimposition

Intent. The intent of code fragment superimposition is to impose predefined types of functionality on a reusable component without directly affecting the source code.

Solution. With this solution, we superimpose product-specific behavior and concern's additional behavior is wrapped around existing behavior. In this case, the binding is realized externally and variability is bound at compilation or linking time, but run-time superimposition is also possible.

Example. Any crosscutting functionality (e.g., aspects) introduced in the system functionality constitutes an example of superimposition (e.g., different authentication methods based on internal or external authentication systems and the user or the system itself can select among one of these). At run-time, the Eclipse platform offers a way to dynamically add or remove plug-ins that include new functionality to the main platform.

Implications. Positively, superimposition enables that different concerns are separated from the main functionality. However, understandability on how the final code works becomes harder.

4.6 Run-Time Variant Component Specialization

Intent. It supports the selection between different specializations inside a component implementation during run-time, as different requirements may demand such capability.

Solution. The component implementation must provide a number of alternative executions that can be switched at run-time. Different design patterns (e.g., strategy, template method, or abstract factory) can be used to separate behavior into several classes and use inheritance or polymorphism to implement the required variability. In this case, the functionality for binding is internal.

Example. The case of a smart home system which provides sensors to detect several data, such as temperature, humidity, smoke, or people. The fire detection system can be activated at run-time to detect fire, as this is required to activate both the home smoke detector and temperature sensors. Different classes provide such functionality that is used by the smart home system control to activate the right sensors in case of the presence of smoke and high temperature.

Implications. Some common functionality might be duplicated when the variants must select between different specializations.

4.7 Variant Component Implementation

Intent. The intent of variant component implementation is to support several implementations of one component architecture that can be chosen at any time dynamically.

Solution. Several design patterns (e.g., strategy pattern, broker pattern, SOA service-broker pattern, etc.) can be used to select between one or several components with high flexibility and changeability. Variability is defined at design time and variants cannot be added later. Variability is bound internally to the system.

Example. Several e-mail protocols like POP and IMAP using the same interface for connecting to the e-mail server.

Implications. The reusability of some code pieces may be low.

4.8 Condition on Constant

Intent. The intent of condition of constant is to support a way to enact one operation from several available. It constitutes a refined version of variant component specialization and is often used to select between different compilation options.

Solution. Conditional #ifdef compilation directives can be used to implement the variability at compilation time. The collection of variants depends on constants that are used to bind the variants at compilation time.

Example. Any software package that uses compilation directives that are selected before the package is installed in the system. Also, configuration executable files are often used to determine the system environment and to drive the selection of the compilation values.

Implications. Using #ifdef directives can be risky and difficult to maintain, in particular when the installation of a software package involves additional packages or modules, as the number of interdependencies may grow exponentially across releases (e.g., the Linux kernel). Also, flexibility of the variability implemented using this option decreases as the number of links and potential paths grow. Moreover, variation points tend to be scattered as it becomes difficult to track what parts of the system are affected by one variant.

4.9 Condition on Variable

Intent. The intent of condition on variable is to support several ways to perform an operation but the choice can be rebound at run-time.

Solution. It replaces the condition on constant by a variable that changes its value dynamically. In this particular case, new variants can be added during implementation and variability is bounded internally.

Example. Any program that wants to control the execution flow can use this technique. Another example may refer to the selection of different web services at run-time according to certain conditions that are stored in variables (i.e., variants in the system) which determine the selection of a particular web service.

Implications. This is a very flexible technique where variants can be instantiated dynamically. However, tracking the value of the variation points can be sometimes difficult if variation points are spread throughout the code.

5 Selecting a Realization Mechanism

This chapter summarizes different variability implementation techniques from a high-level point of view as different languages (e.g., object oriented versus nonobject oriented) and design patterns can be used to implement each technique. Hence, we did not restrict our description to a particular implementation technology. Object-oriented classes, inheritance, variables supporting system features, dynamic libraries, and so on, are examples of different ways to implement the system variability, but selection of a mechanism is driven by the binding time at which the variants are bound.

In general, multiple binding times are hard to combine, so we need to select carefully which binding times we want to support in order to choose the right variability implementation techniques that can be mixed in the code or supported by a specific platform.

The selection of a preferred realization technique is driven by three factors: the mapping to the problem domain variability, the need for late-stage openness, and the expected system evolution.

Ideally, there is a direct, one-to-one mapping between a problem domain variation and a variation point in the solution domain. This significantly simplifies the configuration process and it avoids complex defect detection and repair situations. For instance, in a case where a problem domain variation is mapped to #ifdef statements in every module of the system, it does not require much to make a mistake in one module and have the resulting system act in unpredictable ways due to misconfiguration. Deciding the variability realization technique needs one-to-one map to the problem domain variation.

Second, depending on the system domain, there may be a significant need for late-stage openness of the variation point to allow adding new variants. The selection of the realization technique should explicitly consider the ability to add variants at the required time as many realization techniques cause permanent binding during the compilation and linking stage.

Finally, expected system evolution is an important factor in the selection of the variability realization technique. In practice, the binding time of variation points

tends to be delayed to later stages in the life cycle, meaning that even though a variation point may be bound permanently at compile time at this point in time, it is not unreasonable to assume that over time the binding will take place at installation, start-up, or run-time. Especially for variation points that have system-wide implications, the cost of replacing the selected variability realization technique may be very high and, consequently, it may be better to select a technique that allows for late binding.

6 Outlook

Writing adaptable and evolvable software using variability techniques is not always easy, as the modeling of large variability models is a complex and tedious task in itself. Because customers today push software developers to provide more and more configurable options, the *external variability* becomes more important, and this fact drives the realization of the variability times closer to configuration, run-time, and post-deployment times.

Systems that require run-time binding must implement the dynamic binding condition and use dynamic variability implementation mechanisms in a controlled manner to make the software more adaptable. However, only few variability implementation techniques can be used to realize binding and rebinding during execution time. Regarding the binding time of the variability realization mechanisms, one could think in a post-deployment realization mechanism, suitable for those systems that realize their variants once deployed. However, this new binding is quite similar to the run-time mechanism, and the slight difference between run-time and post-deployment perceived today is more subjective by software engineers because the variability realization mechanisms for architecture, component, and code are almost the same.

Finally, open variability models allow variants to be changed dynamically, but such high evolvability of the structural variability is hard to implement and requires additional codification to support the extensibility of the variability model.

References

1. Fritsch, C., Lehn, A., Strohm, T., Bosch, R.: Evaluating variability implementation mechanisms. In: Proceedings of International Workshop on Product Line Engineering (PLEES), pp. 59–64 (2002)
2. Bosch, J.: From software product lines to software ecosystems. In: Proceedings of the 13th International Software Product Line Conference (SPLC 2009), August 2009
3. Andrade, R., Ribeiro, M., Gasiunas, V., Sabatin, L., Rebêlo, H., Borba, P.: Assessing idioms for implementing features with flexible binding times. In: CSMR 2011, pp. 231–240 (2011)

4. Amin, F., Mahmood, A.K., Oxley, A.: An analysis of object oriented variability implementation mechanisms. ACM SIGSOFT Softw. Eng. Notes **36**(1), 1–4 (2011)
5. Myllymäki, T.: Variability management in software product lines. Tampere University of Technology. Software Systems Laboratory, ARCHIMEDES (2001)
6. Svahnberg, M., van Gurp, J., Bosch, J.: A taxonomy of variability realization techniques. Softw. Pract. Exp. **35**(8), 705–754 (2005)

Chapter 6
Variability Realization Techniques and Product Derivation

Rafael Capilla

What you will learn in this chapter

- *The notion of variability realization and product derivation.*
- *The relationship between binding time and product derivation.*
- *Automated product derivation approaches.*

1 Introduction

One of the ultimate goals of the usage of variability techniques is to allow the configuration of the software products under the product line approach. As different binding times are possible, different variability implementation mechanisms can be used to realize the variability at different stages in the software development lifecycle. Once variability is defined in the architecture and implemented in code, products can be configured at the end of the product line or even reconfigured at runtime. Hence, the variability defined in the architecture can be instantiated for configuring the product portfolio at different stages (e.g., pre-deployment, end of SPL, installation, runtime). Besides, variability realization techniques are intimately linked to the way and the moment products can be deployed, and several alternatives can be chosen to select the best configuration and deployment strategy. In this chapter, we will learn about variability realization techniques.

R. Capilla (✉)
Rey Juan Carlos University, Móstoles, Madrid, Spain
e-mail: rafael.capilla@urjc.es

R. Capilla et al. (eds.), *Systems and Software Variability Management*,
DOI 10.1007/978-3-642-36583-6_6,

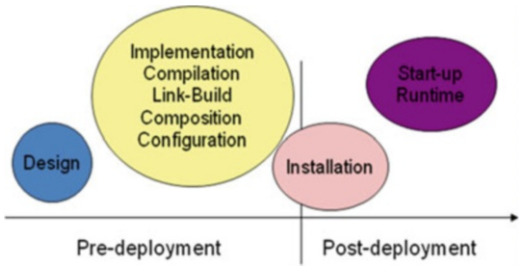


Fig. 6.1 Variability realization stages before and after deployment time

2 Variability Realization

In the solution space, the provide variability is realized by instantiating their variants and variation points in order to configure the products with the right and allowed values. Therefore, the realization of the software products implies to know at a certain time in the software development process which will be the values of the configurable options defined in the architecture and implemented in the core assets and products as well. Variability realization is intimately linked to product derivation, aimed to produce the concrete products once the values of the variants and variation points are known.

Definition 6.1. Variability realization technique
It is the way in which the variants of any family member are realized using a particular variability implementation technique at a given binding time.

The realization of concrete software products implies that the variable interfaces between components must be known, in addition to the invariants described in the architecture. The realization of the variability through the interfaces that may vary is crucial to set the right links between software components, as these interfaces act as a selector of the right component when more than one alternative exist. In addition, the realization of the variability must check the compatibility of the constraint rules, hundreds in commercial software, among the variants selected to avoid incompatibilities during the product derivation.

Definition 6.2. Product derivation
It is a stage in the software product line life cycle where software products become the resultant of a selection and configuration process of the variable design options defined in a variability model.

The software engineer must decide when to realize the variable options, and the flexibility provided by the existence of different binding times offers software engineers a way to delay their design decisions to a later stage. In Fig. 6.1, we organize the different product realization stages based on the moment in which products are or will be deployed.

We have to mention that installation time is not a real post-deployment variability realization stage as it is somehow in the middle, but we preferred to classify the realization of the variability during product installation closer to post-deployment time.

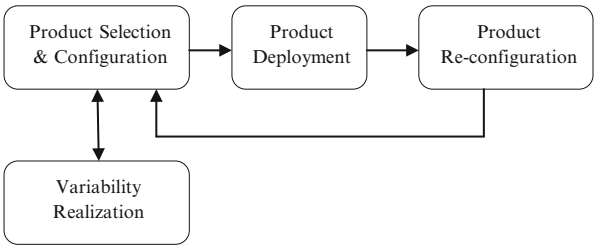


Fig. 6.2 Product derivation activities. The runtime reconfiguration of variants may lead to the selection of new variants and variation points and, in some cases, to a product redeployment phase

An exhaustive taxonomy of variability realization techniques and the factors that are relevant to implement variability can be found in [1], but the current trend in software development for several application domains like self-adaptive systems and service-based system pushes the realization of the variability to runtime modes. In this chapter, we will distinguish three major development stages in which we can realize the variability and according to most common binding times [2].

2.1 Product Derivation Activities

The ultimate goal of a product derivation process, as part of the SPL application engineering lifecycle, is to produce a configurable or configured software product. However, product configuration can be enacted at the beginning of the derivation process at early binding times, or it can be also executed at a very late stage if a product has to be reconfigured once deployed. Configuration is sometimes done to select the variable options that will be included in a product before the variability is realized to concrete values, while in other cases, a reconfiguration process happens at the end of the product line or during system execution. Moreover, product configuration and variability realization can also overlap at the same binding time if we realize the variants at the same time these are selected. At the end of the derivation process, products are installed and deployed in the physical nodes of the system.

As a summary, we show in Fig. 6.2 how these concepts are related and based on the binding times where these activities happen. Initially, product configuration starts by selecting the variable options that will be included in the product, and this activity may happen at different binding times, in which the realization of the variability will take place immediately after. Once the variable options match to concrete values, the executables can be deployed. However, post-configuration operations can be possible when the systems need to be reconfigured at post-deployment time, and dynamic variability plays an important role for systems that require runtime adaptation.

Figure 6.2 describes the major activities of a generic product derivation process. Once the input requirements define the selection of the variants of a new product, a product selection and configuration process chooses the right variants for configuration purposes, and variants are realized according to a particular implementation technique and the allowed values for those variants. Once the variability is realized and the product already configured, installed, and deployed, any post-deployment

activity or runtime reconfiguration of variants may lead to a new selection and configuration of the variable options. In that case, the reconfigured product or the new product (i.e., a different selection of the variable options can lead to different products) can or must be deployed again, while in other situations, no new deployment is required (e.g., the case of dynamic variability used to, for instance, activate a feature at runtime). The figure does not show testing activities that should be carried out to validate the selected product configuration.

In addition to Fig. 6.2, we detail in Table 6.1 which tasks encompass each of product derivation activities. For each of the major activities of Fig. 6.2, we provide the subtasks that are commonly needed and the most suitable binding times under which these tasks may happen.

2.2 Realization at Design Time

At design time, the realization of all variants and variation points is made at the architecture level. The variants in the design are manually operated, as the variability is considered statically in nature. Standard notations like UML offers few mechanisms (e.g., stereotypes, tagged values) to describe the variability of a feature model in the architecture, and the logical formulas describing relationships between variants do not have a direct correspondence in UML diagrams and they must be represented using a different notation or language. Therefore, the steps to realize the variability at design time are:

(a) Selection of variants and variations points defined in the architecture.
(b) Selection of allowed values.
(c) Depiction of the product architecture by instantiating the variants with appropriate values for each single product.

In Fig. 6.3, we show an example of a UML diagram that belongs to the software architecture of system X (left side of the figure) containing five variants and two variation points. At design time, the software engineer selects the variants to realize the construction of system X.1, and he/she derives the product architecture for that system. In this case, variant 3 and variant 4 with their corresponding values have been selected. Variation points and variants are selected and instantiated also for the product architecture.

2.3 Pre-deployment Realization

Products can be configured in the customer site and afterwards installed and deployed in the client side. When the variability of products is realized in the customer site, the variants and variation points can be instantiated at different binding times, depending on how the product is built and configured. At implementation time, the variability can be implemented in variables and the alternatives and

Table 6.1 Tasks encompassing product derivation activities

Activity	Tasks	Binding time	Description
Product derivation	Product configuration Variability realization Product deployment Product installation Product reconfiguration	From design to runtime	Product derivation comprises the main four activities described in Fig. 6.1 in order to build a software product. Product derivation is intimately related to variability realization techniques
Product configuration	Selects the variants to be included in the product Configure the allowed values Check dependency rules	Configuration time Pre-deployment Post-deployment Start-up	Product configuration deals with the selection of the variable options at different binding times. Excludes and requires rules must be checked
Variability realization	Instantiate the values according to the variability implementation techniques used Check constraint rules	From design to runtime	The variants match to concrete and allowed values. Rules delimiting the scope of products must be run
Product deployment	Product installation Node selection	Installation and deployment times, as no further binding time is considered	Installation comprises the nodes where the system functionality will be deployed and maybe some post-configuration activities
Product reconfiguration	Product configuration Variability realization Product deployment	From design to runtime	Reconfiguration procedures may lead to a feedback from runtime to variability selection and realization, and product redeployment activities as well

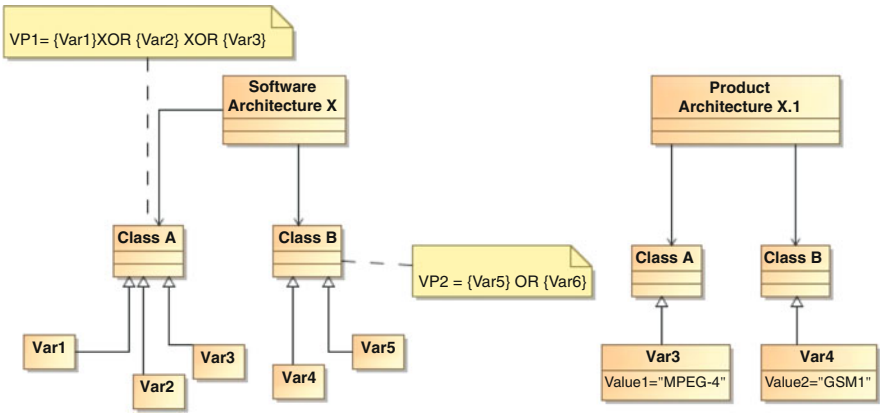


Fig. 6.3 Variability realization at design time

constraints described are often described as *if-then-else* constructions or using constraint programming. The realization of the variability depends on how this is implemented. For instance, the realization of the variants can be done statically in the code changes or more dynamically using dynamic libraries containing the configurable options. If we use an object-oriented approach, variability can be implemented using inheritance to separate the common functionality in superclasses from other variable option defined in subclasses which can be instantiated during the derivation process.

Example 6.1. Variability specialization through OO inheritance

The 3D scene of a virtual reality (VR) system is composed by 3D objects that constitute a hierarchy where objects are successively decomposed in polygons starting from a root object or node. Because the 3D database contains several megabytes and the time for loading the 3D scene during first start-up can delay several minutes, the way in which this hierarchy is organized at the architecture and implementation is critical, as some objects may appear initially hidden or might be unnecessary to show all the details of some of these 3D objects. Therefore, arranging and organizing this hierarchy in a particular form is quite important to reduce the start-up time. In this example, we used a particular object hierarchy (tested using simulation) to reduce the start-up time of the 3D scene and the variability techniques based on inheritance were used to group objects with common behavior [3].

In addition, at compilation and link or build times, directives expanding program macros, variables and compiler flags {e.g., `#ifdef, #include, #define`}, and Makefiles linking the program modules in a specific order are instantiated to produce different configurations or versions of the same product. For instance, a compilation variable can be used to discriminate a stand-alone version from a distributed one or add a security module not present in a different release. Makefiles use variables to make more flexible link and build options when generating the binaries, such as shown in the following code are certain flags that are stored in variables:

```
CC=g++
CFLAGS=-c -Wall
LDFLAGS=
SOURCES=main.cpp hello.cpp factorial.cpp
OBJECTS=$(SOURCES:.cpp=.o)
EXECUTABLE=hello

all: $(SOURCES) $(EXECUTABLE)

$(EXECUTABLE): $(OBJECTS)
        $(CC) $(LDFLAGS) $(OBJECTS) -o $@

.cpp.o:
        $(CC) $(CFLAGS) $< -o $@
```

Moreover, the idea of staged variability fosters the composition of features that can be added or removed to derive different product configurations (e.g., a scientific calculator has several versions of the same products and the product line approach used composed new functionality by selecting new variants). The software engineer selects and deselects features of each new version of the product. Hence, stage configuration becomes an important process applied in an SPL for configuring software products and where people make the right configuration choices at different stages. As described in [4], staged configuration of feature models constitutes a stepwise refinement of the variability model.

In some cases, this refinement leads to a specialization where groups of features are selected during the product configuration process and yields a specialized feature model. Specialization can be seen as a subset of the overall set of configurations and often done via transformations. In this context, baseline architectures play an important role for specialization and derivation processes as new product releases are yield as a result of successive stepwise refinement by adding and removing features from the baselines or from a concrete product configuration. Then, extensibility of the architecture becomes crucial to synthesize different product configurations or release products for different platforms.

2.4 Post-deployment Realization

The configuration of the variability and product realization in the customer site (post-deployment activities) often involves installation and post-deployment procedures where products are configured and deployed on behalf of a set of configurable options (e.g., parameters) that tailor the product to a specific environment or user preferences.

More dynamically, products may change the configuration of their variable options during start-up (e.g., first start-up or on every start-up) time as a system operator can configure certain variable options. For instance, the installation of a

new version of an operating system allows users to configure certain parameters of the target machine (e.g., language, screen resolution) or to select between two different preinstalled versions. On every start-up, the variability can be stored in configuration files (e.g., XML files) or local databases that are uploaded dynamically (e.g., a new user profile that has assigned new privileges). Other situations may deal with the dynamic upload of software modules or libraries that affect to the system configuration, as in some cases, the system needs to be restarted.

Finally, during system execution, the selection of variants happens while the system is running. The ability to select a new variant or to activate/deactivate features is considered a pure runtime variability realization (e.g., an adaptive system that realizes a reconfiguration of certain design options) which often happens at post-deployment time.

As a brief summary, we have to mention that depending on the concrete binding time and on the implementation language selected, the variability realization technique would be different (e.g., parameterization, inheritance, dynamic libraries, user-configurable options, etc.), and runtime variability realization techniques require more complexity and implementation effort.

3 Automated Derivation and Runtime Reconfiguration

The automation of product derivation and configuration tasks is quite important for certain variability management and product derivation operations. As some systems deal with runtime concerns, automating product deployment is increasingly interesting for such systems that require unattended and autonomous manual operations.

Usually, product configuration is perceived as a manual activity but dynamic SPL approaches attempt to manage the automatic activation and configuration of system features or perform an automatic redeployment once the system has been reconfigured dynamically. Some systems with stringent requirements require strict runtime adaptation of their systems options, while in others, it can be a semiautomatic human-guided procedure (e.g., a pluggable smart home system able to plug new software modules automatically and the variability is configured manually afterwards before launching the new functionality).

The automation of product derivation processes can be achieved following a generative approach or a specific model-driven development (MDD) where models are transformed before the final variability realization mechanism realizes the design choices. The input for such automatic process is a feature model or UML model that requires some kind of transformation before the variants are selected. For instance, in [5], an SPL derivation approach, built on the top of Rational Rose RT, provides automated support for developing multiple SPL views in UML and using the feature model as the unifying view.

One important topic in today's automation techniques for product configuration is automatic deployment. As systems or part of them are installed and deployed periodically (e.g., due to the installation of critical updates or because a new hardware is plugged and a reconfiguration operation is needed to support the new

functionality), there is an increasing demand to provide some degree of automation. Therefore, automating configuration and derivation processes in conjunction with deployment activities facilitates the task of software engineers, as many system configuration and installation procedures could be enacted unattended and automatically. In this scenario, software variability can play a key role to handle a set of configurable options that can be managed in automatic mode at runtime.

Some authors [6] suggest a model-driven engineering approach using variability mechanisms under a product line context to automate the customization and deployment of software products. This approach advocates the use of transformation languages such as ATLAS Transformation Language (ATL) and Acceleo, which extends the capabilities of the GenArch[1] software product line tool in order to transform software processes based on the Eclipse Process Framework[2] (EPF) to jPDL workflow language specifications and enable the deployment and execution of such processes. A feature model is used to specify the variability of these software processes and a product derivation tool allows the selection of the relevant features from an existing process, enabling automatic derivation from the software process to a workflow specification. Model-to-model transformations (M2M) facilitate the translation from an EPF specification of an automatic customized process to jPDL elements. Such automatic procedures often exploit model-driven engineering techniques to realize the transformations from high-level models (e.g., a UML specification) to code assets. Another technique which can be used is generation, which realizes stepwise refinements from baselines.

Consequently, the automation of product derivation and configuration activities requires additional coding effort to support automatic management of the variable options, as these configurable choices are sometimes handled by an automatic procedure, while in other cases, the ultimate goal is to leave some of these design choices to be modified by the user at runtime and post-deployment time.

Reconfiguring products at runtime may require in some cases to restructure the entire or a subset of the variability model. Reorganizing the structural variability model at runtime is challenging and hard, but this topic is out of the scope of this chapter. However, other runtime reconfiguration operations may imply automatic activation and deactivation of certain system features in order to meet new context conditions. Any runtime reconfiguration demands automatic redeployment mechanisms to meet the runtime condition, as well as additional runtime checks (even if a system changes its operational mode for some time) to ensure that the new configuration is the right one and properly set. Autonomic computing, pervasive and context-aware systems, service-based systems, and self-*systems are the most suitable candidates for runtime reconfiguration operations supporting variability. Other systems demand reconfigurable operations when new modules are plugged and unplugged and dynamic libraries or software modules can be selected automatically or with minimal human intervention using variable and configurable options. In those more complex cases, policies for runtime changes must be used to manage

[1] http://www.teccomm.les.inf.puc-rio.br/genarch/

[2] http://www.eclipse.org/epf/

the different situations that might arise during the selection of different configurable options and to detect incompatible product configurations.

4 Areas of Practice

4.1 Tooling

Several tools and approaches have been developed to support SPL derivation activities. From a methodological point of view, the "ConIPF Variability Modeling Framework" (COVAMOF) derivation process [7] describes the practical realization of variability for product families through a set of steps that go from the feature model to the component implementation and each of these levels are associated to COVAMOF variability views which capture the dependencies and relationships of the variability model. COVAMOF uses XML-based feature models and *#ifdef* constructs to describe and mange the variability information. The COVAMOF derivation process first configures the product to bind the variations and then realizes the product on the SPL artifacts in order to make effective the values of the variants.

Cirilo et al. [8] compares how three SPL tools (i.e. CIDE, pure::variants, GenArch^{+}) use configuration knowledge to compose the product line variability to derive the SPL products. This knowledge, used in configurable product lines, defines the implementation and composition of the variability for product derivation tasks. The comprehension of this configuration knowledge is crucial to understand domain-specific abstractions which are used for modeling coarse-grained variability and describe the relationships between SPL variability and code assets, annotations in feature models, and fine-grained variability implemented in class attributes and methods.

4.2 Experiences

In several industrial experiences, configuration and variability realization processes become relevant for product derivation. One early experience in the automotive domain [9] enables product derivation through the selection of combined variants aimed to support the right product configuration.

The well-known Koala model for handling the diversity of software products in the consumer electronics domain [10] is a clear example where the size and complexity of software products increasingly growing required a robust variability model able to handle this diversity. The Koala model proposes a strict separation between component and configuration development, as component builders do not make assumptions about the configurations in which components will be used.

Each component provides its functionality through a well-defined set of interfaces (e.g., the signal of a TV tuner is fed by a high-end input processor (HIP) that decodes luminance and color signals which are the inputs to a high-end output processor (HOP). All these devices are controlled by software drivers using a serial IC2 bus, as each driver requires, and IC2 interface that must be bound to an IC2 service during system configuration. A configuration in Koala is a set of components connected to form a product. In Koala, static binding is used during compilation running at configuration time.

In addition, the Koalish modeling language extends Koala and used for automating the product individuals in configurable software product families (CSPFs) [11]. Koalish is built on Koala and adds new variation mechanisms for selecting and configuring the type of parts of components, including constraints for specific individuals. In Koalish, configurations are sets of component and interface instances, and the relations describing which component instances are part of other component instances. The authors introduce the notion of valid configuration as not all possible configurations represent a system. On this basis, the WeCoTin is a prototype configurator tool operating on the product configurator modeling language (PCML) in order to ease the configuration of software product lines and feature models [12]. Reinforcing previous proposals, other authors [13] describe an analysis of the derivation process in two software companies for configurable software product families, from requirements to product delivery.

Regarding automatic product derivation, an experience using multi-agent systems (MAS) under a product line approach is described in [14], where a model-based product derivation tool (GenArch) is proposed for use in the application engineering lifecycle. GenArch consists basically of three steps: (1) automatic models construction, (2) artifact synchronizations, and (3) product derivation, which comprises customization and composition of the SPL architecture.

5 Summary

Evolution is an important aspect for today's software systems, and software variability reduces the barrier for systems that have to evolve more dynamically. Hence, feature models must be ready to support the selection and unselection of features and configuration operations during product derivation and deployment activities.

In this chapter, we have discussed the characteristics of major variability realization and derivation activities. Product derivation tasks can be organized according to pre- and post-deployment binding times, as this separation of concerns is easier to understand when and where (i.e., developer and customer sites) products can be realized. In addition, the categorization of derivation activities becomes important to know which kind of subtasks and which binding times can be used in any derivation process using variability.

The utility to realize the derivation at different binding times will depend in many cases of the type of systems we want to build and deploy, as not all software systems may require to support runtime concerns.

The areas of practice described in the chapter are several and show representative types of systems and applications in various areas that exploit variability realization techniques in different ways and with different binding times, as some of them have different deployment and configuration requirements.

6 Outlook

No one doubts about the importance of product derivation and deployment activities for variability management. In this context, automating reconfiguration and redeployment activities for critical and real-time systems is crucial, as systems using context information are more and more frequent. Systems using variable options evolve much better in dynamic contexts compared to those others than use more rigid approaches.

Finally, regarding the variety of variability realization techniques, we did not want to describe detailed examples on how each variability realization technique can be implemented, as this depends on the language or platform used. Rather, we preferred to provide an overview of the most common techniques used, organized around the time in which variants can be bound.

References

1. Svahnberg, M., van Gurp, J., Bosch, J.: A taxonomy of variability realization techniques. Softw. Pract. Exp. **35**(8), 705–747 (2005)
2. Fritsch, C., Lehn, A., Strohm, T., Bosch, R.: Evaluating variability implementation mechanisms. In: Proceedings of International Workshop on Product Line Engineering (PLEES), pp. 59–64 (2002)
3. Capilla, R., Martínez, M.: Software architectures for designing virtual reality applications. In: 1st European Workshop on Software Architectures (EWSA). LNNC, vol. 3047, pp. 135–147. Springer (2004)
4. Czarnecki, C., Helsen, S., Eisenecker, U.: Staged configuration using feature models. In: 3rd International Conference on Software Product Lines (SPLC). LNCS, vol. 3154, pp. 266–283. Springer (2004)
5. Gomaa, H., Shin, M.E.: Automated software product line engineering and product derivation. In: Proceedings of the 40th Hawaii International Conference on System Sciences (HICSS), p. 285 (2007)
6. Araújo Aleixo, F., Aranha Freire, M., Camara dos Santos, W., Kulesza, U.: Automating the variability management, customization, and deployment of software processes: a model driven approach. In: ICEIS 2010. LNBIP, vol. 73, pp. 372–387. Springer (2011)
7. Sinnema, M., Deelstra, S., Hoekstra, P.: The COVAMOF derivation process. In: International Conference on Software Reuse (ICSR). LNCS, vol. 4039, pp. 101–114. Springer (2006)

8. Cirilo, E., Nunes, I., García, A., de Lucena, C.J.P.: Configuration knowledge of software product lines: a comprehensive study. In: Proceedings of the 2nd International Workshop on Variability & Composition (VARICOMP), pp. 1–5. ACM DL (2011)
9. Thiel, S., Ferber, S., Fischer, T., Hein, A., Schlick, M.: A case study in applying a product line approach for car periphery supervision systems. In: Proceedings of In-Vehicle Software 2001 (SP-1587), pp. 43–55 (2001)
10. van Ommering, R., van der Linden, F., Kramer, J., Magee, J.: The Koala component model for consumer electronics software. IEEE Comput. **33**(3), 78–85 (2000)
11. Asikainen, T., Soininen, T., Männistö, T.: A Koala-based approach for modelling and deploying configurable software product families. In: Proceedings of the 5th International Workshop on Product Family Engineering (PFE-5). LNCS, vol. 3014, pp. 225–249. Springer (2003)
12. Asikainen, T., Männistö, T., Soininen, T.: Using a configurator for modelling and configuring software product lines based on feature models. In: Männistö, T., Bosch, J. (eds.) Proceedings of Software Variability Management for Product Derivation – Towards Tool Support, a Workshop in SPLC 2004, pp. 24–35. Helsinki University of Technology, Espoo, Finland (2004)
13. Raatkainen, M., Soinien, T., Männistö, T., Mattila, A.: Characterizing configurable software product families and their derivation. Softw. Process Improv. Pract. **10**(1), 41–60 (2005)
14. Cirilo, E., Nunes, I., Kulesza, U., Nunes, C., de Lucena, C.J.P.: Automatic product derivation of multi-agent systems product lines. In: Proceedings of the ACM Symposium on Applied Computing (SAC), pp. 731–732. ACM DL (2009)

Chapter 7
Visualizing Software Variability

Steffen Thiel, Ciarán Cawley, and Goetz Botterweck

What you will learn in this chapter

- *Core techniques in Information Visualization*
- *Using Visualization to support Software Variability*
- *Commercial and Prototype tools that utilize Visualization*

1 Introduction

Many of the expected benefits of software product line (SPL) engineering rely on an assumption that the additional up-front effort in domain engineering that establishes the product line produces a long-term benefit. The expectation is that deriving products from a product line during application engineering is more efficient than traditional single system development. However, to benefit from these productivity gains, it must be ensured that application engineering processes are performed as efficiently as possible. This has proven to be extremely challenging with industrial-sized product lines containing thousands of variation points, each of which can be involved in numerous dependent relationships with various other parts of the product line (e.g., [1, 2]). One method of addressing this challenge involves

S. Thiel (✉)
Furtwangen University of Applied Sciences, Furtwangen, Germany
e-mail: steffen.thiel@hs-furtwangen.de

C. Cawley
Dublin Institute of Technology, Dublin, Ireland
e-mail: ciaran.cawley@dit.ie

G. Botterweck
Lero-The Irish Software Engineering Research Centre, University of Limerick, Limerick, Ireland
e-mail: goetz.botterweck@lero.ie

R. Capilla et al. (eds.), *Systems and Software Variability Management*,
DOI 10.1007/978-3-642-36583-6_7,

supporting the SPL engineering activities by providing interactive tools that use, as a central principle, visualization techniques appropriate for the comprehension of large data sets and interrelationships.

Adopting visualization techniques in software product line engineering can aid stakeholders by supporting essential work tasks and enhancing understandings of large and complex product lines. This chapter presents visualization concepts, approaches, and implementations that are used to manage the application engineering phase of the SPL process.

2 Concepts and Techniques

2.1 Visualization

There has been extensive research into information visualization and its applications. Visualization has proven useful in enhancing cognition in numerous ways and application domains [3, 4]. This is particularly the case in relation to externalizing information, thus increasing the "memory" and "processing capacity" available to users, also by supporting the search for information and by encoding the information in a manipulable medium.

Visualization takes abstract data and gives it a form suitable for visual presentation. Such data can, for example, be explicitly collected from software or it can be codified by software engineers utilizing their own implicit knowledge. In this case, we often speak of software visualization, which can be seen as a subdiscipline of information visualization [5]. With suitable techniques, such software visualizations can also amplify cognition about large and complex data sets created and used in industrial software product line engineering.

2.2 Visual Reference Model

Figure 7.1 shows a visual reference model introduced by Card et al. [3]. This model provides a conceptual basis for many visualization approaches. *Source data* is transformed into a format (*data tables*) from which visual abstractions can be created. Various *views* can then operate over those abstractions, which provide the user with a rich interface. By allowing user interaction with the *view*, the different transformation steps can be altered in order to optimize the visualization for specific user tasks. This concept of interactive visualization forms the basis of many dynamic techniques aimed at providing cognitive support to stakeholders.

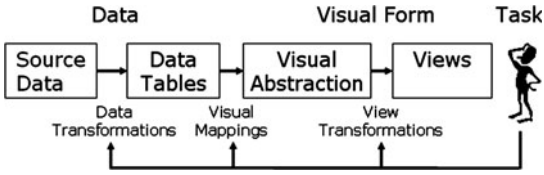


Fig. 7.1 Visual reference model [3]

Visualization techniques developed and discussed within the visualization community (e.g., [3–5]) can be leveraged to support variability management (e.g., [6–9]). By using such techniques, the expertise and experience of that community can be brought to bear on the complexity challenges that exist in that domain. Whereas most established variability management tools do not explicitly aim to utilize such techniques and expertise, recent research tools in that area are attempting to apply visualization concepts to their user interfaces.

2.3 Visualization Techniques

Fundamental visualization techniques and strategies that aim to support user cognition when dealing with large and complex data sets include *Focus+Context*, *Details on Demand*, *Degree of Interest*, *Color Encoding and Iconography*.

- *Focus+Context* describes the general ability to work at a focused level while maintaining the overall context within which you work. A number of techniques can be employed toward this goal such as *fisheye* (magnifying a specific area of a much larger display), *overview/outline windows* (providing a contextual understanding of a given display), and *distortion* (e.g., transparency). An extensive overview of these techniques is, for example, given in [3].
- *Details on Demand* refers to the facility whereby the stakeholder can choose to display additional detailed information at a point where this data would be useful. This point is decided by the user of the system. For instance, the ability to expand/collapse branches within a tree display, *incremental browsing* of such a tree and *filtering*, provides details on demand.
- *Degree of Interest* techniques highlight or expand relevant data with respect to the user's current point of interest. In particular, the degree of interest (as applied to certain parts of the data) can change while the user is navigating the data.
- *Color Encoding* and *Iconography* both serve to encode information visually and are used in conjunction with other techniques to provide additional data that can be identified through visual queries—identifying a visual pattern that will be used by a mental search strategy over a graphical visualization [3]. Examples include a green tick, red X, or a familiar icon.

3 Visualization Support for Software Variability

3.1 Representing Variability

In terms of how to represent and model variability, many SPL research approaches for variability management and product configuration focus on *features*, often represented by dedicated *feature models* (e.g., [10, 11]). *Feature models* usually describe available configuration options of an SPL in terms of "prominent or distinctive user-visible aspects, qualities, or characteristics" [11].

While viewing a product line as a collection of *features* has many advantages, there are some problems as well. Some of the problems include the difficulty in describing cross-cutting features and non-functional requirements, as well as the problems that arise in linking a feature to a concrete component (or set of components) that implement that feature.

3.2 Challenges and Approaches

There are numerous tasks to be performed by various stakeholders during the SPL engineering processes of domain and application engineering (cf. Sect. 3.3). Platform managers, domain engineers, product managers, application engineers, developers, and even customers all take on different roles in the process and require methodology and tool support that facilitates their specific tasks. In many of these cases, a feature model alone is either too detailed or not detailed enough. Using separate models allows different facets of the product line to be managed in a focused manner and supports stakeholder and task-specific representation and manipulation.

One approach to separating the different concerns of a software product line while providing relationships between various elements could be to describe the product line not only in terms of *features* but extend this description by taking *decisions* and *components* into consideration. A *decision model* would then capture a small number of high-level questions and provides an abstract, simplifying map onto *features*. The implementation of features by software or hardware components is then described by a *component model*.

Please note we use the concept of a *decision model* in the sense of a high-level *feature model* that sets the major context of the configuration by answering major questions such as if a particular product is "entry level" or if the product is planned to be introduced in a specific market (e.g., US, Japanese, or European market). In this sense, the *decision model* could be seen as containing the most important questions someone has to ask before configuring the more detailed and fine-grained *feature model*. This is different from other definitions of the term "decision model" in the product line literature, for example, the definition provided in [6].

These three models—the *decision*, *feature*, and *component* (DFC) *model*—can be used as a foundation to support variability visualization and product configuration. One characteristic of the DFC model is that the three underlying models are interrelated. For instance, making a *decision* might cause several *features* to become selected, which in turn requires a number of *components* to be implemented.

In the above approach, *decisions* provide a simplified high-level map onto *features* and can be used to abstract from details by asking a few major questions that are relevant for a particular stakeholder. A *component model* describes components that implement the *features*. Making a *decision* can involve the selection of multiple *features*, each of which may require or exclude sets of other *features*. These *features* in turn may require or exclude sets of *components*. Furthermore, a relationship itself between two *features* may be implemented by a *component*. More details of the underlying model are described in [7].

Visualization of the relationships within a *feature model* alone is challenging, and numerous approaches have been proposed, ranging from filtered lists (e.g., [6]) to graph-based views (e.g., [12]) to methods of only showing the relationships on demand (e.g., [7]). With multiple models in place, visualizing the relationships between each of them becomes even more difficult. Presentation and manipulation of the underlying data in the execution of specific tasks is impeded by their multilayered interrelationships. For example, as mentioned above, making a *decision* can involve the selection of multiple *features*, each of which may require or exclude sets of other *features* and *components*. In such scenarios, stakeholders need to be presented with the relevant data using appropriate techniques. This will enable them to understand the current state and the impact of various required changes. Stakeholders also need to be able to make such changes with ease.

3.3 Task Support

The task of configuring a complete *feature model* can be reduced to a sequence of configuration decisions on individual features. At a basic level, this involves the ability to either select or eliminate a *feature* from the product under derivation which, in turn, usually leads to the inclusion, exclusion, or configuration of related *components*. Additionally, the ability to select or eliminate *features* in groups based on higher-level requirements (*decisions*) is a fundamental task. Whereas these tasks may seem basic, it is the knowledge and understanding (cognition) of the stakeholder that allows these tasks to be performed correctly. Drawing on a variety of research that has been carried out (e.g., [1, 2, 8]), we outline a set of simple cognitive tasks that aim to support the activity of the primary task—namely, to decide which *features* should be included and which should be excluded:

1. Identify/locate a configuration *decision*
2. Understand the high-level impact of a *decision* inclusion (perception of scale and nature of the impact—implements/requires/excludes)
3. Identify/locate a specific *feature*
4. Identify a specific *feature's* context—parent *feature*, alternative/supporting *features*, and *sub-features*
5. Understand the high-level impact of a *feature* selection—a specific *feature's* constraints (requires/excludes relationships)
6. Identify the state of a *feature*—selected/eliminated and why

Visualization approaches can support these cognitive tasks by providing an interactive visual environment.

4 Visual Approaches and Implementations

As discussed, the comprehension and management of large sets of complex data relationships is the primary challenge when presenting variability data. Most approaches to date have utilized existing and well-known visual forms familiar to the software engineering community. The most prevalent of these is the ubiquitous "file explorer style" tree generally presented in the form of a static horizontal tree with expandable and collapsible branches. Recent work such as [13] has expanded on this visual form by introducing more dynamic tree structures and layouts. Other work (e.g., [14]) has focused on leveraging various techniques from the visualization community and utilizing alternative approaches not yet explored for this purpose.

When using visualization techniques for the handling and configuration of variability models, we have to address the cognitive tasks discussed earlier (see list in Sect. 3.3) with corresponding visual and interactive techniques. For instance, a tool environment has to provide interactive techniques to locate a *feature*, to understand a *feature's* configuration state, or to understand the impact of making a configuration decision.

Here, the resulting challenges are mostly related to the complexity and the scale of the models. In other words, the visualization and interactive technique must allow the stakeholder to handle large models and to focus on the relevant information, while abstracting from irrelevant details. This can be, for instance, achieved by techniques that allow to navigate on large models and to focus on elements for a particular task (e.g., a set of currently focused features) and related information (e.g., other elements in the model related to this feature set). A related challenge is that there are multiple ways to structure a model (e.g., which hierarchy to choose, how to modularize) and that the information structure that would be optimal for a particular task is not necessarily identical to the main structure of the model. Since a model is used for multiple tasks, visualizations and interactive techniques have to provide a means to adapt to different usage contexts and to change the focused aspects.

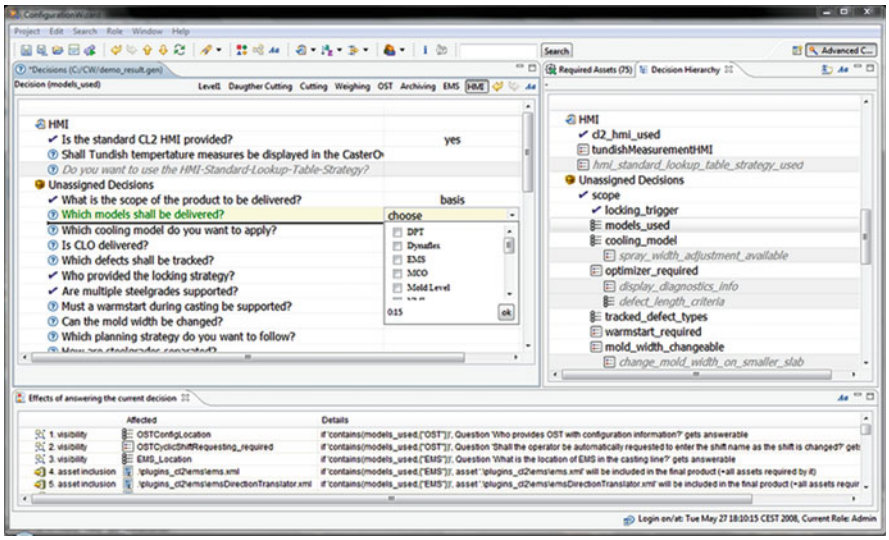


Fig. 7.2 DOPLER configuration wizard [6]

We will now look at particular approaches in more detail. In general, the approaches and techniques to variability visualization can be divided into three broad areas: two dimensional (2D), two and a half dimensional (2.5D), and three dimensional (3D). General characteristics of these approaches and implementation examples from a number of tools supporting variability management are described in the following subsections. The examples are taken from both research and commercially available tool suites.

4.1 2D Visualization

Using 2D approaches such as matrices and graphs to visualize feature models is the normative way to allow feature exploration and model manipulation [6, 8]. More recently, research tools are exploring the use of alternative tree layouts such as dynamic space trees [9] and radial trees [13]. In conjunction with this, the use of varying visualization techniques as described in Sect. 2 is being employed to aid stakeholder cognition in variability management tasks.

4.1.1 Examples

The *DOPLER* tool suite [6] provides decision-oriented variability management through a number of complimentary tools. One of these tools, the *Configuration Wizard*, provides capabilities for product customization, requirements elicitation, and configuration generation. Figure 7.2 shows the use of hierarchical tree structures to display a set of decisions on the left and the decision hierarchy on

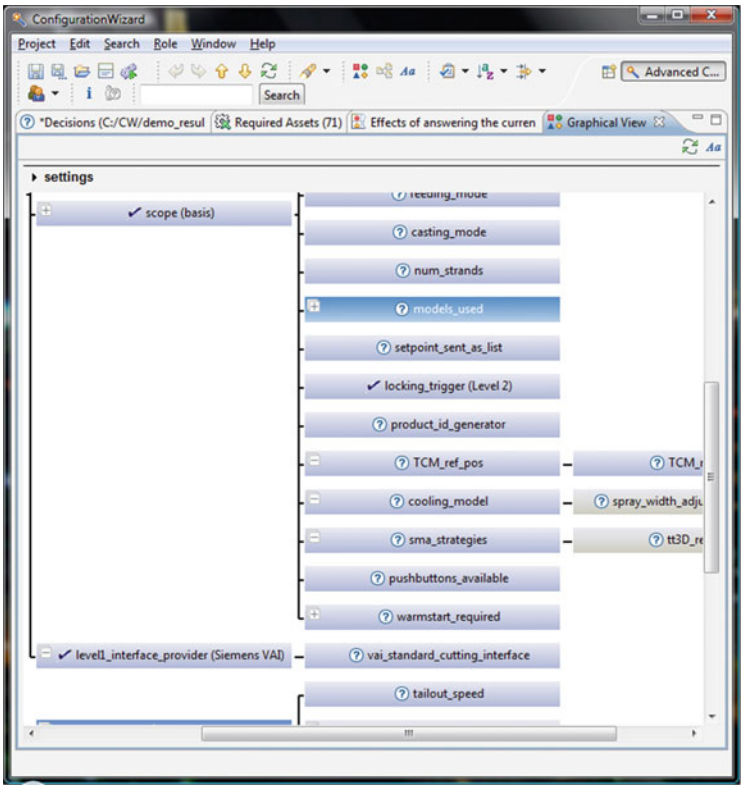


Fig. 7.3 DOPLER tree view [29]

the right. Figure 7.3 shows a more graphical representation of the tree structure. Both visualizations of the model utilize simple iconography to encode selected items (tick symbol icon) and items not yet configured (question mark icon).

The *pure::variants* tool suite [15, 16] is a commercially available product, which provides a set of integrated tools that support various phases of the software product line development and derivation process.

For creating and configuring a new software variant, the tool provides a *Configuration Editor* (see Fig. 7.4). This editor employs a hierarchical "file explorer style" horizontal tree to allow the browsing, selection, and de-selection of features according to their constraints. Iconography is extensively used to identify element types and feature state. Figure 7.5 shows a matrix visualization, which presents a view of the variants identifying the different features available in each of the variants.

The research tool *vivid* [14] primarily explores the use of more dynamic and interactive visualizations in order to provide cognitive support to stakeholders. Figure 7.6 shows a horizontal tree visualization, which represents a variant's *feature* configuration. The visualization allows the stakeholder to incrementally browse the tree structure automatically collapsing and expanding relevant branches

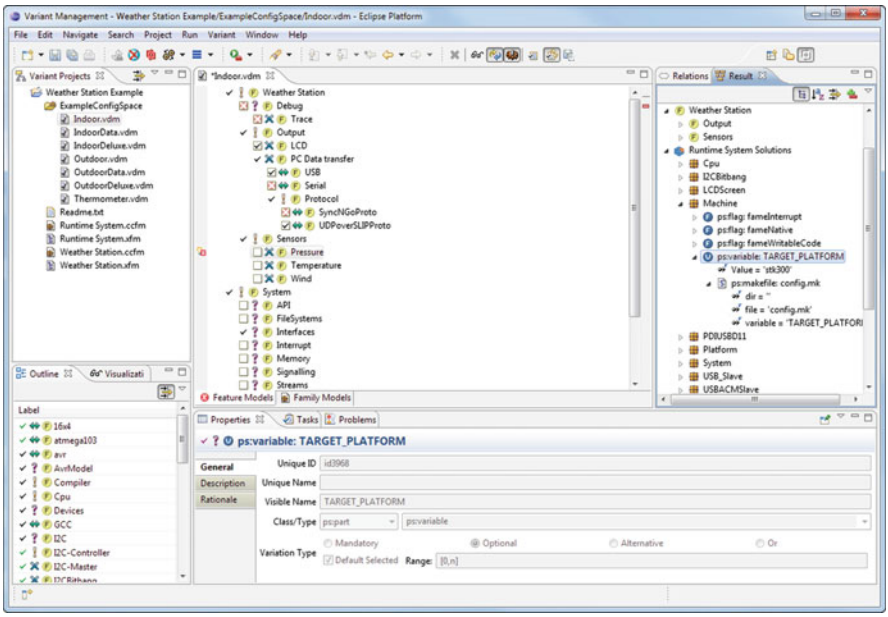


Fig. 7.4 pure::variants configuration editor [16]

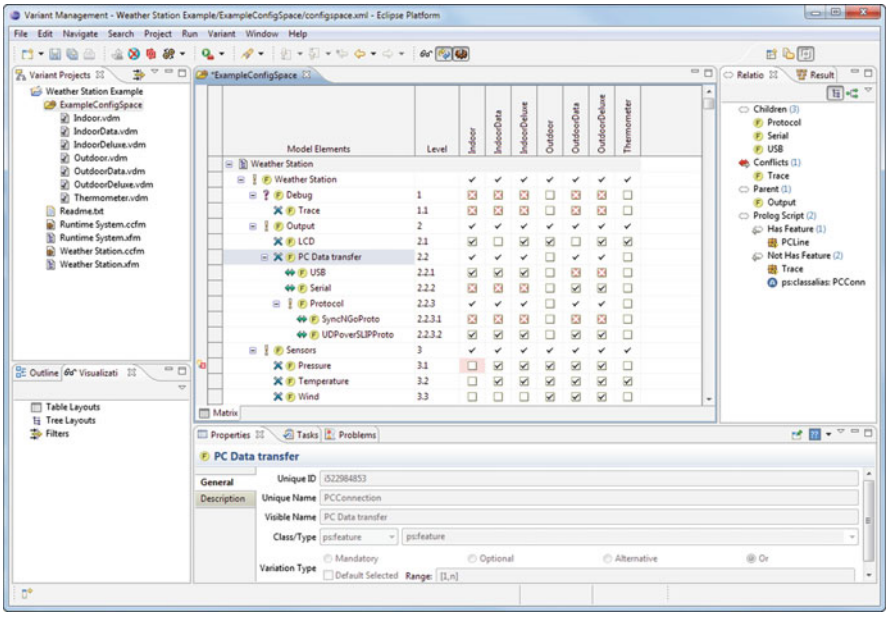


Fig. 7.5 pure::variants matrix view [16]

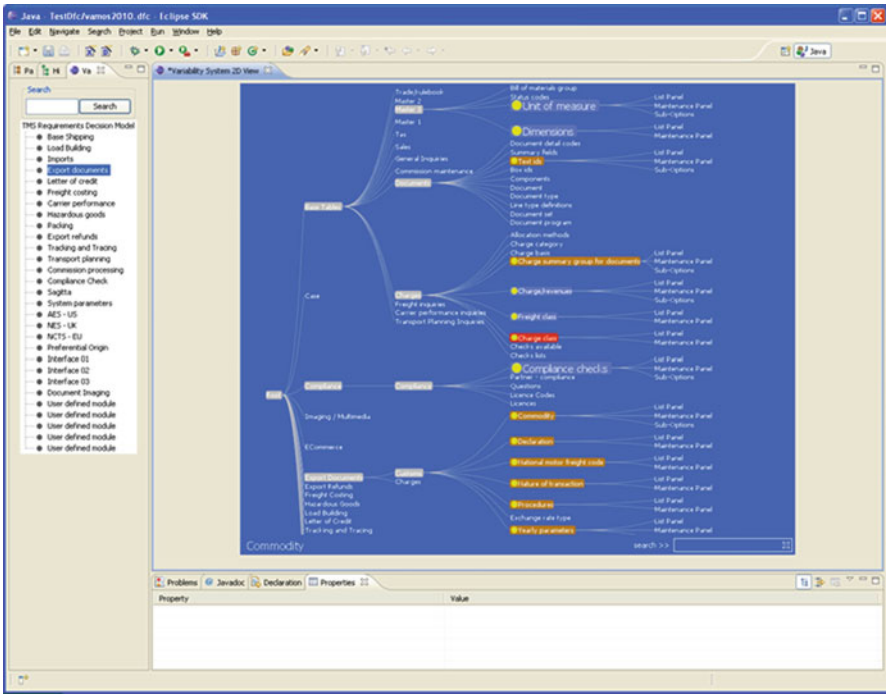


Fig. 7.6 vivid tree view [14]

as the stakeholder progresses. Animation of any visual changes to the display aids the stakeholder in understanding the navigation path. Color encoding allows immediate identification of the state of a particular feature.

The feature configuration tool *S2T2 Configurator* developed in earlier work by Botterweck et al. [17] provides techniques for the configuration of complex feature models and techniques for the joint visualization of feature and implementation models. In [18] the approach was extended by "Feature Flow Maps" to visualize product attributes, which result from configuration decisions, during product configuration (see Fig. 7.7). For instance, the width of the lines indicates the price of the product resulting from the current feature configuration. This visualization is updated incrementally while the feature configuration process is completed.

4.1.2 Advantages and Limitations

The advantages of using visual representations such as lists, "file explorer style" trees, and matrix tables are evident—they are generally familiar and intuitive to stakeholders. However, when the variability that exists within a software product line contains thousands of variation points, it becomes difficult to manage and cognitively challenging to navigate.

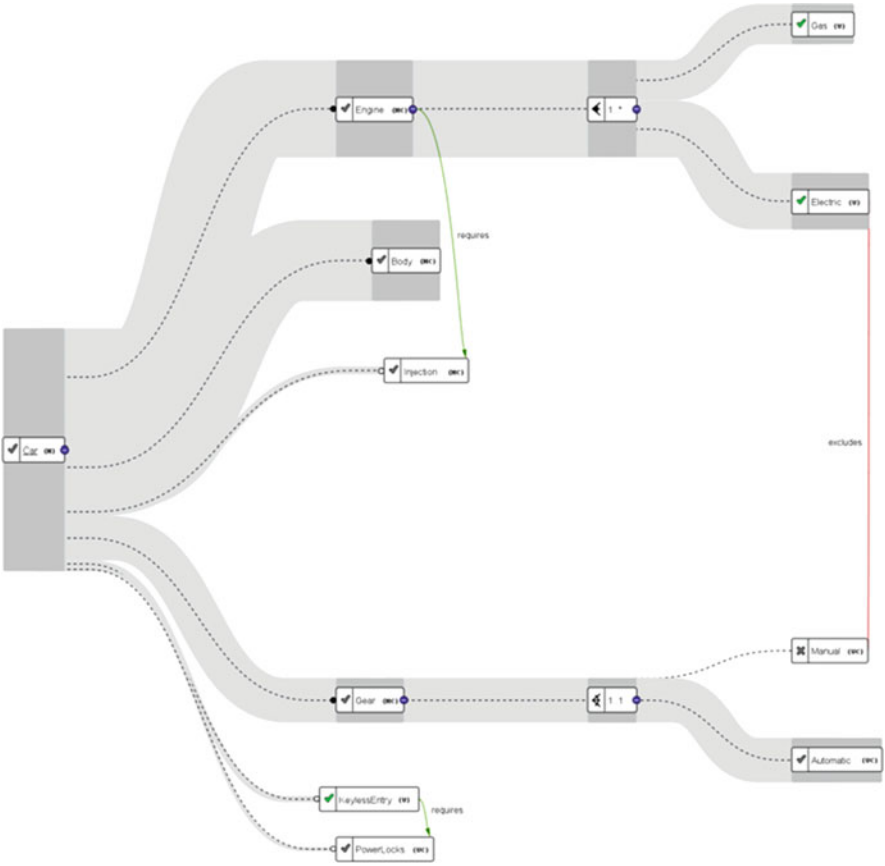


Fig. 7.7 Feature flow maps [18]

Using dynamic tree structures (as with *vivid*) and techniques such as animation, degree of interest, and distortion, visualizations can provide cognitive support to aid with such challenges. However, even with such additional aids, it can still be challenging to configure variability for very large product lines. In the next two sections, we show some examples of alternative visualizations being explored to further enhance the cognitive support provided to stakeholders.

4.2 2.5D Visualization

2.5D visualization techniques use 3D visual attributes in a 2D display [19]. For example, adding 3D attributes such as perspective (e.g., making certain objects smaller to indicate distance) and occlusion (e.g., overlapping objects to indicate layers) to a 2D display can be described as creating a 2.5D display. Work into the

Fig. 7.8 vivid 2.5D view [14]

employment of such techniques uses static 3D planes on which representations of features are presented in an animated interactive environment.

4.2.1 Example

Figure 7.8 shows a 2.5D visualization from the *vivid* prototype. To the left is a simple list of *decisions* (high-level grouping of features). When a selection is made within this supporting view, the main view displays the implementing features along with all features that are required or excluded by them.

The view consists of three stacked planes. Each plane provides a circular grouping of spheres. In the top plane, each sphere in the circle represents a grouping of features. When any one of those groupings in the top plane is selected (by mouse click), then all features that comprise that grouping are displayed in the middle plane in a similar circular format. In the lower plane, all related (required/excluded) features are displayed (for all features presented in the middle plane). The innermost circle on the lower plane identifies features that are directly related (required, excluded) to features in the middle plane. In order of ascending radii, each

subsequent circle in the lower plane represents the transitive relationships that exist; i.e., required features can further require and/or exclude other features.

By hovering the mouse over any sphere in any of the planes, a description of that element will be displayed in the center of the plane. When a sphere is selected in any plane, the circle on which it is presented will rotate so that sphere is brought to the front with its description displayed underneath. These functions aim to implement *Details on Demand*.

Each sphere acts as a representation of a specific feature. A sphere may be color encoded to visually identify its relationship to other variability artifacts (the feature implements a decision or the feature is required/excluded by another feature).

Multiple windows (and multiple planes) are employed to separate and distribute decisions, feature groupings, features, and relationships. Note that the lower plane displays all related features for all the implementing features in the middle plane. This allows an overview of the impact as a whole for this group of features. When a single implementing feature is selected in the middle plane, the circles in the lower plane rotate to ensure all related features are brought to the front while all other features in the plane are distorted (made transparent) in order to highlight the ones of interest. Animation is again used for all movements to preserve context.

4.2.2 Advantages and Limitations

The primary aim of the 2.5D approach is to increase the number of features and relationships that can be represented at any given time within a fixed screen space avoiding the need of panning and zooming across thousands of related on-screen elements. This is achieved by utilizing a depth dimension and providing animated movement and highlighting of relevant information to the foreground when required. *Focus+Context* is a key consideration here.

The presence of a fixed on-screen space may reveal a limitation of this visualization: there will be a point where a very large number of features and relationships may cause unwanted occlusion and selection difficulties. However, this situation would only occur with extremely large and/or complex feature models. Testing and usability studies are required to evaluate the effectiveness of this approach.

4.3 3D Visualization

Differing reports exist on the effectiveness of 3D visualizations to support software engineering but literature suggests that there is acceptance that it can be effective in specific instances (e.g., [20, 21]). Current work into the use of 3D primarily focuses on the visualization of relationships instead of the elements that they relate [9]. Relationships are visualized as objects in a 3D space whose coordinates identify elements within three different models, each model being mapped to one of the three axes.

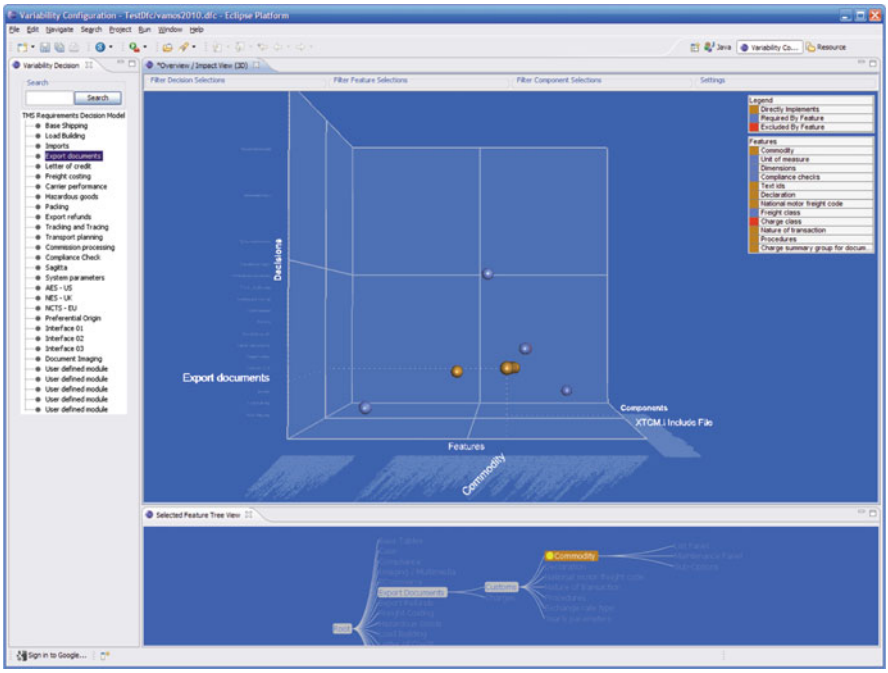


Fig. 7.9 vivid 3D view [14]

4.3.1 Example

Figure 7.9 presents a 3D view which attempts to provide a self-contained representation of all the three models introduced in Sect. 3.2 (*decisions*, *features*, and *components*) and their interrelationships. However, at any given time, only information of interest is displayed.

Here, as in the 2.5D approach, multiple windows are employed to distribute the information and provide the supporting *decision* view. The visualization consists of a 3D space containing *X*, *Y*, and *Z* axes. Sequential lists of *decisions, features*, and *components* are displayed along the *Y*-axis, *X*-axis, and *Z*-axis (moving away from the observer), respectively.

The key idea here is that a point within this 3D space identifies a relationship between all three models. In other words, a sphere plotted at a particular point will identify that the *feature* labeled at its *X*-coordinate implements the *decision* labeled at its *Y*-coordinate and is implemented by the *component* labeled at its *Z*-coordinate. In Fig. 7.9, the stakeholder has highlighted the sphere that represents the "Commodities" *feature*. However, in addition to this, by looking at the highlighted labels on the axes, we can see that it also represents the "Export Documents" *decision* that the *feature* implements and the "XTCM.I Include File" component that implements the *feature*.

Focus+Context and *Details on Demand* are the main techniques guiding this implementation. One goal is that all three models can be perceived to be represented through the listings on each axis. However, the details of any part of any model or its relationships are only displayed when required. For example, when a *decision* is selected, there may be a number of implementing *features*. For each implementing *feature*, a sphere is plotted in the 3D space as described above. Other *features* required or excluded by those implementing features are similarly plotted as spheres and are given a specific color encoding which allows a visual identification of the required or excluded relationship.

Pan & Zoom are combined with rotation to allow a full *world-in-hand* manipulation of the view in three dimensions allowing the stakeholder to position the view depending on the information of interest.

4.3.2 Advantages and Limitations

One advantage argued with this visualization is that it provides a perception of a software product line as a whole within a 3D configuration space while only presenting data that is relevant at a given time. Visually, a stakeholder is enabled to comprehend both the scale and nature of selecting a *decision*, *feature*, or *component*. As such, selecting a *decision* for implementation will require a set of implementing *features* but also require and exclude a large set of other *features*. The impact of such a *decision*, including its nature and magnitude, will be immediately evident allowing the stakeholder to further investigate the details of the impact.

As with the 2.5D visualization, the fixed on-screen space within a 3D configuration may also be a limitation as there is a point at which a very large number of features and relationships will cause unwanted occlusion and selection difficulties. However, this situation would only occur with extremely large and/or complex feature models. Again, testing and usability studies are required to evaluate the effectiveness of this approach.

5 Related Work

Traditionally, many approaches that support feature configuration as part of product line derivation use a hierarchical model. The visualization of hierarchical structures has been studied extensively in the visualization literature, including node-link techniques (e.g., [22]), space-filling techniques (e.g., Tree Maps [23]), and interactive techniques that help to cope with very large models such as *Focus+Context* (e.g., [24]).

Approaches focusing on multiple hierarchies are useful when visualizing the relationships between features and other models as discussed above. Robertson et al., for example, define polyarchies [25] as multiple hierarchies that share nodes.

They describe the visualization design and a software architecture for displaying polyarchy data from a set of hierarchical databases. They use animated transitions when switching between the related hierarchies to allow the user to keep context. Polyarchies are somewhat similar to the multiple related hierarchies found in some product line configuration approaches but lack the intra-model relations and the aspect of progressing configuration.

Moreover, a number of research tools which provide product line configuration capabilities and apply visualization techniques exist in literature. Some of these tools have already been discussed in the preceding section (e.g., [6, 14, 16, 18]). Another example is the research prototype *V-Visualize* developed by Sellier and Mannion [26], which visualizes configuration decisions with a force-directed layout.

Some approaches aim to visualize the variability in the artifacts which are influenced by this variability. For instance, Kästner et al. [27] used color encoding to indicate variability in program code.

In an earlier work, the authors presented Visit-FC, a configuration approach and tool that indicates the configuration state of features by visual clues [28].

6 Summary

The mechanisms by which software variability is presented to a stakeholder through visual representation and interactivity can have a substantial effect on how efficiently the stakeholder can perform their required management tasks. This becomes more and more evident as the size and complexity of the variability increases. Many of today's variability management tools use normative software engineering user interface techniques to present, and provide management of, that variability. For example, variability artifacts such as *decisions*, *features*, and *components* can be presented in one-dimensional lists, as elements in a two-dimensional matrix/spreadsheet, or as nodes in "file explorer" style trees that provide grouping and allow selection/elimination from a variant model.

As industrial-sized product lines grow to the order of many thousands of variation points, these traditional techniques tend to be limited in the cognitive support that they provide to stakeholders in relation to the performance of their tasks. Information visualization techniques have proven useful in enhancing cognition in numerous ways, and in recent work, these techniques are being employed with the aim of increasing the cognitive support provided. Visualization strategies such as *Focus+Context*, *Degree of Interest*, and *Details on Demand* in combination with techniques such as *Iconography*, *Color Encoding*, and *Distortion* are being utilized leveraging the work that has been carried out within the information visualization community. Implementations using dynamic space trees, radial graphs, and more explorative 2.5D and 3D techniques have been developed. Table 7.1 briefly summarizes advantages and limitations of approaches discussed in this chapter.

Table 7.1 Overview of approaches

	Example approaches	Advantages	Limitations
2D	DOPLER [6] pure::variants [15] S2T2 Configurator [18] vivid tree view [14]	"Explorer" interaction style is familiar to stakeholders	Limited on-screen space
2.5D	vivid 2.5D view [14]	Representation of a larger number of elements (features) in a limited space	Interaction requires training
3D	vivid 3D view [14]	Perception of product line as a whole Natural representation of scale	Interaction requires training

7 Outlook

There are a variety of commercial and research tools that provide support for variability management. Many of these are continuing in their development and some are actively exploring the use of more novel presentation and interactive techniques to improve their support for small- and large-scale variability projects. In the immediate future, the use of dynamic graphs and, in particular, the use of *degree of interest* trees seems to become more prevalent. There is also ongoing work into the use of 2.5D and 3D strategies which aims to leverage more of the visualization research that has been carried out to date.

References

1. Deelstra, S., Sinnema, M., Bosch, J.: Product derivation in software product families: a case study. J. Syst. Softw. **74**, 173–194 (2005)
2. Steger, M., Tischer, C., Boss, B., Müller, A., Pertler, O., Stolz, W., Ferber, S.: Introducing PLA at Bosch Gasoline Systems: experiences and practices. In: SPLC 2004, Boston, MA, pp. 34–50 (2004)
3. Card, S.K., Mackinlay, J.D., Shneiderman, B.: Readings in Information Visualisation: Using Vision to Think. Morgan Kaufmann, San Francisco, CA (1999)
4. Ware, C.: Information Visualisation: Perception for Design, 2nd edn. Morgan Kaufmann, San Francisco, CA (2004)
5. Diehl, S.: Software Visualization – Visualizing the Structure, Behaviour, and Evolution of Software. Springer, Heidelberg (2007)
6. Rabiser, R., Dhungana, D., Grünbacher, P.: Tool support for product derivation in large-scale product lines: a wizard-based approach. Presented at the 1st International Workshop on Visualisation in Software Product Line Engineering (ViSPLE 2007), Tokyo, Japan (2007)
7. Botterweck, G., Thiel, S., Nestor, D., Abid, S.B., Cawley, C.: Visual tool support for configuring and understanding software product lines. Presented at the 12th International Software Product Line Conference (SPLC08), Limerick, Ireland (2008)
8. Sinnema, M., Graaf, O. d., Bosch, J.: Tool support for COVAMOF. Presented at the Workshop on Software Variability Management for Product Derivation – Towards Tool Support (2004)

9. Cawley, C., Healy, P., Thiel, S., Botterweck, G.: Research tool to support feature configuration in software product lines. Presented at the 4th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS) Linz, Austria (2010)
10. Czarnecki, K., Helsen, S., Eisenecker, U.: Staged configuration using feature models. Presented at the Proceedings of the Third Software Product Line Conference, Boston, MA (2004)
11. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, S.: Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University (1990)
12. Sellier, D., Mannion, M.: Visualizing product line requirement selection decisions. Presented at the 1st International Workshop on Visualisation in Software Product Line Engineering (ViSPLE 2007), Tokyo, Japan (2007)
13. Rabiser, R.: Flexible and user-centered visualization support for product derivation. Presented at the 2nd International Workshop on Visualisation in Software Product Line Engineering (ViSPLE), Limerick, Ireland (2008)
14. Cawley, C., Healy, P., Botterweck, G.: A discussion of three visualisation approaches to providing cognitive support in variability management. Presented at the 2nd Conference on Software Technologies and Processes (STeP), Furtwangen, Germany (2010)
15. Beuche, D.: Modeling and building software product lines with pure::variants. In: 12th International Software Product Line Conference (SPLC 2008), Limerick, Ireland (2008)
16. pure-systems GmbH. Variant management with pure::variants. pure-systems GmbH (2006)
17. Botterweck, G., Janota, M., Schneeweiss, D.: A design of a configurable feature model configurator. In: Proceedings of the 3rd International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS 09), pp. 165–168 (2009)
18. Schneeweiss, D., Botterweck, G.: Using flow maps to visualize product attributes during feature configuration. In: VISPLE 2010, Jeju Island, Korea (2010)
19. Ware, C.: Designing with a 2 1/2D attitude. Inf. Des. J. **3**, 255–262 (2001)
20. Ali, J.: Cognitive support through visualization and focus specification for understanding large class libraries. J. Vis. Lang. Comput. **20**(1), 50–59 (2009)
21. Risden, K., Czerwinski, M.P., Munzner, T., Cook, D.B.: An initial examination of ease of use for 2D and 3D information visualizations of web content. Int. J. Hum. Comput. Stud. **53**(5), 695–714 (2000)
22. Walker, J.Q.: A node-positioning algorithm for general trees. Softw. Pract. Exp. **20**, 685–705 (1990)
23. Shneiderman, B.: Tree visualization with tree-maps: 2-d space-filling approach. ACM Trans. Graph. **11**, 92–99 (1992)
24. Cockburn, A., Karlson, A., Bederson, B.B.: A review of overview+detail, zooming, and focus +context interfaces. ACM Comput. Surv. **41**, 1–31 (2008)
25. Robertson, G., Cameron, K., Czerwinski, M., Robbins, D.: Polyarchy visualization: visualizing multiple intersecting hierarchies. In: ACM CHI 2002 Conference on Human Factors in Computing Systems, pp. 423–430 (2002)
26. Sellier, D., Mannion, M.: Visualizing product line requirement selection decisions. In: SPLC (2), pp. 109–118 (2007)
27. Kästner, C., Trujillo, S., Apel, S.: Visualizing software product line variabilities in source code. Presented at the VISPLE 2008, Limerick, Ireland (2008)
28. Botterweck, G., Thiel, S., Cawley, C., Nestor, D., Preussner, A.: Visual configuration in automotive software product lines. In: 2nd IEEE International Workshop on Software Engineering Challenges in Automotive Domain (SECAD 2008), held in conjunction with IEEE COMPSAC 2008, Turku, Finland (2008)
29. Rabiser, R.: Flexible and user-centered visualization support for product derivation. In: Proceedings of the 12th International Software Product Line Conference (SPLC 2008), Second Volume, 2nd International Workshop on Visualisation in Software Product Line Engineering (ViSPLE 2008), Limerick, Ireland, pp. 323–328. Lero (2008)

Chapter 8
Variability in the Software Product Line Life cycle

Kyo C. Kang, Hyesun Lee, and Jaejoon Lee

What you will learn in this chapter
- *Understand the software product line life cycle*
- *Understand the issues in variability management*

1 Introduction

Product line (PL) engineering is a software engineering paradigm, which guides organizations toward the development of products from core assets rather than the development of products one by one from scratch [1–3]. Two major activities of PL software engineering are core asset development (i.e., PL engineering) and product development (i.e., application engineering) using the core assets.

For the core asset development, PL requirements are essential inputs. These inputs, though critical, are not sufficient for PL asset development; a Marketing and Production Plan (MPP), which includes guidelines/plans on what features are to be packaged in products, how these features will be delivered to customers, and how the products will evolve in the future. Therefore, it is essential to include a marketing perspective into the PL variability analysis and explores requirements and design issues from the marketing perspective [4, 5]. With an MPP, reuse is not opportunistic; it is carefully planned for specific product and market(s).

Once commonalities and variabilities (C&V) of market needs and their requirements are analyzed and modeled, this information is used to develop software assets, i.e., architectures and components, with appropriate variation points

K.C. Kang (✉) • H. Lee
Pohang University of Science and Technology (POSTECH), Pohang, Republic of Korea
e-mail: kck@postech.ac.kr; compial@postech.ac.kr

J. Lee
School of Computing and Communications, Lancaster University, Lancaster, UK
e-mail: j.lee3@lancaster.ac.uk

R. Capilla et al. (eds.), *Systems and Software Variability Management*,
DOI 10.1007/978-3-642-36583-6_8,

and variants. Once software assets have been developed for a PL, the product development phase involves adaptation and instantiation of these assets for a product. (Asset management is an ongoing process, which includes C&V analysis, and reengineering and refactoring of software assets.)

One of the most difficult and critical tasks in product line engineering is variability management. Various product line lifecycle products, including models, architectures, and components, have C&Vs, which are related vertically among elements within each lifecycle product and horizontally across different lifecycle products. C&Vs must be explored and modeled thoroughly and their consistencies must be maintained while PL evolves. As lifecycle products and their C&Vs are related vertically and horizontally, it is very difficult and also costly to manage and maintain their consistency.

Variability management from the perspective of MPP is the key aspect for managing variabilities of assets because assets are instantiated, adapted, and integrated to support MPP based on market needs. "Features" are abstractions of capabilities or functions that the customer wants/needs. Therefore, feature is the unit of delivery and also the unit of configuration and variability management

In this chapter, we explore various issues of C&V across the entire PL life cycle. As a starting point, Sect. 2 describes PL engineering activities and their inputs/outputs. The elements of MPP are explained and illustrated in Sect. 3 using an elevator controller example described in the box below, and Sect. 4 includes a discussion on how product line "problems" are modeled. The solution space modeling is included in Sect. 5, followed in Sect. 6 by a discussion of how product line artifacts such as architecture and components are designed based on the solution space models. Sections 7 and 8 include a discussion and conclude this chapter, respectively. It should be noted that the Feature-Oriented Reuse Method (FORM) has been used throughout the chapter for the purpose of illustration of various issues in product line variability management.

2 PL Lifecycle Activities: An Overview

PL engineering consists of two major engineering processes: PL asset development and product/application development. (See Fig. 8.1 for activities and their relationships.) The PL asset development consists of activities for analyzing PL problems (analyzing market needs, developing a marketing and product plan (MPP)); modeling C&V of the PL problems (goals aimed to be achieved by the products, product usage contexts, required quality attributes, etc.); solution modeling which includes exploration and modeling capabilities required by PL products and modeling PL requirements, exploring and modeling important design decisions that have significant quality implications, including domain technologies, COTS, external devices, etc.; and developing PL asset architectures and components based on the analysis results. The product development phase consists of a number of activities for analyzing product goals and usage contexts, analyzing product requirements and configuring the product (i.e., variability instantiation,

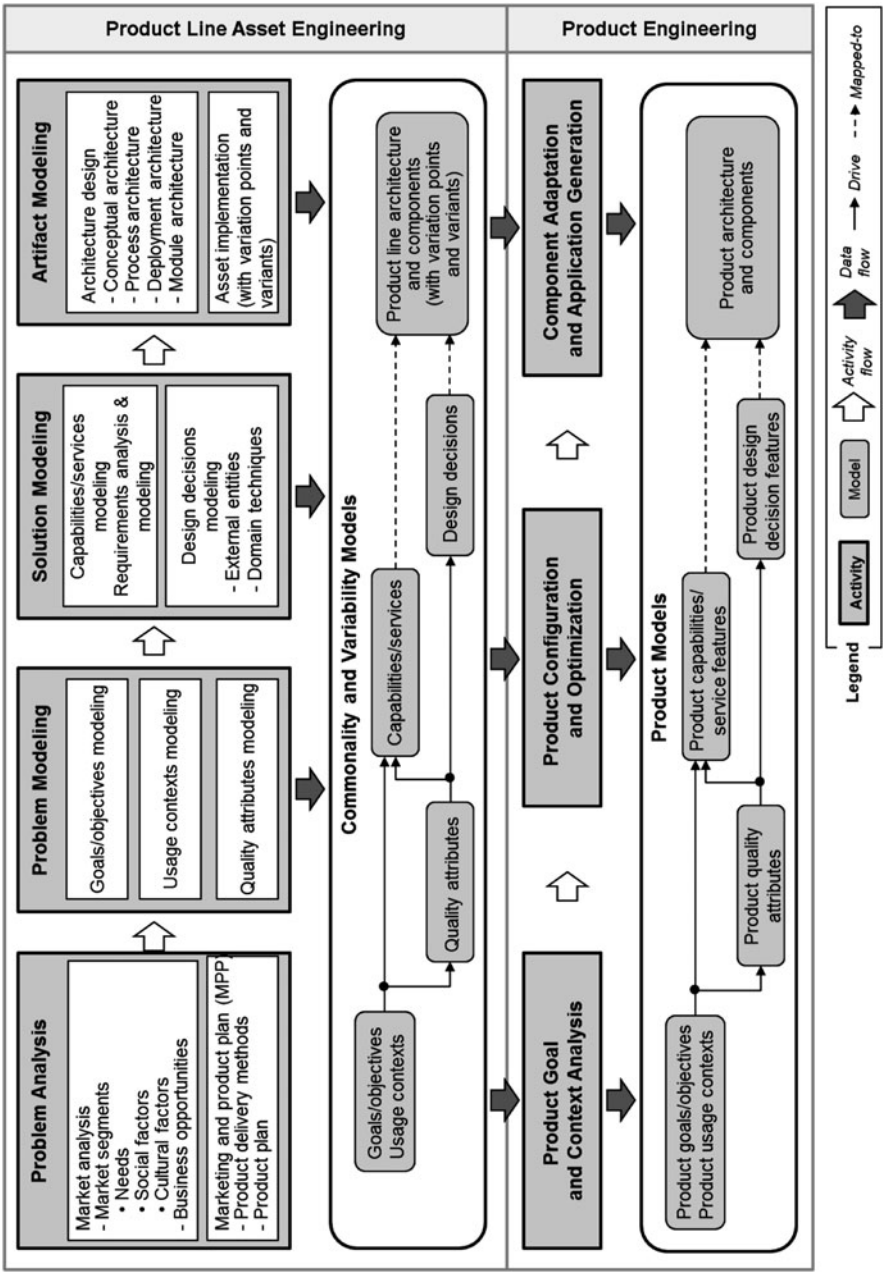


Fig. 8.1 PL engineering process and variability analyses

i.e., feature selection), selecting an appropriate architecture and components, and making necessary adaptations to the selected architecture and components, and generating/developing the product.

Elevator Example.

Elevator control systems are very familiar to everyone and it is easy to understand their behavior from an end user's perspective. However, the internal control and complex interfaces with external devices make it hard to build reliable and reusable software. Elevator products can largely be classified into passenger elevator or freight elevator depending on their main purpose of use. Since passenger elevators have a goal of carrying passengers comfortably, services or devices for comfortable ride are required. Of passenger elevators, those used in hospitals require more stringent floor leveling requirements and a low speed profile for patients on a wheelchair, while elevators in office buildings are less stringent on floor leveling but require a high speed profile. On the other hand, freight elevators do not necessarily require comfortable ride. They instead focus on carrying heavy loads safely, which requires special operational functions and devices for handling heavy loads. As can be seen, there is a wide range of quality requirements, which requires us to select various techniques, devices, and algorithms.

Analyzing the targeted markets and developing an MPP is the starting point for launching a PL. The diversity of market needs drives the development of a family of products, which is reflected in a marketing plan and product plan (MPP). Therefore, MPP serves as a key driver in PL asset development and variability management.

In C&V modeling, product features from MPP, which include functional (e.g., capabilities, services) and nonfunctional (e.g., product goals, product usage contexts, quality attributes) features, are organized into an initial C&V model (We use feature model for illustration. Other modeling techniques such as decision modeling may also be used.) which is refined through product line requirements analysis and then extended further with design features (i.e., operating environment, domain technology, and implementation technique features) as we follow the asset engineering activities. During PL requirements analysis [6], we elicit and organize PL requirements in terms of a PL use case model (variability expressed in terms of <extends> and <includes> stereotypes) and a PL object model, which are used in architecture modeling and component development.

The conceptual architecture design activity allocates features to architectural components and specifies data and control dependencies between architectural components. The result is a "conceptual architecture.[1] " A design object model

[1] The conceptual architecture describes a system in terms of abstract, high-level components and relationships between them.

must be developed based on the conceptual architecture, feature model, PL requirements, and other information such as commercial off-the-shelf (COTS) components and design patterns [7] that are relevant to the PL.

The conceptual architecture is refined into "process and deployment architectures[2]" by allocating components to concurrent processes and network nodes, considering whether to replicate each process or not, and defining methods of interactions between processes. Then the component design activity refines the process and deployment architectures into concrete components by using the design object model.

The MPP provides quality attributes for the architecture design and refinement. For example, the user profile information (i.e., the skill levels and computing environments of potential users) in the MPP is useful in determining the quality attributes (i.e., usability, scalability, etc.) required for the architecture design of the products targeted for each market segment. Also, the MPP is used in exploring design alternatives in the component design for feature delivery method support, feature interaction problem[3] resolution, etc.

The PL engineering processes are iterative and incremental, and repeat until we come to a design that has enough details for implementation. (The arrows in Fig. 8.1 show data flows, i.e., work products generated and used by each activity.) Details of each PL asset development activity are illustrated in the following section.

3 Problem Analysis

Problem analysis starts by exploring the market and developing a marketing and product plan for a product line. This activity initiates PL asset development; an MPP sets a specific context for PL analysis and reuse exploration in the PL. Products developed without consideration of user's needs and their capabilities and how they will be marketed will not be "sold." Functionality alone does not sell. Products must be configurable to meet the needs of and services required by users. Variation points embedded into the PL assets and mechanisms used to implement variants must be able to support the marketing and product plan effectively and efficiently.

3.1 Elements of MPP

The first part of an MPP is a marketing plan, which includes a market analysis with a C&V assessment of a market, and a marketing strategy for realizing business opportunities in the market (see the left portion of Fig. 8.2). The market analysis

[2] The process architecture represents a concurrency structure in terms of concurrent processes (or tasks) to which functional elements are allocated; the deployment architecture shows an allocation of processes to hardware resources.

[3] When products are developed with integration of components implementing various features, these features may interact with each other. The problem of unexpected side effects when a feature is added to a set of features is generally known as the feature interaction problem.

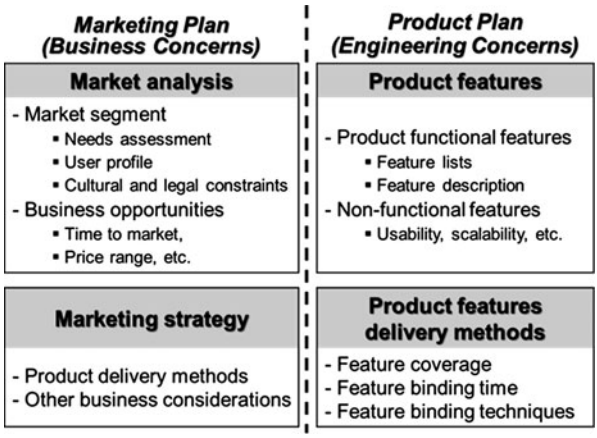


Fig. 8.2 Elements of a marketing and product plan

includes, for each market segment, a C&V assessment of needs, an analysis of potential users, cultural and legal constraints, the time to market, and the price range. Identifying market segments and needs of each market segment may be an iterative process. For example, we may start with the entire elevator systems market but then quickly realize that there are features specific only to elevators in office buildings or to freight elevators. We then focus on analyzing features peculiar to each market segment.

The marketing strategy initially includes an outline of product delivery methods with the information on how products will be delivered to customers and other business considerations. For example, customers may start with products with only core features but then add other features incrementally. Incremental features may be added to the products over the network. If this is the marketing strategy, the product delivery method and variability management method must be able to support this strategy.

Once the marketing plan part of the MPP has been defined, it is important to spend some effort to identify the characteristics of products in a PL in terms of features and develop a plan for incorporating the features. A product plan includes product features and product feature delivery methods (see the right portion of Fig. 8.2).

Product features are distinctive characteristics of products of a PL. They are largely classified into functional and nonfunctional features. Functional features include services, which are often considered marketable units or units of increment in a PL, or operations, which are internal functions of products that are needed to provide services. For example, automatic control, manual control, and VIP service features in elevator PL are functional features. Nonfunctional features include end-user visible application characteristics that cannot be identified in terms of services or operations, but as presentation, capacity, quality attribute, usage, cost, etc. For example, safety, reliability, and fault tolerance are important quality attributes for the elevator PL.

A product feature delivery method defines how product features will be "sold" or delivered to the customers and users, and how they will be installed and maintained. The product delivery method must support the marketing strategy. Some features may be prepackaged in all products as "standard features," and others may be selected at the product negotiation time. There may yet be other features that are specific to a customer and are built into the custom-made product.

3.2 Marketing and Product Planning: A ECS Product Line Example

Market segments affect engineering of assets for the ECS product line. For the purpose of illustration, we assume that the company's marketing strategy is to target the high-rise office building market with high-end products, the general hospital market with mid-level products, and the apartment market with low-end products, based on an analysis of market competitiveness.

Table 8.1 provides/is an example of MPP for the ECS PL. (The actual MPP should be a document describing all elements identified in Sect. 3.1; a simplified example is shown in this paper for illustration of the concept.) The user/maintainer profiles for each market segment are as follows:

- High-end market segment of high-rise office building uses: Dedicated engineers with computer science background are available for maintenance. The computing environment is distributed over the network and maintainers can access the system remotely.
- Mid-level market segment of general hospital uses: No computer skill is assumed for maintainer and ECS software should run on PCs they already have.
- Low-end market segment of household uses: No computer skill is assumed for maintainer and ECS software should run on PCs they already have.

The laws and the cultural traits of each country must also be identified in the marketing plan. For example, standards related to earthquake resilience, fire standards, electrical wiring rule, etc., may vary from country to country. Also, the safety and reliability requirements (e.g., the doors must remain open in case of a fire event) may be different.

Since the high-rise office building ECS has many customer-specific requirements, the "feature selection" method is chosen to adapt and integrate the requirements at the product delivery time. For the general hospital ECS and the apartment ECS, "prepackaged" method with a user-friendly interface is adopted for the users who do not have any computer knowledge.

The product delivery methods are refined to product feature delivery methods, which can be looked at from what features are allowed (feature coverage), when they are incorporated into the product (feature binding time: product build time, product delivery/installation time, or runtime), and how the feature incorporation is made (feature binding techniques: framework, template, load-table, plug-ins, etc.) [8–10]. For example, the apartment ECS have a closed set of features and the feature

Table 8.1 A marketing and product plan example for an ECS PL

Marketing and product plan for ECS product line			
Market segments	High-rise office building	General hospital	Apartment
User/ maintainer profile	Dedicated engineers with computer science backgrounds	No computer knowledge is assumed	No computer knowledge is assumed
Product delivery method	Feature selection from a predefined set of features (feature selection method)	Prepackaged method	Prepackaged method
Legal constraints	Because elevator is part of a building, it must comply with standards relating to earthquake resilience, fire standards, electrical wiring rule, etc.		
Product features	Call handling, indication handling, door control, motor control, hall call cancellation, car call cancellation, emergency driving, group management, etc.	Call handling, indication handling, door control, motor control, hall call cancellation, car call cancellation, emergency driving, hospital emergency, etc.	Call handling, indication handling, door control, motor control, etc.
Quality attributes	Door safety, usability, efficiency, emergency safety	Door safety, usability, smooth and comfortable run, position accuracy, emergency safety	Door safety, usability
Product feature binding time	Product delivery time	Product build time	Product build time

binding occurs at product build time in order to support the prepackaged product delivery method. For the high-rise office building ECS, however, customers may select any features from a predefined list, and feature binding occurs at product installation time using a load table that contains parameter values for instantiation.

4 Problem Modeling

Features in the problem model represent the concrete context of products of a product line, i.e., external forces that drive selection of specific architectures, implementation algorithms, or implementation techniques; these features are important to understand real world problems that the product line should address. That is, the problem space captures the information of:

- Why is the product line required in the market?
- When is a certain product configuration used?
- What are the expected qualities of the product line?

The answers to these questions should be captured in an exploitable form so that we can establish clear traceability, not starting from product functional features, but from the context of a product line.

The problem space can be divided into three disjoint viewpoints: goal/objective, usage context, and quality attribute.

4.1 Goals/Objectives and Usage Contexts Modeling

Organizing goal/objective features and usage context features from real world problems of a product line initiates the modeling process. Goal/objective features specify the boundary and scope of the product line and usage context features set specific contexts for the product line. Figure 8.3 shows an example of them.

Our experience shows that we could explore the following areas while analyzing goals/objectives of a product line; (1) real world problems that the product line addresses, (2) other product lines that address the problems in a similar but different ways, (3) potential benefits that can be accrued from other product lines, and (4) key nonfunctional properties of the product line (e.g., comfortability, efficiency, etc.) that should be achieved. This information is used as an important input for the usage context feature modeling.

Next we analyze the usage context of a product line. The usage context features are to capture various usage contexts of products of the product line. For example, we identified usage context features *Passenger* and *Freight* and each usage context defines the objects to be carried by an elevator. (See *Carrying Object* in Fig. 8.3.) If we select *Passenger*, the scope of the ECS product line is restricted to *Passenger-ECS* products and irrelevant features are removed from selectable ones. If we look further into the product usage, we may be able to identify more specific product usages where different quality attributes and/or functions are needed. For example, an elevator in a hospital may carry patients on wheelchairs, and floor leveling and emergency button are important features. (See *Hospital* in Fig. 8.3 and quality features in Table 8.2.)

With the identified goal/objective and usage context features, the next activity is to analyze quality attribute features.

4.2 Quality Attributes Modeling

In this activity, we analyze quality requirements of a product line and model them as quality attribute features. We can use software requirements specification (SRS), quality requirements document, etc., as inputs of this activity. Suppose, for

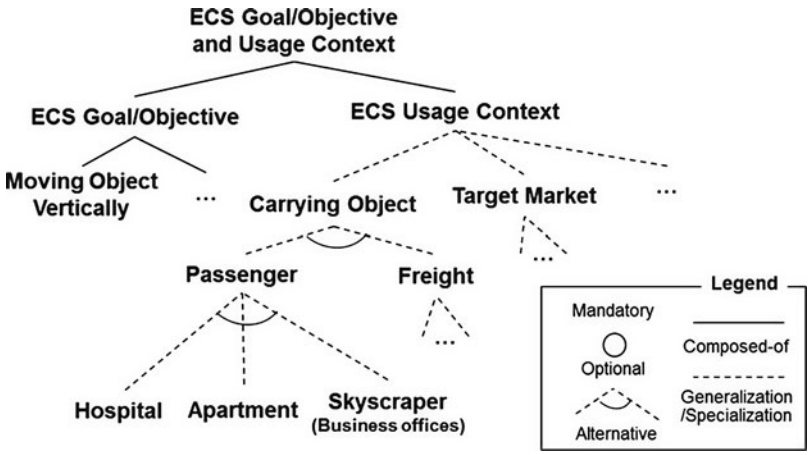


Fig. 8.3 A goal/objective and usage context feature model of the ECS PL

Table 8.2 Relationship between goal/objective, usage context, and quality attribute features of the ECS PL

Quality attribute features Goal/objective and usage context features	Door safety	Usability	Smooth and comfortable run	Position accuracy
{Passenger}	V	V	O	O
{Passenger, hospital}	V	V	V	V
{Passenger, skyscraper}	V	V	O	O

V (required), O (selectable)

example, that one of the quality requirements of ECS is: "If the doors of a cage detect a certain level of friction when the doors are closing, the ECS should open them immediately." From this statement, we can identify the quality attribute feature *Door Safety*. Figure 8.4 shows an example of quality attribute feature model.

Goal/objective and usage context features that we identified in the previous activity have specific quality implications. If a set of goal/objective and usage context features are selected for a product configuration, these features require a certain set of quality attribute features for the product configuration (i.e., the set of selected goal/objective and usage context features *drive* the quality attribute features). We can represent this relation explicitly using a table. For example, in Table 8.2, when we select the usage context feature *Passenger*, the quality attribute features *Door Safety* and *Usability* should be included in a product configuration; this is because user safety and convenience should be guaranteed for passenger ECS products. (See the second row in Table 8.2.) If the usage context feature *Hospital* (which is a specialization of *Passenger*) is selected, the additional quality attribute features *Smooth and Comfortable Run* and *Position Accuracy* should be included in

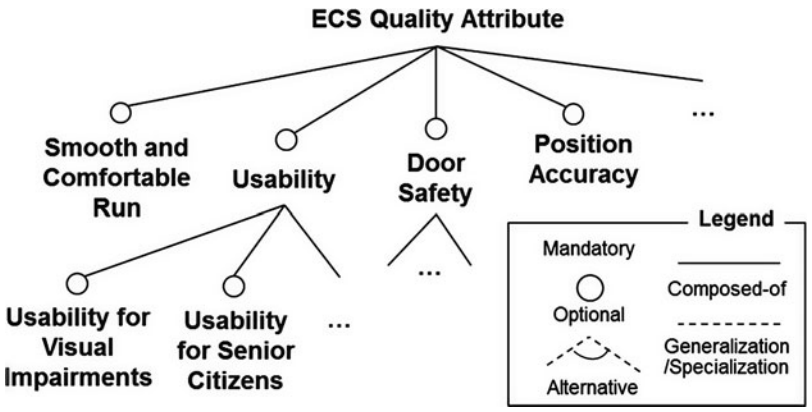


Fig. 8.4 A quality attribute feature model of the ECS PL

a product configuration for moving patients on wheelchairs or beds safely. (See the third row in Table 8.2.)

The goal/objective, usage context, and quality attribute features capture the problem space of a product line. In the next activity, we analyze the solution space.

5 Solution Modeling

5.1 Capabilities/Services and Design Decisions Modeling

We analyze the solution space to satisfy requirements captured by the problem space features. The solution space includes features of the capability/service (e.g., *driving services*, *operations*, etc.) and design decision viewpoints (e.g., *position control*, *speed profile*, etc.). Figure 8.5 shows an example of solution space features.

After we finish feature modeling for the problem and solution spaces, we establish mappings between the problem and solution space feature models. Details are in [11].

5.2 Relationships Between Problem Space Features and Solution Space Features

Goal/objective and usage context features may require specific capability features. We can capture these relationships using a table. In Table 8.3, for example, the usage context feature *Hospital* requires context-specific functional requirements such as "when a patient triggers an emergency alarm, the ECS should call nurses/doctors and stop the elevator in the nearest floor." Therefore, when we select the

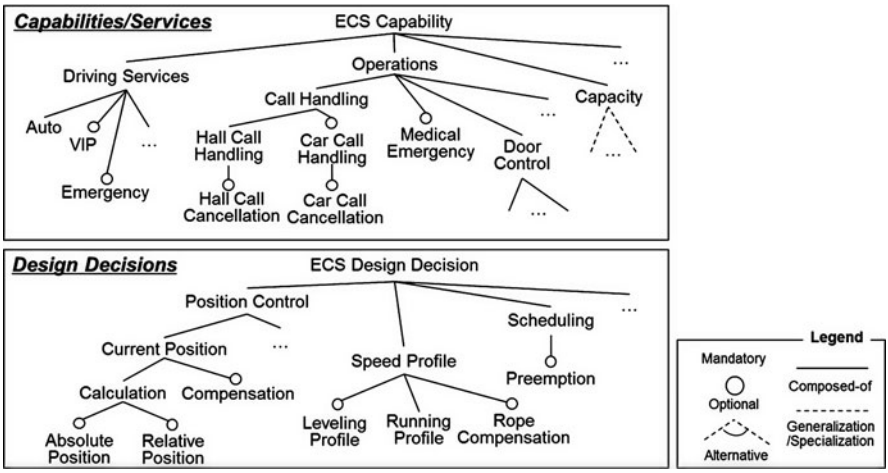


Fig. 8.5 A solution feature model of the ECS PL

Table 8.3 Relationship between goal/objective, usage context, and capability/service features of the ECS PL

Capability/service features Goal/objective and usage context features	Medical emergency	Double-deck	Door hold	Low speed (about 60 m/min)	Large capacity for car (about 3,000–5,000 kg)
{Passenger}	O	O	O	O	X
{Passenger, hospital}	V	O	V	O	X
{Passenger, skyscraper}	O	V	O	X	X

V (required), X (excluded), O (selectable)

usage context feature *Hospital*, the required capability features (e.g., *medical emergency*) must be included in a product configuration and irrelevant features (e.g., *large capacity for car*) must be removed from selectable features. (See the third row in Table 8.3.)

For each quality attribute feature, we should also establish a mapping to the solution space. A capability or a design feature may work for or against each quality attribute feature, and we capture the "strength" of this relation using qualitative measures such as *strongly support* (++), *weakly support* (+), *hurt* (−), and *break* (−−) [12]. This relation is represented using a table as shown in Table 8.4: the table shows relationships between quality attribute features and solution space features in the ECS product line. The design decision feature *Absolute Position* strongly (i.e., ++) *supports* the quality attribute feature *Position Accuracy* and *hurts* (i.e., −) the quality attribute feature *Low Cost*, whereas the design decision feature *Relative Position* weakly (i.e., +) *supports* both *Position Accuracy* and *Low Cost*. If the quality attribute feature *Position Accuracy* has the highest priority for an ECS product, both of the

Table 8.4 Relationship between quality attribute and solution space features of the ECS PL

Quality attribute features Solution space features		Position accuracy	Low cost
Calculation	Absolute position (optional)	++	−
	Relative position (optional)	+	+
Compensation of current position		+	−

++ (strongly support), + (weakly support), − (hurt), −− (break)

design decision features *Absolute Position* and *Relative Position* may be selected, although *Absolute Position hurts Low Cost*. However, if the quality attribute features *Position Accuracy* and *Low Cost* have the same priority, then only *Relative Position* can be selected.

5.3 PL Requirements Analysis

During the PL requirements analysis, functionalities that PL products must provide are captured using a set of models such as a use case model, an object model, etc. [6] A use case model defines interactions between the user and the system; an object model defines allocation of responsibilities. Other models may also be included depending on the domain that a PL is in. Based on this information, a PL component design provides a realization of common functions that can be used across the PL products.

Figure 8.6 shows a use case model of the ECS PL. Each of use case may embed optional/alternative features in the solution space. When we establish the mapping relation, we propose two different types of variability embedded in domain objects, and they are external and internal variability types. In Fig. 8.6, a UML stereotype-based notation introduced by Lee et al. [13] is used for expressing these two types. Other UML-based variability mechanisms such as [14] may also be used.

The external-variability type denotes that an associated use case is entirely related to the specified feature and inclusion/exclusion of the use case depends on the selection result of the feature. For example, '<<●*Car Call Handling*>>' in the *Process Car Calls* use case in Fig. 8.6 indicates that the inclusion of the *Process Car Call* use case depends on the selection result of the optional feature *Car Call Handling*. In other words, if *Car Call Handling* is selected for a product configuration, the *Process Car Call* use case should be included in a product; otherwise it should be removed from the product configuration.

The internal variability type means that a corresponding feature is partially related to the associated use case and specifics on how the feature is related to the use case can be found by looking inside of the use case. For example, "<<○*Hall Call Cancellation*>>" in the *Process Hall Calls* use case in Fig. 8.6 means that *Process Hall Calls* is related to the optional feature *Hall Call Cancellation*, and if we select or not select *Hall Call Cancellation*, the internal interaction(s) of the *Process Hall Calls* use case changes.

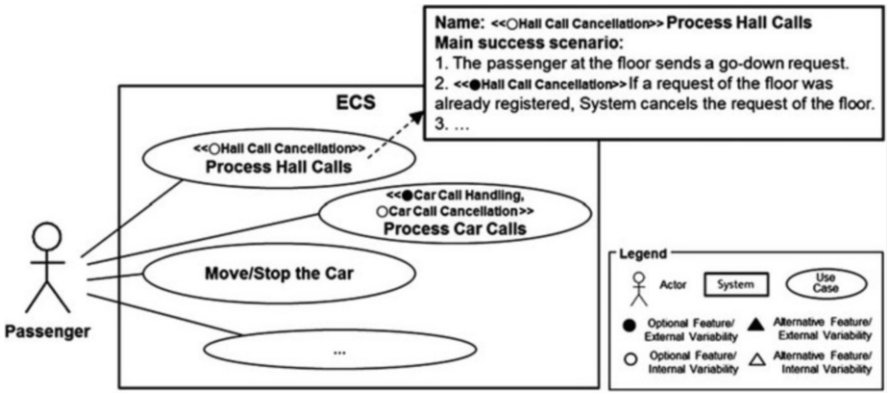


Fig. 8.6 A product line use case model of the ECS PL

In object modeling, we first identify domain objects, which are candidate-reusable objects derived from the feature models based on the guidelines proposed in [11]. Then variabilities captured as optional/alternative features in the solution space are embedded into the object model.

Figure 8.7 shows an example of mappings between solution space features and domain objects. We also use the external and internal variability types. For example, "<<●*Car Call Handling*>>" in the *Car Call Handler* domain object in Fig. 8.7 indicates that if *Car Call Handling* is selected for a product configuration, the *Car Call Handler* object should be included in a product; otherwise it should be removed from the product configuration. For another example, '<<○*Car Call Cancellation*>>' in the *Car Call Handler* domain object in Fig. 8.7 means that if we select or unselect *Car Call Cancellation*, the internal implementation of the *Car Call Handler* object changes.

Once the feature model is refined and PL requirement models are developed, this information is used to refine the MPP as described in Fig. 8.1. Since the initial MPP only contains delivery methods for functional and nonfunctional features, product feature delivery methods for design features (e.g., operating environment features) should be developed during the refinement.

6 Artifact Modeling

After we establish the problem and solution space feature models and their relations, the next activity is to develop product line artifacts based on the feature models. During this activity, the identified solution space features are implemented as product line artifacts including product line architectures and asset implementation.

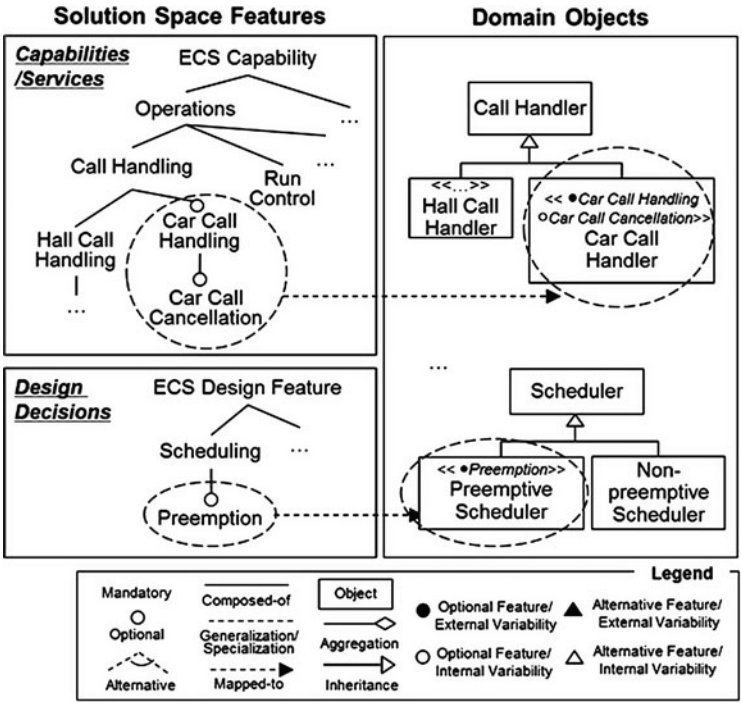


Fig. 8.7 Solution space features and domain objects mappings of the ECS PL

6.1 Architecture Design

In FORM [15], architecture design starts with identifying high-level conceptual components and specifying data and control dependencies among them. During the architecture design activity, the MPP is used as a key design driver. For example, the conceptual architecture for the high-rise office building ECS (see the conceptual architecture in Fig. 8.8.) consists of three major components (i.e., *Call Handler*, *Cage Operation Manager*, and *Safety Manager*) and the *Safety Manager* component is added to meet the quality requirement *Emergency Safety*. The *Safety Manager* monitors various safety related sensors (e.g., smoke sensor) and when an emergency situation (e.g., fire, earthquake) is detected, it notifies users via *Safety Supervision System* and sends the emergency status to the *Call Handler* and the *Cage Operation Manager*. The *Call Handler* receives call requests and schedules floors to visit and the *Cage Operation Manager* controls various external objects (e.g., door, motor) in elevator cars. When they receive the emergency status, they follow predefined emergency strategy.

The next step is to refine the conceptual architecture into process and deployment architectures. The bottom portion of Fig. 8.8 shows the process architecture for the *Call Handler* component of the conceptual architecture. During the

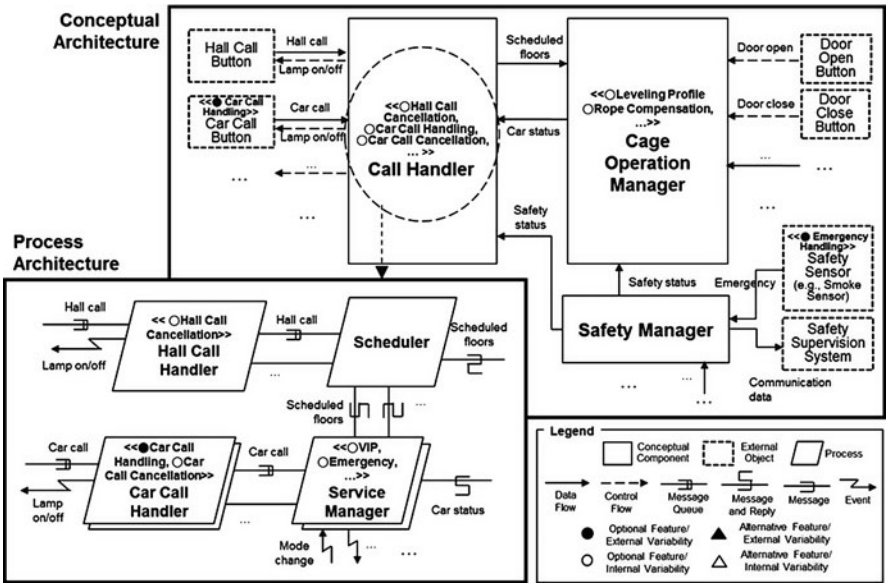


Fig. 8.8 Architecture design and refinement for the high-rise office building ECS

refinement, the quality attributes from the MPP are used for architectural style selection and evaluation [16]. For example, "Independent Component" architectural style [16] is selected, and *Scheduler* is designed to schedule a group of elevators, while each instance of *Service Manager* is designed to control each individual elevator of the group so that efficiency elevator management is achieved (i.e., they addresses the efficiency requirement).

6.2 Asset Implementation

Once conceptual architectures are refined into process and deployment architectures, the architectural components are then refined into concrete components. The PL component design consists of specifications of components and relationships among them. Figure 8.9 includes the component specification of the *Hall Call Handler* component and relationships with other components using UML.

For the component design, the product feature delivery methods in the MPP should be taken into consideration. For example, the use of FORM macro language (i.e., `$IF(;$HallCallCancellation)[-]`) in the component specification of the *Hall Call Handler* component in Fig. 8.9 is to support the prepackaged feature delivery method of the general hospital ECS. When the *Hall Call Cancellation* feature (in Fig. 8.5) is selected as one of prepackaged features in the general

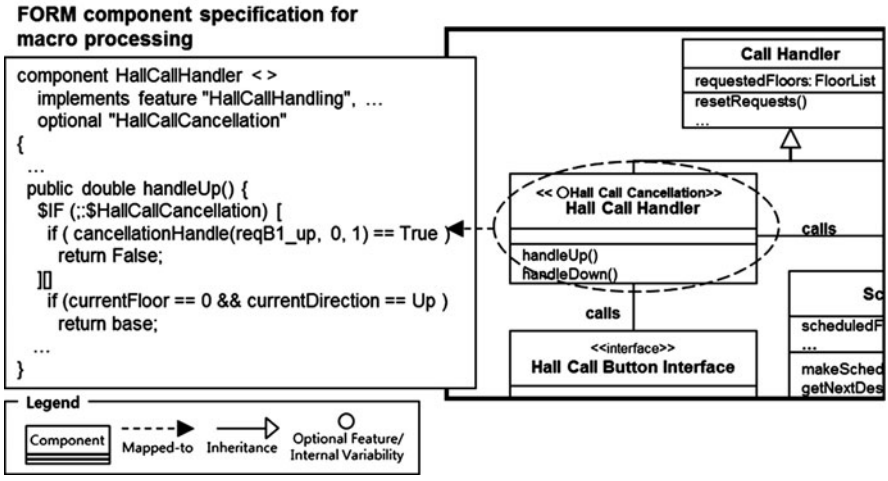


Fig. 8.9 Component specification for Hall Call Handler

hospital ECS, code segments related to the *Hall Call Cancellation* feature are incorporated into the product at product build time.

Depending on the nature of extensions required for product specific features, techniques such as code generation, encapsulation, parameterization, framework, template, etc. may be used. For example, the *Service Manager* component in Fig. 8.8 that encapsulates various elevator control policies (e.g., VIP driving, Emergency driving) may be specified using a formal specification technique (e.g., Statechart specification), from which the program code may be generated after formal verification. Whenever new features are added, the feature interaction specification is modified and tested for correctness and new updated program code for the component is generated.

7 Discussion

In order to develop reusable core assets for a product line, PL software engineering must have an ability to exploit commonality and manage variability. A feature-oriented approach to commonality and variability analysis has been used for PL software engineering both in industry and academia, since the idea of feature-oriented analysis (i.e., Feature-Oriented Domain Analysis (FODA) [8]) was first introduced in 1990 by the Software Engineering Institute.

There are several reasons why FODA has been used extensively compared to other domain analysis techniques. The first reason is that feature is an effective communication "medium" among different stakeholders. It is often the case that customers and engineers speak of product characteristics in terms of "features the product has and/or delivers." They communicate requirements or functions in terms

of features, and such features are distinctively identifiable functional abstractions to be implemented, tested, delivered, and maintained. We believe that features are essential abstractions that both customers and developers understand and should be first class objects in PL asset development. The second reason is that FODA is an effective way to identify variability (and commonality) among different products in a PL. It is natural and intuitive to express variability in terms of features. When we say "the features of a specific product," we use the term "features" as distinctive characteristics or properties of a product that differ from others or from earlier versions. The last reason is that the feature model can provide a basis for developing, parameterizing, and configuring various reusable assets (e.g., PL requirement models, PL architectures, and reusable code components). In other words, the model plays a central role not only in the development of the reusable assets but also in the management and configuration of multiple products in a PL.

Notice that we have not discussed variability in product engineering. It is our opinion that variability of product line must be managed through product line asset engineering; otherwise there will be a proliferation of product versions. Features that are specific only to a product also need to be incorporated into the product through the variability mechanism of the asset management to avoid reworks that may happen when other features of the product evolve.

8 Summary and Outlook

We have explored variability issues over the product line life cycle starting from market analysis through component design to product instantiation. The marketing issues must be studied thoroughly and engineering feasibility of the envisioned products must be analyzed carefully before PL asset development starts.

One of the most important aspects of variability management is managing the explicit horizontal and vertical connections among PL assets including MPP, architecture, and components using various variability models and maintaining their consistencies. Here, the horizontal connection means variability relationships between PL lifecycle products such as MPP, PL architecture, and components, while the vertical connection means relationships between elements of each lifecycle product.

With this connection, marketing, which has traditionally focused only on securing and expanding the market share and on sales strategies, is forced to become more "product aware" and think about how features will be packaged, delivered, and maintained, who will perform these activities, and what the pricing implications are with various alternative approaches. This marketing-oriented perspective can be very effective in uncovering critical quality attributes required for product line architecture and component design. By tightly coupling the marketing with asset development, we can develop PL assets that will support the business goals and satisfy the needs of customers.

In this chapter, we explored explicit connections between business goals/objectives, product usage contexts, quality attributes, and functional and design features. We expect to see more formal treatments of these subjects in a near future.

References

1. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley, Boston, MA (2001)
2. Weiss, D.M., Lai, C.T.R.: Software Product-Line Engineering: A Family-Based Software Development Process. Addison Wesley, Boston, MA (1999)
3. Bosch, J.: Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach. Addison-Wesley, Boston, MA (2000)
4. Kang, K., Donohoe, P., Koh, E., Lee, K., Lee, J.: Using a marketing and product plan as the key design driver for product line asset development. In: 2nd International Software Product Line Conference, pp. 19–22 (2002)
5. Chastek, G., Donohoe, P., Kang, K., Thiel, S.: Product line analysis: a practical introduction. Technical report CMU/SEI-2001-TR-001, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (2001)
6. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Boston, MA (1995)
7. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (FODA) feasibility study. Technical report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, November 1990
8. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley, Boston, MA (2000)
9. Simos, M., et al.: Software technology for adaptable reliable systems (STARS). Organization domain modeling (ODM) guidebook, version 2.0. Technical report, STARS-VC-A025/001/00, Lockheed Martin Tactical Defense Systems, Manassas, VA (1996)
10. Lee, K., Kang, K.C., Lee, J.: Concepts and guidelines of feature modeling for product line software engineering. In: Gacek, C. (ed.) ICSR-7. LNCS, vol. 2319, pp. 62–77 (2002)
11. Lee, K., Kang, K.C.: Usage context as key driver for feature selection. In: Bosch, J., Lee, J. (eds.) SPLC 2010. LNCS, vol. 6287, pp. 32–46 (2010)
12. Lee, H., Choi, H., Kang, K.C., Kim, D., Lee, Z.: Experience report on using a domain model-based extractive approach to software product line asset development. In: Edwards, S.H., Kulczycki, G. (eds.) ICSR 2009. LNCS, vol. 5791, pp. 137–149 (2009)
13. Gomaa, H.: Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures. Addison-Wesley, Boston, MA (2005)
14. Kang, K.C., Kim, S., Lee, J., Kim, K., Shin, E., Huh, M.: FORM: a feature-oriented reuse method with domain-specific reference architectures. Ann. Softw. Eng. **5**, 143–168 (1998)
15. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. Addison-Wesley, Boston, MA (1998)
16. Kang, K.C., Lee, K., Lee, J., Kim, S.: Feature oriented product line software engineering: principles and guidelines. In: Itoh, K., Kumagai, S., Hirota, T. (eds.) Domain Oriented Systems Development: Practices and Perspectives. Gordon Breach Science, UK (2002)

Part II
Review of Research and Commercial Tools

Chapter 9
COVAMOF

Jan Bosch, Sybren Deelstra, and Marco Sinnema

What you will learn in this chapter

- *The basic concepts of COVAMOF and its supportive tool suite.*
- *Experiences from the practical application of COVAMOF.*

1 Introduction

The COVAMOF tool suite is part of an elaborate variability management framework. The framework was specifically developed to deal with aspects of variability management that go beyond formal specification of relations between variation points. The tool suite was constructed as a proof of concept research tool and used to validate the framework in industry. The following sections explain the key difference with respect to traditional variability management approaches and show how the COVAMOF tool suite is used. Throughout this chapter, we use examples from a product line for Intelligent Traffic Systems.

J. Bosch (✉)
Chalmers University of Technology, Gothenburg, Sweden
e-mail: jan@janbosch.com

S. Deelstra
Océ Technologies B.V., Venlo, The Netherlands
e-mail: sybren.deelstra@oce.com

M. Sinnema
Q-Free ASA, Beilen, The Netherlands
e-mail: marco.sinnema@q-free.com

R. Capilla et al. (eds.), *Systems and Software Variability Management*,
DOI 10.1007/978-3-642-36583-6_9,

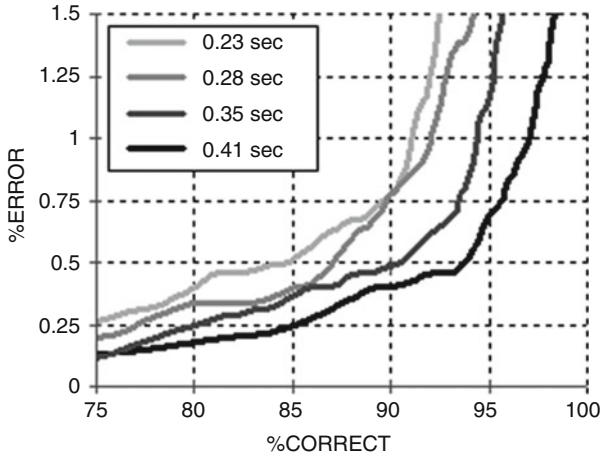


Fig. 9.1 In the ALPR products of Dacolian B.V., the correct rate (%CORRECT) and error rate (%ERROR) are a result of choosing a configuration of explicit variation points and as such pose a *dependency* on these variations points for a given maximum processing time (the *four lines*). Graphs of measurements like the one above capture the *informal knowledge* on the effect of varying maximum processing time on this *interaction* of these dependencies

2 COVAMOF Variability Management Framework

The key difference of the COVAMOF Variability Management Framework in comparison to other approaches is in how it deals with informal knowledge. An example of using informal knowledge can be found in ALPR development for large tolling projects at Dacolian B.V. The derivation of high-performance ALPR products is a complicated task, where trade-offs have to be made to deal with the dependencies between the various configuration options. These trade-offs used to be available as tacit knowledge in expert minds or as informal graphs like Fig. 9.1. As a result, only a few experts were capable of performing a directed optimization during the configuration process. Moreover, their knowledge was hard to capture in formalized dependencies.

COVAMOF addressed this issue through its use of variation points and dependencies as first-class entities. The variation points (Fig. 9.2) provide a view on the variability provided by the product family. These variation points specify for each variant or value the actions that should be taken in order to effectuate a choice. These actions can be specified *formally*, for example, for automatic component selection by a tool, or *informally* in natural language, for example, a guideline for manual steps that should be taken by the software engineers. Variation points that have no effectuation specified are realized by variation points on a lower level of abstraction (e.g., variation in the design realizing the variation in the product features). Such realizations are represented by realization relations between variation points and contain rules that describe how a selection of variation points

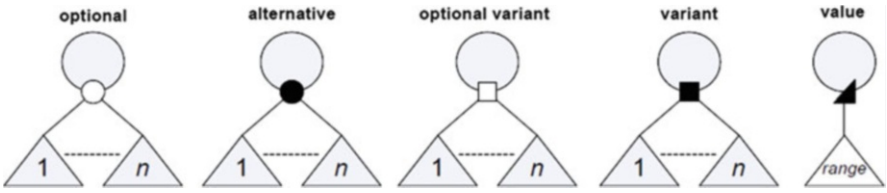


Fig. 9.2 Variability in COVAMOF is modeled by five types of variation points (*large circles*), where the *triangles* represent the possible variants. The *optional* variation points (*white marker*) allow for selecting no variant and *variant* variation points (*square marker*) allow for selecting multiple variants in one product. For *value* variation points, a specific value should be selected within the specified range

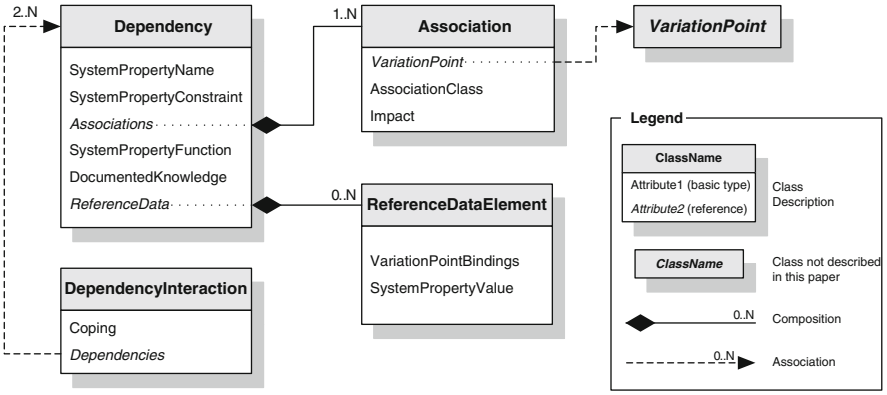


Fig. 9.3 Dependencies in the COVAMOF Meta-model: *dependency* entities contain one or more *associations* to *variation points* and zero or more Reference Data elements. The *dependency interaction* entities specify relations between two or more *dependency* elements

directly depends on the selection of the variation points in a higher level of abstraction.

Also modeling dependencies as first-class entities allow for specifying that relations exist without knowing or writing down all the details of its nature. Treating dependencies as objects furthermore enables storing additional information (see also Fig. 9.5) such as measurements and descriptions of how choices at a variation point influence the dependency and how choices influence multiple dependencies. This information is modeled by the COVAMOF *reference data elements*, the type of an *association*, and *dependency interaction*, respectively. See also the meta-model for dependencies in Fig. 9.3. For more information on the details of a COVAMOF variability model, see [1].

3 Examples and Recommended Areas of Practice

With the key aspect of the COVAMOF framework in mind, this section explains in what way the COVAMOF tool suite contributes to the daily work of software engineers. The COVAMOF tool suite is implemented as Add-Ins for Microsoft Visual Studio and provides integrated variability views on the active Solution, which contains the product family artifacts. Examples of those artifacts are the source files (e.g., C, C++, and C#), start-up configuration files, and XML-based feature models. The variability information in these artifacts is either directly interpreted from language constructs (such as #ifdef) or from special constructs of the COVAMOF variability language XVL within the source files.

For domain engineering, the COVAMOF Variability Assessment Method can be used to construct a model of the variability that is provided by the product family [2]. To assist the engineer during that process, the COVAMOF tool suite provides a graphical model editor, visualized in the left part of Fig. 9.4. To edit the details of individual COVAMOF model entities, the engineer can use the property editor shown in the right part of Fig. 9.4. Basic verification of the model is offered by automatic consistency checking.

In addition to this generic model editor, COVAMOF also provides specialized views that address specific tasks. Examples of these specialized views are the Dependency Editor and the Derivation Assistant. The Dependency Editor is used to maintain the references to documented and tacit knowledge related to dependencies and dependency interactions. In Fig. 9.5, we show how the view is used to model the example from Fig. 9.1.

The COVAMOF tool suite not only supports engineers in specifying the variability provided by the software, but with help of the specialized Derivation Assistant, engineers are assisted to configure the individual products as well. The derivation functionality in COVAMOF supports an iterative configuration process. In this process, the inference engine calculates any implied choices based on the information specified by realization relations and formal dependencies. Where complex dependencies are involved, the tool provides hints on how to continue. These hints assist the engineer to converge towards a solution that addresses all customer requirements. To develop a product, the application engineer needs to follow the following steps:

- *Product Definition*. In this first step, the engineer creates a new Product entity in the variability model, by opening the Product view and typing the product and customer name in the “Name” and “Customer” input fields. After clicking the “New” button, COVAMOF stores his product in the active Solution of Visual Studio, and the engineer can start to configure his product.
- *Product Configuration*. To start the actual configuration, the engineer selects the Product entity from the available products drop-down menu in the COVAMOF toolbar. From that point, COVAMOF is in configure mode, and additional configuration information about the product at hand is shown in both variability views (see also the middle-right section in Fig. 9.6).

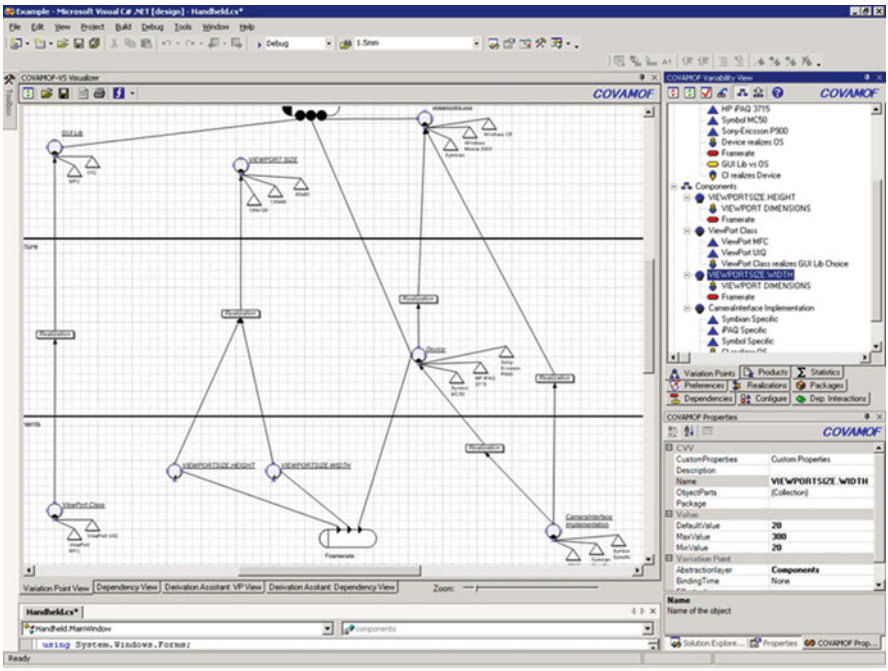


Fig. 9.4 Domain engineering with COVAMOF, as plug-in for Microsoft Visual Studio

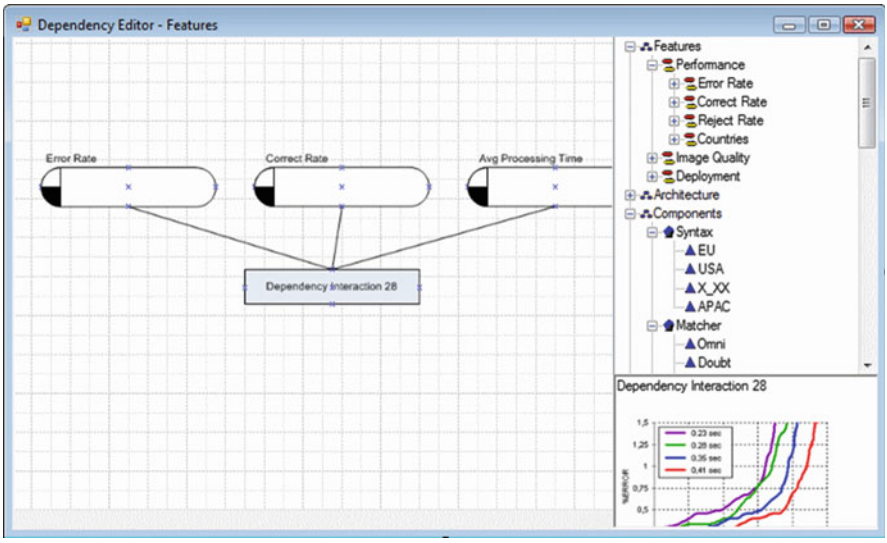


Fig. 9.5 Domain engineering in COVAMOF-VS—maintaining the dependencies and associated interactions. The screenshot shows how the dependency interaction presented in Fig. 9.1 is captured through a combination of formal entities (*left panel*) and informal knowledge (*graph* in the *bottom-right corner*)

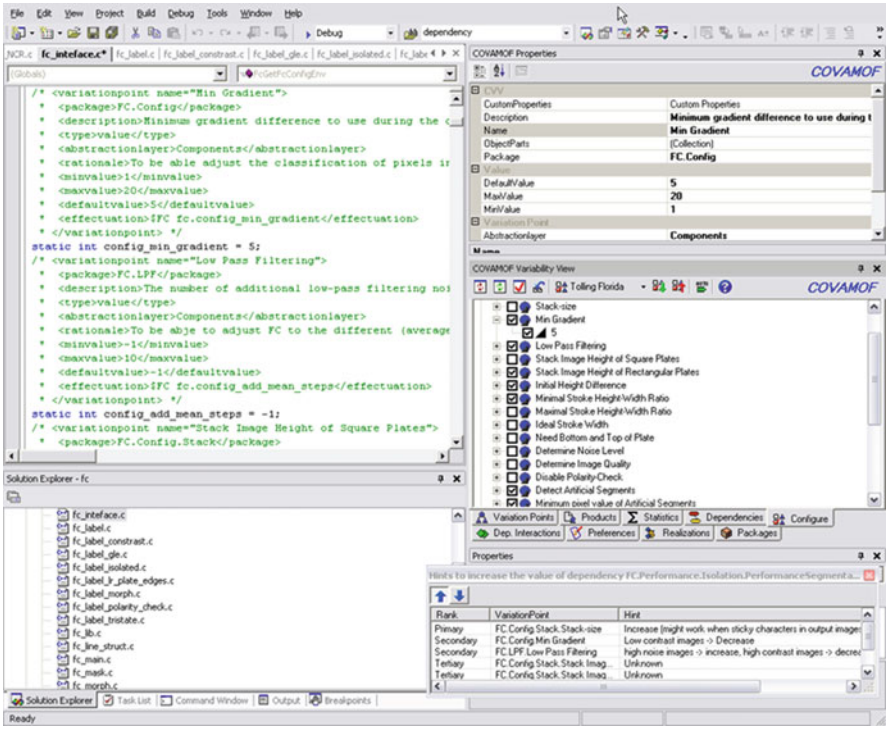


Fig. 9.6 Application engineering in COVAMOF-VS

During the Product Configuration step, the engineer binds one or more variation points to new values or variants based on the customer requirements. In order to bind these variation points, he marks the new variants or specifies the new values in the variation point view. The order in which variation points are bound is dynamically determined by the realization relations and dependencies in the variability model. To specify requirements on dependency values, he clicks the respective dependency in the configure view, which allows to set a minimum and maximum value. This view will also show the inferred dependency value of the configuration at hand, which may be a specific value or may be the value "unknown." In the first case, requirements are automatically verified and where necessary, the tool warns the user. In the latter case, a test is required to determine the value.

After a test, when the result has been fed back to COVAMOF, the measured value will show in the configuration view. For convergence to an acceptable configuration, the engineer can use the hints shown in the Derivation Assistant (see also the bottom-right corner in Fig. 9.6).

- *Product Realization*. In order to get a complete software product, the Product has to be realized to the point where it can be installed and executed. In COVAMOF, the product is realized by pressing the Realize button in the toolbar of

COVAMOF. As a result, COVAMOF executes the effectuation actions specified in the variability model for each of the variants that are selected for the Product entity.

- *Product Testing*. The goal of the testing phase is to determine whether the product meets both the functional and the nonfunctional requirements. When, during testing, the realized product appears to satisfy all requirements, the software product can be packed and shipped to the customer. Otherwise, one or more additional iterations of a Product Configuration and Product Realization steps are required. In any case, the values of the dynamically analyzable dependencies that have been determined during the test are fed back into the variability model as Reference Data. In this way, the COVAMOF variability model is gradually enriched and improved to provide better estimated values during Product Configuration steps in the future.

4 Results and Lessons Learned

Although we explained that the tool suite is useful both in domain and application engineering, the benefits of the tool were only validated for application engineering. To learn about the effects of the use of COVAMOF, an experiment was conducted early in 2004 [3]. During the experiment, engineers were asked to derive several products from a product line, some with, and some without, the use of the COVAMOF tool suite. The experiment showed that indeed both experts and novices were able to derive their product faster with the tools aid, particularly because they required less iterations (see Fig. 9.7). Rather than discussing the experiment and its results in this chapter, however, we will share three interesting observations on the tools use.

- Although the number of product iterations and thus total derivation time decreases with COVAMOF, *NonExperts* spent significantly more time in the configuration step of each iteration when using the tool suite. The cause of this difference most likely lies in the amount of existing background knowledge of the participants. As *NonExperts* lack relevant background knowledge, without COVAMOF, they oftentimes resort to guessing during the configuration step. With COVAMOF, however, they require time to study the suggestions provided by the tool.
- Being an expert can be a drawback at first. From practice, we knew that several experts can have conflicting insights; where one would think the effect of a change would be positive, another expert can think the exact opposite. These effects are taken into account when creating COVAMOF variability models but can also be experienced firsthand during the product derivation process. Some experts will first deviate from the hints provided by COVAMOF, as they believe from their own frame of reference that they should do otherwise. When studying the test results, they will be surprised to see that the original hints were in fact correct.

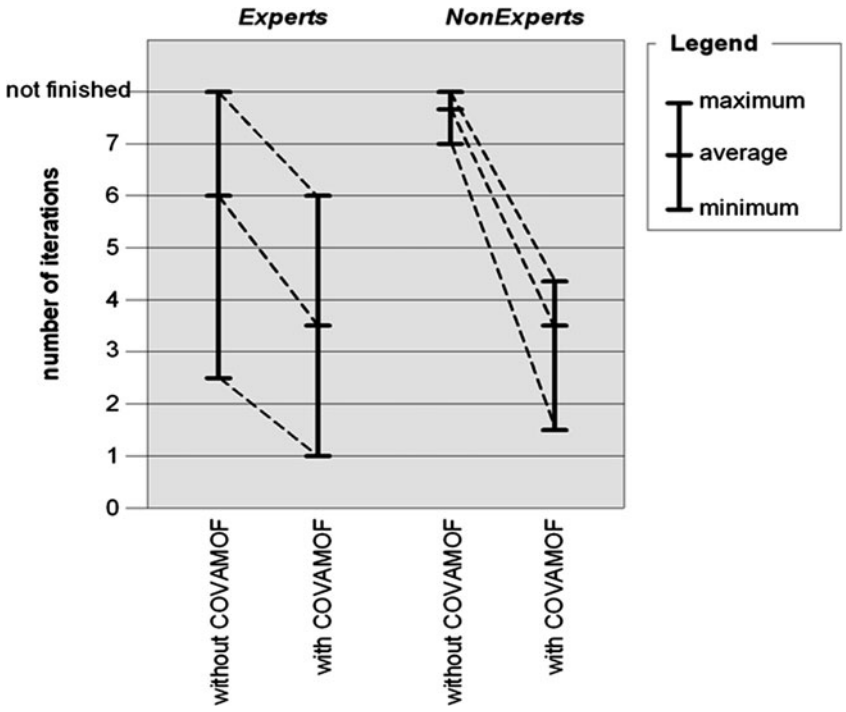


Fig. 9.7 The effect of the COVAMOF tool suite on the number of iterations during product derivation

An important effect of the tool's use is that it improves knowledge sharing, as differences between insights of experts now become visible.

- Users have very different ways to tackle a configuration problem. This was clearly visible during the experiment both with and without using COVAMOF. On the one hand, for example, COVAMOF-VS calculates a ranking of VariationPoints that is based on the impact property of associations. This is the most likely best order in which the selection at VariationPoints should be changed. One participant would instead first exclude a few of the other possibilities prior to following the order suggested by COVAMOF. On the other hand, without COVAMOF, some participants would rigorously change choices to determine the effect of changes, while others would only marginally change them.

5 Outlook

The COVAMOF tool suite was developed up to the point where it would sufficiently serve its purpose of proofing the feasibility and value of the COVAMOF Variability Management Framework. Naturally, there are several ways where the tool suite and framework can be improved:

- *Including scale*: The first improvement is directly related to the last observation in the previous section. While the tool currently suggests that the value of certain variation points have to be increased, an easy extension is to include the scale of the change as well.
- *Test integration*: Since test effort is usually the largest step in deriving products using COVAMOF, a significant improvement would be a tighter integration of COVAMOF and automatic test suites. Viable expansions are launching tests directly from the tool and storing their results automatically as Reference Data in the COVAMOF variability model.
- *Automatic configuration*: The way that the use of the COVAMOF tool suite is depicted in this chapter is a situation where it is used to assist engineers during product derivation, that is, through consistency checking, configuration guidance, and automatic inference. This automatic inference is currently based on formalized knowledge. Once a better test integration is achieved, the tool can be extended so that it also uses the fuzzy rules, hints, and intermediate test results to derive a product automatically.
- *Language integration*: Currently, variation points are embedded in source code by annotation. The COVAMOF variability modeling elements could also be incorporated as first-class entities in (programming) languages.
- *Runtime adaptability*: With the previous improvement in place, dependencies can also be checked on runtime, and the system can be reconfigured using variation points [4].

Even without these improvements, however, COVAMOF can already contribute to more than it was initially intended. Parts of the framework are being used in industry today. You can download the tool suite and related research papers at the Tool website [5].

References

1. Sinnema, M., Deelstra, S., Nijhuis, J., Bosch, J.: Modeling dependencies in product families with COVAMOF. In: Proceedings of the 13th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2006), pp. 299–307, March 2006
2. Deelstra, S., Sinnema, M., Bosch, J.: Variability assessment in software product families. Inf. Softw. Technol. **51**(1), 195–218 (2009)
3. Sinnema, M., Deelstra, S.: Industrial validation of COVAMOF. J. Syst. Softw. **81**(4), 584–600 (2008)

4. Siljee, J., Bosloper, I., Nijhuis, J., Hammer, D.: DySOA: making service systems self-adaptive. In: Proceedings of the Third International Conference on Service Oriented Computing (ICSOC05), pp. 255–268 (2005)
5. Tool website. http://www.msinnema.nl

Chapter 10
PLUM: Product Line Unified Modeler Tool

Cristina López and Jason X. Mansell

What you will learn in this chapter
- *The concepts underlying the PLUM tool*
- *Using PLUM for variability management*

1 Introduction

PLUM[1] [1] is a tool suite for the design, implementation, and management of product lines that follow a Model-Driven Software Development approach. PLUM is a domain agnostic tool suite. It aims to provide an integrated set of tools which support adopting Product Line Engineering (PLE) approach in any domain.

With PLUM, the variability of the domain products is captured in what is called a Decision Model. Decision Modeling implies analyzing domain variability in terms of decisions and establishing dependencies among them (this is further detailed in Chap. 2). This technique, which comes from previous European research projects such as FAMILIES,[2] CAFÈ,[3] and ESAPS[4] finds in PLUM its latest implementation for PLE automation support. Decision Modeling is an approach alternative to Feature Modeling [2]. While Feature Modeling represents all the possible elements of the domain in terms of features, Decision Modeling [3] only focuses in the variable elements, representing them with decisions/questions. In this

[1] PLUM tool, http://www.tecnalia.com/plum

[2] FAMILIES ITEA research project, http://www.esi.es/Families/

[3] CAFÈ ITEA research project, http://www.hitech-projects.com/euprojects/cafe/

[4] ESAPS research project, http://www.esi.es/esaps/

C. López • J.X. Mansell (✉)
Fundación TECNALIA Research & Innovation, Derio, Bizkaia, Spain
e-mail: cristina.lopez@tecnalia.com; jason.mansell@tecnalia.com

R. Capilla et al. (eds.), *Systems and Software Variability Management*,
DOI 10.1007/978-3-642-36583-6_10,

way, PLUM differs from Feature Modeling approaches implemented in well-known commercial tools like pure::variants[5] or Gears.[6]

PLUM is based on Eclipse,[7] which facilitates the integration of tools in the same framework. PLUM uses a wide range of well-established Eclipse technologies, including, but not only restricted to, the Eclipse Modeling Framework [4], GMF [5] for graphical asset editors, EMF Validation Framework, OCL [6] for Decision Model's dependency engine, BIRT[8] for reporting valuable PLE metrics, the Modeling Workflow Engine (MWE)[9] for the execution of model transformations, ModelBus[10] to allow clients to ask for product generation at a remote server, and SVN[11] for version control and for obtaining historical metrics. The transformation languages supported are Xpand2[12] and Xtend, previously parts of the OAW project[13] but now integrated in the Eclipse Modeling Project.

2 PLUM Implementation Process

In this chapter we will guide the reader in the use of PLUM to implement a software product line. To illustrate the utilization of PLUM, we will use the example from the "*Airbus Simplified Doors and Slides Control System (SDSCS)*" case study, in which Tecnalia cooperated with Airbus and EADS. As a first step we present the actual process for defining a software product line. Then we will follow this process in the execution of the SDSCS case study.

As any other variability tool, the success of using PLUM depends on three principles: the knowledge of the specific domain, the adequate use of the tool, and variability management background (in other words, previous experience dealing with variability). The adoption of a PLE approach requires a high investment of effort and cost, but the benefits obtained afterwards are very well known as reported in [7].

The initial phase on the implementation of a product line is performed by the Domain Engineer in what is called Domain Analysis. The Domain Engineer will be responsible of determining the domain concepts that are going to be used to build the PLE. The Domain Engineer identifies the variability, as well as the dependencies among the elements of the product line, in order to implement the

[5] pure::variants, http://www.pure-systems.com/

[6] Big lever, gears, http://www.biglever.com/

[7] Eclipse, http://www.eclipse.org/

[8] Business intelligence and reporting tools, http://www.eclipse.org/birt/phoenix/

[9] Modeling Workflow Engine, http://www.eclipse.org/modeling/emft/?project=mwe

[10] ModelBus, http://www.modelbus.org

[11] Subversion, http://subversion.tigris.org/, http://subversion.apache.org/

[12] Xpand, http://www.eclipse.org/modeling/m2t/?project=xpand

[13] Open Architecture Ware, http://www.eclipse.org/workinggroups/oaw/

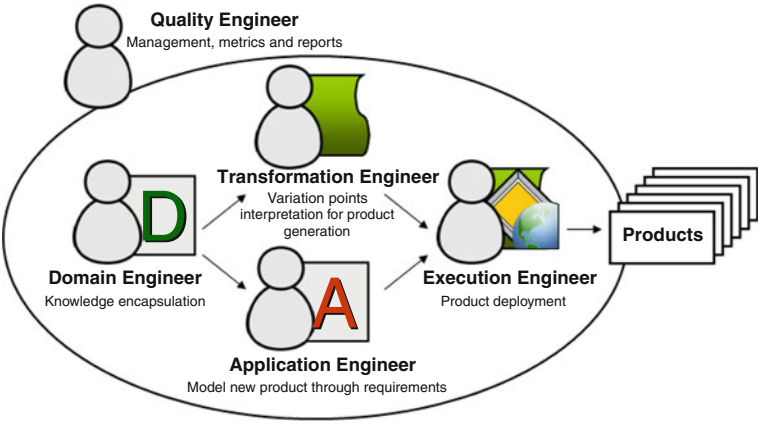


Fig. 10.1 Roles in PLUM

corresponding Decision Model(s) that captures the mentioned knowledge. Figure 10.1 gives an overview of how the roles that are part of the PLUM implementation process are related to each other.

As a result of the Domain Engineer's work, a Domain Model is created. A Decision Model is a set of decisions and dependencies that represent all possible dimensions of variation of the product line products. This Decision Model serves as the basis for creating the product line architecture. It includes the variability points that provide the architecture of the required logic that enables producing all the products aligned to the Decision Model. This architectural knowledge is captured by the Transformation Engineer in the Flexible Components, which are "*smart*" components capable of managing the variability of the Decision Model and consider the commonalities by embedding this domain knowledge in the form of transformation rules (e.g., IF-ELSE statements).

The mechanism used to actually configure a specific product from the product line is called Application Model. The Application Engineer is responsible of creating as many Application Models as necessary to represent the wide range of products of the family and fulfill the product line requirements. The Application Model is an instance of a Decision Model in which the variability has been resolved by giving value to each of the decisions of the model, in other words, by "making decisions."

Finally, the Execution Engineer is the person who defines and executes the Workflows in order to generate the products. The Workflow is a file that enumerates the sequence of steps that have to be executed, establishing the necessary relationships among the Application Models and the Flexible Components, in order to generate the desired products. The products can be of diverse nature, from compiled code (e.g., Java, C), Programmable Logic Controller code, HTML code, XML models, documentation, UML models, etc. In the SDSCS case study, the goal is to obtain UML models representing the architecture of the aircraft. The implementation process of the SDSCS case study is depicted in Fig. 10.2.

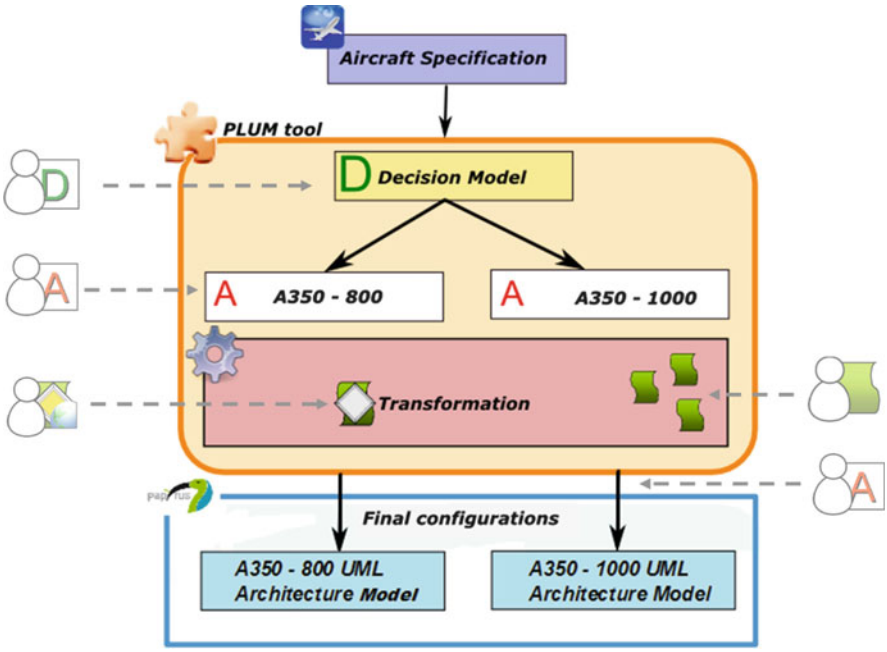


Fig. 10.2 PLUM implementation process overview

In order to get additional information about the PLUM development process, check the tool user manual online [8]. Examples explaining how to use the tool and how to define the different assets of a PLUM project are also available in the tool web site, under the documentation section [9]. In the following section, we introduce a case study that exemplifies the described implementation process and the PLUM assets that have to be created.

3 Airbus Simplified Doors and Slides Control System

This case study is related to the system components that are necessary to include in an aircraft in order to ensure the pressurization of the cabin thorough the correct management of the doors systems. The Doors and Slides Control System of an Airbus is mainly composed by passenger doors, emergency doors, cargo doors, the sensors and actuators associated to them, and the circuitry and components that internally manage the signals received by the sensors and actuators. Currently the safety standards require a strict manufacturing process in which a component-based approach is used but in which each provider supplies components with different response times and more important different failure rates. The design of the aircraft must accomplish an error safety objective in order to fulfill the standardized safety

requirements. What determines the total failure rate of the aircraft is the sum of the individual failure rates of the sensors, actuators, warning lights, and related internal elements that compound the system. The components of the system are of different suppliers, which offer diverse failure rates for each of the components, so, for example, Supplier A may offer a failure rate of $2e^{-8}$ for a pressure sensor, while the Supplier B's sensor has $5e^{-07}$. The key of the aircraft design relies on encountering the balance between the supplier failure rates per component and the number of sensors of each kind that are necessary in the system to reach the regulated safety objectives.

The example at hand is a simplified version of the Doors and Slides Control System. So for instance, the example establishes that an aircraft model must have the same supplier for all the components. Each supplier has a general failure rate associated. Therefore, the supplier selected will determine the minimum and maximum number of components that the system may have in order to fulfill the safety objective. Once the supplier is chosen, the number of sensors, actuators, and warning lights selected will also establish the number of internal components (RDC, CPIOM, etc.)[14] that the systems tolerate. Furthermore, the example also considers the model and layout of the aircraft, characteristics of the systems that decide the total number of doors and indeed the number of associated sensors, actuators, and warning lights allowed. These properties of the SDSCS problem, and how they relate to each other, are outlined in Fig. 10.3.

3.1 Creating the Decision Model

The Decision Engineer performs the Domain Analysis to extract the variability and commonalities and therefore create the associated Decision Model. Based on the results of this analysis of the SDSCS example, the variability as well as the dependencies among the variations is captured in PLUM in the manner that Fig. 10.3 depicts.

The Decision Model defined in the case study is divided into three groups: Model, Supplier, and Fuselage. The first group, "Model," contains decisions related to the family model and layout of the aircraft, which affect the total number of doors of the aircraft. The "Has high comfort layout" decision is only visible when the "A350-1000" model is selected in the previous decision ("Choose Model"). This behavior has been implemented by adding a dependency between both decisions and associating a "Validity Action" that controls the visibility of the affected decision. The group "Supplier" contains a decision to choose the component supplier of the whole system. The value selected in the "Choose the component

[14] There are different internal HW/SW components that are connected to the sensors and actuators. RDC stands for Remote Data Concentrator, whereas CPIOM stands for Core Processor Input/Output Mode.

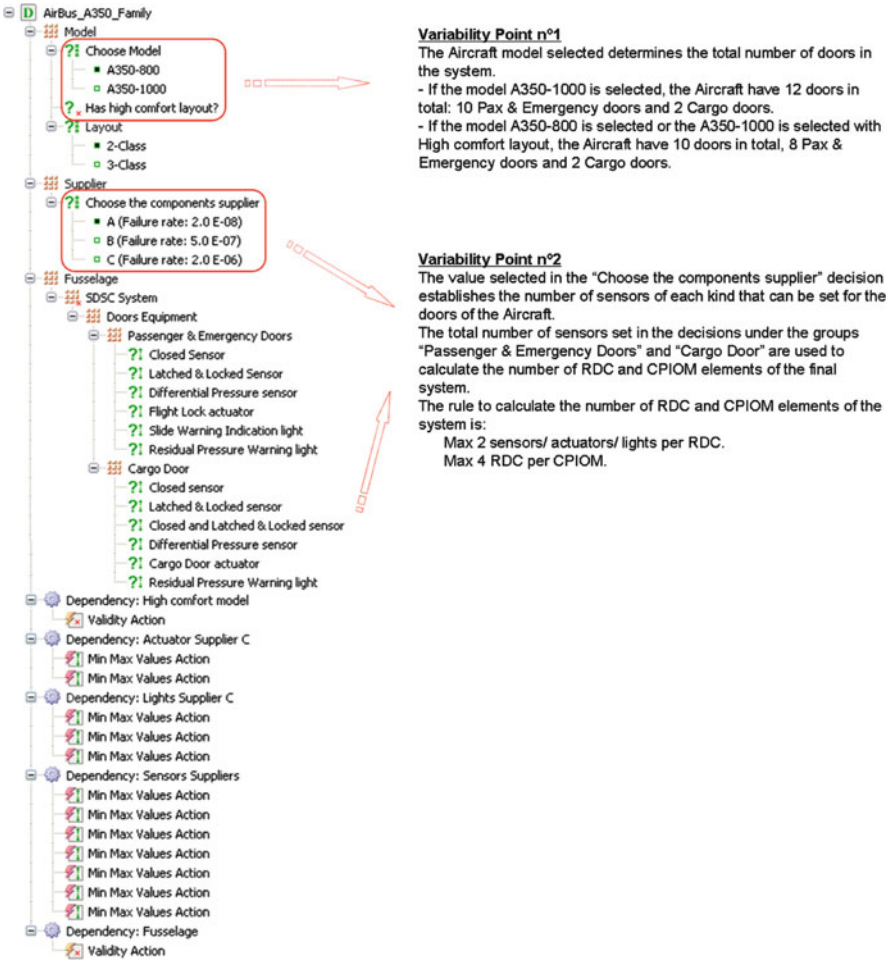


Fig. 10.3 Decision Model of the SDSCS case study

supplier" decision determines the number of sensor, actuator, and warning light components that the system can accommodate, which is calculated taking the average failure rate of the supplier. The last group, "Fuselage," contains a subgroup named "Doors Equipment" which is composed of decisions that allow selecting the desired number of sensors, actuators, and warning lights. These decisions are of type "Min Max," which defines the minimum and maximum bounds of an Integer value.

The Decision Model must not only consider all the possible dimensions of variation of the domain but also the relationships between the elements. The object "Dependency" is used to represent these relationships and "Action" prepositions are attached to the dependencies to implement the desired behavior (defined with OCL [6]). There is a wide range of predefined Actions to choose from (described in

detail here [8]) and are mainly used to guide the Application Engineer in the decision-making process when defining a new product. These dependencies and actions are used not only to guide the resolution making process but also to ensure model compliance and prevent the Application Engineer from committing any error. Additionally, the underlying OCL language gives an extra support to define complex conditions and constraints within the dependencies among the decisions.

Some dependencies have been defined in the model to implement the described behavior as can be seen at the bottom of Fig. 10.3. These dependencies contain "Min Max Values" Actions that with the support of OCL queries are used to change the range of selectable elements in the "Doors Equipment" subgroup depending on the supplier previously chosen. For instance, if *Supplier A* is selected, the number of *Pressure Sensors* for a cargo door must be between two (min) and four (max); in others words, the door can have two, three, or four pressure sensors. Otherwise, if *Supplier C* is selected, the number of *Pressure Sensors* for a cargo door must be obligatorily four, due to the high failure rate of the selected supplier.

3.2 Resolving the Application Models

As depicted in Fig. 10.2, we have defined two product configurations using the Application Models: A350-800 and A350-1000.

For simplification, the values selected in both models are depicted in Fig. 10.4, instead of explained with text. The configuration of the A350-800 is on the left side of the figure, while the configuration of the A350-1000 is on the right side. In order to give values to the Application Model, the user navigates through the model decisions and, with the help of a properties window, establishes the answers to the different questions that are part of the model. The checkboxes of the Application Model decisions indicate that the decision has been made and hence has a value, which is shown after the decision name. Comparing the aircraft configurations illustrated in Fig. 10.4, it is easy to see that the answers of the decisions are different in each case.

3.3 Creating the Flexible Components

The Flexible Components contains the reusable information of the product line. These components are capable of managing the variability described in the Decision Models, interpret the values given by the Application Model, and reuse the commonalities of the domain-defining transformation rules (a transformation is just a piece of code that maps one type of concepts to another type of concepts). The Flexible Components transformations can be either Model-to-Text or Model-to-Model transformations. The languages chosen to implement these transformations are Xpand and Xtend, respectively [10], both part of the Eclipse Modeling Project.

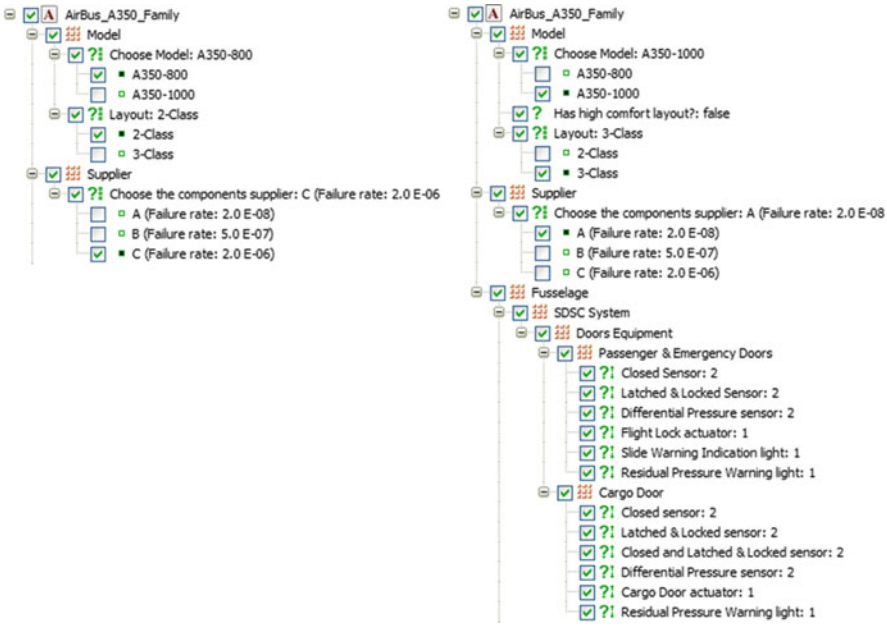


Fig. 10.4 Application Model resolution comparison

For the understanding of the example, it is not worthwhile explaining in detail the logic behind the implementation of these Flexible Components.

On the right side of Fig. 10.3, it is explained how the elements that the final system architecture must contain are calculated depending on the Application Models decision values. The transformations of the Flexible Components are programmed following this logic. Knowing this and taking into account the decision values depicted in Fig. 10.4, we know that the design of the A350-800 aircraft should have 8 Pax & Emergency doors, 2 Cargo doors, 38 sensors in total, 19 RDCs, and 5 CPIOMs. On the other hand, the design of the A350-1000 aircraft should have instead 10 Pax & Emergency doors, 2 Cargo doors, 19 sensors, 10 RDCs, and 3 CPIOMs. The transformations are coded in such a way that, given the input from the Application Models, a new UML model is created, following the logic behind the variability points explained at the right side of Fig. 10.3.

3.4 Execution of the Product Line

Being at this stage, it is the responsibility of the Execution Engineer to launch the execution of the product line project. In the case study at hand, the results generated by the product line are UML models implementing the simplified architecture of the Doors and Slides Control System of the aircraft. The UML models generated are

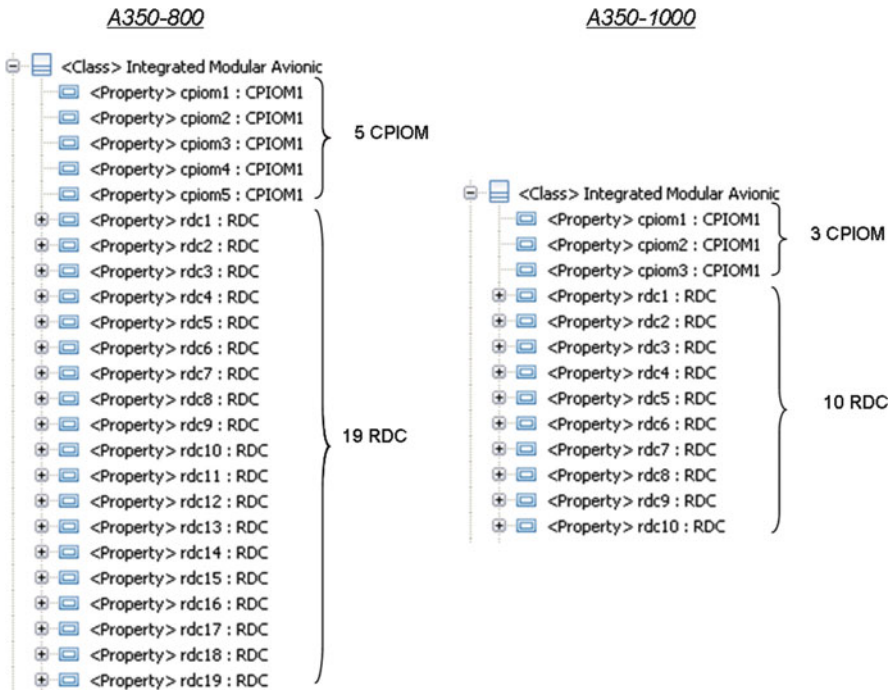


Fig. 10.5 Extract of the product line execution results of UML models

not created from scratch. As depicted in Fig. 10.2, the PLUM product line receives as input the Aircraft Specification, which internally contains, among others, a UML containing all the possible dimensions of variation of the use case. Then, after the execution of the product line, a new version of the UML model with the variability solved is generated taking into consideration the decisions made in the Application Model. Figure 10.5 illustrates an extract of the UML models obtained for each of the product configuration (Application Models) described above. Comparing both models, it is quite straightforward to see how the results differ from one configuration to another. This output is correctly obtained; thanks to the transformation rules that are defined in the Flexible Components as mentioned before.

4 Success Stories

PLUM has proven successful so far in very different domains. Even though this chapter addresses the PLUM tool, actually the way it is provided to the market is by means of technological transfer, that is, the PLE approach is introduced within the culture of the target organization, with PLUM just being a tool to automate the PLE process. The PLUM tool has undergone implementations in the following domains:

enterprise resource planning (ERP) [11], manufacturing (PLCs), avionics (UML modeling and transformations) [12], content providers (content manager and web generation automation) [13], and requirements management (requirement-guided reasoning) [14, 15]. Basically we are talking of any field where companies need to increase productivity, reduce risks, and industrialize software production.

The major added value that is identified by the customers that have used the PLUM, among others, is that since it is fully Eclipse based, it allows the easy integration with another tools and the implementation of new enhancements by the customer. Once the tool has been deployed within the company, they are not married to the tool provider and the consequent restrictions. This means, in other words, that they do not totally depend on the tool provider in order to integrate new functionalities.

The major issues the adopters identify as potential improvements are directly linked to user interfaces and customer oriented, such as the use of a natural language at Decision Model level, the ability for the tool to learn (based on the user reasoning when defining decision models as well as variability/dependencies), scalability issues, and one which is considered as the next generation of PLUM, the capability of specifying Domain-Specific Languages (elements, images, etc.) which can be used as an add-on to the Decision Model.

5 Outlook

The effort of the PLUM development team is devoted to improve the user experience when working with the tool and minimize the impact of adopting new technologies. One way to achieve this, and our goal for the near future, is to provide an interface intelligent enough to self-adapt its capabilities to the domain at hand. The basic idea behind this self-adaptation is that the tool should have a capability of enabling each company adopting it to define their own DSL which will be directly mapped to the actual PLUM underlying technology. We are confident that, with effort, this could become a reality in the coming years.

One key improvement of the tool support will provide full bidirectional traceability from decision to variations points in order to enable the automation of maintenance efforts and easy the evolution of the product line.

References

1. Martinez, J., Lopez, C., Aldazabal, A., Mansell, J., del Hierro, M.: “PLUM (Product Line Unified Modeller)” Tool demo. In: 13th International Software Product Line Conference, SPLC 2009, San Francisco, CA, USA, 24 Aug 2009
2. Czarnecki, K., Grünbacher, P., Rabiser, R., Schmid, K., Wąsowski, A.: Cool features and tough decisions: a comparison of variability modeling approaches. In: Proceedings 6th

International Workshop on Variability Modelling of Software-Intensive Systems, Leipzig, Germany, pp. 173–182 (2012)
3. Schmid, K., Rabiser, R., Grünbacher, P.: A comparison of decision modeling approaches in product lines. In: Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems (VaMoS '11), pp. 119–126. ACM, New York (2011)
4. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: Emf: Eclipse Modeling Framework, 2nd edn. Addison-Wesley Professional, Reading, MA (2009). http://dl.acm.org/citation.cfm?id=1197540. ISBN 0321331885
5. Graphical modeling framework. [Online]. http://www.eclipse.org/modeling/gmf/
6. Warmer, J., Kleppe, A.: The Object Constraint Language: Getting Your Models Ready for MDA. Addison-Wesley, Boston, MA (2003)
7. Mansell, J.: Experiences and Expectations Regarding the Introduction of Systematic Reuse in Small and Medium-Sized Companies. Springer, Berlin (2006)
8. PLUM tool user manual. Available online in the website: http://www.esi.es/plum/docum/PLUM_User_Manual_v3.0.pdf
9. PLUM examples with tutorials. Available online in the website at the documentation section: http://www.esi.es/plum/documentation.php
10. Xpand & Xtend languages Eclipse reference. http://help.eclipse.org/galileo/index.jsp?topic=/org.eclipse.xpand.doc/help/ch01.html
11. Hamza, H., Martinez, J., Alonso, C.: "Introducing product line architectures in the ERP industry: challenges and lessons learned" industry track. In: 14th International Software Product Line Conference, SPLC 2010, Jeju Island, South Korea, 13–17 Sept 2010
12. Saratxaga, C.L., Alonso-Montes, C., Haugen, O., Ekelin, C., Mitschke, A.: Product line tool-chain: variability in critical systems. In: 3rd International Workshop on Product LinE Approaches in Software Engineering (PLEASE 2012), Zurich, Switzerland, 4 June 2012
13. Martinez, J., Lopez, C., Ulacia, E., del Hierro, M.: Towards a model-driven product line for web systems. In: 5th Model-Driven Web Engineering Workshop, MDWE 2009, San Sebastian, Spain, pp. 1–15, 22 June 2009
14. Dubois, H., Ibanez, V., Saratxaga, C.L., Machrouh, J., Meledo, N., Mouy, P., Silva, A.: The product line engineering approach in a model-driven process. In: Online Proceeding of the Embedded Real Time Software and Systems (ERTS2 2012) Conference, Toulouse, France, 1–3 Feb 2012. Available at: http://www.erts2012.org
15. da Silva, A., Becker, M., Graubmann, P., Schmid, R., Ibanez, V., Meledo, N.: Theory and application of product line engineering. In: Proceedings of the Embedded World 2012 Exhibition and Conference, Nuremberg, Germany, 29 Feb–1 Mar 2012

Chapter 11
FaMa

David Benavides, Pablo Trinidad, Antonio Ruiz-Cortés, and Sergio Segura

What you will learn in this chapter

- *The scope of the automated analysis of variability models.*
- *How to use a tool for the automated analysis of variability models.*
- *A product line architecture to build analysis tools.*

1 Introduction

Extracting relevant information from variability models is an important task to support decision-making in product line development and product configuration. Examples of questions that can arise during development are:

- Which are the products that contain a certain subset of features?
- Does the variability model contain any errors, i.e. contradictory information?
- How many products are we able to build?
- How much does it cost to produce a certain product?

Answering the above questions can be a tedious and error-prone task, and it is infeasible to perform manually with large-scale models; so it requires to be automated.

Using tools for the Automated Analysis of Variability Models (AAVM, see Definition 11.1) is part of the life of software product line developers. As an example of such tools, we present in this chapter the FeAture Model Analyser (FaMa) Tool Suite. It is an ecosystem of tools that focuses on the most used variability modelling languages: feature models. It contains several software tools some of which are:

D. Benavides (✉) • P. Trinidad • A. Ruiz-Cortés • S. Segura
University of Seville, Seville, Spain
e-mail: benavides@us.es; ptrinidad@us.es; aruiz@us.es; sergiosegura@us.es

R. Capilla et al. (eds.), *Systems and Software Variability Management*,
DOI 10.1007/978-3-642-36583-6_11,

- FaMa Framework: a customizable analysis framework.
- FaMa Test Suite: a set of implementation-independent test cases to test feature model analysis tools such as those created by the FaMa Framework.
- FaMa Integrations: a set of developments to integrate FaMa Framework into other tools such as Moskitt Feature Modeler.
- BeTTy Framework: Betty [1] is a highly configurable framework supporting benchmarking and functional testing of variability model analysis tools.

The tool suite is under LGPL v3 licence and can be found at http://www.isa.us.es/fama. In this chapter we focus on FaMa Framework, the masterpiece in the puzzle of tools provided by the ecosystem.

1.1 Definitions and Examples

Definition 11.1. Automated analysis of variability models

The Automated Analysis of Variability Models (AAVM) is about the automated extraction of information from variability models using automated mechanisms to assist decision-making in PL development or life cycle. This is a key activity in variability management because it allows checking and extracting information from the variability model which is central in the development of a PL.

Example 11.1. Example of analysis operations

Let's think about a feature model as a possible variability model. A feature model can be analysed automatically extracting valuable information from it. For instance, dead features could be automatically detected. Dead features are features that are represented in the model but can never be part of a concrete product because model internal inconsistencies. At the time of writing this book, there are 30 different analysis operations reported in the literature [2].

2 FaMa Framework

FaMa Framework (FW) is a product line of tools to analyse variability models. It mainly performs analysis operations transforming a variability model into a suitable logic, which is used to reason about the model using off-the-shelf logic solvers. FaMa FW is developed as a product line, which contains many variant features. For example, there exist different variability meta-models (a.k.a. feature model dialects) and file-formats, several analysis operations (a.k.a. *questions*) to be performed over them. These questions are answered using different *reasoners* or logic solvers, each of them performing better or worse for a kind of model, which is determined by the *reasoner selectors*. Since some analysis operations can take long time to solve, the framework provides for several *transformations* to improve the response time.

Figure 11.1 represents a feature model of the FaMa FW. It offers a wide variety of variant features, such as meta-models, logic reasoners, analysis operations,

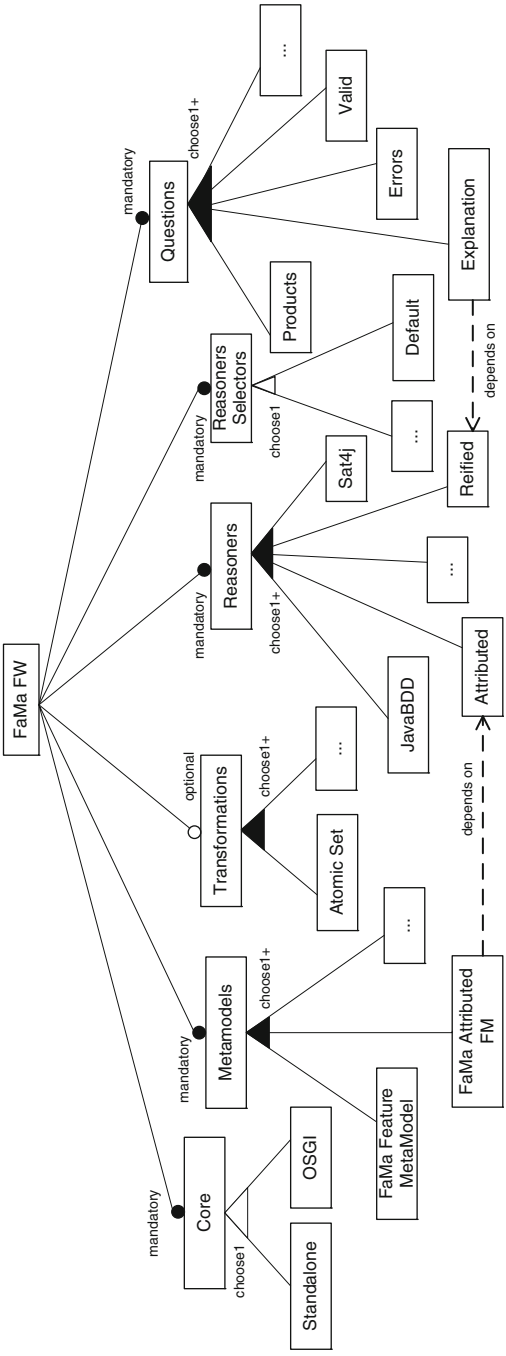


Fig. 11.1 A feature model describing FaMa Framework variability

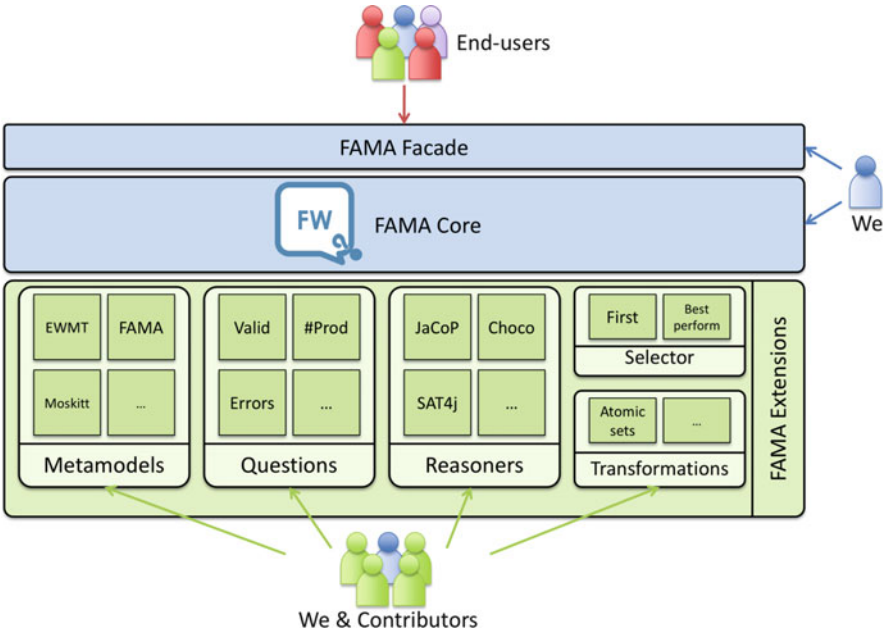


Fig. 11.2 FaMa Framework architecture

transformations and reasoner selectors. A product in FaMa FW product line is therefore a selection of the variant features that are useful for a certain context, such as a CASE tool for PL, a product configuration tool or a reconfigurable smart home.

2.1 FaMa Architecture

FaMa FW has a component-based architecture as shown in Fig. 11.2. Every variant feature is built as an independent component or *FaMa Extension*. These extensions can be classified as follows:

- Meta-models: support different FM representations and file formats.
- Reasoners: map a feature model into a concrete logic. This logical representation is used to solve analysis operations.
- Questions: interfaces that define the available analysis operations independently of the specific reasoners that answer them.
- Criteria selectors: select the most efficient reasoner for a particular question and a given feature model.
- Transformations: transform a feature model into an equivalent one in terms of products but easier to analyse in terms of performance.

A FaMa FW product is customised selecting a valid set of FaMa Extensions that comes together with the *FaMa Core*, which is the common and mandatory part of the architecture among different FaMa FW products. It mainly communicates meta-model and reasoner components through mappings, registers the available questions and use criteria selectors to search for the reasoners that may answer the question a user demands. All this process is performed independently of specific components, avoiding any kind of coupling among them. We can build our own customised FaMa FW product selecting the components we are interested in. We assume that the SPL developers, as end-users, are not interested in the internal aspects. To this purpose, FaMa FW just wants to analyse feature models with the best performance. We provide *FaMa Facade* as a transparent way to integrate FaMa FW into existing systems. It is a stable facade that hides the complexity of FaMa Core into a unique interface. It is designed to reduce the changes that are carried to users; so a new extension can be deployed while the façade remains unaffected.

FaMa FW aims to be integrated into existing CASE tools. Eclipse has become a standard in CASE tool development. FaMa Core and Extensions are OSGi-compliant [3] which is a requirement for being Eclipse-compliant. FaMa FW not only uses this technology as an alternative to support componentization but also provides for its own componentization technology that permits it to work as a standalone application wherever OSGi is not available.

3 Examples and Recommended Areas of Practice

FaMa FW has been designed to be an important piece of third-party products. It provides for an API offered as an OSGi bundle and a Java library distribution. At date, it has been used in four kinds of products:

- Feature Modeling Tools: Incorporating analysis capabilities to visual editors and other CASE tools that use VMs somehow. Moskitt Feature Modeller [4], an Eclipse-based visual editor of feature models is an example of it.
- Product Configurators: Providing product configuration capabilities to validate feature selections and to assist end user by propagating user decision and suggesting corrections for invalid configurations. These products usually use a fixed feature model. FaMa Debian Package and ISA Packager use FaMa FW for this purpose.
- Dynamic Systems Reconfiguration: Restoring and reconfiguring dynamic systems such as smart homes [5] and TV broadcasting systems [6] whenever errors happen or system preferences change. FaMa Lite for Smart Homes supports the decision of the features to activate or deactivate services whenever a new feature is deployed or an existing one fails.
- Fast Prototyping Framework: Building tools for the development of new variant features for FaMa FW PL. We have developed the BeTTy Framework [1] and

the FaMa Test Suite and FaMa Random Generator to test the functionality and performance of FaMa FW reasoners and FaMa SDK, an environment to develop new variant features.

For a first contact to FaMa FW, it is recommended to use its console interface. It permits a complete interaction with all the elements in the infrastructure and may be used to evaluate its capabilities. Next, we show an example of how to specify an FM using our file-format and how to use FaMa FW from the console and a Java application as a library.

3.1 An Example Input Feature Model for FaMa FW

FaMa FW supports several feature meta-models and also offers a plain-text format to represent all the kinds of relationships that can be found in the bibliography. The example below describes the FM in Fig. 11.1 in a textual format.

Notice that a FM is divided into hierarchical relationships and cross-tree constraints. Any relationship follows the next syntax:

```
Parent: [min_card,max_card] {Child1 Child2 ...};
```

This cardinality-based relationship allows the definition of mandatory ([1,1]) and optional ([0,1]) relationships for a one-child relationship and alternative ([1,1]), or ([1,N]) and set relationships for a multiple-child relationship. Constraints can represent *require* and *exclude* constraints and any other constraint that can be represented in terms of a Boolean constraint. Although there is a full support for extended feature models, allowing working with attributes, this example avoids them for the sake of simplicity.

Example 11.2. A definition of a FM using FaMa plain-text format

```
%Relationships

FaMaFW: Core Metamodels [Transformations] Reasoners
        ReasonerSelectors Questions;
Core: [1,1]{Standalone OSGi};
Metamodels: [1,3]{FaMaFeatureMetamodel FaMaAttributedFM
AnotherFM};
Transformations: [1,2]{AtomicSet Attributed2Basic};
Reasoners: [1,5]{JavaBDD Sat4j Choco Reified Attributed};
ReasonerSelectors: [1,2]{Default OptimalSelector};
Questions: [1,23]{Valid Products Errors ValidProduct Explanation
[...]};

%Constraints
FaMaAttributedFM REQUIRES Attributed;
Explanation REQUIRES Reified;
```

3.2 Using FaMa Console

FaMa console is a recommended way to give the first steps to learn the scope and capabilities AAVM provides. You only need to download the last available distribution and execute the main Java jar file and the console will be automatically launched. Example 11.3 shows an example of interaction that loads the FM in Example 11.2, validates it and counts the number of products the FM describes.

Example 11.3. Interacting with FAMA FW through its console

```
C:>java -jar FaMaSDK-1.1.0.jar
Welcome to FaMa shell
$>load fama-fm.fama
Loading model...
Loaded!!
$>valid
Model is valid
$>#products
Number of products: 87240
```

3.3 Using FaMa Façade

FaMa FW is also a Java library that can be integrated in third-party tools through *FaMa Façade*. It hides FaMa FW insides offering a simple facade to interact in a question–answer manner. Since a change on it will make all the coupled products change, the facade must provide a stable set of interfaces that mainly consists of question interfaces and a feature meta-model at choice. One of the advantages of using this facade is that new versions of reasoners, reasoner selectors and meta-models may reach end-users with no adaptation on their applications since there is no need to couple to them.

Example 11.4. Java code to analyse the previous FM

```
// Instantiating FAMA Framework facade
QuestionTrader qt = new QuestionTrader();

// Loads a FM from a file
VariabilityModel fm = qt.openFile("fama-fm.fama");
qt.setVariabilityModel(fm);

// Validates the FM and then counts its products
ValidQuestion vq = (ValidQuestion)
                        qt.createQuestion("Valid");
qt.ask(vq);
if (vq.isValid()) {
```

```
      NumberOfProductsQuestion npq =
                (NumberOfProductsQuestion) qt.
                            createQuestion("#Products");
      qt.ask(npq);
      System.out.println("The number of products is: "
                + npq.getNumberOfProducts());
  } else {
      System.out.println("Your  feature  model  is  not
valid");
  }
```

The façade is also available as an OSGi service providing the same interface the facade does.

4 Results and Lessons Learned

After 4 years of development, 12 FaMa Extension projects and 7 products that are used by 20 institutions, we have learned many things regarding PL development [7]. Those conclusions that have surprised us most are summarised next and we expect they serve to build other SPLs:

- *PL is a growth concept rather than specific architectures*: Many people explore books for finding the silver-bullet architecture for PL. We do not have to search for brand new architectures for PL, but using whole-life architectures and apply innovative PL management procedures.
- *Analysing core and variant helped on defining a stable core*: We built FaMa FW aiming that transferring new results in AAVM does not mean delivering new and incompatible versions every month. A thorough study in AAVM commonalities and waiting for the right moment when we had all the needed background to make stable decisions helped on defining stable interfaces that have suffered no changes since their definition.
- *Deploying brand new features transparently to end-users is possible*: In our context there are much functionality that just improves the performance of our tool such as reasoners, transformations and reasoner selectors. We have defined a solution that delivers new features to end-users while their existing systems suffer no change at all.
- *Open-source and SPL are compatible concepts*: open-source helps on disseminating FaMa FW. Managing a SPL is complex; even more if third party contributors are involved in the project. Using an adequate architecture reduces coupling among projects and allow reducing the collateral effects new projects might produce. However, critical parts such as the core and facade must be under the control of only one party to ensure the maintainability of the SPL.
- *Using tools instead of processes*: dealing with a large amount of components is not a trivial task. Incorporating Maven and SVN have reduced a large amount of

errors we had in the beginning to synchronise project versions by hand. We have also used the own FaMa FW to build tools to manage our SPL such as FaMa Benchmark to compare the performance of our reasoners, FaMa Test Suite to test every reasoner prior a delivery and FaMa itself to analyse dependences among variant features.

5 Outlook

The area of automated analysis has recently reached 20 years of existence [2]. Although the area is mature enough and technology transfer for production is ready (FaMa is an example on this direction), some challenges remain open and will be explored and leveraged in the following years. One of them is the introduction and exploitation of attributes inside variability models such as cost, time, versions, etc. This will bring a new generation of automated analysis tools. FaMa already includes some of those features but some investigation and practical use cases are need to complement the current state of the practice.

We will see in the future also how analysis tools will be integrated into product line development ecosystems: the research community has mainly built variability model analysis tools as stand-alone prototypes that have hardly been integrated as part of large-scope CASE tools. FaMa has also made some steps forward but more will come in the future.

A key issue in variability model analysis tools is performance. The community is investigating this but new results will be released in the following years. The BeTTy framework is one of our main contributions in this direction. Refer to FaMa's web page at http://www.isa.us.es/fama for any further information.

References

1. BeTTy framework. http://www.isa.us.es/betty/
2. Benavides, D., Segura, S., Ruiz-Cortés, A.: Automated analysis of feature models 20 years later: a literature review. Inf. Syst. **35**(6), 615–636 (2010)
3. OSGi Service Platform, Core Specification. OSG Alliance – OSGi Specification
4. Moskitt Feature Modeler. http://www.pros.upv.es/mfm/
5. Cetina, C., Pelechano, V., Trinidad, P., Ruiz-Cortés, A.: An architectural discussion on DSPL. In: SPLC (2), pp. 59–68. Lero International Science Centre, University of Limerick, Ireland (2008)
6. Trinidad, P., Ruiz-Cortés, A., Peña, J., Benavides, D.: Mapping feature models onto component models to build dynamic software product lines. In: SPLC (2), pp. 51–56. Kindai Kagaku Sha Co. Ltd., Tokyo, Japan (2007)
7. Trinidad, P., Muller, C., García-Galán, J., Ruiz-Cortés, A.: Building industry-ready tools: FAMA Framework and ADA. In: WASDeTT-3 (2010)

Chapter 12
pure::variants

Danilo Beuche

What you will learn in this chapter
- *The underlying concepts of the pure::variants tool*
- *Examples using pure::variants*

1 Introduction

This chapter provides a comprehensive description of the tool pure::variants. The pure::variants tool [1] is a commercial variant and variability management suite. Its goal is to provide a uniform way to express and relate variability and variant information throughout the life cycle of a product line. It has been designed to complement existing development tools and provide necessary links between those tools in order to support efficient development of variant-rich systems. Thus pure::variants does not replace existing tools such as requirements management tools, IDEs, or test tools but extends them.

2 pure::variants Overview

This section is split into a brief overview of pure::variants based on a product line workflow, a description of the underlying concepts of pure::variants, the provided variability meta-model, and the tool architecture.

The following walk-through gives a rough outline how pure::variants can be used in a holistic approach. It introduces some key terms used in pure::variants and

D. Beuche (✉)
pure-systems GmbH, Magdeburg, Germany
e-mail: danilo.beuche@pure-systems.com

R. Capilla et al. (eds.), *Systems and Software Variability Management*,
DOI 10.1007/978-3-642-36583-6_12,

use a fictive example to illustrate possible use cases along the product line development life cycle.

2.1 A Brief Walkthrough

Variability and variant-related information is produced and used throughout the development of product line core assets and the use of these assets in product development. Variability does not start at architecture level and does not end with source code. It covers all phases of product (line) development from portfolio and product management over engineering to testing and product deployment. In general, standard development tools do not provide the necessary means to handle variability and connect the related aspects to a common concept for expressing and exchanging variability and variant knowledge. The pure::variants tool suite has been designed and developed to close this gap by complementing existing tools to enable efficient product line development.

In a typical pure::variants use case portfolio and product managers express the intended and planned product commonalities and variabilities in *feature models* by assignment of features to products in *variant models*. Each variant model is used to capture a single product's configuration. This can be done in pure::variants directly or by using company-specific spread sheets in an application like Excel for instance. In the latter case pure::variant models are generated and updated from/to external data sources using its integrated synchronization capabilities.

The walkthrough starts at the specification level. In order to achieve reuse for requirements among products in a product line, requirements shall be selectable based on the needs of the individual products. Feature models like the one shown in Fig. 12.1 can be used to add variation points to the reusable requirements through an integration with the respective requirements management tool (e.g., Rational DOORS, Microfocus CaliberRM, PTC Integrity Requirements ...). Figure 12.2 shows a sample requirements document in DOORS which is annotated with variability information used by pure::variants to generate product variant-specific requirement specifications from the reusable "master" requirements.

Usually the detail level of the product definitions provided in pure::variants by the product management's feature models is not sufficient for a full product configuration. Thus additional variability models are created and connected to the existing models. This allows for stepwise refinement of variability and variant information and allows separating ownership of related information fragments.

For example, product Line architects can connect flexible architecture descriptions expressed in UML or DSL models to variability models in pure::variants. In case of UML, pure::variants provides off-the-shelf integration with a number of leading UML tools like Rational Rhapsody, Enterprise Architect, and most EMF-based tools like Papyrus, Topcased, and Rational Software Architect. Figure 12.3 shows a simple UML state machine annotated with pure::variants restrictions to mark variation points in UML.

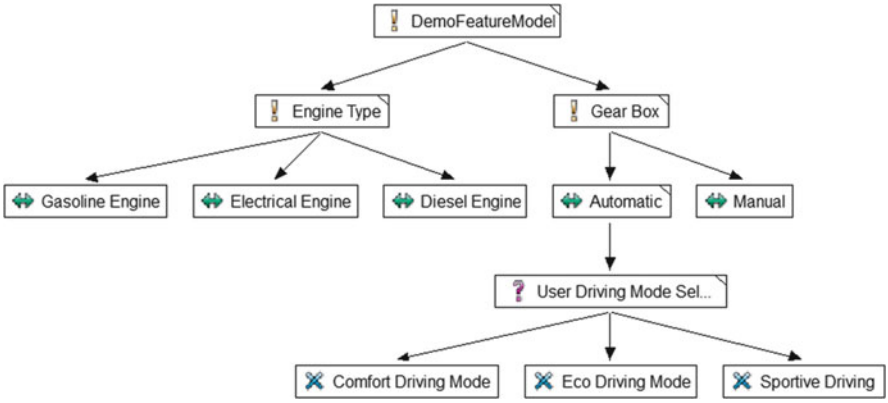


Fig. 12.1 pure::variants feature model sample

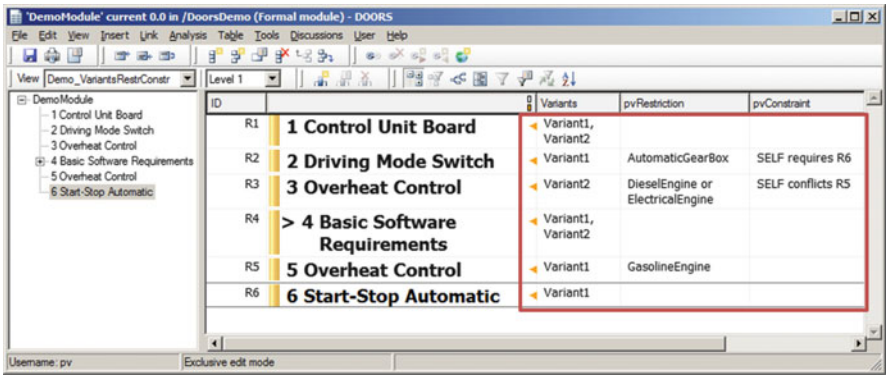


Fig. 12.2 Requirements reuse in DOORS with pure::variants

On source code level, software developers can attach selection rules and/or processing instructions to source code assets, configuration files, etc. For instance, Fig. 12.4 shows a feature model (left) and a family model representing source structures (right). The highlighted boxes on the right show rules which link the existence (i.e., selection) to the existence of the boxed features on the left.

If the provided out-of-the-box processing capabilities are not sufficient, pure::variants allows to create new custom *transformation modules* to generate code, e.g., using the integrated Java or JavaScript interfaces or by running external programs.

Last but not least, tests can be handled like any other artifact in pure::variants. This means that based on feature selections pure::variants derives variant-specific test plans and also configures tests if necessary. Furthermore pure::variants can link test results and change requests to its model elements for a number of lifecycle tools such as PTC Integrity, Rational ClearQuest, Jira, or Bugzilla. This gives the users a quick way to check the state of the product line and product variant development.

While in several cases variability information can be stored directly in the assets (for instance in UML model as profiled constraints, see Fig. 12.3), this is not always

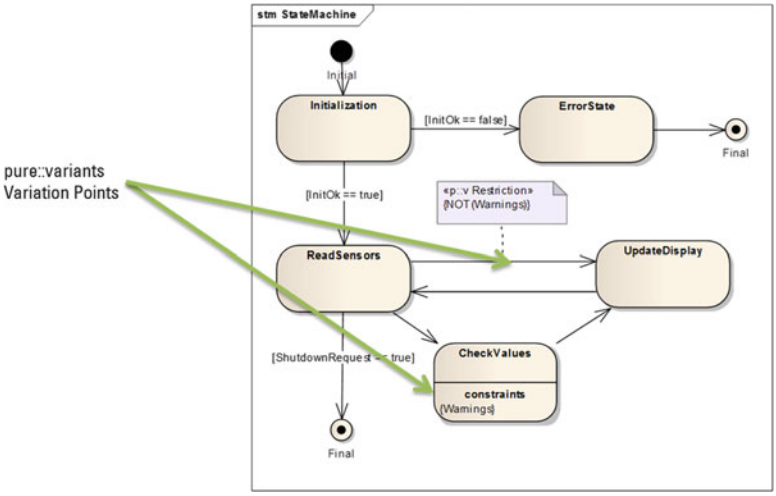


Fig. 12.3 Representing variation points in UML with pure::variants

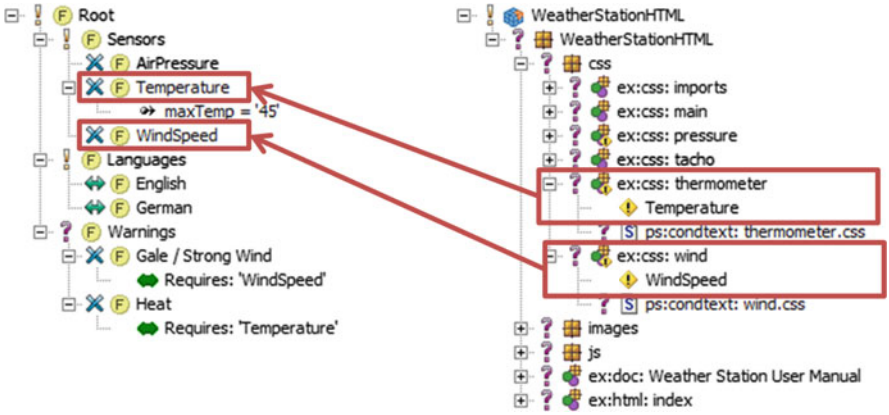


Fig. 12.4 Connecting assets with features in pure::variants using restrictions

possible. Therefore pure::variants provides a generic model type called *family model* for that purpose. For instance, in case of source code assets it can be used to store rules for selecting the proper file set (see marked rules in Fig. 12.4). In the pure::variants EMF integration, for instance, family models are used to store information about variation points in EMF models and thus can support any EMF meta-model without changes to the original meta model.

Of course, since introduction of product line engineering often does not start on a green field and happens incrementally, often pure::variants is deployed only in a specific activity of the development process such as management of source code assets. Due to the modular nature of pure::variants model structures, the set of activities in which pure::variants is used can be extended easily.

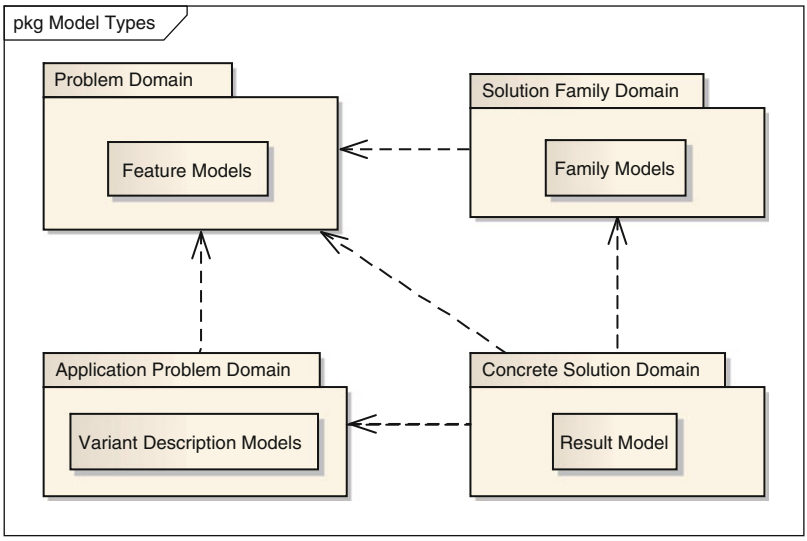


Fig. 12.5 pure::variants model types and main dependencies

2.2 Basic Concepts

The pure::variants tool uses a uniform meta-model to express variability in the problem space (*feature models*) and solution space (*family models*) and a related meta-model to describe variant configurations (variability instantiation information), the *variant description models*.

The derivation/instantiation of variant description models creates instances of the feature and family models which no longer contain variations, i.e., describe a single variant. The instances are represented in so-called *variant result models* and can be further processed either internally by pure::variants' integrated model and asset transformation or used by external tools (Fig. 12.5).

Since all pure::variants meta-models are generic by nature, they can (and have to) be tailored to represent variability-related information from different domains. In practice this means that pure::variants' extensions for tool connections usually add information to the meta-model, for instance, to introduce specific element types representing tool-related assets.

All artifacts (models, additional assets) in pure::variants are organized in *Variant Projects*. A product line may be composed of multiple variant projects containing data, e.g., separated by the ownership or development activity the assets relate to.

Projects are either stored using a file-based approach or a database approach. In case of file storage, sharing models happens using external version control systems such as Subversion, Mercurial, Rational ClearCase, etc. In case of database storage, pure::variants provides an integrated version management. In both cases model compare and merge is supported with a graphical user interface.

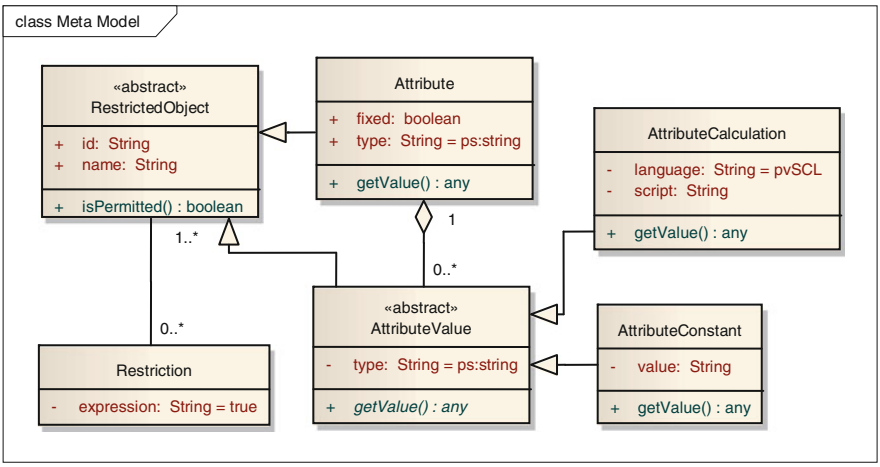


Fig. 12.6 Attributes in the pure::variants meta model

2.3 *Variability Meta Model*

The pure::variants meta-model is an evolution of the original FODA feature models, providing additional semantics such as default selection state, freely definable variation group cardinality, arbitrary attributes, element types, or constraints in various forms.

The predefined variation group cardinalities are provided according to the common feature model concepts: optional, alternative, or and mandatory elements, If the standard cardinalities are not sufficient, the freely defined cardinalities allow definition of sets of arbitrary integer range definitions such as "1,[4-8],10."

The attributes concept (see Fig. 12.6 for a structural overview) allows users to define attributes with constant values, with values selected by rules (called restrictions) and with calculated values. Opposed to fixed attributes (values/calculations defined at feature/family modeling time, actual value is selected/calculated automatically at variant configuration time) the values of variable attributes have to be entered at variant model configuration time to allow variant-specific user-defined values.

Directed relations provide a simple way to express element-to-element relations such as *conflicts* and *requires*. In addition to relations, textual model constraints can be used to express complex configuration knowledge. Constraints can query all relevant properties (element existence, attribute values) of the related models. Currently pure::variants provides two constraint languages: a Prolog-based dialect and pvSCL, a subset of OMG Object Constraint Language (OCL).

2.4 *Tool Architecture*

The concrete internal architecture of pure::variants (see Fig. 12.7) is complementing the meta-model so that it provides the necessary flexible yet powerful and efficient infrastructure for handling complex variant management projects.

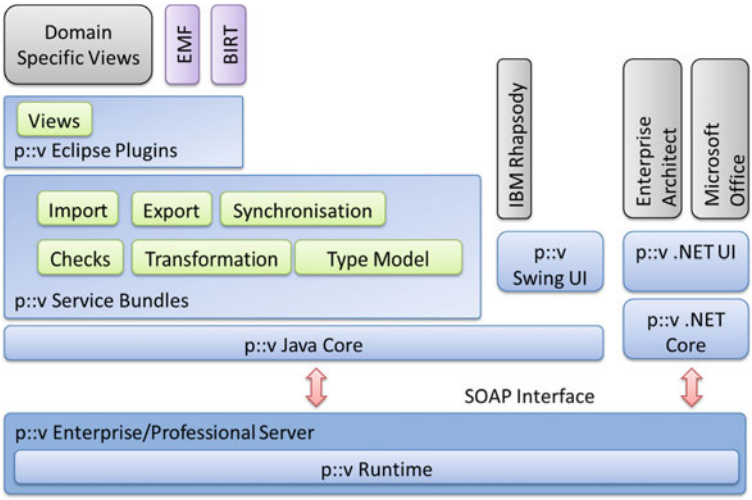


Fig. 12.7 pure::variants internal architecture

The pure::variants architecture is a client/server architecture where the server is responsible for model management, access control, and model storage. The server is accessed by clients sharing a common core API for both the Java language and the .NET language. The main user interface is implemented as a set of Eclipse plugins. The Eclipse infrastructure is also used to provide a large set of extension points and service interfaces. This allows users to add complex views for models, consistency checks, transformation modules, importer, and exporter. pure::variants also provides interfaces to other elements in the Eclipse ecosystem such as BIRT (reporting) and the Eclipse Modelling Framework (for reading and writing pure:: variants models in EMF-based applications). For integration with other tools which are not based on Eclipse, pure::variants provides a set of APIs and components based on Java (Swing) and Microsoft .NET which can be easily embedded into standard tools. These APIs provide a similar user experience as the main Eclipse client. The API of pure::variants' Java core and the Eclipse plugins are public and can be used by anyone to extend or customize pure::variants.

3 Examples and Recommended Areas of Practice

pure::variants is being used in a wide range of domains and use case. In most cases it has been used in the area of embedded software development projects in domains such as automotive industry, automation, mobile phone software, or consumer electronics. It has been used also for research purposes by a number of institutions, e.g., for testing [2] or dealing with qualitative measures for product lines [3].

In most industry projects the number of features in a project ranges from a few dozens to several hundreds. The number of variation points in total is usually in the range of four to ten times the number of features in the project.

Publically documented industrial applications of pure::variants include an automotive product line using model-based development with MATLAB Simulink and pure::variants [2] and the application of pure::variants as part of a product line migration for frequency converter products [3].

Naturally, the performance of pure::variants has an impact on the use cases in which it can be successfully applied. The tool can handle large models with several ten thousands of elements even on today's typical desktop computers/laptops easily. Thus since in almost all cases this is above the required amount of elements, pure::variants does not impose a technical limitation. And in cases of extreme variability, often structuring into a hierarchy of variability domains solves performance issues in a natural way.

4 Results and Lessons Learned

The following paragraphs give some insights based on experiences with pure::variants in industrial applications.

4.1 Integration Concepts and Capabilities

A key factor for the quick acceptance of pure::variants turns out to be the deepness of integrations. Since pure::variants does not replace traditional development tools, the level of integration and ease of use of the integration with the existing tool chain is crucial. Especially in complex scenarios and/or if users are rarely confronted with variability information maintenance and use, a deep integration is the best fit. An example is the pure::variants for Simulink integration where most activities (creation of variation points in models, assignment of elements to variation points) can be done inside MATLAB Simulink. This decision was based on experiences with users of a predecessor version of this integration. In this special case the current integration provides lesser functionality in terms of variability modeling capabilities than the former pure::variants Connector for Simulink but fits the existing workflows much better.

However, this integration also provides a way to maintain almost all data within pure::variants. The variability modelers thought it to be more natural to keep everything in pure::variants so that all variability information can be kept in pure::variants. This created challenges in terms of keeping data consistent between two different worlds.

4.2 PLE Adoption Strategies

Since green field starts are a rarity in current product line engineering projects, pure::variants has to provide the necessary concepts to support migration of existing assets into product line assets. Especially in migration scenarios where parts of the whole development are migrating only product by product, this creates a number of challenges for tools. Taking in new products, which are clones of others, with automatic generation of variation points from it right is possible. However, this requires in many cases investment in creation of the proper tool (extension) for the specific use case and also manual refactoring as part of the migration.

4.3 Scalability and Performance

For most activities users demand interactive or near interactive operation of pure::variants. However, variability information can grow very quickly, and not everything which works well for small models can be applied for larger models such as automatic problem solving (BDD and SAT solvers have exponential growth of resource usage). Users have to make a compromise between available functionality and size of data being handled. Users do not always seem to accept the (unavoidable) loss of functionality and/or performance when the amount of information to be handled grows.

4.4 Out-of-the-Box vs. Flexible Customization

Another key point to learn was that almost no two customers have similar enough tool chains and processes. Therefore almost all integrations contain more or less complex customization capabilities in terms of the data model, deployed variation point concepts, etc. One-right-way-to-do-it solutions usually break with the second or third customer.

However, it is also obvious that too many customization points slow down the speed of migration.

4.5 Complexity vs. Simplicity

For most customers pure::variants provides more capabilities in terms of modeling flexibility and ways to express and use variability and other information than required. This leads to certain barriers when starting with pure::variants, since a lot of decision whose consequences are not yet known have to be made.

5 Outlook

The pure::variants suite shows how the concept of variability modeling based on feature models can be applied throughout all activities of the product line development. Integration with existing tools and support of incremental migration are some of its strength.

Future developments will not only bring more and more powerful integrations with development tools but also improved support for migration of legacy systems to product lines and better support for collaboration and coordination of the concurrent domain and application engineering.

More information about pure::variants itself plus demonstrations and tutorials for its integrations as well as a freely downloadable Community Edition can be found in [1].

References

1. pure::variants homepage. http://www.pure-systems.com/pv
2. Oster, S., Zorcic, I., Markert, F., Lochau, M.: MoSo-PoLiTe: tool support for pairwise and model-based software product line testing. In: Proceedings of VAMOS 2011 (2011)
3. Bartholdt, J., Medak, M., Oberhauser, R.: Integrating quality modeling with feature modeling in software product lines. In: Proceedings of ICSEA 2009, pp. 365–370 (2009)

Part III
Industry Experiences

Chapter 13
Philips Healthcare Compositional Diversity Case

Frank van der Linden

What you will learn in this chapter

- *The complexity of our systems leads to a hierarchical, local, and heterogeneous variability management solution.*
- *The present state of the art both in research and in tooling is still lacking good solutions supporting this.*

1 Introduction

The aim of this chapter is to provide a comprehensive description of the issues on variability management within the development of the product line of Philips Healthcare. Many issues originate from distributed and heterogeneous development. Solutions are sought in localization and hierarchical variability.

The remainder of this chapter is structured as follows in Sect. 2, the context of the product lines within Philips is described, especially it shows the organizational and business context of the product line development. In Sect. 3 the evolutionary growth of the product line development towards a large distributed development is sketched. It describes the distributed development process—inner source—for sharing of product line knowledge and assets. Section 4 describes several diversity aspects originating from the distributed organization, the evolution, and the incorporation of third-party assets. Important elements are hierarchy and localization of variability. Although no definite solution for local variability is available, some theory developed 15 years ago is used to illustrate how localization will support distributed variability management. In Sect. 5, an example shows how local variability management was used for TV-set development. In Sect. 6, the variability

F. van der Linden (✉)
Philips Healthcare, Eindhoven, The Netherlands
e-mail: frank.van.der.linden@philips.com

R. Capilla et al. (eds.), *Systems and Software Variability Management*,
DOI 10.1007/978-3-642-36583-6_13,

issue at Philips healthcare is described. Unlike the TV-set development, an own proprietary tool implementation is no option. A heterogeneous hierarchical solution is found, including default configurations at several levels in the hierarchy. It is explained that different mechanisms are needed at different levels and parts of the hierarchy. In addition, default configurations are available at several levels of the hierarchy. These originate from the use of different levels of abstraction and from the needs of different disciplines. Especially several domain-specific languages are in use. In Sect. 7, variability management supports for the own domain-specific languages at different places in the hierarchy. The underlying mechanisms are similar, but these languages differ for each set of components they apply to. Proprietary tooling is available to couple them. In Sect. 8, several measured advantages of the present set of methods, tools, and techniques are provided. In Sect. 9, the achievements are summarized. Finally in Sect. 10, an outlook is given. It provides some information in the way the Philips platform evolves. In addition it provides some information for possible future directions of research for local variability management.

2 Industrial Context

Philips is a global company with a focus on health and well being. It is a global leader creating value through meaningful innovations and improving lives with sense and simplicity. Philips Healthcare is a division of Philips that provides products in medical imaging systems, home healthcare solutions, patient care, and clinical informatics, and it provides services around these systems. The healthcare division of Philips is almost 100 years old. In 1998 the development was done at two sites. Since then it has grown to globally 34 sites in 2011. Presently software development within Philips Healthcare involves more than 2,000 developers worldwide. This development is structured around product groups, although there is an increasing need for cross product group reuse. Consequently, Philips Healthcare has initiated in 1996 a product line initiative to provide the core functionality to all application groups. An account on this can be found in [1], Chap. 15. The core functionalities of the platform developed by a separate domain engineering group are: storing, retrieving, distribution, processing, and viewing medical images in 2D, 3D, and 4D. However, presently the platform also provides other functionality that support medical diagnosis, treatment, and measurements.

After several years of product line introduction, the domain engineering group became the bottleneck for innovation. Many departments were asking new or updated features, and the limited size of the domain engineering group was not able to deal with all these requests in time. The solution was found in the application of open source principles within the company: *inner source* [2, 3]. This involves open access by all application developers worldwide to all development information and source code of the platform software. This use of inner source was successful: application engineering groups are involved in domain features that are

important for them. It led to lower pressure for domain engineering and faster deployment of new features.

Within the platform development of Philips Healthcare, variability development and configuration are done distributed and heterogeneous, a practice that is already described elsewhere. For distributed variability and configuration, i.e., there are several places that introduce variability and similar for configuring, see [4]. For the heterogeneity, i.e., use of diverse mechanisms for different parts of and no common global variability model, see [5]. Commodification [6] leads increasingly to components from third parties to be integrated, each of them having their own variability model [7, 8], leading even to more heterogeneity. Several partial solutions for this are proposed in the literature. For instance, Elsner et al. [7] propose an environment for configuring different models from different vendors. Several others, [9–11], propose different ways to compose several feature trees. Some papers describe the solutions for keeping the variability local to components [12, 13]. However, a practical solution is not yet available. In this chapter, the present heterogeneous situation is sketched and the main issues are discussed.

3 SPL Approach

Within Philips Healthcare there is a central group providing domain engineering. They deliver the Medical Imaging Platform (MIP) determining the architecture, interfaces, information models, common components, methodologies for dealing with common quality issues, and tooling, including configuration, collaboration, and test tools. More than 20 application engineering groups around the world use this platform. Some of these groups make imaging products for a diversity of modalities: ultrasound, X-ray, or MRI. Some of these groups make imaging products aimed at different markets for diagnosis or intervention. Some groups make monitoring systems, e.g., measuring and ECG signals from the heart. Other groups make hospital IT systems dealing with remote viewing, diagnosis, advanced image processing, or storage, retrieve, and exchange of the images.

The development is supported by virtual teams that are set up, involving people with a similar organizational role from different departments. These teams deal with all kinds of cross-cutting issues, such as diverse aspects of the architecture, information models, tooling, change control, roadmaps, requirements, and tests.

The MIP architecture is a product line layered reference architecture standardizing component middleware such as the GUI, workflow, data handling, image rendering, printing, reporting, and field service. The MIP platform mission is to run anywhere with excellent interoperability. To enforce the platform stability, the architecture determines many interfaces and several information models. The dark boxes in the layers represent interfaces. Components in these layers support several of these interfaces. One of the standard interfaces is a configuration interface that allows setting values to diversity parameters. The most important layers are (bottom to top; see Fig. 13.1):

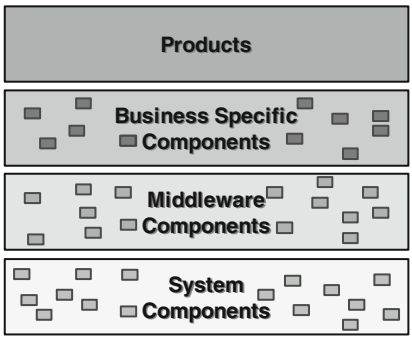


Fig. 13.1 MIP—high-level architecture

- *System components*—these are mainly third-party components providing the basic system services, including operating system and UI primitives. This layer is constantly growing, including increasing third-party solutions for quality aspects, or configuration tooling. In addition, this layer shields away the details of these components and gives a uniform access to system resources, independent on its implementation.
- *Middleware components*—these are mainly in-house developments, but the amount of third-party components is rising here as well. These components provide the basic functionality described above: image processing, distribution storage, rendering, viewing, measurements, UI widgets, and icons. This layer also provides basic standardized quality solutions for several important qualities, such as remote maintenance support, availability, security, and privacy. Within this layer several integrated components (IC) are defined that encapsulate a default configuration of a group of related components.
- *Business-specific components*—these are mainly in-house components' healthcare-related services. These involve integrated support for specific procedures, including the workflow support, and screen layouts. In addition this layer has tooling for configuration support of components of the level below. For several subsystems there are graphical, domain-specific languages that describe which sets of components can be connected, and in which ways. These languages are connected to tools that support actual configuring; they are discussed in Sect. 7. Also this layer provides several integrated components that encapsulate a default configuration of a group of related components.
- *Products*—these are the final applications, combining several services of the business-specific components' layer together. These products are mainly developed by the application engineering groups. However, the domain engineering group provides a "semifinal" Philips Medical Workspot (PMW) that is a system on its own, involving a default configuration of lower layer functionality.

In each layer, the components group and organize components at lower layers to perform specific functionality. Each layer has a subdivision of internal layers as well.

Application engineering groups discuss their roadmaps with the domain engineering group. This leads to a roadmap for domain engineering, prioritizing the new features in, and adaptation to the platform. In addition, the domain engineering group needs to serve problem reports and change requests from the application engineering groups. As the number of application engineering departments is very large, the platform team cannot fulfill all change requests in time. To deal with this situation, an *inner source* model is introduced [2, 3]. The essence is that all available information from all domain assets is available. This includes binaries, source code, configuration tooling, test scripts, test results, and documentation. If an application needs a new feature, there are several ways to obtain it

- If it is part of a platform release, it can be used immediately.
- If it is already in development, the user can take the component, the department can be a testing site for it, or even it can help further development of the software, increasing its development speed.
- If there is no component available, the department can issue a change request, and if nobody else takes it, the department can do the development itself.

The platform defines the reference architecture for all applications. It defines a Hierarchical structure of subsystems, layers, and components. The hierarchy is a key in comprehending the architecture and its details. It is an important carrier for communication between different teams. The hierarchical structure defines units of reuse and for the distribution of work.

Variability management is supported by a collection of default configurations that are organized hierarchically along the layers. The medical imaging platform consists of a hierarchy of components that can be configured separately. However, for related components, at a given layer in the architecture, integrated components (IC) deliver default configurations. These default configurations are combined in a default way into the PMW. Figure 13.2 gives a simplified architecture involving configurations. For the MIP, it shows only the *Business-specific* layer and some products in the *Product* layer. Components are depicted as dark boxes. Integrated components are shown in dashed shapes combining several components. Arrows mean the use of an (integrated) component including the assignment of configuration values. Note that an integrated component can have configuration parameters as well. Configuring an integrated component leads to consistent set of configuration values for the involved components.

Application groups can use the PWM unaltered, meaning that they do not have to configure the lower layers; see *Product 1* in Fig. 13.2. The developers only need to provide configuration values at the PMW level. However, the PMW configuration may be too restricted for some products. Each developer is allowed to override the default configuration and select other (integrated) components. Configuration tooling helps to select the right parts. In addition the inner source tooling helps to disseminate configuration knowledge via discussion forums. For instance, *Product 2* in Fig. 13.2 needs another integrated component and also another configuration of a components selected by the PMW. Note that circumventing the PMW is also possible for (integrated) components at the *Middleware* layer.

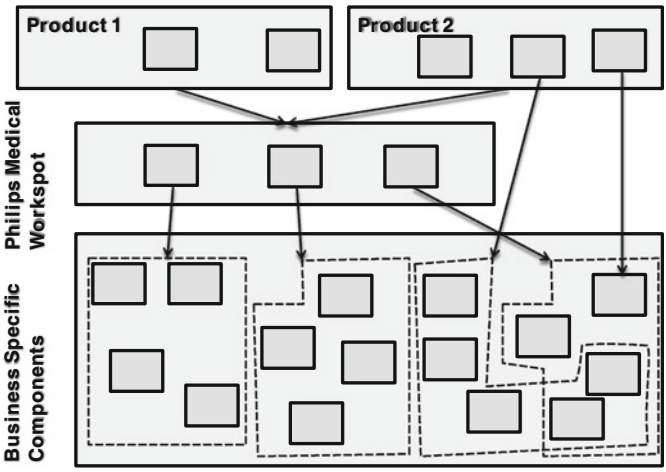


Fig. 13.2 MIP semifinal: Philips Medical Workspot

4 Diversity, Hierarchy, and Localization

Variability is just one aspect of *diversity*. It is important to deal with all aspects of diversity in a coherent way. These different aspects are [12, 13]:

- *Commonality*, what is common to all applications
- *Variability*, how can the applications differ
- *Configuration*, consistent selection of variants for an actual application

Commonality and variability define the scope of the product line. These are determined and designed during domain engineering. Configurations are designed during application engineering. However, it is an important task for domain engineering to support application engineering, to provide variation management, and to ease the configuration activity. Variation management involves the availability of diversity models, tools, methods, and default configurations.

As the product line architecture is often hierarchical component based, the components, at different layers of the architecture, play an important role in the diversity. A component often has its own diversity—incorporating its own commonality, variability, and configurations. To enhance comprehensibility and configurability, the component should hide parts of its diversity and abstracts away from it [4]. For instance, the component diversity can internally be described in resource use (memory, processing, bandwidth, etc.) externally it may show diversity only in terms of services provided. This means that ideally there should be abstract diversity interfaces between components, showing the internal diversity in a reduced and abstract form. However, for such interfaces, no solution proposed up to now is satisfactory. Several papers, mentioned in the introduction, discuss this issue. They propose a configuration link between different feature models [9, 11],

having just a single feature as interface [10] or model fragments [5]. However, none provides convincingly both abstracting and hiding.

An additional concern in a hierarchical environment is that both top-down and bottom-up configuration should be possible [4, 12]. Development and configuring proceed partially top-down and partially bottom-up. Top-down configuring, from architecture to lower layer component, is necessary when dealing with concerns of system (quality) requirements and the exploitation of commonality. Bottom-up configuration, from existing component to complex compositions, is needed when dealing with concerns of existing legacy or COTS assets, subdisciplines, distributed development, or system (hardware) parameters.

Local diversity is a means to decouple configuration decisions from each other, and enables to start configuring everywhere in the structure allowing top-down, bottom-up, or even middle-out configuration procedures [13]. Having diversity local was a prerequisite to the notions of *product populations* or *composable* systems instead of decomposing them [14].

We ourselves have proposed a component-based diversity model [13]. At that time, there was not much standard notation available for dealing with diversity. In any case, abstraction was served by the restriction that a variant at a certain level of the hierarchy is only related to variation points at the level below. This leads to invisibility of any variant or configuration at the level below. Hiding was served by *local diversity*: only relationships exist between the diversity at succeeding levels of the hierarchy, i.e., between a component and its subcomponents only.

Figure 13.3, based on [13], describes the way local variability can help in a hierarchical system. This figure is adapted to the UML and OVM notation, the original was drawn before UML became standard software design notation. At the top row a component at a certain level in the hierarchy is shown, at the bottom its subcomponents. At the left-hand side the commonality is presented, and at the right-hand side the variability. The right–left direction is the direction of increasing the abstraction and hiding. The left-hand side provides commonality information, in this case structural UML diagrams. At the right-hand side, the variability is shown. In this case, it is an extended UML diagram, incorporating all variants and a connected OVM model. We can imagine other format here as well.

At the top left-hand side, we see that component A has three subcomponents: B, C, and D, in a certain configuration and call structure. Also external interfaces are represented. At the right-hand side the variability of the component A is shown. It has two variants of subcomponent B, each with a different call structure. The variability in the subcomponents is not visible at the commonality level of A. At the bottom left of Fig. 13.3, the commonality of each of the subcomponents is shown. This should incorporate enough information to develop A. Each of the subcomponents has their own variability, which is here only partially shown for B_1.

Local diversity keeps the design within a hierarchy comprehensible. If all diversity would be present in a single model, then each level of the hierarchy multiplies the complexity of this model. Different levels in the hierarchy will have different parts of the variability, at their own abstraction levels. However, in practice, for lower levels, only commonality information is needed. Similarly, for

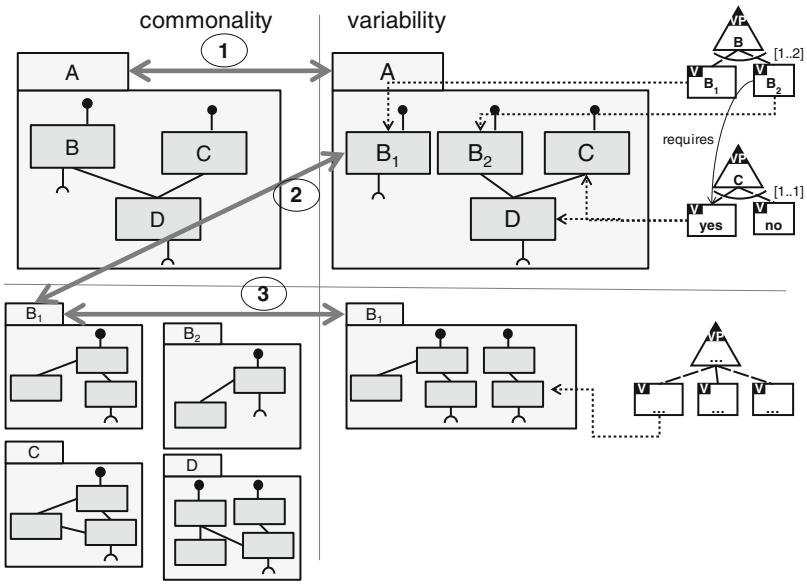


Fig. 13.3 Local variability and hierarchy

higher levels, only specific variants may need to be known. In addition, in a single model, it may be hard to share common parts of the diversity model, as there are no clear "module" boundaries. Localization implies that in case of shared subcomponents, the diversity has to be designed only once.

Localizing also supports both top-down and bottom-up development. In a top-down way, we proceed according to 1→2→3, in Fig. 13.3. Start with a top level component. Develop its commonality, next its variability, and subsequently develop the subcomponents. In a bottom-up way, we proceed according to 3→2→1, in Fig. 13.3. Start with several low level, existing, components for specific task. They together determine the variability at their level. Extracting the commonality makes them available to be used is several configurations, which lead to variability of a higher level component. Similarly, configuration can proceed top-down following the 1→2→3 path, and bottom-up following the 3→2→1 path.

Figure 13.3 is just an ad hoc notation, improving the one of [13]. To make it really useful, several questions need to be answered. For instance, for the boxes at the left-hand side, we need "*commonality of the diversity model*," whatever that may mean. The UML diagram only provides a possible view on this. There are several questions to be answered: is it only an abstract (smaller) version of the right-side model? Does it contain features, variation points, or variants, or something else? Can it be used as a diversity interface? How to interpret the links in Fig. 13.3 for diversity models both for left–right (1, 3) and top-right–bottom-left (2)?

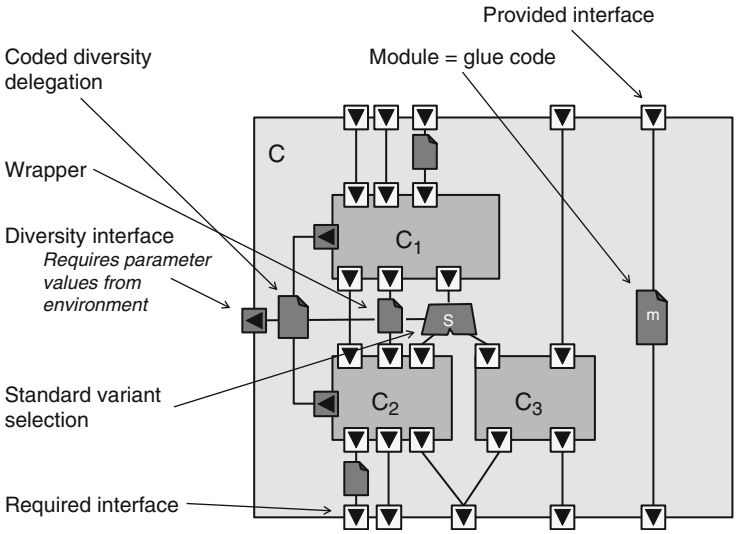


Fig. 13.4 Koala component—with subcomponents

5 Koala

The ideas of the previous section found their way into Koala [14, 15]. This was used in the development of TV-sets within Philips. Also in Koala hiding is performed by linking variability only between components and subcomponents; see Fig. 13.4 for a typical Koala component diagram.

Koala components have several provided and required interfaces. These are shown as a square with a triangle, pointing in the direction of the call. The component implementation consists of *modules*, which are pieces of code, without formal interfaces, like "m" in Fig. 13.4. Any relevant code of the component will be in modules, including glue code to translate between interfaces. Part of the functionality may be delegated to subcomponents, with interfaces—"C_i" in Fig. 13.4. Pieces of code may be used as wrappers to translate between interfaces; they appear at several places in Fig. 13.4. A specific diversity interface is used to exchange information between a component and its surrounding component. A diversity interface is like a normal interface. It has function calls to extract diversity parameters from the surrounding component. The parameters in these function calls are to communicate internal diversity parameters to the external component. The main mechanism to provide abstraction translates values between subcomponent and component diversity interfaces. A module may be present to translate the internal variability values into those that are presented at the interface. For specific situations a standard translation mechanism is present: the variant selection module "S" in Fig. 13.4 selects between the call of either "C_2" or "C_3" based on the value of a specific variable.

Koala abstracts away from binding times. A component developer does not need to know when external diversity, or the diversity of subcomponents, is bound. Koala computes the configuration at compilation time, evaluating as much as possible the function calls. As certain (or all) parameters are defined to be constant, such static evaluation may give definite values for complex expressions and fixed selections for branches. The Koala pre-compiler will replace the expressions by these values and discard code for nonselected branches. This leads to very compact code, which is an important requirement in TV-set development. Evaluation at compilation time is not restricted to diversity interfaces, leading to very compact executable code, which was extremely important at the moment Koala was developed. Especially the diversity interface calls are used to select certain subcomponents. In practice, this means that only the relevant subcomponents will be selected for compilation and binding, again leading to minimal code. In cases that require dynamic selection of subcomponents, the pre-compilation evaluation will not lead to a specific choice of subcomponent to be selected. In that case, specific "if"-statements or branches will be generated for selecting at run-time, and all involved subcomponents will be compiled and linked.

In the example of Fig. 13.4, the variant selection module "S" needs a Boolean value to select between calls to "C_2" or "C_3." This value is obtained via the diversity delegation module from the diversity interface of the component "C." If the environment provides a fixed value "false," the only calls to "C_3" will be selected, and "C_2" will not be part of the configuration. In the case that the value cannot be computed at compile time, both components will be integrated. Note that a fixed value may be originating from a fixed value in the surrounding component, but it may also be based on calls to other components in the configuration.

The component developer does not see any difference between binding at compilation or at run-time. For instance, for the variant selection module "S" in Fig. 13.4 when the value of the variable is computable at compile time, only one of the modules C_2 or C_3 will be compiled and linked. If the value cannot be determined at compile time, both are compiled and linked, and the generated "if"-statement selects which call should be made.

6 Variability at Philips Healthcare

For Philips Healthcare, the Koala solution was not usable as its maintenance could not be outsourced and third-party solutions could not be easily integrated. However the locality of diversity, as explained above, was necessary to keep the diversity manageable. A study was performed using the orthogonal variability model (OVM) [16] for variability modeling. OVM provides a low-level decomposition of the diversity model. In fact each variation point and its variants can be seen as separate variants. However, diversity constraints can be laid between any pair of variation points and/or variants. There is no mechanism provided for information hiding or abstraction.

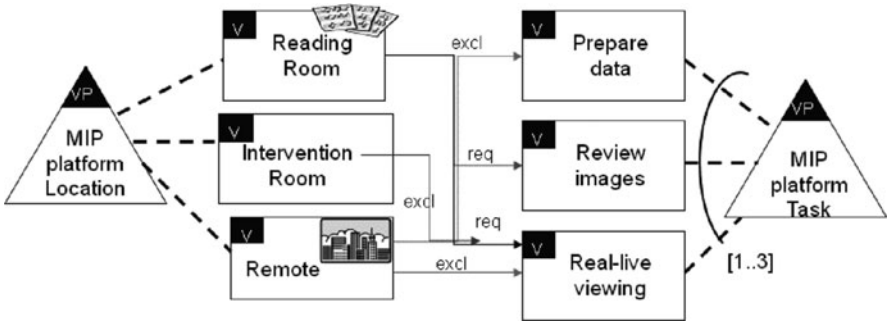


Fig. 13.5 OVM model of imaging workflow

For the platform the .Net framework is used to ease connecting components at run-time. In principle, binding above component level is done at initialization time or at run-time. For this the standard .Net tooling is available. In this way the components can be, and are, developed independent of each other. Therefore their diversity is also developed independent from each other, and only part of the diversity is exposed to external components. In most cases a component has a list of configuration parameters, for which standard tooling is made available for setting them in a consistent way.

As the MIP platform is very large, we selected for the OVM case study a service at the *Business-specific component* layer. This service deals with the hospital workflow of making images of a patient. The most important variation points are related to components in the *middleware components* layer, and are the following:

- *Location*—dealing with the place where the viewing will be performed. Examples of variants are: intervention room, control room, office, and remote.
- *Tasks*—dealing with the activity the workflow supports. Examples of variants are: prepare data, review, real-live viewing, and archiving.
- *Environment*—dealing with the system in which the service works. Its variants are related to the kind of imaging equipment involved, its hardware, and operating system.
- *Language*—dealing with the language of the people working with the equipment. Example variants are US English, German, and Chinese.
- *User*—dealing with the role of the person executing the workflow. Example variants are clinician, technician, and radiologist.

Unfortunately the model became too complex, as there is too much interrelation between variants and variation points, for instance, in Fig. 13.5, a part of this model dealing with location and task. It has two variation points and six variants, but it already involves six requires/excludes relationships. It was diagnosed as that there is too much implicit knowledge available in the system and architecture, and a refinement of variability is necessary. It can be observed that the relationships chosen here are related to the *products* layer, whereas the model deals with the

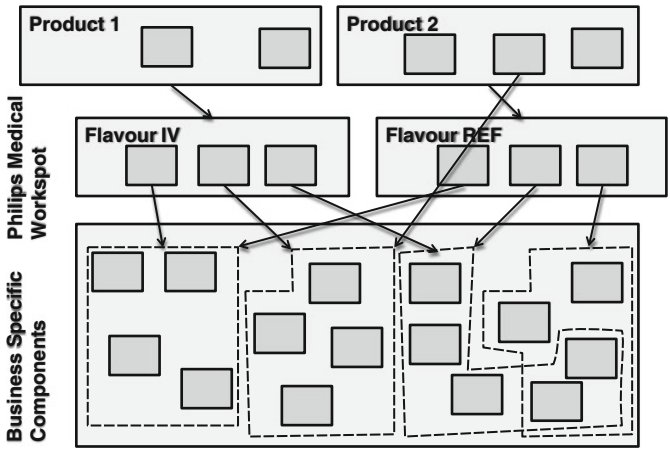


Fig. 13.6 Platform flavors—common diversity

business-specific layer in the architecture hierarchy. However, it is unclear how to avoid this situation.

Other means of modeling and local diversity were used to solve this problem. We have defined a diversity hierarchy; see below. In addition default configurations were defined at many levels in the hierarchy. These are the integrated components and the PMW itself, which are described above. In a way the default configurations can be regarded as the *common diversity* at that level of the hierarchy. This also expresses itself in the situation that there are several integrated components involving the same components, and even the PMW has several flavors; see Fig. 13.6, showing several semifinals to be used by different products.

Configuration can be done at each level for the hierarchy with elements of the level below. But if there is a default configuration available, this can be used unaltered. Bottom-up default, or standard, configurations are built. These can be combined in higher level configurations. However, to keep the platform flexible, default setting may be overridden top-down, traversing the hierarchical layers. This is also shown at the left-hand side of Fig. 13.6. Special tools, often supporting domain-specific languages, are used to ease configuration for a specific element in the hierarchy. Here, heterogeneity comes in. Different elements at the same hierarchical level deal with different aspects of the system leading to different domain-specific languages. However, the languages are similar, and the same proprietary tooling is used for all of them. Each language can configure dedicated groups of components, and only certain connections are allowed. These connections relate to actual binding of interfaces; for more details see below.

A growing part of the platform consists of third-party components. Such a component often comes with its own configuration mechanism, adding to the heterogeneity. However, locality helps to focus only on the neighboring elements in the hierarchy. At any time, during development and configuration, only a small set of configuration mechanisms are relevant.

Note that the heterogeneity is not regarded as a problem. Heterogeneity might be seen as an increase of complexity. However, it decreases complexity, as only the configuration parameters and mechanisms that are needed for the specific part of the system, at the right level of abstraction, need to be modeled. If a uniform solution was selected, a single monolithic model has to be used to deal with any local configuration issue. This also relates to experiences in other technical fields. For instance, in the car industry, a wheel has different variation mechanisms and parameters than a dashboard, but the car manufactures are able to deal with this.

For local diversity, see Sect. 4, the architecture hierarchy needed to be refined. Each architecture layer has its own components, calling components at lower layers in the hierarchy. However, each component has its own internal configuration parameters, and it is related to a set of specific other components acting together for a specific piece of functionality. We have called these sets of components a *component suite*. These are collections in between components and subsystems. An integrated component is usually a configuration for a specific component suite. If we look at the hierarchy for variability, MIP identifies to at least the following set of layers:

1. Component parameters—diversity: enumerated lists, strings, Boolean, or numeric values.
2. Components—350 different components in MIP. Diversity: often a configuration file with <50 parameter-value pairs.
3. Component suite—18 different default configurations, each with about 50 parameters, called *integrated components*. Diversity often solved via a domain-specific language; see Sect. 7.
4. MIP platform—several (<10) default platform configurations for different kind of applications, called *platform flavors*. Diversity solved via a domain-specific language; see Sect. 7.
5. Application product line platform—several (~30) in different application domains. Diversity solved by own configuration means.
6. Application product—many (>1,000) products. Diversity solved by own configuration means.

Each of these will be discussed below. Note that layers 1–4 deal with the domain components, layers 5 and 6 are at the application engineering site, where often own platforms are developed and components are introduced, on top of the MIP platform (cf. Fig. 13.6). This means that there may be more levels at the application site hierarchy.

Note that in the list above we see that at different levels different mechanisms are used for variability management. This has several causes. Different levels of the hierarchy deal with different levels of abstraction. At component level these can be usually easily expressed as configuration values for simple types. However, at different levels of the hierarchy, the configuring has to deal with more or less complex subcomponent configurations. Consequently, domain-specific languages are more expressive to determine the configurations. As several subsystems have different disciplines involved, and different aspects or different abstraction levels

are relevant, there is a need for several domain-specific languages. Finally the development (especially at levels 5 and 6) is distributed, and partially based on legacy code. It takes too much effort, with too low added value, to fix a single mechanism for all. Of course, every department is free to change their own configuration mechanism. Especially a move towards a domain-specific language is encouraged. As people performing configuration at a certain level in the hierarchy may have problems with lower layer configuration mechanisms, it is crucial to have default configurations. Using them will lead to a workable solution. If the quality, like performance, is not good enough, fine tuning by configuring at limited parts of lower layers is a second step, leading to situations as shown in Fig. 13.6, Product 2.

Configuration tooling is part of the platform. Therefore, the application engineering groups can apply the same mechanism (component parameters, domain-specific languages, and integrated components) for their own systems.

7 Domain-Specific Languages

The diverse domain-specific languages in use are supported by domain-specific editors. These are all part of the platform. Figure 13.7 shows a screen shot of the editor for one of the languages. Although not visible in Fig. 13.7, different groups of components are distinguished by their own color, and their role in the configuration is also expressed by the default position of the component in the layout. The relevant component lists, *asset base*, are available in a hierarchical view at the left-hand side. Additional views are available for setting parameters (*component properties*), whenever a component is selected.

Configuration, from scratch, proceeds by dragging components from the asset base into the configuration pane. Connections are drawn, and are only allowed if interfaces match. For each component, the specific parameters may be set any time it is selected. In most cases, configuration does not proceed from scratch. Instead an earlier version of the configuration is used, or an integrated component. A variation of the configuration is made by adding or removing components, redirecting connections, or setting the configuration at components lower in the hierarchy.

The results of such a configuration will be a component at a higher level in the hierarchy. It can be used in this way to build up higher levels. Note that the default configurations are available as inputs to the editor. This means that when adaptations are necessary, the developer can still proceed from the default configuration, adapt it, and save it. The domain-specific language can be easily translated in executables code, since the configuration mainly consists of linking calls between components. In this way, the developer can have fast validation of the adaptations.

Whenever a lower level of the hierarchy need to be (re-)configured, the relevant editor can be called via the selection of the appropriate element. The resulting editor may be either a new domain-specific editor or an editable parameter-value list. For instance, in Fig. 13.7, at the right-hand side, an editor shows the attributes of a

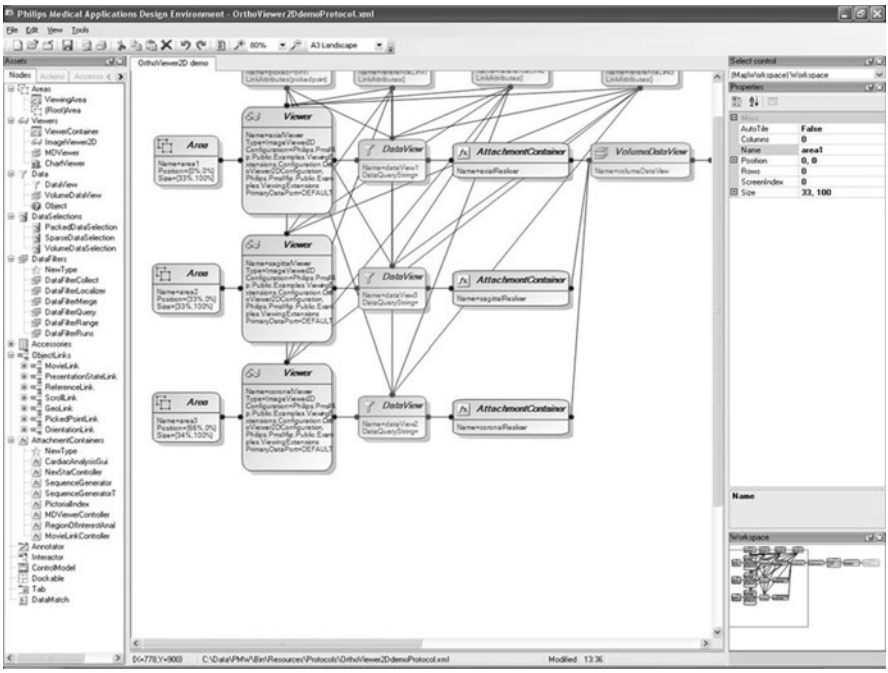


Fig. 13.7 Domain-specific editor

selected component. Certain of these components are interface to a group of lower level components, e.g., a component suite. In that case, an editor of that specific configuration can be called via the corresponding element in the component property list. The domain-specific languages and editors are mainly proprietary build, as there was no commercial solution available that fulfills the hierarchy requirements. The editors may spawn an executable version of the configuration, which means that the configuration can be tested immediately, without leaving the editors. This eases reconfiguration as unwanted behaviour can be repaired easlity.

8 Cost and Benefits

The benefit of software product line engineering is high. Without it we would not have been able to produce in time the amount of diverse products we presently have, with the right quality. As we are working already on software product lines for more than 15 years, it is difficult to quantify. Note that the platform and its architecture are in continuous evolution. These are continuous actions on the increase of the quality and also on the integration of third-party solutions, freeing our own resources for innovation.

In any case, several metrics on the development performance are measured continuously, giving insight in our own performance. At least the following results are obtained in our development:

- To build a reusable component, it costs about 1.6 times the effort for a component with the same functionality for a single product. This means that reuse already pays off if there are two or more users. Since most platform components are used by more than ten departments, the development effort is reduced by a factor of 6. This also incorporates the own single product components in the different application groups.
- Since 1996, we were able to integrate several additional companies that are now presently users of the MIP platform.
- The time-to-market is reduced by more than 50 %, since the platform functionality only need to be configured, and both application and domain engineering development can concentrate on new features.
- The product defect density reduces much faster than before, e.g., <50 % lower after 6 months than without software product line engineering. This is caused by the fact that the platform has many diverse users leading to many different tests.
- The maintenance cost is reduced by more than ~60 %, because the products have many similar parts, the maintenance personnel can serve many different products, and changes in new products are usually not disruptive.
- The products have a common look and feel, since they are all build on the same user interface components. This serves the clients, since they do not need to train the operators for each product anew.

9 Results and Lessons Learned

After 15 years of product line development in Philips Healthcare, it is the only way to proceed. More than 20 groups are dependent on the MIP platform, and they can only efficiently build products if the platform configuration is comprehensible. As the amount of third-party software in our products is growing, configuration has to deal with external diversity mechanisms, in cooperation with those of our own.

Over time a heterogeneous, hierarchical, and local variability management has grown. This is the only way to deal with the complexity of the platform and the large amount of applications that are built on it. Most of the configuration tooling is still own proprietary, since there is not yet a good commercial offering for it. Different mechanisms are used at different levels of the hierarchy. These originate from the diverse background of the development population, but also from the different levels of abstraction and the needs of the diverse disciplines. Tooling exists to connect configuration at several levels in the hierarchy, whenever necessary. This is not always needed, as there are several default configurations, aiding higher level configuration.

Configuration is both done in bottom-up and top-down ways. Bottom-up, default or standard, configurations are built that can be use unaltered at higher layers. Top-down, specific parts of a pre-configuration can be accessed and altered according to specific wishes.

10 Outlook

The MIP platform is in continuous evolution incorporating more third-party elements, and it is to be expected that configuration support will increasingly rely on third-party solutions as well. In order to deal with the configuration complexity, a hierarchical, local, and heterogeneous solution is crucial. Diverse levels of the hierarchy need diverse abstractions to work with. In addition, several disciplines have their own requests. This situation will stay, and only become more complex, not less. In particular, monolithic solutions will not work.

As can be observed in the discussion above, there is not much information about the interfaces between diversity at different layers in the hierarchy. These interfaces are needed to link the different levels of abstraction involved. However, we do not have a good idea how such interface should look like, and there is not much publication in this field. In the solutions used within Philips, the translation between levels of abstraction is mainly performed through actual code without many guidelines. This is a good issue for future research, as we may need more automation to navigate through the diversity hierarchy.

References

1. Van der Linden, F., Schmid, K., Rommes, E.: Software Product Lines in Action. Springer, Heidelberg (2007)
2. Van der Linden, F.: Applying open source principles in product lines. Upgrade **X**(3), 32–40 (2009)
3. Wesselius, J.: The bazaar inside the cathedral: business models for internal markets. IEEE Softw. **25**(3), 60–66 (2008)
4. Schmid, K.: Variability modeling for distributed development – a comparison with established practice. In: Proc. 14th SPLC'10, pp. 155–165. Springer (2010)
5. Dhungana, D., Grünbacher, P., Rabiser, R., Neumayer, T.: Structuring the modeling space and supporting evolution in software product line engineering. J. Syst. Softw. **83**(7), 1108–1122 (2010)
6. Van der Linden, F., Lundell, B., Marttiin, P.: Commodification of industrial software: a case for open source. IEEE Softw. **26**, 77–83 (2009)
7. Elsner, C., Ulbrich, P., Lohmann, D., Schröder-Preikschat, W.: Consistent product line configuration across file type and product line boundaries. In: Proc. 14th SPLC, pp. 181–195. Springer (2010)
8. Schirmeier, H., Spinczyk, O.: Challenges in software product line composition. In: HICSS, pp. 1–7 (2009)

9. Abele, A., Johansson, R., Lönn, H., Papadopoulos, Y., Reiser, M.-O., Servat, D., Törngren, M., Weber, M.: The CVM framework – a prototype tool for compositional variability management. In: Proc. VaMoS 2010, ICB-Research Report No. 37, University of Duisburg Essen, pp. 101–105 (2010)
10. Czarnecki, K., Helsen, S., Eisenecker, U.: Staged configuration through specialization and multilevel configuration of feature models. Softw. Process Improv. Pract. **10**, 143–169 (2005)
11. Reiser, M.-O.: Core concepts of the compositional variability management framework (CVM). Technische Universität Berlin, Forschungsberichte der Fakultät IV – Elektrotechnik und Informatik, Bericht-Nr. 2009-16, ISSN 1436-9915 (2009)
12. Haber, A., Rendel, H., Rumpe B., Schaefer, I., Van der Linden, F.: Hierarchical variability modeling for software architectures. In: Proc. SPLC 2011, pp. 150–159 (2011)
13. Van de Hamer, P., Van der Linden, F., Saunders, A., Te Sligte, H.: An integral hierarchy and diversity model for describing product family architecture. In: Springer LNCS, vol. 1429, pp. 66–75 (1998)
14. Van Ommering, R.: Building product populations with software components. Ph.D. dissertation, University of Groningen (2004)
15. Van Ommering, R., Van der Linden, F., Kramer, J., Magee, J.: The Koala component model for consumer electronics software. IEEE Comput. **33**, 78–85 (2000)
16. Pohl, K., Böckle, G., Van der Linden, F.: Software Product Line Engineering. Springer, Berlin (2005)

Chapter 14
Variability in Power Plant Control Software

Masami Okamoto, Makoto Fujii, and Yoshihiro Matsumoto

What you will learn in this chapter

- *We study how to successfully develop automatic start-and-stop control system families for steam power plants.*
- *The formalization of the system semantics based on frequent field analysis concerning operational modes, events, and other data of the operated devices.*
- *The basic schemes and the major features of the software product line described in this industrial experience.*
- *How we evaluated the ROI of our software product line.*

1 Introduction

The automatic start-up of large-scale fossil fuel-type power station from the cold plant state until to the state where synchronization of rated generating capacity to the power grid completes, accompanied by its shutdown, has been implemented by TOSHIBA Corporation in 1968 at Hachinohe power station of Tohoku Electrical Company (250 MW). This was the first automated power station in the world. Lately, most utility companies are required to furnish electric power utility for both base load and variable load. In daily operations of those utility companies, the amount of generated electricity must be adjusted so as to meet variation of daily power demands in timely manner each day. For example, the peak load within a day reaches as much as twice the minimum load during the day.

M. Okamoto (✉) • M. Fujii
Fuchu Complex, Toshiba Corporation, Fuchu-shi, Tokyo, Japan
e-mail: masami.okamoto@toshiba.co.jp; makoto2.fujii@toshiba.co.jp

Y. Matsumoto
The ASTEM Research Institute of Kyoto, Shimogyo-ku, Kyoto City, Kyoto, Japan
e-mail: yhm@mvg.biglobe.ne.jp

R. Capilla et al. (eds.), *Systems and Software Variability Management*,
DOI 10.1007/978-3-642-36583-6_14,

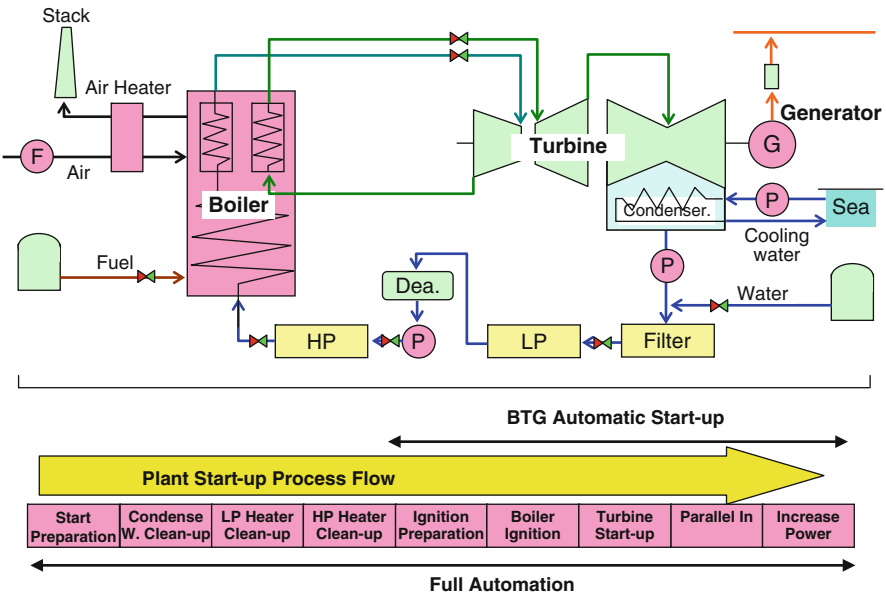


Fig. 14.1 Schematic illustration of power plant and plant start-up process flow

To cope with those circumstances, it is required that (1) the reduction of start time (the time span covering boiler ignition, steam-turbine acceleration, various equipments buildup, and generator synchronization: see the lower part of Fig. 14.1), (2) the reduction of stop time (the time span covering generator disconnection, turbine stop, and boiler extinguishment), (3) the reduction of the number of the operating personnel, (4) the reduction of start-up loss (the energy loss in fuel, electrical power, and water flow spent during plant start-up), and (5) the enhancement of equipment lifetime by reducing mechanical stress can be implemented.

In order to meet those requirements, an automatic plant start-and-stop control system (hereafter called APSS system) was developed as one of the key solutions. For putting the manufacturing of the APSS systems to commercial base, we developed the framework, which enabled tailoring of application systems from a variability-based software product line, called Electric Power Generation Software Product Line (EPG-SPL) [1, 2]. This software product line realizes the customization of variability by the interpretation of rules described with using a domain-specific language, called Domain-Specific Language for Electric Power Generation (DSL-EPG). EPG-SPL has been utilized by the EPG application development teams to generate the application system for every power station that was ordered by different electric power utility companies not only from Japan but also from worldwide. The number of power stations built by the EPG-SPL adoptions by the year of 2010 amounts more than 150 including up to 1,000 MW power stations.

Our accomplishment has been recognized by the participants of Software Product Line Conference in 2007, and EPG-SPL was inducted into the Product Line Hall of Fame of Software Engineering Institute/Carnegie Mellon University in 2008.

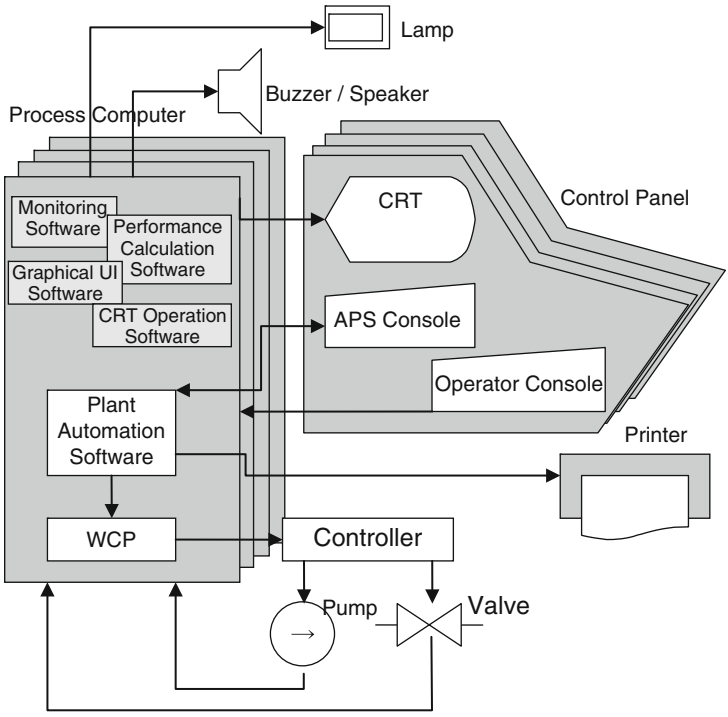


Fig. 14.2 Plant monitoring and control system

2 Background

In the fossil fuel-type power stations, a set of boiler, steam turbine, electric power generator, and several other subsystems are connected as shown in Fig. 14.1.

Figure 14.2 illustrates the configuration of a plant monitoring control system (only a part is shown), which highlights the automatic control of a pump and the valve associated with it.

The plant monitoring control system shown in Fig. 14.2 is connected in fault-tolerance mode with a set of ten units of embedded type computers, and graphic operators' consoles. Each embedded computer is connected with a number of device controllers and actuators through industrial Ethernet (100 Mbps).

The graphic operators' console serves the following functions:

- Remote manipulation of plant control devices with using touch screen and mouse
- Monitoring of plant status through graphic displays
- Audio–visual guidance for plant operations
- Audio–visual annunciation in case of plant abnormality

– Recording of plant events and status
– Plant performance monitoring

The major challenges that we encountered when we started APSS are summarized in the following:

1. Complexity of plant dynamics and indeterminable plant disturbance: The APSS's design, devised in the 1960s, was a theory-based one, so to speak, for example, main steam pressure value at some condition can be presumed with using the values of fuel flow and feed water flow time. However, theory-based presumption often failed because of the complexity of system dynamism and external disturbance that we frequently suffer even in regular operating conditions. In order to resolve these circumstances, we devised what we call "man-simulation" scheme, where "man" means a plant operator, and computer simulates operator's behaviors in "man-simulation" scheme. The plant operations can be interpreted as the sets of rules, where each rule defines a relationship between a particular plant status, events, conditions, and operator's actions. APSS memorizes all these rules by learning operator's behaviors. In the execution stage, APSS monitors plant status, events, and conditions in real-time base. Using the monitored results and the memorized rules, APSS selects one of the operator's behaviors that it learned previously and controls the target plant.
2. Customer's collaboration: The collaboration by the customer, especially the cooperative participation by the plant operators, was crucial to formally describe operating procedures and actions at each plant start-and-stop step. In order to establish solid bridge between plant operators and APSS developers, formal field analysis described in Sect. 3 was conducted. The results of the field analysis were presented with using the formal documents called "role description cards." The responsibility for the role description cards was shared both by the customer, plant operators, and APSS developers. The role descriptions are transformed to the descriptions in DSL-EPG as is described in Sect. 4.

3 Development of EPG-SPL

3.1 Field Analysis

Preceding the development of DSL-EPG, the field analysis, detailed in the following, was conducted to classify plant properties into invariants and variants.

EPG plant is usually controlled through a central control room, where operators' teams work on a rotating schedule. A team, in each shift, consists of one duty supervisor, three group leaders, and several operators, organized as shown in Fig. 14.3.

The duty supervisor monitors and supervises plant-operation processes, makes high-level decision with regard to plant conditions, and provides direction to group

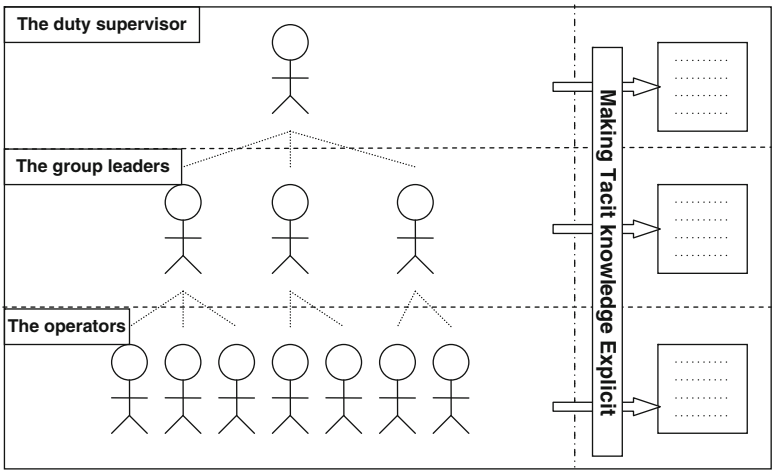


Fig. 14.3 Power plant operators' team

leaders. The group leaders, each of who is responsible for a plant subsystem, check up equipments and devices accommodated in the responsible subsystem and ask operators to start operating steps.

3.2 SPL Approach

The field analysis for the development of DSL-EPG was made through the following steps:

1. Developers of APSS system interviewed operators' teams ten times, each after plant start-up operation at the target power station, involving selected duty supervisors, group leaders and operators, based on the operational guidance prepared in the power utility companies. The result of each interview was recorded in the formalized role description cards, which includes monitored variables, plant and environmental conditions, and operational timing, procedures, and actions that were undertaken.
2. The data collected and recorded in the formalized cards are classified according to the roles of the supervisor, group leader, and operator, to bring out the timing and logic charts exemplified in Figs. 14.4 and 14.5.

In the first part of Fig. 14.4, an exemplified logic diagram is shown, which is used to identify plant master status (PMS) with the measured sensor values. In the second part of Fig. 14.4, master control sequencers (MCS), which are produced by the result of logical additions of one or more PMS values, are shown. For example, MCS05 is produced by the result of logical addition of PMS001, PMS002, and

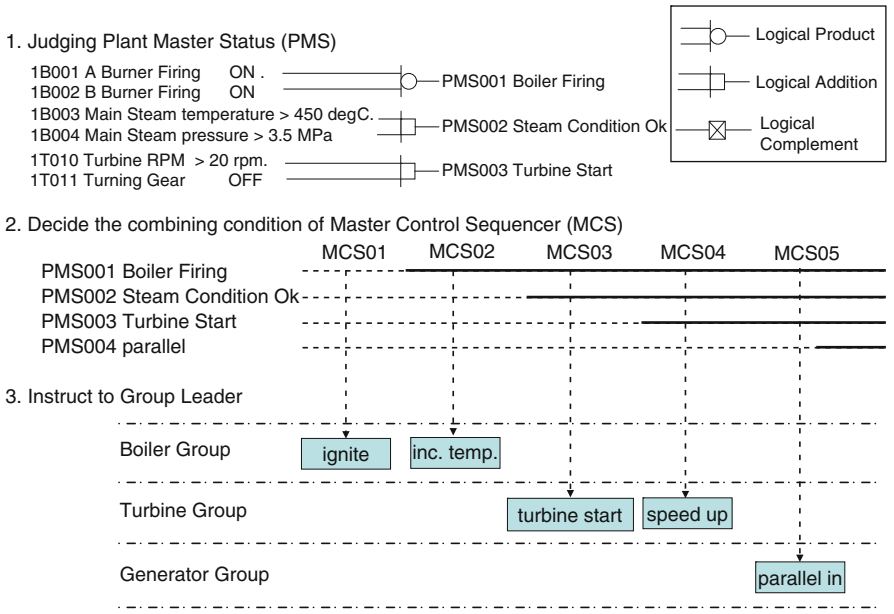


Fig. 14.4 Example of duty supervisor role description

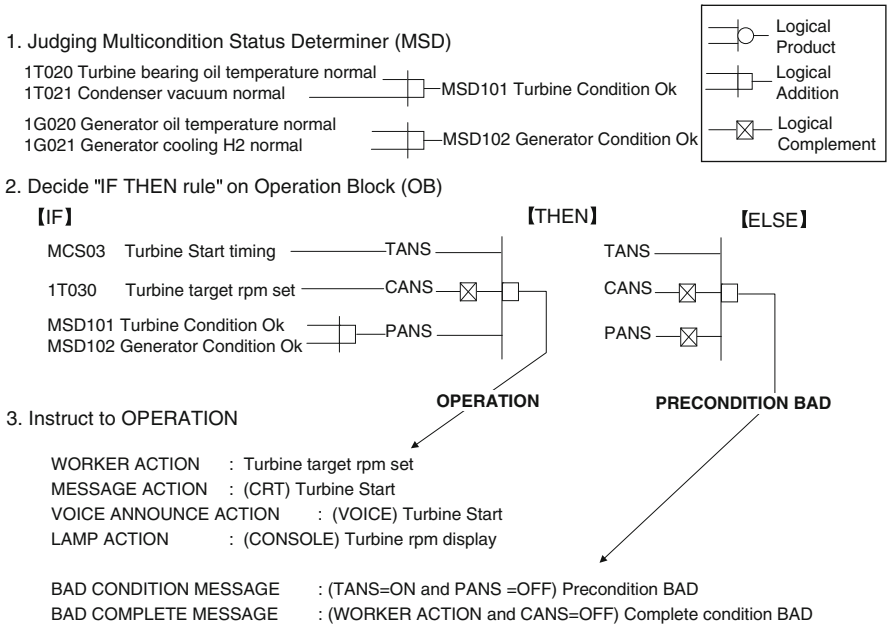


Fig. 14.5 Example of group leader's role description

PMS003. In the third part of Fig. 14.4, the activities to be triggered by the respective MCS are shown.

Each activity enclosed in a shaded box is initiated by the respective MCS. These activities are conducted by the group leaders and operators.

The refinement of activity "turbine start," defined in Fig. 14.4, is illustrated in Fig. 14.5. In the first part of Fig. 14.5, it is shown that MSD101, the logical product of 1T020, and 1T021, which are logical variables, triggers MSD101. In the second part of Fig. 14.5, the "IF-THEN rule" that triggers OPERATION or PRECONDITION BAD is illustrated. Each OPERATION is controlled by the shown logical expression on the logical variables: TANS, CANS, and PANS (TANS corresponds to trigger condition, CANS to post-condition, and PANS to precondition). As shown in Fig. 14.5, OPERATION is refined into atomic actions: WORKER, MESSAGE, VOICE ANNOUNCE, and LAMP. PRECONDITION BAD is refined into several individual messages.

Using the duty supervisor role descriptions and group leader role descriptions shown in Figs. 14.4 and 14.5, how to classify invariants from variants are analyzed. The results of the analysis are summarized as follows:

1. Invariants
   (a) The semantics of the EPG-SPL framework, which is described in Sect. 5
   (b) The logical expression to define conditions for controlling every activities
   (c) The operational semantics for APSS called SCIA, which is described in Sect. 4
2. Variants
   (a) The expressions used to identify variables in PMS, MCS, MSD, and OB
   (b) The measured values to be used in the conditions defined in the logical expressions
   (c) The expressions used to identify actions
   (d) The expressions used to identify messages

Each set of role descriptions is transformed to each table respectively in the way as described below:

(a) The role descriptions of the duty supervisor shown in Fig. 14.4 are transformed to the table called Master Control Status (MCS) and Plant Master Status (PMS).
(b) The role descriptions of group leader shown in Fig. 14.5 are transformed to the table called Operation Block (OB) and Multicondition Status Determiner (MSD).
(c) The role descriptions of operator, not shown in the Figs. 14.4 and 14.5, are transformed to the table called Worker Control Driver (WCD).

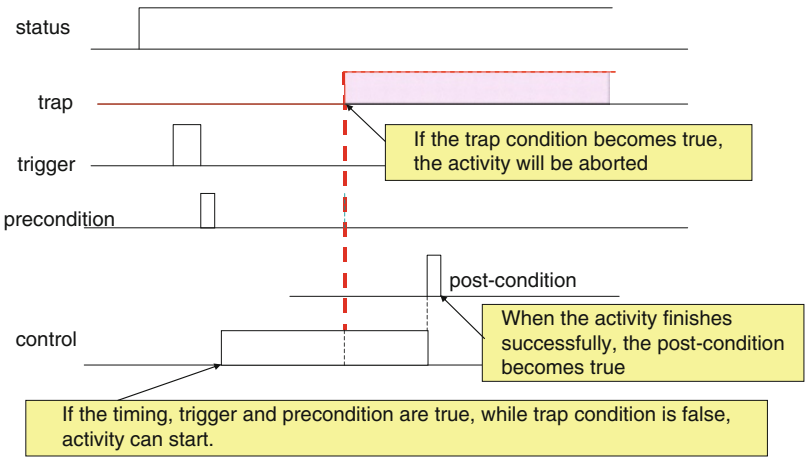


Fig. 14.6 Timing chart [3]

4 APSS Framework

Status, Condition, Interaction, and Control (SCIA) is an operational semantics that satisfies the execution of the component-based software architecture called APSS framework shown in Fig. 14.11. The APSS framework allows building hierarchically structured program components for implementing APSS systems. In SCIA, the execution of every atomic component is concurrent and their coordination is expressed in terms of event-oriented architecture.

- Status is the state that is common to all plant start-up operations such as "Start Preparation," "Condense Water Clean-up," "Low Pressure Heater Clean-up," "High Pressure Heater Clean-up," "Ignition Preparation," "Boiler Ignition," "Turbine Start-up," "Synchronization," and "Increase Power."
- Condition is the logical variable defined by a logical expression that consists of logical variables called events and logical operators. Logical composition of atomic events, each of which denotes a deviation of input value from some threshold values.
- Event is defined by interaction involves plant status, events, conditions, and controls, and defines rules to activate or deactivate controls using logical expressions that comprise status events and conditions.
- Control presents logical expressions to define preconditions, trigger conditions, trap condition, and post-conditions that used to control corresponding actions.

Figure 14.6. illustrates an example of temporal relationship between status, trap condition, trigger condition, precondition, post-condition, and control.

Table 14.1 illustrates an example of the semantic relationships between conditions, variables, rules, and controls. The controller Cont.1521 can be activated or deactivated by the result of logical conjunction between condition Cond.0311

Table 14.1 Conditions for controller initiation

		Rules			
	Variables ID	1	2	3	.....
Conditions	Cond.0311	True	False	True	False
	Cond.0312	True	False	False	True
Controls	Cont.1521	Activates	Deactivates	Activates	Activates

From Matsumoto 2009 [1]. ©2010 Taylor and Francis Group, LLC. With permission

Table 14.2 Definition of controller activities

	Rules	
Activities of: Cont.1521	1	2
Timing event (in regular expression)	$\lambda \times$ e(Cond.0311 $\cap$ Cond.0312)	$\lambda \times$ e(Cond.0311 $\cap$ Cond.0312)
Trigger event	e(^L1)	e(^L2)
Precondition	(^V1 $\cap$ ^V2)	(^V1 $\cap$ V2)
Trap (exit) condition	^(L1 $\cap$ ^L2)	^(L1 $\cap$ ^L2)
Activity	a(V1) $\times$ a(^V2)	a(^V1) $\times$ a(^V2)
Post-condition	L1 $\cap$ ^L2 $\cap$ V1 $\cap$ ^V2	L1 $\cap$ ^L2 $\cap$ ^V1 $\cap$ ^V2

Note: X1, X2, Y1, and Y2 are exampled logical variables. a(·) represents action. For example, a(Y1) represents action to operate Y1. e(·) represents event. For example e(X1) represents event to be activated by X1. λ represents initiation event

From Matsumoto 2009 [1]. ©2010 Taylor and Francis Group, LLC. With permission

and Cond.0312, shown in Table 14.1. The activities to be performed by Cont.1521 are defined in Table 14.2.

DSL-EPG for the APSS system consists of the six kinds of fill-in-the-blank formats, which are Plant Master Status (PMS), Multicondition Status Determiner (MSD), Master Control Sequencer (MCS), Alarm Group (ALG), Operation Block (OB), and Auxiliary Tables.

1. DSL-PMS: DSL for defining Plant Master Status (an example is shown in Fig. 14.7): the logical tables to specify plant master status and to set system flags such as boiler ignition, turbine start-up, synchronization, etc.
2. DSL-MCS: DSL for defining Master Control Sequencer (an example is shown in Fig. 14.8): the decision tables that specify parameters that are used to choose control sequence codes.
3. DSL-MSD: DSL for defining Multicondition Status Determiner (an example is shown in Fig. 14.9): the logical tables that specify parameters used to select processes to be activated.
4. DSL-ALG: DSL for defining Alarm Group definition: the logical tables that specify alarming devices and how to drive those devices.
5. DSL-OB: DSL for defining Operation Block (an example is shown in Fig. 14.10): the logical tables that are used to select the objective WCD and to specify preconditions, post-conditions, and logical expressions.

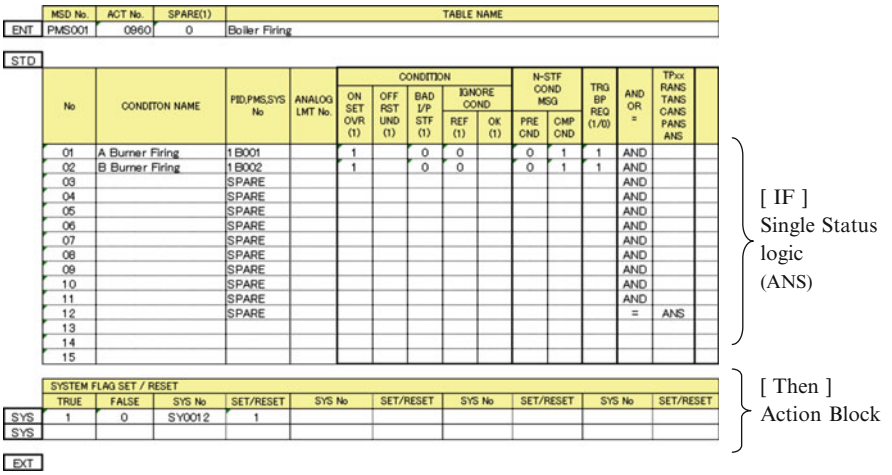


Fig. 14.7 DSL-PMS

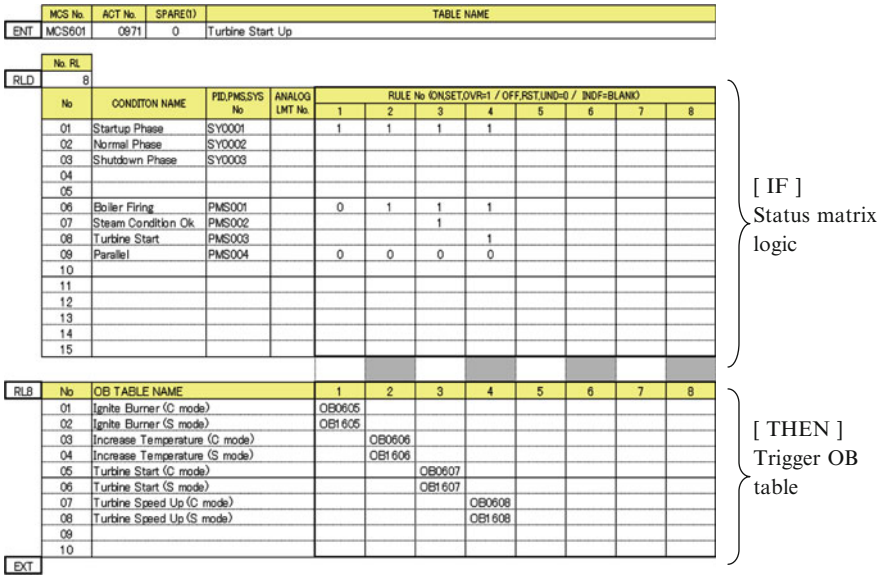
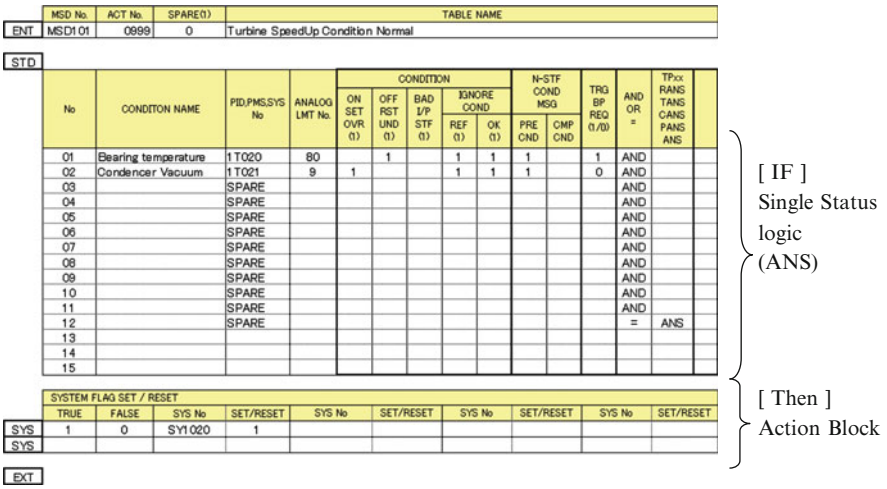


Fig. 14.8 DSL-MCS

6. Auxiliary Table:

   (a) I–O List: Input–Output List
   (b) WCD: DSL for defining control of worker drivers



	MSD No.	ACT No.	SPARE(1)	TABLE NAME
ENT	MSD101	0999	0	Turbine SpeedUp Condition Normal

STD

No	CONDITON NAME	PID,PMS,SYS No	ANALOG LMT No.	CONDITION: ON SET OVR (1)	CONDITION: OFF RST UND (1)	CONDITION: BAD I/P STF (1)	CONDITION: IGNORE COND: REF (1)	CONDITION: IGNORE COND: OK (1)	N-STF COND MSG: PRE CND	N-STF COND MSG: CMP CND	TRG BP REQ (1/0)	AND OR =	TPxx RANS TANS CANS PANS ANS
01	Bearing temperature	1T020	80		1		1	1	1		1	AND	
02	Condencer Vacuum	1T021	9	1			1	1	1		0	AND	
03		SPARE										AND	
04		SPARE										AND	
05		SPARE										AND	
06		SPARE										AND	
07		SPARE										AND	
08		SPARE										AND	
09		SPARE										AND	
10		SPARE										AND	
11		SPARE										AND	
12		SPARE										=	ANS
13													
14													
15													

[IF] Single Status logic (ANS)

SYSTEM FLAG SET / RESET										
	TRUE	FALSE	SYS No	SET/RESET	SYS No	SET/RESET	SYS No	SET/RESET	SYS No	SET/RESET
SYS	1	0	SY1020	1						
SYS										

[Then] Action Block

EXT

Fig. 14.9 DSL-MSD

Inside the APSS framework, seven software components (Process-scheduler, ACP, EAC, BOC, WCP, CIS, and MAS) and three code storages (I/O-code base, plant-code base, and worker-code base), shown in Fig. 14.11, are accommodated. The APSS framework will be included as a constituent of the provisional machine shown in Fig. 14.14.

The roles of those components are as follows:

- Process-scheduler: The role of Process-scheduler (PS) is the mediation between other six framework components and the operating system. PS schedules, monitors, and controls processes to drive processes called ACP, EAC, BOC, WCP, CIS, and MAS.
- Contact Input Scan (CIS): Driven periodically by PS, CIS scans states of switch contacts using the data provided by I/O-code base. Whenever any event (deviation from the defined range or condition) is found, CIS sends action to EAC.
- Multiple Analog Scan (MAS): Driven periodically by PS, MAS scans states of analog inputs using the data provided by I/O-code base. Whenever any event (deviation from the defined range or condition) is found, MAS sends action to EAC.
- Executive Action Control (EAC): EAC is activated periodically in the highest priority, and receives actions sent from CIS and MAS. Accordingly, EAC transfers to ACP, the names of the occurred actions and the identification numbers of OB tables (OB assignment) that are related with the actions.
- Activity Control Processor (ACP): Driven by the messages from EAC, ACP takes the codes of the assigned OB tables from the plant-code base and interprets. As the result of interpretation, WCP or BCO is activated.

	OB No.	ACT No.	SPARE(1)	TABLE NAME
ENT	OB0607	0988	0	Turbine Start

STD

No	CONDITON NAME	PID,PMS,SYS No	ANALOG LMT No.	CONDITION ON SET OVR (1)	CONDITION OFF RST UND (1)	CONDITION BAD I/P STF (1)	CONDITION IGNORE COND REF (1)	CONDITION IGNORE COND OK (1)	N-STF COND MSG PRE CND	N-STF COND MSG CMP CND	TRG BP REQ (1/0)	AND OR =	TPxx RANS TANS CANS PANS ANS
01	Turbine Start timing	MCS003		03								OR	
02		SPARE										OR	
03		SPARE										=	TANS
04	Turbine target rpm set	1 T030		1			1	1		1		AND	
05		SPARE										AND	
06		SPARE										=	CANS
07	Turbine Condition OK	MSD1 01		1					1		1	AND	
08	Generator Condition OK	MSD1 02		1					1		0	AND	
09		SPARE										AND	
10		SPARE										AND	
11		SPARE										=	PANS
12		SPARE											
13													
14													
15													

[IF]
3 Status logic
(TANS,PANS,CANS)

START MESSAGE REQUEST

	DEV No	KEY No	TIMER(sec)	COLOR	MSG IDX	TREND PID	MESSAGE
SMR	CRT1		30	CG	412		TBN SPEED TARGET 400RPM SET
SMR	TYP1	0		TB	412		TBN SPEED TARGET 400RPM SET

WORKER CONTROL START

	Req	TYPE	WCD No.	TARGET PID	RATE PID	WCD NAME
WKC	1	KAW	173			SET TURBINE SPEED 400RPM

[Then]
Action Block

PRE CONDITION TROUBLE MESSAGE

	Req	SET SYS No
PCT	1	

COMPLETE CONDITION TROUBLE MESSAGE

	Req	SET SYS No	TIMER(sec)
CCT	1		30

[ELSE]
Action Block

EXT

Fig. 14.10 DSL-OB

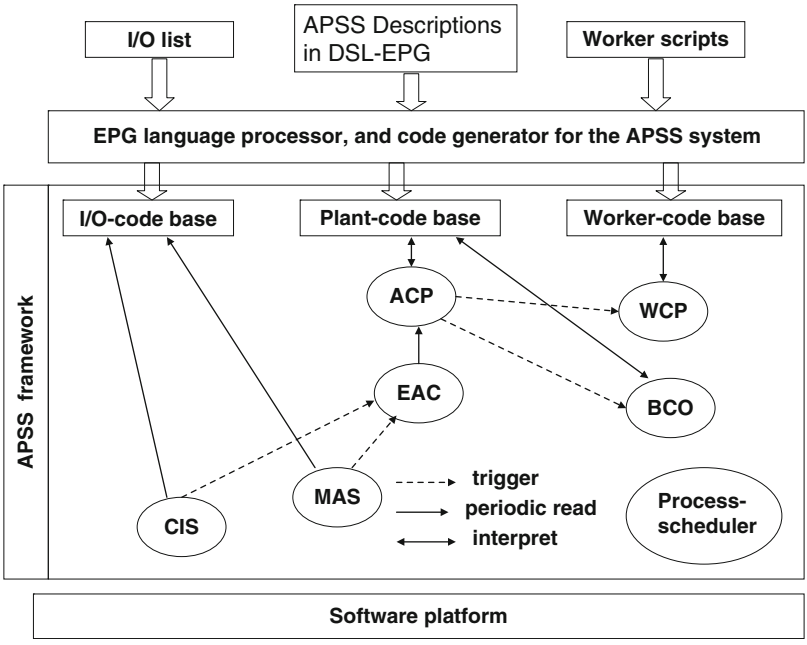


Fig. 14.11 APSS framework

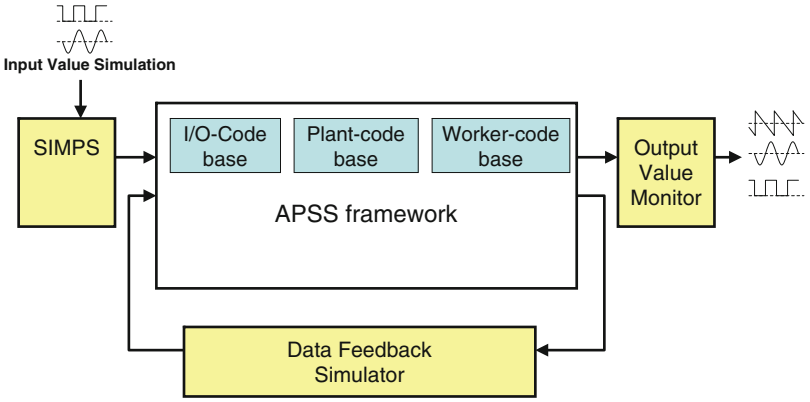


Fig. 14.12 Testing environment for APSS codes generated from DSL-EPG

– Worker Control Processor (WCP): This is the process to drive various external devices such as relays, switches, servo motors, or valve controllers. WCD interprets the codes in the worker-code base and plant-code base assigned by ACP and sends outputs to control external devices.
– Blocking Conditioning Output (BCO): This is the process to select and block some outputs for the purposes of protection by interpreting the assigned conditions from the I/O-code base and plant-code base.

In order to verify the function of APSS software, Toshiba has developed the testing software environment shown in Fig. 14.12. The testing software environment implements the following subfunctions:

– Plant input SIMulation Program System (SIMPS): The function to apply time sequential data to each input of APSS software.
– Output Value Monitor: The function to enable monitoring each output of APSS software by using graphical method.
– Data Feedback Simulator: The function to simulate the feedback from output data to input of APSS system based on the combination of software code. This function is applied to verify the functions of some important part of the plant such as main turbine.

These functions effectively work for verifying the APSS software functions, for example, validity of data provided I/O-code base, activities, controlled by plant-code base and process driven by worker-code base.

5 Software Product Line

5.1 *Product Line Scoping Supports Business Strategy [7]*

The scoping [4, 5] of EPG-SPL has been decided based on the Toshiba Corporation's business strategy that defines market segments in the EPG world. The market segments covered by the EPG-SPL are fossil fuel fired (coal fired, oil fired, or liquefied natural gas fired) steam cycle, steam/gas combined-cycle and nuclear power plants (e.g., boiling-water type) to be provided by public utility companies, as well as by private companies. The EPG-SPL scope also includes other parameters such as listed as follows:

1. Type of software platform
2. Type of EPG middleware
3. Type of DSL-EPG interpreter
4. Various specialties that are required depending on each market segment
5. Various specialties that are derived from the properties of customers, plant-equipment manufacturers, and type of plant/control devices
6. Type of functionalities covered by the EPG-SPL

The functionalities covered by the EPG-SPL are plant-monitoring, plant-performance calculation, APSS, man–machine interactions, plant-operational graphics display. The EPG-SPL constituents are classified in accordance with those EPG-SPL functionalities. It means that APSS is one of the EPG-SPL constituents, and the operational semantics SCIA, described in Sect. 4, is the semantics that specifically supports only APSS.

The model of EPG-SPL-code configuration is shown in Fig. 14.13. It comprises four major parts: DSL-EPG-code generator, code based for the defined four market segments, functional programs for DSL-EPG interpreter and specialties used for the particular functionalities, and EPG middleware.

5.2 *Architecture of EPG-SPL [8]*

The architecture of EPG-SPL and how to use it is modeled in Fig. 14.14. The invariants described in Sect. 3.2, e.g., the operational semantics (SCIA) for APSS, commonly used logical expressions, and event-driven architecture, are classified in accordance with the market segments and associated parameters, explained in Sect. 5.1., and mounted on the repository described in the left edge of Fig. 14.14 through Meta Class format sheets. The variants described in Sect. 3.2, which are variables in PMS, MCS, MSD, and OB, variables to identify actions, and variables to identify messages, are input to the code generators through the DSL documents shown in the top right corner of Fig. 14.14.

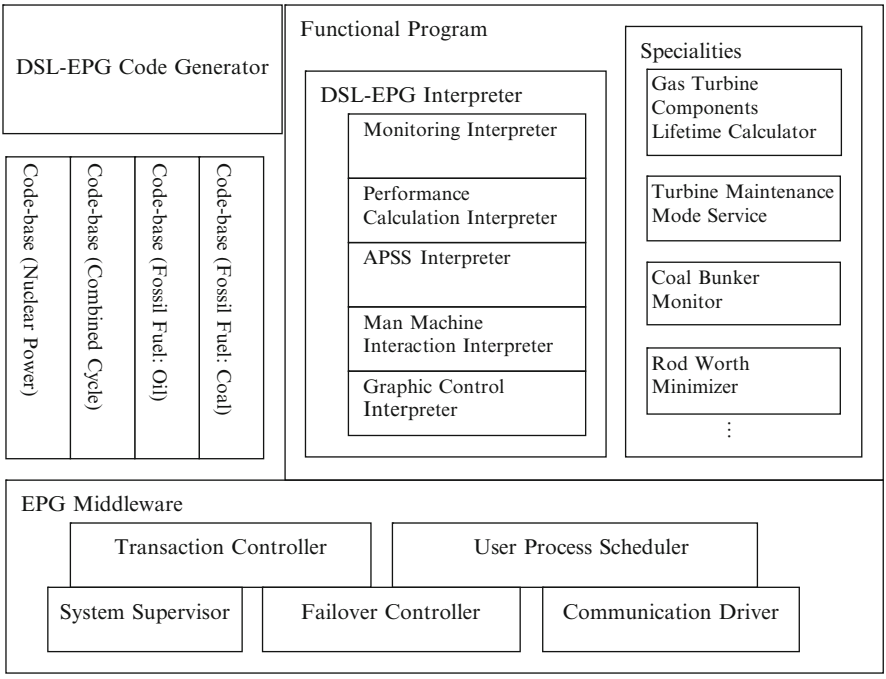


Fig 14.13 Model of code configuration of the EPG-SPL

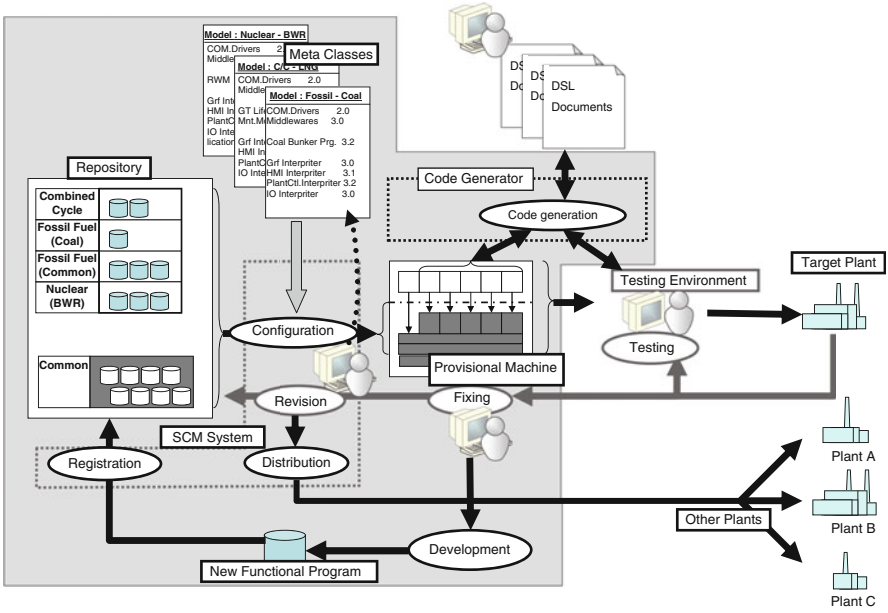


Fig. 14.14 Architecture of EPG software product line

In case that we need to construct a whole set of codes to be released to a target power plant, the plant-specific invariants defined by the meta-classes, described using the Meta Class format sheets shown in the left-top part of Fig. 14.14, are put into the EPG-SPL by the developers. The task called "configuration," shown in Fig. 14.14, selects the necessary sets of invariants, defined in the Meta Class format sheets, from the repository, and put those invariant sets into the provisional machine automatically. The provisional machine consists of code bases, interpreters, EPG software framework core, platform and hardware. The interpreters can be made to execute using all the defined variants and invariants using EPG middleware, platform, and hardware, on the provisional machine. The codes system mounted on the provisional machine can be tested using plant input SIMulation Program System (SIMPS) by the human task "testing".

In case that the codes released and deployed in the target plant should be changed as results of maintenance, improvement or enhancement, contents of the repository will be updated accordingly in order to match with the deployed codes through human tasks "testing," "fixing," and "revision." The fact that any revision has been made will be announced to other plants by the human task "distribution." When any new invariant features are developed, those invariants can be uploaded to the repository through human task "registration."

6 Cost and Benefits

In order to manage and control cost and benefits during the development and adoption of EPG-SPL, a guideline was developed [1, 2]. The guideline consists of three kinds of tables shown in Tables 14.3, 14.4, and 14.5, namely ROI calculation form, variable cost table, and fix cost table. Using this guideline, cost and benefits are controlled so that shortage of return, to be entered in the bottom row of Table 14.3, should never get into the red in each fiscal year.

The ordinal numbers shown in the top of every table identify the sequential order of the fiscal years in the adoption stage. Usually, the corporate level of the company defines the values such as standard interest rate (cost of capital), the tax rate, and the depreciation. The number of shipments in fiscal year i, or n(i), means the number of systems released and installed within this fiscal year, developed by the adoption of EPG-SPL. The sales amount S(i) means the sum of customer's payment within fiscal year i.

The breakdowns of "variable cost" and "fixed cost" are listed respectively in Tables 14.4 and 14.5. The explanations for every symbolized item are noted in the rightmost columns.

In Table 14.4, the cost of maintenance, enhancement, and configuration management required for running installed systems should be included in team cost TC(i).

Table 14.3 ROI calculation form

No.	Item	Symbol	Notes
#1	Ordinal number	i	The "i" of the fiscal year when the product line is first adopted is set 1
#2	Standard interest rate (unit value)	SIR	SIR is the value that is internal to the company. SIR corresponds to the cost of capital, and its value is defined a little higher than the cost of capital
#3	Tax rate (unit value)	T	–
#4	Number of shipment	n(i)	–
#5	Sales amount (¥)	S(i)	–
#6	Variable cost (¥)	VC(i)	The cost breakdown is shown in Table 14.4
#7	Marginal profit (¥)	MP(i)	MP(i) = S(i) − VC(i)
#8	Fixed cost (¥)	FC(i)	The cost breakdown is shown in Table 14.5
#9	Profit (¥)	P(i)	P(i) = MP(i) − FC(i)
#10	Profit after taxes (¥)	PAT(i)	PAT(i) = P(i) * (1 − T)
#11	Depreciation (¥)	DP(i)	–
#12	Residual Value (¥)	RV(i)	–
#13	**Cash in (¥)**	**CI(i)**	**CI(i) = PAT(i) + DP(i) + RV(i)**
#14	Additional investment (for software) (¥)	IS(i)	The cost breakdown is shown in Tables 14.4 and 14.5
#15	Additional investment (for hardware) (¥)	IH(i)	The cost breakdown is shown in Tables 14.4 and 14.5
#16	**Cash out (¥)**	**CO(i)**	**CO(i) = IS(i) + IH(i)**
#17	**Cash flow (¥)**	**CF(i)**	**CF(i) = CI(i) − CO(i)**
#18	Discount rate (unit value)	DR(i)	$DR(i) = 1/(1 + SIR)^i$
#19	Present value of cash flow (¥)	PV(i)	PV(i) = CF(i) * DR(i)
#20	**Shortage of Return (¥)**	**SR(i)**	**SR(i) = (initial investment) − (sum of PV(1), . . ., and PV(i))**

From Matsumoto 2007 [2]. ©2010 IEEE. With permission

Table 14.4 Calculation of variable cost

No.	Item	Symbol	Notes
#1	Ordinal number of fiscal year	I	–
#2	Cost spent for acquiring products from outside vendors (¥)	AC(i)	–
#3	Cost spent for product outsourcing (¥)	OC(i)	–
#4	Cost spent by the domain engineering team (¥)	TC(i)	Develop additional assets, improve the existing assets, and develop sub-domain product lines
#5	**Variable cost (¥)**	**VC(i)**	**VC(i) = AC(i) + OC(i) + TC(i)**

From Matsumoto 2007 [2]. ©2010 IEEE. With permission

The costs of activities, such as development of new items necessary to satisfy individual customer's requirements, development of additional assets, improvement of the existing assets, and the development of sub-domain product lines,

Table 14.5 Calculation of fixed cost

No.	Item	Symbol	Notes
#1	Depreciation (¥)	DP(i)	–
#2	Personnel cost (¥)	PC(i)	–
#3	Operative overheads (¥)	OO(i)	–
#4	Department expense (¥)	DX(i)	–
#5	**Fixed cost (¥)**	**FC(i)**	**FC(i) = DP(i) + PC(i) + OO(i) + DX(i)**

From Matsumoto 2007 [2]. ©2010 IEEE. With permission

modification efforts (such as development of sub-domain EPG-SPL, and modification of EPG-SPL adoption programs) should also be included in TC(i).

In Table 14.5, the organizational costs necessary for keeping the fixed activities and tasks, such as the ones conducted by the fixed organizational elements should be included in department expense DX(i).

Using VC(i) and FC(i) calculated in Tables 14.4 and 14.5, the shortage of the return SR(i) should be calculated using the equation in the bottom row of Table 14.3. This value suggests the residual return that should be recovered by the cash flows in the residual years.

The guide addresses the following processes:

1. Time scale: The length of EPG-SPL life and the time span necessary for conducting each stage should be estimated at the beginning of or in the early stage of development. At every fiscal year in the adoption stage, its ordinal number starting from the first year should be identified and entered in the row #1 of the tables.
2. PV calculation: The expected cash flows which could be obtained at every fiscal year in the adoptions stage should be predicted at the end of development stage at the row #19 of Table 14.3. And NPV (accumulation of the PVs) should be calculated and used for getting SR(i) at the row #20 of Table 14.3.
3. Cost control in adoption stage: The cost limit for conducting adoption stage could be determined using the calculated SR(i) described in item (2), so that the cost limit should not exceed the NPV. The adoption stage should be managed and controlled so that the actual cost should be less than the cost limit.
4. Renewal of SPL generation: If it becomes clear that the shortage of return will not to be recovered within the predetermined EPG-SPL lifetime, EPG-SPL adoption plan should be modified and improved totally. In our case, such an overall improvement was made in 1982, 1988, 1993, and 1996, as is described by Matsumoto 2007 [2].

7 Conclusion

In this chapter, major features of APSS system and the software family, including DSL-EPG, its processor and EPG-SPL, which supports automatic code generation for each APSS application, are introduced. The sets of rules for operating the plant,

where each rule defines a relationship between plant status, events, conditions, and operator's actions, are defined in the collaborative work participated by customers, plant operators, and APSS developers. The variants included in each rules can be specified, and described with using DSL-EPG. The DSL-EPG processor analyzes those descriptions to generate application codes [9]. The paper also illustrates how EPG-SPL, which comprises APSS, is structured and maintained from the viewpoint of cost and benefit.

The text of this paper was mostly produced by the third author, one of the original creators of the system covered by this paper, and was compiled with the figures, tables and references by the first and second authors.

References

1. Matsumoto, Y.: Management and financial controls of a software product line adoption. In: Kang, K.C., et al. (eds.) Applied Software Product Line Engineering, pp. 399–419. CRC Press, Taylor Francis Group, Boca Raton, FL. 33497–2742 (2009)
2. Matsumoto, Y.: A guide for management and financial controls of product lines. In: Proceedings of IEEE 11th International Software Product Line Conference, pp. 163–170 (2007)
3. Matsumoto, Y.: A method of software requirements definition in process control. In: Proceedings of COMPSAC77, pp. 128–132 (1977)
4. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley, Reading, MA (2002)
5. Czarnecki, K., Eisennecker, U.W.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley, New York (2000)
6. IEEE Std 828-1988, IEEE Standard for Software Configuration Management Plans. Software Engineering Standards Committee of the IEEE Computer Society (1998)
7. ISO/IEC DIS 26551, Software and systems engineering – Tools and methods for product line requirements engineering. International Organization for Standardization/International Electrotechnical Commission (2012)
8. ISO/IEC DIS 26555, Software and systems engineering – Tools and methods for product line technical management. International Organization for Standardization/International Electrotechnical Commission (2012)
9. Matsumoto, Y.: Mapping Dynamic Software Product Line Properties to the Current Toshiba Software Product Line for Electric Power Generation, http://www5d.biglobe.ne.jp/~y-h-m/Mapping_DSPL_Properties.pdf

Chapter 15
Second-Generation Product Line Engineering: A Case Study at General Motors

Rick Flores, Charles Krueger, and Paul Clements

What you will learn in this chapter

- *An introduction to the basic concepts of the factory paradigm of product line engineering, including feature declarations, feature profiles, shared assets, variation points, and configurator*
- *The characterization of first-generation vs. second-generation product line engineering (2GPLE)*
- *How 2GPLE is being applied at General Motors and why the 2GPLE concepts have been critically important: How it has led to the creation of new roles and responsibilities, how organizational units at different levels and in different domain areas are cooperatively building PLE models that will all work together to define a vehicle*

1 Introduction

This chapter is the story of a product line engineering effort under way at General Motors. The product line involves the electronic control systems placed aboard vehicles during manufacturing. These control systems include electrical components (sensors and actuators), electronic control units laid out in a given topology around the car, wires and data networks to connect the components appropriately, and the software that runs it—all loaded correctly onto each vehicle. This story focuses on a particular set of aspects:

R. Flores
General Motors, Detroit, MI, USA
e-mail: rick.r.flores@gm.com

C. Krueger (✉) • P. Clements
BigLever Software, Austin, TX, USA
e-mail: ckrueger@biglever.com; pclements@biglever.com

R. Capilla et al. (eds.), *Systems and Software Variability Management*,
DOI 10.1007/978-3-642-36583-6_15,

- How solving this product line engineering problem requires every dimension of what has come to be called the *second-generation approach* to product line engineering.
- How a very small but consistent set of product line constructs are proving to be adequate to provide the necessary expressive power for this product line.
- How the automation that is required to power the product line solution depends not only on its own technical capabilities but also on vendor business partnerships that allow it to work seamlessly with a variety of lifecycle engineering tools that store artifacts in proprietary formats—artifacts that need to have variation points injected into them.

These aspects are made compelling because of the unprecedented complexity involved in this product line. If these solutions work here, it is unlikely they will be found wanting anywhere else.

2 Overview of Product Line Engineering

Systems and software product line engineering, often abbreviated as *product line engineering* (PLE), refers to the disciplined production of a portfolio of related products using a shared set of assets and a common means of production. The products in the portfolio are related by the features they have in common with each other; the variations among the products are also expressed as variations in the features they offer. The products can be

- Software
- Systems in which software runs or
- Non-software products that have software-representable artifacts (such as requirements, engineering models, or development plans) associated with them

In all cases, PLE works with the "soft" artifacts associated with the products and their production. PLE, then, includes and extends software product line engineering.

The key strategy behind PLE is to capitalize on commonality and manage variation in order to reduce the time, effort, cost, and complexity of creating and maintaining a product line of similar software systems:

- Capitalize on commonality through consolidation and sharing within the asset inputs, thereby avoiding duplication and divergence.
- Manage variation by clearly defining the variation points and decision model for exercising the variation points, thereby making the location, rationale, and dependencies for variation explicit.

The essence of PLE—for systems, software, and for manufacturing—is the focus on the single system rather than the many products. The "system" in this case consists of the *production line*, which enables the rapid production of any version of any of the products in the portfolio. A production line consists of a

collection of software *assets*, a set of *feature profiles* that define the products, and the *configurator* that applies a feature profile to the assets in order to produce each product in the portfolio. Once the production line is established, products are instantiated rather than manually created.

PLE stands in contrast to classical product-centric development, in which each individual product is developed and evolved independently from other products, or (at best) starts out as a cloned copy of a similar product that is then changed to suit the new product's specific needs. Product-centric development takes very little advantage of the commonalities among products in a portfolio.

3 Basic PLE Concepts: The Factory Paradigm

PLE can be described in terms of the following five concepts:

- *Feature declarations* are parameters that express the diversity in a product line. Feature declarations are analogous to the choices that are available to you when you buy a new car: Two door or four door? Sport package, luxury package, or economy package? Moon roof? Feature declarations typically express the customer-visible diversity among the products in a product line.
- *Feature profiles* are used to select and assign values to the feature declaration parameters for the purpose of instantiating a product. A feature profile is associated with each product and reflects the actual choices you make: Two door with sport package but no moon roof or four door with luxury package and moon roof.
- *Systems and software assets* are configurable artifacts—such as models, source code, requirements, and test cases—engineered to be shared across the product line. They are the building blocks of the products in the product line. Assets can be whatever assets are representable with software and either compose a product or support the creation of a product.
- *Variation points* define the variations in the system and software assets used to build products. Feature declarations are mapped to these variation points, and a feature profile is mapped to the choices made at each variation point when building a product.
- *Configurator* is the automation that takes the feature choices reflected in a feature profile for a product and applies them to the variation points in the assets, so as to produce instances of the assets that are the building blocks of the product being built. It is possible to perform this step manually, but the task quickly becomes unmanageable without an automated tool. An example of an industry-leading configurator is Gears by BigLever Software [8], which GM chose to power its PLE effort.

An analogy with factory-base manufacturing serves to illuminate the concepts. Manufacturers have long used analogous engineering techniques to create a product line of similar products using a common factory that assembles and configures parts designed to be reused across the varying products in the product line. For example, automotive manufacturers can now create thousands of unique variations of one car

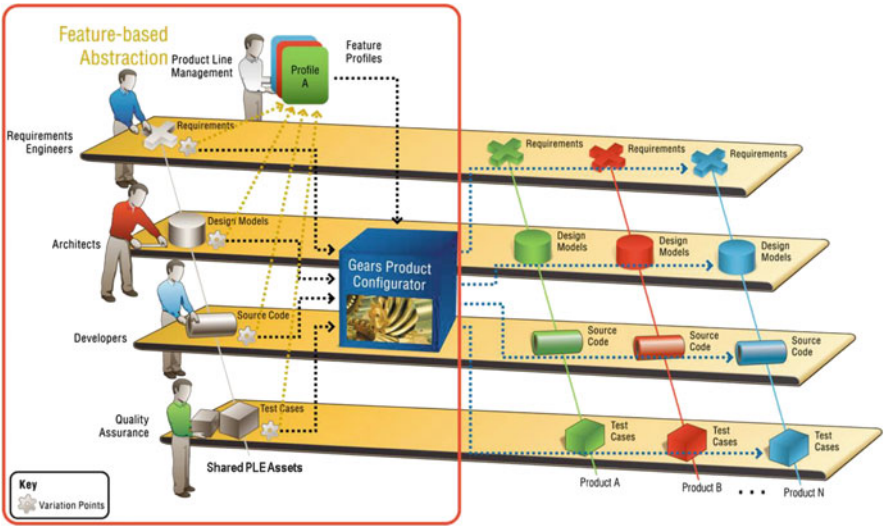


Fig. 15.1 A production line. Feature profiles drive instantiation of assets' variation points, which are exercised by the configurator (here, Gears) to produce product-ready instances

model using a single pool of carefully architected parts and one factory specifically designed to configure and assemble those parts.

In PLE, the configurator is the factory and the assets represent the factory's supply chain. Figure 15.1 illustrates.

4 First-Generation PLE

Product line engineering now has roots that span at least five decades, going back at least as far as Parnas's seminal paper on product families for software in 1976 [11]. Examining the rich historical legacy of this community reveals patterns of evolution in the state of the art and practice.

"Generations" are hard to pin down precisely and do not have rigid boundaries, but that does not prevent the concept from being useful. We can identify the Baby Boomer, Gen-X, Gen-Y, Tween, and Millennium Generations. Fighter aircraft are generally thought to be in their fifth generation [6] and programming languages in their fourth or fifth (opinions vary) [21]. Current standards for mobile broadband devices are known as 4G.

In the same spirit, we characterize some of the early and long-standing approaches to product line engineering as first generation. First-generation PLE (1GPLE) includes, among other things:

- A strong dichotomy between domain engineering and application engineering, or core asset development and product development. Application engineering

(or product development) is often said to include creating any assets used in a single product and promoting them to core assets only if subsequently used in more.
- Explicit inclusion of non-software artifacts in the collection of core assets, but with an unmistakable emphasis on software (under the umbrella of an all-encompassing software architecture) as the principal kind of core asset.
- Focus on features as the language to describe a product line's domain and a way to discriminate products from each other.
- Acknowledgment of configuration management as an essential practice under PLE but without a strong distinction between core asset CM and product CM.

These approaches have yielded a rich legacy of product line success, as evidenced by a plethora of case studies [3, 4, 9, 13, 16]. The newer approaches we describe in this chapter build on them. These newer approaches came about because of situations where more robust methods are needed to (among other things) deal with very large-scale product lines. "Scale" can refer to size, complexity, and number in terms of products, core assets, lifecycle phases involved, and evolutionary revisions over time.

5 Second-Generation PLE

PLE has been evolving a new set of concepts and technology that has been referred to as *second-generation product line engineering (2GPLE)*. This characterization represents seen-in-practice extensions to the earlier paradigm that was centered mainly on core asset production and product derivation.

Second-generation PLE can be said to comprise five aspects. None of these facets of 2GPLE are incompatible with or contradict earlier approaches to software product line engineering [4, 13, 19]—indeed, all five are mentioned as possible. The difference is that in 2GPLE they have emerged in a central role, essential to support large-scale practice. The five facets of 2GPLE are:

- Reliance on features as the *lingua franca* to express product differences in all phases of the life cycle
- Consistent and traceable variation management in artifacts across the full engineering life cycle
- A simplified configuration management model that maintains versioning of assets, not products or asset instantiations
- Feature models with encapsulating constructs to facilitate hierarchical product lines and cooperative feature model development across organizational boundaries
- Industrial-strength automation

GM could not accomplish its product line engineering goals without each one of these. We will discuss each in turn here and show how each is put into play at GM in the second half of this chapter.

5.1 *Features as the* Lingua Franca *to Express Product Differences Across the Life Cycle*

The concept of a feature as applied to families of software systems is thought to date to feature-oriented domain analysis methods, beginning with FODA [7]. In that work, the authors adopted the definition of feature right out of an ordinary dictionary: "A prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems," and this definition still serves us well in the 2GPLE world. The property of being visible to the user is perhaps the central notion; the choices a buyer can make when purchasing a new car is a helpful analogy.

Referring again to the manufacturing paradigm, the set of features to be exhibited by the product under construction drives the manufacturing process: The features determine which parts should be used in the product, how they should go together, and how they should be tailored to fit the product. All of the assets that go into building products will include variation points that will be exercised based on the features the product under construction needs to have.

The concept of "feature" allows a consistent abstraction to be employed when making choices from vehicle configuration all the way down to the deployment of software components onto an electronics architecture. As we will see, GM is elevating what they call a *bill-of-features* to the role of communication vehicle between business, product marketing, and engineering units. The goal is to use this to express and automatically derive content for vehicles in terms of desired features and capabilities, rather than describing vehicles in terms of its bill-of-materials—that is, its listing of parts and pieces. Although a bill-of-materials will still be needed for manufacturing, the vision of GM's PLE effort is that the bill-of-materials for a vehicle's electronics is generated from its bill-of-features.

The product line literature is rife with feature modeling languages and constructs, few of which have seen industrial application. The GM experience is providing a compelling argument that a very small and simple set of feature modeling constructs suffices for describing all of the necessary feature information for large and complex product lines.

To capture features, here is the set of feature-modeling constructs (provided by Gears) that GM is using for its product line work. These constructs have evolved over 10 years based on experience in ever-larger and more complex industrial applications. The set of constructs has remained stable and small. They are:

- *Feature declarations* are parameters that express the diversity in the product line for a system or subsystem. Feature declarations typically express the customer-visible diversity among the products in a product line.

  Feature declarations have types. When a feature is chosen for inclusion in a product, it must be given a value consistent with its type. Table 15.1 shows the feature types supported by Gears.
- *Feature assertions* describe constraints and dependencies among the feature declarations. Feature assertions in Gears express REQUIRES or EXCLUDES

Table 15.1 Gears feature types

Boolean	True, false
Integer, Float	Signed or unsigned numeric value
String	Character string delimited by double quotes
Character	Single character delimited by single quotes
Enumeration	*Select exactly one* value from subordinate features
Set	*Select zero or more* values from subordinate features
Record	*Select all* values from subordinate features
Atom	Named member/value of a set or enumeration

relations. They express the constraint that a feature (or combination of features), if present, either requires or excludes the presence of another feature (or combination of features). For example, an assertion could express the need for software-actuated brakes to be present whenever the park assist option is on the vehicle or the need for certain switches to be present if certain lights are installed.

- *Feature profiles* are used to select and assign values to the feature declaration parameters for the purpose of instantiating a product. A feature profile is associated with a product and reflects the actual choices you make: Two door with sport package but no moon roof or four door with luxury package and moon roof. The values assigned in feature profiles must satisfy the constraints and dependencies expressed by the assertions in the feature declarations.
- *Assets* are the abstraction for systems and software artifacts in a production line. They are the building blocks of the products in the product line. Assets may be requirements, architecture and design documents, source code files, calibration sets, test cases, and so forth—artifacts from any phase of the development life cycle.
- *Variation points* encapsulate the variations in the assets used to build products. Feature declarations are mapped to these variation points, and a feature profile is mapped to the choices made at each variation point when building a product. In Gears, a variation point is instantiated from one or more variants, one of which will "stand in" for the variation point when a feature profile is used to build a product. A variant can "stand in" as is (in which case, the variation is accomplished by choosing which variant to use), or it can "stand in" after being transformed by applying a match-substitution pattern expressed in the regular-expression language of Perl. Also encapsulated within each variation point is the logic, expressed as a sequence of rules, that maps feature values to the different instantiations of that variation point.

There are three more Gears constructs that come into play in hierarchical product lines; these will be discussed shortly.

Figure 15.1 illustrates how this small set of constructs give us the concept of a *production line* (the part of the figure inside the red box). Assets are built and maintained on the left; each is endowed with one or more variation points (indicated by the gear symbol). Feature profiles determine how the assets are instantiated

(by exercising their variation points) to produce product-ready artifacts. Under this paradigm, organizations become production-centric rather than product centric.

5.2 Consistent Variation Management in Artifacts Across the Full Engineering Life Cycle

The 2GPLE paradigm treats artifacts across the full engineering life cycle as equals, as current applications of product line engineering are demanding it.

It has long been a stated tenet of product line practice that core assets include more than software. For example, the Software Engineering Institute's Framework for Product Line Practice [14] states that "architecture, requirements specifications, testing-related artifacts, budgets, schedules, plans, and production infrastructure can all constitute core assets." However, a complete systems and software PLE lifecycle solution requires more than just a statement of eligibility. It requires consistent treatment of the artifacts' variation points under the production infrastructure, so that a full set of demonstrably consistent supporting artifacts can be systematically generated for each product. The alternative, trying to translate between the different representations and characterizations of features and variations across the boundaries between stages in the life cycle, is untenable.

To illustrate, imagine that a requirements engineering team has embraced a PLE requirements management technique based on tagging requirements in a requirements database with attributes that differentiate feature variations in requirements. Further, the design team has adopted a UML tool and has embraced inheritance as the mechanism for managing PLE design variations. The development team is using a FODA [7] feature model drawn in a graphical editor, plus #ifdefs, build flags and configuration management branches to manage implementation variations. Finally, the test team has adopted clone-and-own of test plan sections, stored in appropriately named file system directories to manage their PLE test plan variations. Now imagine what would be needed to create a complete PLE lifecycle solution that integrates into a larger business process model. How do the requirements database attributes and tagged requirements relate and trace to the subtypes and supertypes in the design models? How do these attributes and supertypes relate and trace to the #ifdef flags, CM branches, FODA features, and test case clone directories? Trying to translate between the different representations and characterizations of features and variations creates dissonance at the boundaries between stages in the life cycle.

To resolve this quagmire, a key aspect of 2GPLE is not just inclusion of non-software artifacts, but consistent and traceable treatment.

The artifacts to support this process include requirements, system architectures and designs, source code implementation, calibration parameters, test cases, and documentation. Some of the documentation could be for suppliers, who will provide some of the necessary software and hardware components. At a company

such as GM, the long-term goal can be that all of these are endowed with variation points, which can be exercised to correspond to feature choices.

Common representation of variation points is key to achieving traceability from requirements to deployment. Traceability is of great concern for GM. Every requirement needs to be traceable to one or more design elements that satisfy that requirements, and each design element should be traceable back to one or more requirements that it satisfies. Each design element needs to be traceable forward to its implementation and vice versa. Each requirement needs to be traceable to one or more test cases that validate whether or not the requirement is satisfied in the final product. Managing all of these artifacts consistently, by tying their variations to features, is the key to achieving this.

5.3 *CM That Maintains Assets, Not Products or Asset Instantiations*

Configuration management (CM) for a product line must allow the rapid reconstruction of any product version that may have been built using various versions of the PLE assets and development/operating environment. This capability is essential for rapid response to and remediation of any anomalies that arise in the field.

The most important aspect of CM in 2GPLE is that the full superset of available PLE assets (and not the individual products or systems) are managed under CM. A new version of a product is not derived from a previous version of the same product, but from the shared superset of PLE assets themselves.

Contrast this to product-centric CM, illustrated in Fig. 15.2. Suppose a defect is discovered in Product B after it has been deployed, and the defect is traced back to product B's requirements. The Product B team fixes the defect and redeploys. But Product B's requirements might have been borrowed from Product A's requirements, and Product N's code might have been borrowed from (defective) product B's. By the time all of the potential dependencies have been run to ground to make sure the defect is eliminated from every place it might occur in n products, $n(n-1)$ interactions have occurred, for an $O(n^2)$ complexity.

By contrast, using the scheme shown in Fig. 15.1, the requirements defect will be fixed in the asset, not the products. The affected products will be regenerated. This is an $O(n)$ proposition. All that is required to reconstruct any product version is to store the *temporal context* for that product version. A temporal context is a vector of assets and the version of each that was used to build a version of product.

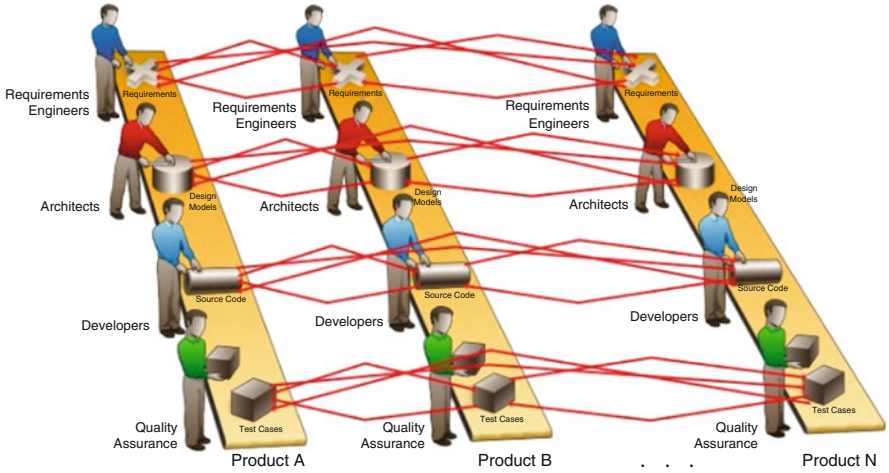


Fig. 15.2 A product-centric perspective with O(n^2) complexity

5.4 Product Lines Across Organizational Boundaries

For PLE to work at large organizations, it may be impractical to have a single organizational unit tasked with the care and feeding of the shared PLE assets [17]. Certainly having one global collection of feature declarations for an entire production line is impractical. (At GM, a single feature model for a car would comprise a few thousand features and have to be shared among thousands of engineers.) Further, subsystem engineers have no interest or need to see all of the feature diversity in other subsystems. For example, engineers for an automotive transmission system do not need to see feature abstractions that capture the diversity in the entertainment or GPS navigation system. It makes no sense to comingle them.

It makes much more sense to modularize the feature model in a way that corresponds to the organizational structure of the enterprise. Although these structures can change over time [5], they make an excellent starting point and let the organization begin to adopt PLE using familiar units.

At GM, a vehicle is composed from a set of integration areas (such as *safety* or *human–vehicle integration*), which assemble combinations of subsystems, which are in turn composed of functional elements, which are implemented by compositions of software components and calibrations that are loaded onto hardware components arranged in one or more physical architecture topologies. At each level in this decomposition—which is not necessarily hierarchical—engineers are assigned responsibility for managing the artifacts and configurations at that level, all of which are imbued with rich and numerous kinds of variation. Assembling a vehicle from the most primitive elements would simply be intractable. By contrast, a vehicle is more like a system of systems [10], which is managed as a product line of product lines. At GM the nesting is at least four levels deep.

Each of these units represents a domain, by which we mean a body of knowledge [7]. Integration areas and subsystems are part of the fabric of the company. Building a subsystem for a vehicle, or combining subsystems in an integration area, or implementing a functional element requires specialized knowledge. In a PLE context, that specialized knowledge becomes knowledge about the variations that are possible, and the result is a number of product lines that each contribute instances to the overall vehicle product line.

Because some of the domains are quite large (as are the bodies of knowledge they embody), domains have sub-domains, and their product lines are the result of still finer grained nested product lines. (Section 8 will show an example of this nesting.) Features at the highest level of the hierarchy include things that vehicle customers would resonate with, such as daytime running lights or lane keep assist, while "features" take on a different meaning at the lower levels. Here, features represent a variation of a lower level "product" (such as a component that implements one of the available varieties of cruise control) being offered up to higher level product lines. But the whole chain starts at the highest level with the Bill-of-Features for the vehicle, each of which causes a cascade of lower level choices to be made. At every level, the same small and elegant set of concepts presented earlier work to capture the inherent variation. This lets engineers work largely independently within the confines of their own organizational units and domain expertise.

A hierarchical product line constitutes an architecture-like construct, in that there are interfaces and relationships among the nested product lines. There is the parent–child relationship for product lines that typically mirrors the system–subsystem decomposition in the vehicle architecture. Product line features can be partitioned, encapsulated, and scoped within the primary subsystems that realize the features. Features can also be shared among product lines by establishing an import relationship, which is crucial for establishing feature constraints and asset variation points among interrelated subsystems (e.g., a high-end flavor of cruise control that slows the car if there's traffic ahead requires a flavor of the braking system that supports braking via software command).

Gears provides three more constructs that facilitate the interfacing and coordination between levels in the hierarchical product line: mixins, matrices, and imported production lines.

- *Mixins*. Although many feature declarations will fall cleanly into the realm of one asset or another, there are many cases where a feature declaration applies to two or more assets. For example, the automotive platform (Buick Regal? Chevy Cruze? Cadillac CTS?) and the region for which the vehicle is being marketed (North America? Brazil? China?) constitute features that determine how an asset should be configured at many levels: Integration area, subsystem, functional element, component. Rather than duplicating the same feature declaration in multiple assets, Gears provides the mixin abstraction to allow creation of a feature declaration in one place and then "mix it into" the feature declarations of multiple assets, by reference.

AutoCommerce	Global	Showroom	Site	Showroom	Requirements	UserGuide
ChevroletOfUS	Minimalist_ChevUS	BasicNew	Simple	$defined$	$defined$	US
ChevroletOfCanada	Minimalist_ChevCan	BasicNewAndUsed	Desktop	$defined$	$defined$	NorthAmerica
CadillacOfFrance	Extreme_CadillacFr	FullService	Mobile	$defined$	$defined$	Europe

Fig. 15.3 A Gears matrix, with three rows for three products. The "Global" and "Showroom" columns show feature profile choices for mixins; the last four columns show feature profile choices for assets.

Mixins are more than a convenience to avoid typing of feature declarations. They also encapsulate, in a single location, the feature profiles for the feature declaration parameters. Having a single location for the feature profiles prevents inconsistencies when composing assets to create a complete system.

Imagine, for example, that we need the lane-keep-assist option to be supported by two assets. If this option were declared independently in both assets, it would be possible to inadvertently create a system where one asset assumed that the feature was supported and the other asset assumed it was not supported. Using mixins, there is a single feature profile for the lane-keep-assist option that is "mixed into" both assets. It is either true for both assets or false for both.

- *Matrices*. A Gears production line is the "virtual factory" that knows how to build products by configuring assets in accordance with selected feature profiles. To build a product, you need to tell Gears what feature profile to use for each asset and each mixin in the production line. A matrix is a table showing the choices to build a complete and consistent product. Each row specifies one product. Each column specifies a choice of feature profile for a mixin or an asset (Fig. 15.3).

  A complete product instance is "actuated" by actuating every asset and nested production line column that has an entry for that product. Each asset and nested production line is actuated according to its cell value in the row. If an asset imports a mixin, the mixin feature profile to be used is determined by its cell value in that row.

  Some products may not need all of the assets. For example, low-end products in a production line may not include "luxury" assets that are aimed at high-end products. Each matrix allows you to include or exclude individual mixins and assets to accommodate such product diversity.
- *Imported production lines*. Gears allows you to create a hierarchy of production lines by nesting one production line into another production line. In order to use a production line as a nested production line, it must first be imported. An imported production line will be added as a column in the matrices for the importing production line, just like an asset or mixin. For example, engineers at GM have defined a production line for the safety integration area. In order to provide a safety package to a vehicle, the safety production line must include specifically configured subsystems from a number of subsystems (such as body and active safety), which are their own production lines. A subsystem production line, in turn, can import production lines corresponding to functional elements, and so forth.

Hierarchical product lines are not new in the product line literature [2], but GM is turning out to be the largest (and deepest) realization seen in practice. Also, hierarchical product lines are not needed just because the product line is large. We know of another product line of multi-million-line systems with equally astronomical product variation. However, their products are structured as a set of a dozen or so major subsystems with limited influence on each other in terms of variation selection. That is, choosing features for one subsystem does not have much influence on the features of the other subsystems, obviating the need for a hierarchical production line.

5.5 Industrial-Strength Automation

The last ingredient in 2GPLE is a configurator employed to maintain configurations and translate feature profiles into assets with their variation points exercised in prescribed ways. The tooling needs to be able to support the construction and management of feature models (including feature declarations, assertions, and profiles), assets and their variation points, support hierarchical production lines, and map from feature choices to asset instances (in Gears, this is the job of the matrices). Further, it needs to either provide version control for the models and artifacts or (even better) work seamlessly on top of the user's own choice of change management system.

A major requirement for the tooling is that it supports the specification and selection of variation in assets and artifacts from across the entire spectrum of the product life cycle. This means that in addition to working with open-format artifacts, the tool will have to support variation proprietary-format artifacts such as IBM Rational DOORS requirements modules, Microsoft Word documents, and Excel spreadsheets, build files for Make or Ant, Rhapsody UML models, and many more.

Gears supports these and more using *bridges*. A bridge is a piece of software that "knows" the other-tool representation and presents a "product-line-aware" user interface for that tool that allows product line engineers to insert variation points in the artifacts maintained by that tool.

First-generation approaches always discussed the need for automation; second-generation approaches require it, and it must interface with other system and software engineering tools.

6 A Mega-scale Product Line at GM

General Motors is the largest automotive manufacturer in the world [1]. In 2011 it sold over nine million vehicles, produced (with its partners) in 31 countries around the world. That works out to over 1,000 vehicles rolling off assembly lines *every hour*.

Fig. 15.4 Chevy Volt: Ten million lines of code, nicely packaged (© GM Company)

The product line we describe is built under the Next-Generation Tools (NGT) initiative at General Motors. GM introduced NGT to tackle the complexity brought on by (among other things) the introduction of hybrid and alternative-fuel vehicles and new "active safety" features that require intricate and unprecedented orchestration among vehicle subsystems. Product line engineering is a key ingredient of NGT.

General Motors may well represent the most challenging domain in all of product line engineering. We characterize it as *mega-scale PLE* due to the fact that engineers must deal with multiple product line characteristics that measure in the millions although, as we will see, even this term's implied order of magnitude fails by a wide margin to do justice to the problem space:

- *The vehicles are complex*. As a group, GM vehicles comprise some 300 engineered subsystems such as brakes, exterior lighting, interior lighting, entry controls, and many more. The Chevrolet Volt runs approximately ten million lines of code [12], which is several million more than either the Boeing 787 or the F-35 Joint Strike Fighter 13 (Fig. 15.4).
- *The variation among vehicles is enormous*. GM builds over 60 models under seven brands and divisions. The vehicles may be internal combustion, electric, or both. Customer-visible options include everything from power windows to "lane keep assist" (a system to help the car stay in the correct highway lane). These options, and many dozens more, fundamentally affect the electronics and software aboard the vehicle.

  Legislation, not to mention cultural preferences, in the 150+ countries where GM does business also imposes feature constraints. To choose one of many dozens of examples, there are complex interactions between the vehicle's exterior lights (low beam headlights, high beam headlights, tail lamps, brake lights, parking lights, daytime running lights, front fog lamps, rear fog lamps, cornering lamps, reversing lamps, and hazard flashers) in terms of which lights are allowed, disallowed, or required to come on with which others. The "lead me to my car" feature makes lights come on or flash when the driver presses a button on the key fob. Which lights come on, whether they flash or not, and how long they stay on all are specific to the region and (of course) what exterior lights are

actually on the vehicle. The electronics aboard *every* car has to get that behavior right for *that* car.

A simple thought experiment helps to grasp the astronomical magnitude of the variation involved. We can think of vehicle rolling off an assembly line as the result of making a very large set of yes-or-no decisions. The set of all possible vehicles results from all possible combinations of those yes-or-no choices. The size of that product space is 2^x, where x is the number of decisions. If $x > 260$, then the product space comprises more combinations than the number of atoms in the observable universe 21. For GM, x is in the low thousands. (The number of variants that GM actually produces is much less than that, obviously—a number in the low tens of thousands.)

- *Feature interaction abounds*. The lighting example above illustrates interactions within a subsystem (exterior lighting), but other features require complex interactions among completely different subsystems. For example, the presence of "park assist" (a feature to help park the car) requires the presence of a sensor to gauge the car's position relative to the parking space. In some cars this will be a sonar detector, while on others it will be a camera. Park assist also requires brakes that accept software control, and some versions of park assist require particular versions of steering controls. Thus, the presence of a customer-visible feature can affect multiple subsystems, requiring communication and coordination among the subsystems on the car, and among the groups that are responsible for the subsystems involved.
- *The product line must be in lockstep with current and future model years*. GM has to plan their production years in advance. Features that won't be in the showroom for 3–5 years are already part of today's engineering. And the entire product line marches in unwavering lockstep with the calendar, fixed and unforgiving, which defines each new model year. This means that the product line infrastructure must support concurrent engineering streams for each of the fixed yearly cadences, as well as concurrent development cadences for release cycles scheduled every 6 weeks throughout the year. There may be as many as 15 active, concurrent engineering baselines that engineers must contribute to and coordinate among.

  Another thought experiment illustrates the astronomical combinatorics of the temporal dimension. Each of the 300 or so GM subsystems will typically undergo enhancements or fixes within ten or more cadences within a 2-year period, resulting in 10^{300} possible subsystem version combinations. As with the number of feature combinations, this also vastly exceeds the 10^{80} atoms in the observable universe [18].
- *Consistency and traceability across the life cycle are required*. Each vehicle is the result of an engineering process that spans requirements, design, implementation, calibration, layout and interconnection of electronic control units (ECUs), allocation of software to the ECU network, production of a manufacturing bill-of-materials, and testing. Each of the artifacts must be consistent with each other, in that they must all be accurate with respect to the vehicle to which they

apply. Further, that consistency must be demonstrable through feature interdependency constraints as well as traceability among lifecycle phases.
- *The organization is very large*. Ultimately up to 5,000 engineers will be directly working on artifacts that are part of the product line, some in roles newly defined expressly to support the PLE effort.

The emergence of hybrid and alternative fuel vehicles and new active safety features, which dramatically increase the amount of product line diversity, plus the new economic reality in the automotive industry that leaves little margin for technical error, drove GM to plan to overhaul its engineering tools and processes. The result is the Next-Generation Tools (NGT) initiative.

7 GM's Approach for Mega-scale PLE

This section describes in greater detail how GM has adopted 2GPLE as their technical roadmap for the future.

7.1 GM's Architectural Decomposition

GM's architectural strategy plays a key role in how it is rolling out PLE. The strategy is one of logical decomposition as a way to gain control over the complexity of building a vehicle's electronics and a way to allot the thousands of engineers into organizational units with clearly scoped roles and responsibilities.

- *Functional architecture*. First, a vehicle consists of a number of *domains*. These are "containers" for capturing the requirements necessary to describe the electronics terms applicable to an entire vehicle. Domains define areas of related functionality. For example, Powertrain is a domain, as is HVAC (heating, ventilation, and air conditioning).

  Orthogonal to domains are *integration areas*. Integration areas can be thought of knowledge areas for satisfying high-level stakeholder requirements for vehicles. Requirements here span domains. For example, Noise and Vibration is an integration area; it "touches" any domain that can contribute noise or vibration to the occupants' driving experience: Powertrain, Body, Chassis, HVAC, and more.

  GM refers to integration areas and domains together as its *functional architecture*. The functional architecture provides the overarching structure to host the hierarchical PLE models. Each domain or integration area team will build the PLE models for their area of concern in corresponding part of the functional architecture hierarchy. Figure 15.5 illustrates.
- *Implementation architecture*. Domains comprise *subsystems*. Subsystems represent physical systems on vehicles. There are subsystems for brakes, external

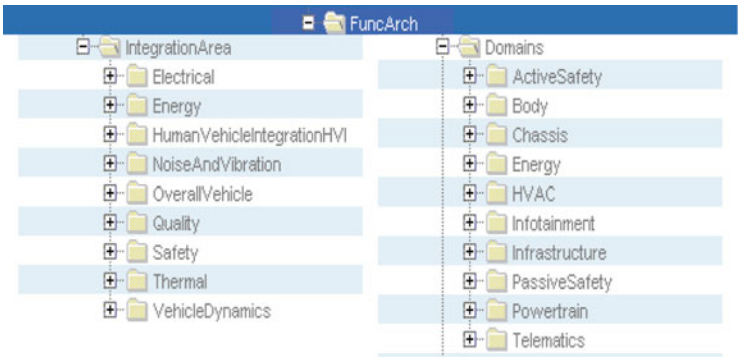


Fig. 15.5 Tool view of GM's functional architecture, showing some of the integration areas and domains

lighting, internal lighting, entry and egress, and many more. Subsystems have their own requirements, which must permit the subsystems to play their proper role in the domains and (in turn) integration areas that need them. Subsystem designers in turn decompose their subsystems into *functions*, and functions into *functional elements*, and write requirements for each. *Components* are units of implementation that satisfy the requirements for functions and functional elements. Components are arranged in a decomposition hierarchy; leaf nodes are components; higher nodes (which are just aggregations of their descendants in the tree) are called *compositions*. Components may be software components or hardware components, depending on how the functional elements are satisfied. GM calls this component structure (with components mapped to the functional elements they satisfy) its *implementation architecture*.

- *Deployment architecture*. Next, the components have to be assigned a place in the onboard electronic architecture topology. Software components need to be assigned to an electronic control unit (ECU), and hardware components have to be assigned a spot in the topology. The selection of a topology from a small number available, the assignment of ECUs to spots in the topology, and the assignment of software to ECUs all constitute what GM calls its *deployment architecture*.
- *Vehicle application architecture*. Finally, the components need to be laid out on a vehicle. This architecture determines where the ECUs are stationed, and the type, position, and routing of the wire harnesses to connect the sensors, actuators, and ECUs.

These architectures—functional, implementation, deployment, and vehicle application—institutionalize and add structure to concepts that are deeply ingrained in the organizational and technical fabric at GM. For instance, there are centers of deep expertise in brakes and lighting and keyed/keyless entry systems and dozens of other domains. As part of PLE adoption, these centers are not going to be

discarded in favor of a massive reorganization involving the reorientation, reassignment, and retraining of some 4,000–5,000 engineers.

In order to maintain the functional architecture and its rich set of decomposition structures, the hierarchical production line approach of 2GPLE is needed. Integration areas, domains, subsystems, and functional elements can all be represented with their own production line, importing the production line of smaller, child units. Software components and ECUs can be represented as assets within the production line at the appropriate decomposition level. Across all levels, requirements, design models, and specifications can also be represented as assets. The result is a small and nondisruptive change from the organizational schemes that GM has employed successfully for years.

7.2 Roles

GM's embrace of 2GPLE has led to the creation of a few new and refined engineering roles that have come about as a direct result of piloting their hierarchical product line. The major PLE roles and their broad responsibilities vis-a-vis maintaining the PLE models and artifacts include:

- *Feature Owner*. Feature owners take ownership of GM features (customer-visible features such as cruise control or lane keep assist or hundreds of others that are visible and bring value to car owners). These features are, in a sense, abstractions. They only become tangible when realized by concrete artifacts: requirements, functions, software components, electronic control units, and wiring. In GM's PLE environment, each of those artifacts will also embody variation. It is the feature owner's job to make sure that all of those artifacts in "supporting roles" are adequate and correctly provide the feature to GM's customers.

  A feature owner is the main technical contact to external teams who need to know about the feature from the point of view of assembling a vehicle from this and other features. This engineer is the recognized expert for the functional area regarding the feature's required variants, the system constraints it imposes, and how it integrates onto a vehicle platform.
- *Functional Architect*. This engineer owns a specific area of the functional architecture and as such establishes ownership and boundaries between system-level assets. Together, the functional architects maintain the functional architecture taxonomy introduced above.

  With the advent of the NGT 2GPLE effort, functional architects have taken on a new and critical role. Together, they are the keepers and centralized owners of all of the PLE models. Their job is to ensure that the PLE models produced inside their assigned area by feature owners, asset owners, and others are consistent, fit together, and represent best PLE practice.

Functional architects are each assigned a domain, which will involve several subsystems. As PLE practices are introduced into each domain, functional architects will actually build the PLE models, working with feature owners who are the subject matter experts in each area. For example, the functional architects will mine the feature owners' knowledge about what constitute the features in an area, what profiles (choices of feature combinations) they should provide, and what feature combinations require or exclude other feature combinations. For example, the feature owner for the wipers and washers knows that rear wipers cannot be installed on any vehicle with a rear window that slides open; the functional architect will capture this with an "EXCLUDES" assertion between the rear window type and the wiper/washer configuration.

Under this scheme, the feature owners remain the subject matter experts about their features; the functional architects translate (or help the feature owners translate) that knowledge into well-structured and consistent PLE models.

The PLE models built with Gears for each domain or subsystem take the form of production lines that are then combined by importing them into an overarching production line for the entire vehicle, making full use of the cross-organizational, hierarchical product line aspect of 2GPLE.

- *Product Line Integration Engineer*. This is another new role at GM, brought about by PLE. This engineer collaborates with Vehicle Product Teams in the selection of a "bill-of-features" for a vehicle being planned. The product line integration engineer also collaborates with the feature owners in the identification of the top-level subsystem production line "products" that will be offered up to vehicles. The vehicle team for a vehicle will need the services and advice of a team of product line integration engineers, who together will put together the bill-of-features for that vehicle's electronics system. When the bill-of-features for a vehicle is created, the product line integration engineers will be at the table.

  For example, the vehicle team for the Buick Verano wants to understand what kind of climate control options they can offer with the car (or, to put it another way, what climate control features are eligible for the Verano's bill-of-features, and what the downstream implications are of each). The product line integration engineer responsible for heating, ventilation, and air conditioning (HVAC) systems will offer up various automatic and manual climate control systems. If a vehicle might 1 day be powered by hybrid or next-generation energy and propulsion systems, this might mandate another kind of HVAC system.

  The vehicle teams aren't interested in the details of the features' implementations, but only in how the features will appear to the customer and how they interact with each other. The product line integration engineers, then, manage subsystem "products" that are exposed at the vehicle and bill-of-features level.
- *Asset Owner*. These engineers manage various kinds of assets across their life cycle, and establish variation points in the assets. A requirements engineer is one kind of asset owner. Their responsibilities include migrating requirements from legacy requirements assets (mostly Word documents) into DOORS and, along the way, imbuing those requirements with variation points that support features.

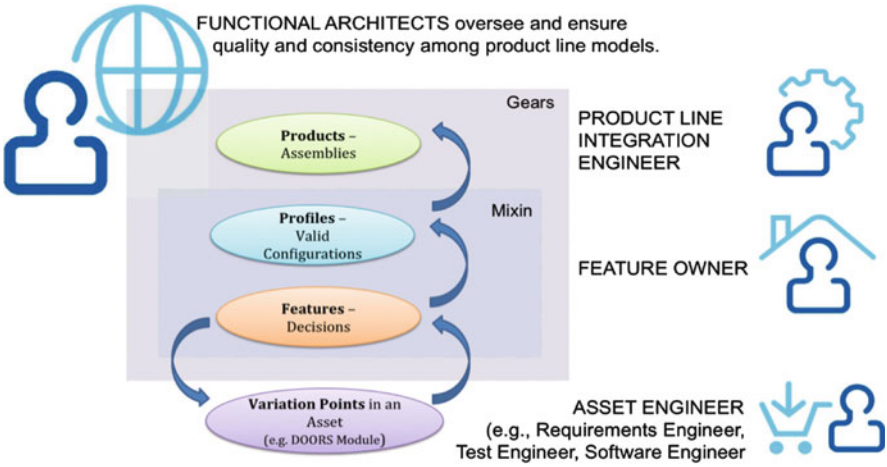


Fig. 15.6 PLE constructs and roles at GM

Asset owners, including requirements engineers, are responsible for modeling the features that their assets make available to the consumers of those assets, and the variations in those features. These features are often strongly suggested, if not identified outright, in existing technical specifications. Thus, feature creation is more of an identification and extraction process, as opposed to an invention process. This helps things go more smoothly and predictably.

Figure 15.6 shows how these roles relate to the PLE concepts discussed previously.

7.3 Organizational Adoption

Launching and institutionalizing [15] this approach at GM has required significant investment over the last 2 years or more, and that investment is ongoing. There has been a group of champions and advocates of the PLE approach throughout the effort. Early on, they sponsored a 2-week workshop to show how the approach (using Gears in concert with DOORS) could tame the requirements for a major subsystem, with variations clearly identified and managed. This pilot effort produced more strong advocates and steered GM towards their current tooling approach.

After that followed a steady series of workshops and technical meetings with senior engineers to work out how to apply the concepts at GM; the eventual results of these meetings include the architectures and roles described above, plus a vision of how features could be used across all of the architectures to describe variation. All the while, the champions practiced internal evangelizing, advocating the approach to management and engineers alike. One-day requirement workshops

were held with subsystem owners to duplicate the results of the first 2-week workshop.

The latter part of 2011 saw the launch of a series of some two dozen *Bill-of-Features Workshops*. These workshops bring together a small group of feature owners and subsystem experts in a particular area—for example, interior lighting. They spend a day learning the PLE approach and then actually using Gears to model the features in their domain. An important goal is to have participants experience the "PLE epiphany," when they see how 2GPLE and the NGT tool suite will help them do their jobs better.

At the start of 2012, after 2 years of establishing buy-in, GM launched a series of training courses. The course series kicks off with a short introduction to PLE at GM and continues with 1-day classroom courses in each of the tools and how they will be used. In concert with the training is the establishment of resources to help engineers once they go back to their desks: Discussion boards, FAQ lists, help desks, and the like. The Bill-of-Features Workshops have continued, with the features created for the domains and subsystems being used to provide a full pallet of vehicle-level variation choices, plus create variation points in assets such as requirements specifications. Thus, the features and profiles in the middle of Fig. 15.6 are being used to inform the product (vehicle) assemblies at the top and the requirements assets at the bottom. The PLE roles identified earlier are all carrying out their respective responsibilities using features as the *lingua franca* for what they do.

Answers to recurring questions are being captured and stored in a "GM PLE Cookbook," which will include a set of patterns and anti-patterns for good practice, a list of FAQs, and a set of naming conventions for product line objects shared across organizations. This will represent a trove of practical knowledge not usually divulged in the product line literature, as well as another aid to institutionalization at GM.

7.4 What Is the End Game?

One of GM's senior electronics engineers characterizes the electronics division's job this way: "We build silver boxes," he said, "load software in them, and wire them together." If they can do that correctly for every vehicle they build, their job is done.

Whatever PLE and NGT can do to help achieve that purpose is a win for GM. The long-term vision is to create a bill-of-features for a vehicle (which manifests as a vehicle-level feature profile in Gears) and automatically derive as much as feasible of the bill-of-materials for that vehicle, including requirements, designs, models, software, calibration data, tests, documentation, allocation of software to hardware, wiring diagrams, and so forth. That vision is years away from being achieved.

However, short of that, there are some intermediate steps that GM is working towards. Examples include

- Migrating all requirements to incorporate Gears variation points that formalize feature-based variations in the system, subsystem, and component requirements
- Generating calibration files and values that will need to be loaded onto the electronics modules or
- Given a set of features on a car and the components that need to be onboard to support those features, generating a list of all of the digital signals that the serial networks will have to accommodate

Longer term goals include calculating certain additive nonfunctional properties of the electronics, such as weight or generated heat or cost.

Even short of this capability, GM is already getting value out of their PLE efforts even before they have started producing instantiated engineering artifacts. Just defining an internally consistent vehicle with consistent versions of subsystems, functional elements, components, and hardware allocations represents a very big step in managing the complexity at hand. To be able to do this in an end-to-end fashion under the auspices of fully interoperating tool suite is a capability not available at GM before now. The automation—in this case, Gears—can do a semantic check on the feature model and report anomalies, such as the fact that this vehicle is supposed to support the lane-keep-assist feature but the instrument cluster chosen for it doesn't have the correct display for that feature, or the chosen physical architecture topology cannot support the serial data communication required.

8 Example: Daytime Running Lights

We conclude by illustrating some of the points in this chapter through an example, which necessarily must be a small one. A *daytime running light* (DRL) is a "lighting device on the front of a roadgoing motor vehicle, installed in pairs, automatically switched on when the vehicle is moving forward, emitting white, yellow, or amber light to increase the conspicuity of the vehicle during daylight conditions" [20].

8.1 DRL Requirements

DRLs are considered a feature at GM; they're certainly visible to the user. But not all cars have them. DRLs are required equipment in Canada, Norway, and Sweden, prohibited in Japan and China and optional in the USA, Europe, Australia, and the rest of the world. (Region of sale turns out to be a major discriminator among features, permitting or precluding a plethora of other features.)

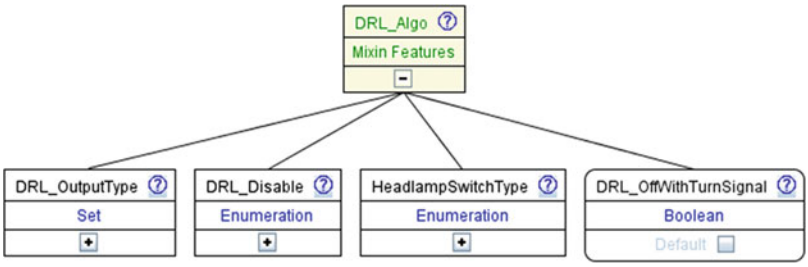


Fig. 15.7 Feature model for daytime running lights

In vehicles that have them, DRLs can be "implemented" by lamps dedicated to that purpose, or by front turn signal lamps, reduced intensity low beam headlamps, full intensity low beam headlamps, parking lamps, or a combination of parking lamps and dedicated lamps.

Just as there are many ways to realize DRLs, there are many choices for how the customer can turn them on and off (including none at all, leaving it up to the car to do so automatically). There is a thicket of requirements concerning when DRLs may, must, and may not be on. For example, in Europe, DRLs must switch off automatically when the front fog lamps (if the car has front fog lamps) or headlamps are switched on, except when the headlamps are "used to give intermittent luminous warnings at short intervals"—that is, flashed.

These and other impinging factors consume page after page in the requirements document for the exterior lighting subsystem, of which DRLs are a member. These requirements are rife with information about what requirements apply under what conditions and be used to identify variations in the DRL feature.

Besides being a feature by themselves, DRLs play a part in other features as well. Some realizations of the "Lead me to my car" feature flash the DRLs when the key fob is pressed. Police vehicles have turn-everything-on features, which include dedicated DRLs if the car is so equipped. Cornering lamps, another feature, can only come on under certain conditions and affect DRLs if they share output devices.

8.2 Modeling DRLs

The feature owner for DRLs is responsible for understanding how the DRL feature is realized, the variations it includes, and any variant capabilities required because of its appearance in other features.

Figure 15.7 shows a preliminary feature model for DRLs. The feature model captures the output type (what lamps on the vehicle can be used), if and how DRLs can be disabled, and how DRLs are integrated with the car's headlamp controls turn signals, respectively.

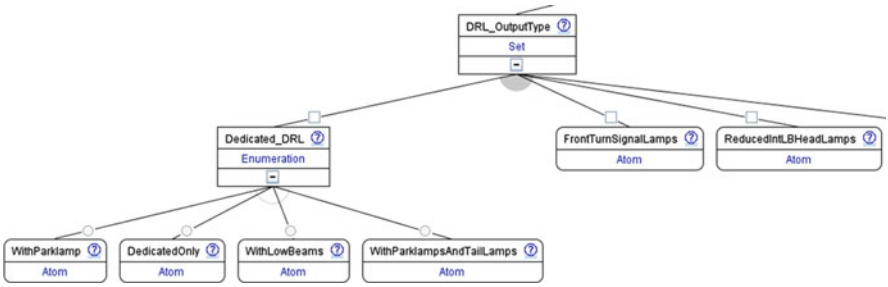


Fig. 15.8 Partial expansion of the OutputType feature of DRLs

Figure 15.8 shows how the output type subfeature is expanded to take into account all the possibilities. The output type is modeled as a set; a vehicle can have any number of ways of realizing the DRL feature, or none at all.

However, if a vehicle realizes the DRL feature with low-intensity headlamps, then it cannot realize it with high-intensity headlamps as well and vice versa. An assertion captures this:

```
NOT DRL_Algo.DRL_OutputType >= {ReducedIntLBHeadLamps,
FullIntLBHeadLamps}
```

This says that the OutputType set cannot include both of the indicated values; in Gears, the symbol ">=" (when used in an expression involving sets) means "is a superset of."

The DRL feature owner builds these feature models in Gears, under the conventions and standards developed by the functional architects, and in particular the functional architect for the exterior lighting domain. He or she will also build a matrix for the DRL feature model that specifies a small number of flavors of the DRL feature that can be made available to Product Line Integration Engineers working to assemble vehicles from features.

Simultaneously, requirement engineers who own the requirements for this feature work to turn the DRL requirements into a Gears asset, with variation points that (when exercised) produce requirements that correspond to the requirements needed for each case.

8.3 DRLs and Deployment

Those responsible for choosing a deployment architecture are another kind of asset owner; their asset is the set of ECUs needed for the features on board, and the variation points they can provide based on features and feature combinations chosen. These variation points include basic network topology (currently two are available; more may be added), how many ECUs will populate a topology, what the choice of ECU hardware will be, and the allocation of software components to each ECU.

8.4 DRLs and Other Features

Feature owners for other features that interact with DRLs (such as the lead-me-to-my-car feature owner) will need to reference DRLs in their feature models and profiles. They will coordinate with the DRL feature owner, under the auspices of the functional architects for the including domain or domains, to make sure that the DRL feature can be referenced, by importing the DRL domain-level mixin into their domain production line.

Other domains than exterior lighting will need the same ways to refer to DRL in their feature models. For instance, the switches that turn DRLs on and off are part of the Body domain, whereas any indication that DRLs are on are part of the Displays domain.

8.5 DRLs and the Vehicle

Finally, those defining a vehicle type and the myriad of features combinations that GM wishes to offer with it, can do so by importing all of the domains' and integration areas' production lines, adding the highest layer to the product line hierarchy. They will also define a matrix of "products" for each vehicle, defining combinations of features in concert with each other.

9 Outlook

The guiding PLE vision at GM is the ability to engineer vehicles—across the full life cycle—according to a "bill-of-features" rather than the traditional "bill-of-materials." Although still very much a work in progress, the GM experience has already revealed a number of lessons about mega-scale product line engineering.

First, the product line experience at General Motors can be seen as intensively applying aspects of what has been called Second-Generation Product Line Engineering. This new perspective brings the following ideas, which previous approaches always allowed but never stressed, to the forefront:

- A focus on features as the "lingua franca" of variation and product selection; the "bill-of-features" replaces the "bill-of-materials" as the key engineering artifact for product derivation. At GM, functional architects and feature owners cooperate to capture the features in Gears models across domains and subsystems and integration areas. Vehicle-level engineers can choose from the profiles provided from across the functional architecture, and asset owners design and install variation points in their assets that are expressed in terms of those very same features.

- Treatment of artifacts across the entire life cycle completely consistently with each other, and consistently with the software, as first-class components of the product line and the derived products. At GM, requirements and calibration sets are the near-term artifact targets, with, wiring data and source code components on the horizon.
- An emphasis on high-quality automation at the center of a production line, to quickly turn a bill-of-features into a set of instantiated lifecycle assets. At GM, this automation takes the form of the Gears configurator, working at all levels of the functional architecture and in separate groups, from vehicle-level engineers, down through domains and subsystems, as well as assets.
- A CM and PLE approach geared to multi-baseline multiproduct management in a way to reduce the order of complexity from $O(n^2)$ to $O(n)$. GM has embraced this configuration management paradigm by only managing the shared assets and not their auto-configured instances.
- Taking multi-organizational management in stride, by providing feature model concepts such as mixins and imported (hierarchical) production lines, to reflect the structure of engineering activities and domain knowledge present in an ultra-large organization. This is perhaps the most overarching aspect of the GM story. PLE would not have worked at GM by overthrowing their longstanding multi-level functional architecture and corresponding organizational hierarchy. Such a radical departure from their current way of thinking about and organization to build vehicles would probably have ruled out any attempt at a large-scale PLE effort; the organizational change would have been too forbidding. Instead, they are able to apply PLE at every level and in every group of their functional architecture and make their PLE models work together using the Gears constructs of mixins, matrices, and imported production lines.

GM's PLE approach embodies a compelling need for each one of these characteristics. They have embraced feature-based variation at all levels of their product line to the extent that they are transitioning from an organization dominated by subsystem owners to one where *feature owners* play the key role.

Second, the GM experience also validates that a small set of consistent concepts is sufficient to model product lines of inordinate complexity. Features (declarations, types, assertions, and profiles), assets (that embody features, as well as variation points), mixins, and matrices constitute a production line, the "factory" that turns feature choices into asset instances. Allowing production lines to import other production lines gives us unlimited hierarchy, which can map to any organizational structure in which specialized bodies of knowledge are encapsulated throughout, no matter how many levels deep.

Third, one of the most important aspects of PLE and GM is the application of consistent variation management in artifacts from all across the life cycle (the second bullet above). In order to accomplish this, the automation engine has to embody business partnerships with important tool vendors.

Future work involves continuing the march towards the ultimate "end game": Generating a complete bill-of-materials for a vehicle by starting with its bill-of-features.

Ultimately, GM may investigate merging PLE with product lifecycle management (PLM), which is the technology used in vehicle manufacturing. That would represent a convergence with repercussions across the entire manufacturing industry.

Acknowledgments Our thanks to everyone at General Motors who is involved in making their adoption of 2GPLE a success, and especially those who have been involved through the roll-out effort. Their contributions have made this case study, and the story it tells, possible.

References

1. G.M. regains the top spot in global automaking. Business Day, New York Times, 19 Jan 2012
2. Bosch, J.: Organizing for software product lines. In: Proceedings of the 3rd International Workshop on Software Architectures for Product Families (IWSAPF-3). Las Palmas de Gran Canaria, Spain, 15–17 Mar 2000, pp. 117–134. Springer, Berlin (2000)
3. Catalog of Software Product Lines. Software Engineering Institute, http://www.sei.cmu.edu/productlines/casestudies/catalog/index.cfm
4. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley, Reading, MA (2002)
5. Clements, P., Brownsword, L.: A case study in successful product line development. Software Engineering Institute CMU/SEI-96-TR-016, September 1996
6. Cohen, A.: Russia trails U.S. in pursuit of a fifth-generation jet. UPI, 14 Jan 2009, United Press International, retrieved 2012: http://www.upi.com/Business_News/Security-Industry/2009/01/14/Russia-trails-US-in-pursuit-of-a-fifth-generation-jet/UPI-35761231951126/
7. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A.: Feature-Oriented Domain Analysis (FODA) Feasibility Study (CMU/SEI-90-TR-021, ADA235785). Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (1990)
8. Krueger, C.: BigLever software gears and the 3-tiered SPL methodology. In: Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion (OOPSLA '07), pp. 844–845. ACM, New York (2007). doi:10.1145/1297846.1297918, http://doi.acm.org/10.1145/1297846.1297918
9. van der Linden, F.J., Schmid, K., Rommes, E.: Software Product Lines in Action. Springer, New York (2007)
10. Office of the Deputy Under Secretary of Defense for Acquisition and Technology. Systems and Software Engineering. Systems Engineering Guide for Systems of Systems, Version 1.0. ODUSD(A&T)SSE, Washington, DC. http://www.acq.osd.mil/sse/docs/SE-Guide-for-SoS.pdf (2008)
11. Parnas, D.L.: On the design and development of program families. IEEE Transactions of Software Engineering **SE-2**(1), 1–9, March 1976
12. Paur, J.: Chevy Volt: King of (Software Cars). Wired, 5 Nov 2010, http://www.wired.com/autopia/2010/11/chevy-volt-king-of-software-cars/
13. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer, Berlin (1998)
14. Software Engineering Institute. A Framework for Software Product Line Practice, version 5.0. http://www.sei.cmu.edu/productlines/frame_report/index.html
15. Software Engineering Institute. A Framework for Software Product Line Practice, version 5.0: Launching and Institutionalizing. http://www.sei.cmu.edu/productlines/frame_report/launch.inst.PL.htm
16. SPLC Software Product Line Hall of Fame. http://splc.net/fame/gm.html
17. Van der Linden, F., Schmid, K., Rommes, E.: Software Product Lines in Action. Springer, Heidelberg (2007). Chapter 5

18. Villanueva, J.C.: Atoms in the Universe. The number of atoms in the observable universe is approximately 10^{80} or 2^{260}. http://www.universetoday.com/36302/atoms-in-the-universe (2009)
19. Weiss, D.M., Lai, C.T.R.: Software Product-Line Engineering: A Family-Based Software Development Process. Addison-Wesley, Reading, MA (1999)
20. Wikipedia. Daytime running lamp. http://en.wikipedia.org/wiki/Daytime_running_lamp
21. Wikipedia. Fifth-generation programming language. http://en.wikipedia.org/wiki/Fifth-generation_programming_language

Part IV
Emerging and Research Topics in Software Variability

Chapter 16
Dynamic Software Product Lines

Svein Hallsteinsen, Mike Hinchey, Sooyong Park, and Klaus Schmid

What you will learn in this chapter

- *The importance of dynamic software product lines.*
- *The role of software product lines in adaptive systems.*
- *The underpinnings of dynamic software product lines.*

1 Introduction[1]

In emerging domains such as ubiquitous computing, service robotics, unmanned space and water exploration, and medical and life-support devices, software is becoming increasingly complex with extensive variation in both requirements and resource constraints throughout its lifetime. This is partly due to the dynamic nature of modern computing and network environments and the way they are used and partly due to the need to survive evolution of the same. For example, computing

[1] This chapter is partially based on a previous article by the authors (cf. [4]).

S. Hallsteinsen
SINTEF ICT, Trondheim, Norway
e-mail: svein.hallsteinsen@sintef.no

M. Hinchey (✉)
Lero-The Irish Software Engineering Research Centre, University of Limerick, Limerick, Ireland
e-mail: mike.hinchey@lero.ie

S. Park
Sogang University, Mapo-gu, Seoul, South Republic of Korea
e-mail: sypark@sogang.ac.kr

K. Schmid
University of Hildesheim, Hildesheim, Germany
e-mail: schmid@sse.uni-hildesheim.de

R. Capilla et al. (eds.), *Systems and Software Variability Management*,
DOI 10.1007/978-3-642-36583-6_16,

and communication resources, user requirements, and interface mechanisms between software and hardware devices such as sensors can change dynamically at runtime. As a result, a higher degree of adaptability is demanded from software systems. At the same time, developers face growing pressure to deliver high-quality software with extensive functionality, on tight deadlines, more economically.

This challenge is by now widely recognized and has led to a significant number of different efforts to achieve (self-)adaptation properties in software systems. Examples for this are self-adapting systems [1], autonomous systems [2], agent-based systems [3], reflective middleware, emergent systems, etc. All these approaches share a common underlying goal: to make software systems more flexible than ever before. In this chapter aim at a discussing a particular subclass of approaches addressing this goal, which borrows essential ideas from product line engineering to achieve flexibility at runtime and, hence, is called *dynamic software product lines* [4].

Software product lines (SPL) have been successful in coping with varying requirements by allowing the derivation of product variants from a common asset base. However, to cope with the challenges discussed above, *dynamic* SPLs (DSPL) aim at producing software capable of adapting both to fluctuations and evolution in user needs and resource constraints at runtime. DSPLs may bind variation points initially when software is launched to adapt to the current environment as well as during operation to adapt to changes in the environment.

Although traditional SPL engineering recognizes that variation points are bound at different stages of development, and possibly also at runtime, the main body of research focuses on binding variation points at the latest at system startup. Traditional approaches to software product line engineering focus in their methods and techniques on this scenario. In contrast, DSPL engineers typically aren't concerned with pre-runtime variation points. However, they recognize that in practice mixed approaches might be viable, where some variation points related to the environment's static properties are bound before runtime and others related to the dynamic properties are bound at runtime [4].

1.1 Product Lines

Henry Ford (1863–1947), founder of the Ford Motor Company, is often viewed as the father of factory automation and the use of assembly lines, which he introduced and expanded in his factories between 1908 and 1913 in building his Model T line of motorcars.

Ford is famously quoted as saying that "Any customer can have a car painted any colour [sic] that he wants so long as it is black" [5]. He is noted for his introduction of mass production and the assembly line. What is less known is that this was achieved through the use of interchangeable parts, based on earlier ideas by Honor Blanc and Eli Whitney, which significantly streamlined the process over previous approaches where parts were often incompatible and one difference in a product meant that the

entire development process had to be restarted. The result was economies of scale and a line of motor cars that were affordable, built quickly, and to a high quality standard, even if the choice of colors, etc., was extremely limited.

Ford's ideas have been influential in the development of the idea of using product lines sometimes called Product Family Engineering or Product Line Engineering (PLE), which affords economies of scope. As Greenfield et al. point out: "Economies of scale arise when multiple identical instances of a single design are produced collectively, rather than individually. Economies of scope arise when multiple similar but distinct designs and prototypes are produced collectively, rather than individually" [6]. Economies of scope imply mass customization. Mass customization is defined as "producing goods and services to meet individual customer's needs with near mass production efficiency" [7].

PLE provides an alternative to mass production in order to create customized products as similar variants of the mass produced products; that is, reusing core assets (a "platform") to develop similar products for a market segment. Its key aim is to create an underlying architecture of an organization's product platform, based on commonality and variation. Product variants can then be derived from the basic product family, reusing common components and using various combinations to create a variety of products.

The use of product lines involves engineering new products in such a way that it is possible to predictably reuse product components and offer variability and choice while simultaneously decreasing costs and development lead time. The software development community has caught on to the usefulness of the approach with the idea of Software Product Lines.

1.2 Software Product Lines

The Software Engineering Institute defines a Software Product Line (SPL) as "a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way" [8]. The approach has been successfully used to develop a wide variety of product lines in a number of different domains, ranging from avionics over medical equipment to information systems, in a wide variety of organizations, ranging from five developers to more than a thousand.

Consistently, strong achievements in terms of time-to-market, cost reduction, and quality improvement have been achieved. The interested reader is directed to the Product Line Hall of Fame [9]. In-depth discussions of product lines case studies are given in many literature sources like [10].

Fundamental to product line engineering is a shift from a single system point-of-view to an integrated understanding of a set of products. As a result, the differences or variations among products become a primary concern. Thus management of variation—so-called variability management—is a core capacity of PLE. The idea

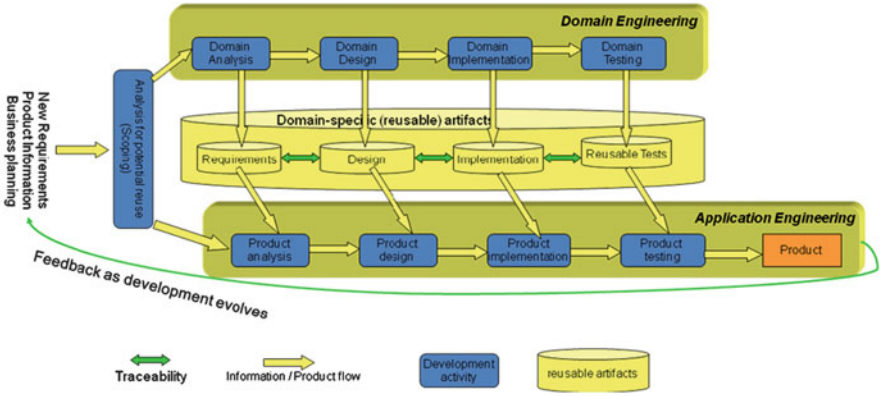


Fig. 16.1 The product line engineering process

is to separate all products in the product line into three parts and to manage them throughout development:

- *Commonality*: artifacts, which are common to all products in the product line. On the requirements side these are common requirements; in the implementation this results in common components.
- *Reusable variation*: aspects that are common to some, but not all, products in the product line. This is the powerhouse of product line engineering; by providing a low-effort mechanism with predictable properties, it is easy to assemble new products by reusing existing assets.
- *Product specifics*: no matter how well designed a product line is, there will always be requirements that are specific to individual products. Here, it is key not to waste any effort on generic development for aspects that will be used only once.

In addition to variability management, a second key principle of product line engineering is the use of a twin-lifecycle approach. We separate development into domain engineering and application engineering (cf. Fig. 16.1). Domain engineering is responsible for an analysis of the product line as a whole and for producing any common and (reusable) variable parts. Application engineering is then responsible for, for all aspects peculiar to individual products, such as the derivation of product requirements, the selection and integration of reusable parts, and possibly the production and integration of product-specific parts. Both life cycles can rely on fundamentally different processes (e.g., agile application engineering combined with plan-driven domain engineering).

The Software Product Line approach has received increased attention as a means of coping with product diversity, especially as software engineers and developers are faced with increasing pressure to deliver high-quality software with more functionality on strict release deadlines ever more economically.

Traditional product line engineering supports the binding of variability (i.e., the selection of the variations to be exhibited in a specific product) at different points in time: the so-called *binding time*. However, the focus was always on development time binding, ranging from the design stage to the moment of initialization of the system, with a strong focus on preprocessing, compiling, and linking. In the light of the need for a higher degree of adaptability, the need arose to provide variation also at runtime. This is the main characteristic of a system that qualifies as a DSPL, as we will further discuss in the next section.

2 DSPL

Dynamic Software Product Lines (DSPL) may be viewed as an area of research where we try to apply ideas developed in the SPL community like variability modeling, common assets shared by product variants, and automated product derivation to build software that adapts dynamically to fluctuations in user needs, environmental conditions, and resource constraints at runtime [4].

Relying on product line ideas and transferring them to the realm of runtime adaptation is one approach to build self-managed systems, i.e., software systems, which can modify their own behavior with respect to changes in their operating environment and thus adapt at runtime to the changing environment. Such systems are becoming more essential for complex network management, for use in unmanned space and underwater exploration, for complex medical and life support devices.

The central shift from the traditional view of software product lines to dynamic software product lines is that variation points are bound at runtime. First, when software is launched to adapt it to the current situation and subsequently to re-bind variation during operation to adapt the software to changes in the situation.

Thus, DSPL is basically not concerned with pre-runtime variation points. However, it recognizes that in practice mixed approaches may be viable, where some variation points related to static properties of the environment of use are bound before runtime, while others related to the dynamic properties are bound at runtime. Examples of approaches that seek to unify pre-runtime and runtime variation points into the same development framework are EASy-Producer [11] or CAPucine [12]. Both use aspect weaving as their primary mechanism for variability.

If a dynamic software product line is only variable at runtime, it will be perceived as a single adaptive system. On the other hand, if also development time binding is involved, it will still be perceived as a product line, where some or all products are adaptive systems. Some approaches support such a combination of binding times, even for the same variability, thus avoiding the need to duplicate the implementation of features [11, 13, 14].

Transferring product line concepts to the dynamic situation mostly focuses on the aspect of variability management. Variability management is on the one hand responsible to model the variability that is supported by the product line and on the

other hand responsible for mapping this to the level of implementation, i.e., to determine what implementation impact a certain variability has. This can be readily transferred to the level of runtime binding by introducing a model that makes the potential variation explicit and supports the mapping to implementation consequences *at runtime*. Along the lines of traditional product lines, this model is a discrete model, mapping out certain alternatives and optional characteristics, and relating them to implementation consequences, just as existing approaches to product line variability do. Due to its importance, we regard the existence of such an explicit variability model as a defining piece for a DSPL. Actually, we will refer to any approach that relies on variation management at runtime for dynamic adaptation as a DSPL.

In dynamic software product lines, monitoring the current situation and controlling the adaptation are central tasks. This may be performed manually by an operator or by the user, or automatically performed by the application or by generic middleware. In this aspect, the DSPL approach is less restrictive than other approaches to adaptive systems like autonomous computing [2] or self-adaptive systems [1], etc., which demand that the system is able to autonomously perform the data gathering and decision making. Another difference is that the models that guide these variabilities are akin to product line variability models, although evaluated at runtime. This excludes certain ways of dealing with adaptation. As a DSPL we explicitly do not take into account cases where the whole system needs to be halted and restarted in order to perform an adaptation, no matter how flexible it seems (like [15]).

Evolution has been a concern in traditional SPL research. It is recognized that it is in general impossible to foresee all functionality or variability that will be needed in an SPL up-front, so evolution must be expected. In application areas such as those mentioned above it becomes more and more important to be able to evolve both the functionality and the adaptation capabilities of deployed systems. Here, DSPL can help. On a first level a DSPL is able to adapt to varying circumstances at runtime. Thus, it can, for example, address evolution needs that are created by changes in the environment. However, despite these capabilities the basic range of variability, which is supported typically remains fixed. However, approaches that combine DSPL with open variability (i.e., variability is identified, but dynamically more alternatives of realizing a variability can be added) may have an answer, which enables to dynamically extend the scope of the DSPL at runtime. The details of how to do this while providing certain correctness guarantees are still to date a challenging research problem.

It is worth noting that although Dynamic Software Product Lines are built on central ideas of SPL, there are also differences. For example, the focus on understanding the market and let the market drive the variability analysis may not be so relevant to DSPL, where concern is about adaptation to variations in individual needs and situations rather than market forces. Also reconfiguring a running product has additional challenges compared to configuring a new product instance, like pausing the execution and transferring state.

Dynamic Software Product Lines is an emerging and promising area of research with clear overlaps to other areas of research besides SPL, notably:

- Research related to self-* (adapting/managing/healing...) systems and autonomic computing is also concerned with situation monitoring and adaptation decision making [1], and DSPL may be seen as one among several approaches to building such systems
- Agent-based software engineering [3] and multi-agent systems, which focus on the use of agents and organizations (communities) of agents as the main abstractions

DSPL brings to this arena a number of techniques, which have proven very successful in a different, but strongly related field. This supports the expectation that these techniques can also contribute to the general research direction of adaptive and runtime-flexible systems.

3 Conclusion

We have presented the notion of *dynamic software product line (DSPL)*. In summary, to qualify as a DSPL we envision a system, which was developed based on (some) SPL principles and has the following properties:

- Dynamic (runtime) variability: (re-)configuration and binding happen at runtime, changing the binding several times during the lifetime of the system
- Dealing with changes in the environment or triggered by users (e.g., changes in functional and/or quality requirements)
- The system can be seen as a product line where the running system switches at runtime from one variant to another
- It uses a variability management approach for identifying the different possible runtime variants
- An integrated model of the various runtime variations of the system exists (comparable to a domain model for a design-time product line)
- An architecture exists that describes the architectural variation that may happen (comparable to a reference architecture in classical PLE)

DSPL is suited to develop systems, which are adaptable at runtime by manual intervention, as well as autonomic or self-adaptive systems. In the latter case it is also necessary to address context or situation awareness and automatic adaptation reasoning and decision-making.

At this point there exists a small, but growing community, which aims to take the DSPL way to transfer existing knowledge and solutions to the realm of adaptive systems. First surveys exist that capture the state of the field [16, 17], which is at this point still rather fragmented. One of the most important challenges at this point is probably to address open runtime variability, i.e., variations that are not foreseen

at development time, but need to be integrated at runtime. This might provide a pathway for dynamic evolution of product lines.

References

1. Cheng, B., Giese, H., Inverardi, P., Magee, J., de Lemos, R.: Software engineering for self-adaptive systems: a research road map. In: Software Engineering for Self-Adaptive Systems, Dagstuhl Seminar Proceedings, 08031, available at: http://drops.dagstuhl.de/opus/volltexte/2008/1500 (2008)
2. Kephart, J., Chess, D.: The vision of autonomic computing. IEEE Comput. **36**(1), 41–50 (2003)
3. Jennings, N.: On agent-based software engineering. Artif. Intell. **177**(2), 277–296 (2000)
4. Hallsteinsen, S., Hinchey, M., Park, S.Y., Schmid, K.: Dynamic software product lines. Computer **41**(4), 93–95 (2008)
5. Ford, H: My Life and Work – An Autobiography of Henry Ford. Heinemann, London (1923)
6. Greenfield, J., et al.: Software Factories, Assembling Applications with Patterns, Models Framework, and Tools. Wiley, Indianapolis, IN (2004)
7. Tseng, M.M., Jiao, J.: Mass customization, In: Handbook of Industrial Engineering, Technology and Operation Management. Wiley, New York (2001)
8. Software Engineering Institute. Software product lines – overview. Online available at: http://www.sei.cmu.edu/productlines (last checked: 7.4.12)
9. The product line hall of fame. Online available at: http://splc.net/fame.html (checked: 7.4.12)
10. van der Linden, F., Schmid, K., Rommes, E.: Software Product Lines in Action – The Best Industrial Practice in Product Line Engineering. Springer, Heidelberg (2007)
11. Schmid, K., Eichelberger, H.: Model-based implementation of meta-variability constructs: a case study using aspects. In: Second International Workshop on Variability Modeling of Software-Intensive Systems (VAMOS), ICB-Research Report No. 22, ISSN 1860-2770, pp. 63–71 (2008)
12. Parra, C., Blanc, X., Cleve, A., Duchien, L.: Unifying design and runtime software adaptation using aspect models. Sci. Comput. Program. **76**(12), 1247–1260 (2011)
13. van der Hoek, A.: Design-time product line architectures for any-time variability. Sci. Comput. Program. **53**(30), 285–304 (2004)
14. Dolstra, E., Florijn, G., de Jonge, M., Visser, E.: Capturing timeline variability with transparent configuration environments. In: International Workshop on Software Variability Management. ICSE Workshop (2003)
15. White, J., Schmidt, D.C., Wuchner, E., Nechypurenko, A.: Automating product-line variant selection for mobile devices. In: Software Product Line Conference (SPLC), pp. 129–140 (2007)
16. Bencomo, N., Lee, J., Hallsteinsen, S.: How dynamic is your dynamic software product line? In: Workshop on Dynamic Software Product Lines, pp. 61–68 (2010)
17. Burégio, V., de Lemos Meira, S., de Almeida, E.: Characterizing dynamic software product lines – a preliminary mapping study. In: Workshop on Dynamic Software Product Lines, pp. 53–60 (2010)

Chapter 17
Variability in Autonomic Computing

Carlos Cetina and Vicente Pelechano

What you will learn in this chapter

- *That it is possible to use variability models to support building autonomic systems from system design to execution.*
- *The role that variability models can play in the reference model for autonomic control.*

1 Introduction

Autonomic Computing transfers maintenance responsibilities to the software itself. By automating tasks such as installation, healing, or updating, system operation is simplified at the expense of increasing its internal complexity [1]. A system with autonomic capabilities installs, configures, tunes, and maintains its own components at run-time. Overall, an autonomic system must function consistently and reliably in the absence of detailed human involvement by means of fulfilling some externally defined purpose [2, 3]. This chapter shows how variability models can be used to provide a richer semantic base for the run-time decision-making related to Autonomic Computing.

C. Cetina (✉)
Universidad de San Jorge, Villanueva de Gállego, Zaragoza, Spain
e-mail: ccetina@usj.es

V. Pelechano
Universidad Politécnica de Valencia, Valencia, Spain
e-mail: pele@pros.upv.es

R. Capilla et al. (eds.), *Systems and Software Variability Management*,
DOI 10.1007/978-3-642-36583-6_17,

2 Motivational Example

Variability models enable to specify not only current features of a system but also potential features since they may be activated in the future. In response to changes in the context, the system itself can query these variability models in order to determine the necessary modifications to its architecture. For instance, a smart home system can trigger the activation of both *In Home Detection* and *Occupancy Simulation* features when all the inhabitants leave the home.

Autonomic computing capabilities can address some of the adaptation and reconfiguration challenges of the Smart Home domain [4]. First, because of its nature as a shared environment, different users use the same room over time. Each user has its own preferences for the room, which should be adjusted to improve the quality of their stay; second, the preferences of the users change depending on the activity performed (e.g., the users usually have different preferences when they are watching a movie than when they are working). In particular, to reduce this configuration effort, the following autonomic capabilities can be provided:

- *Self-configuring*. New kinds of devices can be incorporated to the system. For example, when a new movement/presence detector is added to a home location, the different smart home services such as security or lighting control should automatically make use of it without requiring configuration actions from the user.
- *Self-healing*. When a device is removed or fails, the system should adapt itself in order to offer its services using alternative components to reduce the impact of the loss of the device. For example, if an alarm fails, the Smart Home can make the home lights blink as a replacement for the failed alarm.
- *Self-adaptation*. The needs of users are different and change over time. The system should adjust its services in order to fulfill user preferences. For example, when all users leave home, services in the home should be reorganized to give priority to security.

In fact, the autonomic community is more and more identifying a system as autonomic only if it exhibits more than one of the self-management properties described earlier.

3 Autonomic Computing Through Variability Models

To achieve autonomic computing, variability models can provide support to autonomic system engineers from system design to execution. At design time (see top of Fig. 17.1), it is possible to take advantage of current variability and context modeling techniques in order to specify the context and architecture of the system, and how the system architecture can be adapted to manage context changes.

Furthermore, this stage also benefits from the whole range of typical gains brought by SPL approaches (i.e., reuse and automation). In fact, it is also possible to take advantage of current techniques for variability analysis (see Chap. 11) in

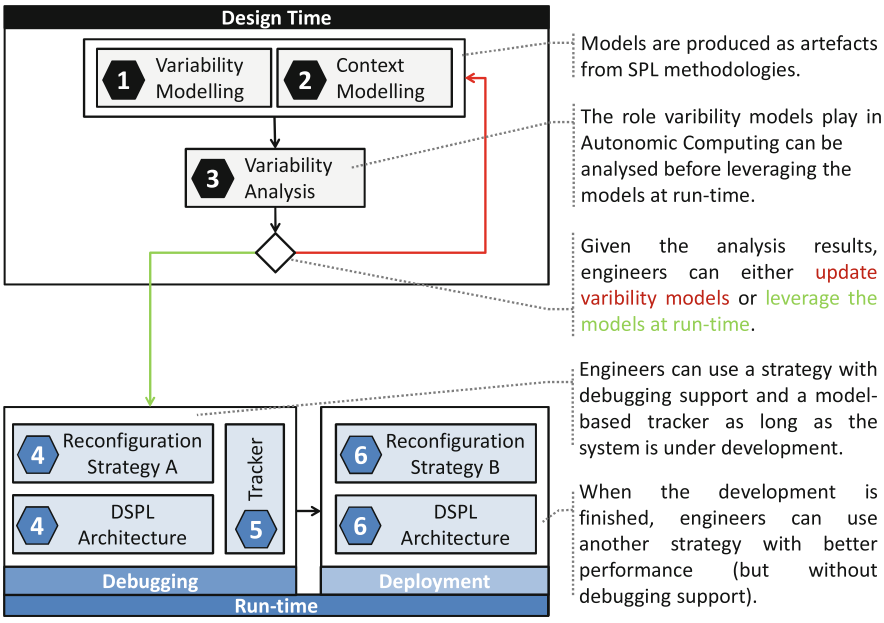


Fig. 17.1 Overview of the process

order to conduct a thorough analysis of the models for the purpose of validating the autonomic behavior.

At run-time (see bottom of Fig. 17.1), the variability knowledge can be used to support Autonomic Computing. In this way, the modeling effort made at design time is not only useful for producing the system but also for providing autonomic behavior during execution. To enable autonomic behavior, the system must evolve from one configuration to another by itself. Since the reconfiguration is driven by variability models at run-time, a Model-based Reconfiguration Engine is required. MoRE [5] is an implementation of this Model-based Reconfiguration Engine that uses the variability models to determine how the system should move from a consistent architecture to another consistent architecture by means of reconfiguration actions. These reconfiguration actions modify the system components accordingly. Specifically, MoRE defines how a set of components cooperate to change from one configuration of the product line to another.

3.1 Process to Achieve Autonomic Computing

Figure 17.1 presents an overview of the process to achieve autonomic computing through variability models. Specifically, this process features six steps. For each one of these steps, the following information is provided: name of the step, a brief description, and tool support.

- *Step 1*. Specifies the variability of the reconfigurable system.

  Description. The design of an autonomic system is guided by scope, commonality, and variability (SCV) analysis [6]. SCV captures key characteristics of the reconfigurable system, including its (1) scope, which defines the domain of the system, (2) commonalities, which describe the attributes that come up across all feasible configurations of the system, and (3) variabilities, which describe the attributes unique to the different configurations of the system.

  Tool Support. MOSKitt (http://www.moskitt.org) is a free Modeling platform, built on Eclipse. This modeling platform provides editors for several modeling languages such as Feature models, BPMN or UML, as well as code generation capabilities.

- *Step 2*. Specifies the context of the reconfigurable system.

  Description. The context of the reconfigurable systems is specified by means of the OWL language. This language provides a vocabulary for describing system context knowledge and for specifying conditions in the context. The fulfillment of these context conditions triggers a set of changes in the variants that conform the system configuration.

  Tool Support. Protégé-OWL (http://protege.stanford.edu) is a free open source ontology editor and knowledge-base framework. An OWL ontology may include descriptions of classes, properties, and their instances. Given such an ontology, the knowledge-based framework specifies how to derive its logical consequences, i.e., facts not literally present in the ontology, but entailed by the ontology instances.

- *Step 3*. Analyzes the reconfigurations before performing them.

  Description. The configurations resulting from the simultaneous fulfillment of context conditions are validated at design time. This enables us not only to obtain a valid–invalid tag for each configuration but also to know the reasons why a particular configuration is invalid. Given this information, either the variability constrains or the context conditions can be updated to achieve a specification free of invalid configurations that can be used at run-time.

  Tool Support. FaMa is a framework for automated analysis of feature models that integrates some of the most commonly used logic representations and solvers proposed in the literature. This framework enables to determine if a system configuration is valid (according to variability constraints), and it can also provide explanations about invalid configurations.

- *Step 4*. Debugs the run-time reconfigurations.

  Description. Given the fact that not all potential run-time failures can be anticipated during system design [7], it is possible to set up MoRE with a debugging-enabled reconfiguration strategy. This strategy keeps the history of system configurations. Therefore, the suggestion is to use this strategy as long as the system is under development.

  Tool Support. MoRE featuring a debugging-enabled reconfiguration strategy.

- *Step 5*. Keeps track of the reconfigurations.

  Description. In the context of experimentation, MoRE can store trace entries about the reconfigurations. This provides information for a posterior analysis, which ranges from context conditions to reconfiguration plans.
  Tool Support. MoRE featuring the reconfiguration tracker.

- *Step 6*. To deploy the system in the target platform.

  Description. Once the development is finished, there is no interest in debugging information any longer. Therefore, MoRE can be set up with another reconfiguration strategy which lacks debugging support but achieves better performance.
  Tool Support. MoRE featuring a performance-oriented reconfiguration strategy.

Although some of the steps that conform the process to apply the approach can be skipped (for instance, variability analysis or reconfiguration debugging), the recommendation is to perform all of them. The whole process (as is proposed in this section) is conceived to achieve a system free of unexpected reconfigurations at run-time.

4 Run-Time Platform for Variability Models

To achieve autonomic computing, it is also required an execution platform for variability models at run-time. In particular, MoRE is a model-based version of the reference model for autonomic control, which is called the Monitor, Analyze, Plan, Execute (MAPE) loop [8]. The overall reconfiguration steps are outlined in Fig. 17.2. A context monitor uses the run-time state as input to check context conditions. If any of these conditions is fulfilled, then MoRE queries the run-time models about the necessary modifications to the architecture. Given the model response, MoRE elaborates reconfiguration actions which (1) modify the system architecture and (2) maintain the consistency between the models and the architecture.

The above model-based version of IBM's reference model for autonomic control makes an intensive use of models. Context events and system variability are represented by models. Context events are represented by means of OWL ontologies, and system variability is captured by means of variability models. For performing the system reconfiguration, information is extracted from these models. Different model query technologies are used at run-time by MoRE depending on the models involved. MoRE uses SPARQL for OWL manipulation and Eclipse Model Query (EMFMQ) for variability model manipulation.

Finally, the reconfiguration of the system is performed by executing reconfiguration actions that deal with the activation/deactivation of components and the creation/destruction of channels among components. Although the general approach is not platform dependent, MoRE takes advantage of the concrete platform to implement the reconfiguration actions. MoRE makes use of the OSGi framework [9] for implementing the reconfiguration actions by means of the

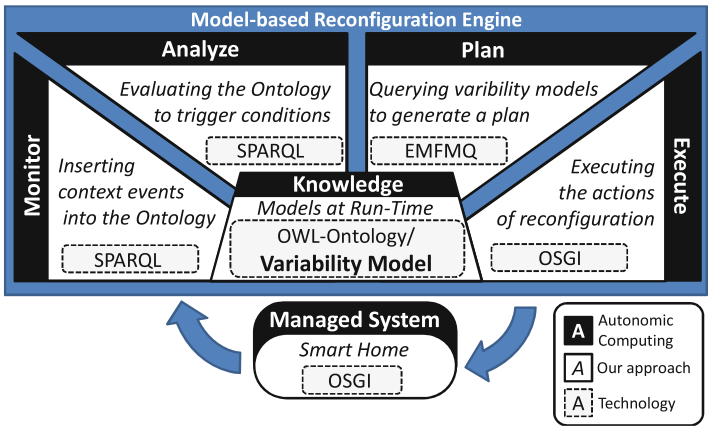


Fig. 17.2 Overview of the run-time reconfiguration

OSGi capabilities to install, start, restart, and uninstall components without having to restart the entire system.

5 Results

The presented work has been validated from three different perspectives (1) scalability of the approach, (2) reliability-based risk of run-time reconfigurations, and (3) degree of autonomic behavior achieved as follows.

Scalability of the approach. The introduced model-based reconfiguration is still subject to the same efficiency requirements as the rest of the system because the execution of the reconfiguration impacts the overall system performance. Therefore, it is interesting analyzing to what extent system performance could be affected using complex models at run-time. Experimentation results show that the approach gathers the necessary knowledge from the run-time models to perform the reconfiguration without drastically affecting the system response.

Reliability-based risk of run-time reconfigurations. The presented approach encompasses systems that are capable of modifying their own behavior with respect to changes in their operating environment by using run-time reconfigurations. However, a failure in these reconfigurations can directly impact the user experience. Thus, it is important to assess the reliability-based risk of run-time reconfigurations, which depends on both the probability that the software product will fail in the operational environment (availability) and the consequences of malfunctioning (severity). Experimentation revealed that the reconfigurations achieved a high level of reliability.

Degree of autonomic behavior achieved. To determine the level of autonomic behavior that can be achieved, it is possible to obtain theoretical results about the autonomic behavior specified by variability models at run-time. Furthermore, users can be asked whether or not they considered the system reaction to be adequate

taking into account the defined context events. Experimentation reveals that acceptance for the reconfiguration scenarios was high. Most of the users considered the behavior provided to be a good response to the context events.

To evaluate the above concerns, a Smart Hotel case study has been developed (see http://www.autonomichomes.com). The Smart Hotel reconfigures its services according to changes in the surrounding context. In particular, a hotel room changes its features depending on users' activities to make their stay as pleasant as possible.

6 Outlook

Autonomic Computing plays a key role in simplifying the use of systems by reducing the need for maintenance. Variability models at run-time turn out as an important contribution to the field of autonomic computing providing metainformation to drive autonomic decision-making. This is done by means of a planned reuse of the efforts invested at the SPL. The benefits are immediate, as the design knowledge and existing variability knowledge can be reused at run-time. The run-time variability models support the autonomic behavior of systems when triggered by changes in the environment. This approach has been applied to an application in the smart-homes domain, obtaining valuable validation of the approach. Finally, the details about variability models supported by MoRE, the underlying technology and data that support the results can be found at [10].

References

1. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. IEEE Comput. **36**(1), 41–50 (2003)
2. Sterritt, R.: Autonomic computing. Innov. Syst. Softw. Eng. **1**, 79–88 (2005)
3. Dobson, S., Sterritt, R., Nixon, P., Hinchey, M.: Fulfilling the vision of autonomic computing. IEEE Comput. **43**, 35–41 (2010)
4. Okeyo, G., Chen, L., Wang, H., Sterritt, R.: Ontology-enabled activity learning and model evolution in smart homes. In: Ubiquitous Intelligence and Computing, 7th International Conference, pp. 67–82 (2010)
5. Cetina, C., Giner, P., Fons, J., Pelechano, V.: Autonomic Computing Through Reuse of Variability Models at Run-time: The Case of Smart Homes. IEEE Computer Society Press, Los Alamitos, CA (2009)
6. Coplien, J., Hoffman, D., Weiss, D.: Commonality and variability in software engineering. Software, IEEE, 15(6), 37–45 (1998).
7. Liu, Y., Ali Babar, M., Gorton, I.: Middleware architecture evaluation for dependable self-managing systems. In: Proceedings of the 4th International Conference on Quality of Software-Architectures, pp. 189–204. Springer, Berlin (2008)
8. IBM. An architectural blueprint for autonomic computing. Technical report (2003)
9. Marples, D., Kriens, P.: The open services gateway initiative: an introductory overview. IEEE Commun. Mag. **39**(12), 110–114 (2001)
10. Cetina, C.: Achieving autonomic computing through the use of variability models at run-time. Ph.D. Thesis, Universidad Politécnica de Valencia (2010)

Chapter 18
Variability in Web Services

Matthias Galster and Paris Avgeriou

What you will learn in this chapter

- *Why service-based systems need special treatment of variability*
- *What types of variability can exist in service-based systems*
- *How we can address variability in service-based systems*

1 Introduction

Service-based computing and associated development paradigms, including service-oriented architecture (SOA), web services, or the idea of "Software as a Service," have gained significant attention in software engineering industry and research. The aim of this chapter is to provide an introduction to *variability* in service-based systems. Within this chapter, we use the term "service-based" for systems that are largely or entirely built from web services [4], with SOA as the primary architectural style.

To briefly illustrate variability in service-based systems, let us consider the example of an online travel agency that communicates with various external businesses (such as airlines, hotel companies, rental car companies) to obtain airfares, hotel prices, etc., and to make reservations. Web services provide a standardized way to exchange information between the online travel agency and the information systems of these external businesses. In case a web service of an

M. Galster (✉)
Department of Computer Science and Software Engineering, University of Canterbury, Christchurch, New Zealand
e-mail: matthias.galster@canterbury.ac.nz

P. Avgeriou
Department of Computer Science, University of Groningen, Groningen, The Netherlands
e-mail: paris@cs.rug.nl

R. Capilla et al. (eds.), *Systems and Software Variability Management*,
DOI 10.1007/978-3-642-36583-6_18,

airline becomes unavailable, a web service of a different airline that offers the desired flight can be used. In this example, variability could be expressed in a *variation point* for selecting an airline web service. *Variants* (or options of alternative services) in this case would be the different airline web services.

In this chapter, we first provide a brief introduction of service-based systems. Then, we motivate and illustrate variability in service-based systems in a real-world example from the e-government domain and argue why variability handling is useful. We will discuss why service-based systems need special treatment with regard to variability and explore how variability in service-based systems could be addressed.

2 Service-Based Systems

Service-based systems are distributed systems that are built from loosely coupled software services. Services are autonomous, platform-independent computational elements that can be described, published, discovered, orchestrated, and programmed using standard protocols for building interoperating applications [1]. Services are developed independent from a particular technology, for example, could be implemented in Java or .NET, as long as they comply with standards and protocols. For example, in local e-government, a municipality can use services offered by the central government, such as a citizen registry, or services to support the processing of taxes. In service-based systems, a service registry enables service consumers to discover, bind, and assemble available services, often at runtime. A service infrastructure (such as an Enterprise Service Bus) connects services to service consumers. Service consumers query the registry and compose applications using a service or a composition of services. Consequently, service-based systems facilitate interoperability and reuse within and across different systems.

Individual services usually correspond to business functions and provide functionality to a large number of anonymous users (end users or other software artifacts), often distributed across organizations [5]. Thus, service-based systems support flexible environments and infrastructures for adaptable business processes. However, the reusability of individual services and service-based systems is determined by their ability to support the variability required to adjust them to different contexts. This means, if services or service-based systems cannot adapt to changing situations and contexts, they can only be reused in a very limited scope. Therefore, enhancing variability in service-based systems and providing methods that help explicitly model and manage variability facilitates highly reusable and configurable services and service-based systems.

3 Variability in Service-Based Systems

Before discussing variability in service-based systems, we present an industrial case from the e-government domain to motivate and illustrate variability in service-based systems: the implementation of national laws in local municipalities in the Netherlands. The national government may approve a law, which then is implemented in municipalities. In the Netherlands, there are more than 400 municipalities; each municipality would implement the law including processes and software systems that help implement the law autonomously. Differences between municipalities are too big to have solutions as one product for all municipalities, yet not too big to be covered by one generic solution to cover possible variants.

A concrete example for this phenomenon is the implementation of the Dutch Law for Social Support (known as the Wet Maatschappelijke Ondersteuning—WMO law). This law mandates rules for providing social support to citizens, such as domestic care. The responsibility and the execution of the WMO law lie with the municipalities. This means, even though the law has been approved by the Dutch national government, the solutions chosen to implement this law differ substantially between municipalities. Variability must cope with static variability (originating from differences between municipalities) but also dynamic variability (i.e., changes to the WMO law once a solution is deployed) and the evolution of the system in its environment. Throughout Dutch e-government initiatives, the "Software as a Service" (or SaaS) model is used. As a result, software providers offer solutions for the WMO law as software services, in a municipality-independent way. To cover the needs of as many municipalities as possible, the SaaS must be customizable to fulfill variations in business processes, functionality, and quality requirements of municipalities.

As discussed in the previous section, service-based systems provide some degree of flexibility by definition. However, it is difficult to build generic service-based systems that can be adapted in different organizations and changing situations. Even the eight fundamental design principles of service-orientation do not consider variability as a key issue when designing service-based systems [6]. Thus, there is a need for handling variability in service-based systems [12] for the following reasons:

- It helps meet Quality of Service and optimize quality attributes. When a currently deployed service does not perform adequately, it can be replaced by a better performing service. This means, configurations of service-based systems can be changed. In e-government, there are multiple vendors for the same software service. Due to the regulated nature of e-government, these services have to provide the same functionality but may differ in terms of reliability or performance. Based on these differences, services can be selected.
- It can enhance the availability of the system. When a service becomes unavailable, an alternative service with the same functionality can be used.

- It allows for runtime flexibility. This means, rebinding of services can be performed at runtime, potentially automatically when needed.
- It allows to handle different instances of one service-based system in different organizations and versions and to adjust the system to operate in diverse environments. This means, in Dutch e-government, the same system could be used in multiple municipalities.
- Individual services are usually not designed with variability in mind to make services highly customizable. By handling variability, artifacts in service-based systems (such as specifications and models) can be designed with variability for planned and enforced reuse [15].
- Related to a more technical aspect, if variability is not handled systematically and parts of a service-based system are adapted in an uncontrolled manner, interoperability problems occur [10]: Other parts of the system affected by the adaptation might not be adjusted properly. For example, in the case of the WMO law, many external parties (such as health care providers, doctors) are involved in completing a business process. If a service is changed in an unsystematic manner, it might not work with these external parties anymore.

3.1 Why Service-Based Systems Need Special Variability Treatment

Service-based computing includes its own design paradigm and design principles, design patterns, a distinct architectural model (SOA), technologies, and frameworks [6]. Thus, in this section, we explore why service-based systems are different compared to traditional reuse-based paradigms (such as product line engineering, component-based development, or object orientation) and therefore need special "treatment" with regard to variability.

- *Dynamic execution environment*: The most significant difference to other reuse-based paradigms is that the dynamic execution environment of service-based systems allows changing systems at runtime. This means, services can be replaced or reconfigured while the system is running [18]. Product lines, for example, focus on compile-time support. Consequently, to fully support variability in service-based systems, events that occur in such systems must be linked to rules to reason about alternatives [7]. This is particularly true in the context of a volatile, distributed service composition in which services can change, fail, become temporarily unavailable, or disappear.
- *Different levels of abstraction*: Service-based systems usually comprise different levels, i.e., a business process level, the architecture level, and a service level with the actual implementation. Each of these three levels might again comprise different levels or descriptions (e.g., the architecture level usually consists of different architecture views and/or layers). The alignment of business and IT is a key concern in service-based systems, more than in other domains. This means,

variability must be traceable through all these levels and variation points in one level need to be translated into variation points in other levels to ensure business/IT alignment. This also means that variability in service-based systems occurs at different levels of abstraction. For example, variability might be provided through parameter values used to invoke a service (service level), or by replacing complete services (architecture level), or by changing the sequence in a workflow (business process level).

- *Nature of individual services*: Another difference lies in the nature of individual services. Services as computational units must accommodate the challenge of meeting requirements for each organization that might use them while crossing boundaries between organizations. This means, when handling variability in service-based systems, there is no centralized authority that handles variability concerns in the individual parts of the system. Also, a high heterogeneity in customer requirements occurs due to anonymous service users. As a result, the range of possible variations between services and service-based systems might be very broad and extremely difficult to anticipate.
- *Organizational issues*: Organizational differences occur as services and service-based systems are no longer developed, integrated, and released in a centrally synchronized way [14]. Instead, services are developed and deployed independently and separately in a networked environment. Developers need to consider the integration of services, third-party applications, organization-specific systems, and legacy systems. Thus, rather than self-contained and isolated software development, service-based systems extend towards enterprise level collaborative exchange of services and components over networks. Therefore (and similar as stated in the previous paragraph), coordinating variability concerns is problematic during service-based development.
- *Quality attributes*: Quality attributes (e.g., performance, maintainability, security, reliability) in service-based systems are more diverse than in other domains and difficult to achieve [8].

3.2 Types of Variability in Service-Based Systems

As mentioned in the previous section, there are different reasons why service-based systems require special treatment with regard to variability. This also means that there are particular types of variability in service-based systems that may not occur in other types of systems. In this section, we therefore discuss types of variability that exist in service-bases systems. However, rather than providing a complete taxonomy, we provide an overview of basic types as well as where variability might occur in service-based systems. Please also note that we do not discuss variability based on technological aspects. A good overview in this regard has been presented by Robak and Franczyk [16]. From a high-level perspective, we differentiate two categories of variability:

1. Variability inside a service, with services as reusable units that can be adapted for different contexts [2]. In the e-government case, one example of variability inside a service is a difference in quality requirements of municipalities with regard to response time.
2. Variability in the service-based architecture (i.e., the composition of services). In the e-government case, examples for variability in the service-based system are different situations in which a service for assessing the need for a wheelchair through a government authority is called, or is not called, based on local regulations in the municipality.

Going into more detail of the first category, variability inside a service, the following types of variability can be found:

- Variability in parameters required by a service [20]: The type of data sent at service invocation can vary. For example, data sent to a service might be a single variable or an array of variables. This type of variability is usually expressed in Web Service Definition Language (WSDL) documents.
- Variability in parameter values [11]: The value of a parameter used at invocation might vary. For example, the age requirement for a wheelchair subsidized by a municipality differs between municipalities.
- Variability in the protocols [20]: Different protocols might be used by clients to communicate with services.

On the other hand, more detailed types of the second category, i.e. variability in a service-based architecture, include the following:

- Logic variability [2, 20]: A service includes operations for providing a certain functionality. The logic is the algorithm or logical procedure used in the operations of the service. A service can provide different logics depending on the requested functionality.
- Variability in the web service flow [2, 17]: A web service flow is a composition of service using a process-based approach. It specifies sets of tasks which are executed by the participants of a process. Additionally, a web service flow defines the execution order of tasks, the data exchange among the participants, and business rules. Web service flow variability expresses that services can be alternatively or optionally executed in a workflow, in different orders. This type of variability is described in Business Process Execution Language (BPEL) specifications for business process models and service flows. A detailed discussion of variability in a web service flow, which also considers technical and implementation aspects (e.g., message exchange), can be found in [17]. The service flow has two abstractions: the service and the related business process. Services are modeled as sets of operations, while the business process defines a flow of activities. Each activity is implemented by executing an operation on a service. Variation points affect the description of the flow needed to perform a set of ordered operations (see variability model in [11]).
- Composition variability [2]: The business process consists of several services to fulfill end user needs. For one service in the workflow, there may be more than

one possible service interface which implements the service with different implementation logics or quality attributes. Variability occurs in selecting the most appropriate service.

- Variability in quality attributes: Quality attributes might vary from one system to another. For example, one municipality might have higher privacy or performance requirements than another one. This type of variability might be specified in Web Service Level Agreement (WSLA) specifications for web service level agreements between the service consumer and the service provider [11]. This type of variability is most difficult to handle and even a major challenge in software product lines.

4 How to Address Variability in Service-Based Systems

We consider three major strategies for addressing variability in service-based systems. First, as currently argued by many authors, we could adopt product line approaches in the service domain [13]. Referring to the types of variability from the previous section, this strategy can be applied for handling variability in the composition of a service-based system if services are treated as features.

Second, as product lines focus on feature and decision models, we can apply new methods and concepts—beyond the product line domain. This strategy has not yet been thoroughly explored. Examples for this strategy include modeling variability from a pattern point of view [20]. Here, pattern approaches can be used to describe variation points. Again, referring to variability types from the previous section, this strategy could be applied for variability in quality attributes.

Third, we can combine existing methods from the product line domain and concepts based on the specific requirements of service-based systems. For example, to support variability at the architecture level of service-based systems, new viewpoints for the product line architecture could be introduced. This strategy could be applied to variability in BPEL, WSDL, or WSLA specifications; variability in parameters requirements by a service and actual parameters; variability in protocols and logic; as well as variability in the web service flow.

In the remainder of this section, we first list principles for how to address variability. Then, we discuss handling variability by utilizing product line paradigms together with traditional web service development (third strategy) as the strategy that is followed most. The other two strategies are omitted due to lack of space.

4.1 General Principles

The four following principles, as general guidelines, can be used to address variability in service-based systems [3]:

1. Recognize commonalities and variations across the scope of multiple service-based products within and across an organization. In the e-government example, commonalities and variations occur due to differences in business processes, local regulations, and infrastructures.
2. Leverage the recognized commonalities by building "core assets" that exist in all product variants, including services across product variants with established points of variation. In the e-government example, core asset services are services that are required in all municipalities, such as billing citizens for obtaining a wheelchair.
3. Address enterprise integration needs that service-based systems must offer. Integration must take variability into consideration. Integration could encompass many systems and involves sharing services across systems. For example, in municipalities, existing legacy systems or third-party software (such as enterprise resource planning systems or data base systems) must be integrated. Legacy systems and third-party software occur in all municipalities but vary between municipalities.
4. Address end-user needs for variation within the service-based system. For example, municipalities can select services as needed to accommodate unique workflows.

4.2 Product Lines and Web Service Development

As mentioned at the beginning of this section, using the product line paradigm to address variability in service-based systems has been the most popular strategy so far [19] as service-based systems and product lines share certain commonalities [3]. A technical comparison between the variability in product lines and variability in service-oriented computing is given by Chang and Kim [2].

Recently, Sun et al. proposed a framework and a tool suite for modeling and managing variability of web service-based systems [18]. This framework addresses runtime and design-time variability and is an extension of COVAMOF. COVAMOF was originally developed to manage variability in software product lines. Sun et al. use UML diagrams and a specifically developed UML profile to model variability. This work builds on VxBPEL an extension of BPEL to support variability in web-based systems [12]. VxBPEL has extra XML elements to support the expression of variation points and variants in a BPEL process. An example of a VxBPEL fragment to code a variation point is shown in Fig. 18.1 (adapted from [12]). Variation points can be placed inside a BPEL process at any place where a single activity can be placed.

VxBPEL supports service replacement, different service parameters, and changing system composition. Compared to Sun et al., VxBPEL variability is only modeled in the implementation layer rather than at higher levels of abstraction. Sun et al. make full use of COVAMOF to model variability also at the architectural level to help understand the composition of a service-based system.

Fig. 18.1 VxBPEL fragment to code a variation point (adapted from [12])

```
<vxbpel:VariationPoint name="VP1">
  <vxbpel:Variants>
    <vxbpel:Variant name="default">
      <vxbpel:VPBpelCode>
        <invoke .../>
      </vxbpel:VPBpelCode>
    </vxbpel:Variant>
    <vxbpel:Variant name="alternative1">
      <vxbpel:VPBpelCode>
        <sequence ...>
          ...
        </sequence>
      </vxbpel:VPBpelCode>
    </vxbpel:Variant>
  </vxbpel:Variants>
</vxbpel:VariationPoint>
```

In the example of WMO implementation, the VxBPEL fragment above could contain a `VariationPoint name="PersonalBudget"`. This variation point expresses variability in the way the personal budget of a citizen who applies for societal support is determined. Variants for this variation point are `Variant name="Inhouse"` and `Variant name="External"`. Based on the selected variant, either a service for in-house processing of the personal budget is invoked or the request for budgeting is sent to an external provider.

5 Outlook

Handling variability in service-intensive systems enables systems which are highly adaptable and reusable in different environments. Handling variability helps construct systems that do not only reuse services but can be reused as a whole. In this chapter, we highlighted the need for and benefit of variability handling in service-based systems. Moreover, types of variability as it might occur in service-intensive systems were discussed as well as how variability could be addressed.

To further leverage variability in service-oriented systems and web services, new tools would need to be created that support the concepts discussed in this chapter. Moreover, to facilitate dynamic service variability, "self-adaptive" and "plug-and-play" architectures can be investigated. Recently, utilizing dynamic product lines in a service-oriented context has been explored [9].

References

1. Asadi, M., Mohabbati, B., Kaviani, N., Gasevic, D., Boskovic, M., Hatala, M.: Model-driven development of families of service-oriented architectures. In: First International Workshop on Feature-Oriented Software Development, Denver, CO, pp. 95–102. ACM (2009)
2. Chang, S.H., Kim, S.D.: A variability modeling method for adaptable service in service-oriented computing. In: 11th International Software Product Line Conference, Kyoto, Japan, pp. 261–268. IEEE Computer Society (2007)
3. Cohen, S., Krut, R.: Managing Variation in Services in a Software Product Line Context. CMU SEI, Pittsburgh, PA (2010)
4. Curbera, F., Duftler, M., Khalaf, R., Nagy, W., Mukhi, N., Weerawarana, S.: Unraveling the web service web: an introduction to SOAP, WSDL, and UDDI. IEEE Internet Comput. **6**(2), 86–93 (2002)
5. Erl, T.: Service-Oriented Architecture (SOA): Concepts, Technology, and Design. Prentice Hall, Upper Saddle River, NJ (2005)
6. Erl, T.: SOA Design Patterns. Prentice Hall, Upper Saddle River, NJ (2009)
7. Gannod, G.C., Burge, J.E., Urban, S.D.: Issues in the design of flexible and dynamic service-oriented systems. In: International Workshop on Systems Development in SOA Environments, Minneapolis, MN, pp. 118–123. IEEE Computer Society (2007)
8. Gu, Q., Lago, P.: Exploring service-oriented system engineering challenges: a systematic literature review. Serv. Orient. Comput. Appl. **3**(3), 171–188 (2009)
9. Hallsteinsen, S., Jiang, S., Sanders, R.: Dynamic software product lines in service-oriented computing. In: 3rd International Workshop on Dynamic Software Product Lines, San Francisco, CA, pp. 28–34 (2009)
10. Jiang, J., Ruokonen, A., Systa, T.: Pattern-based variability management in web service development. In: Third European Conference on Web Services, Vaxi, Sweden, pp. 83–94. IEEE Computer Society (2005)
11. Kim, Y., Doh, K.: Adaptable web services modeling using variability analysis. In: Third International Conference on Convergence and Hybrid Information Technology, Busan, Korea, pp. 700–705. IEEE Computer Society (2008)
12. Koning, M., Sun, C., Sinnema, M., Avgeriou, P.: VxBPEL: supporting variability for web services in BPEL. Inf. Softw. Technol. **51**(2), 258–269 (2009)
13. Lee, J., Kotonya, G.: Combining service-orientation with product line engineering. IEEE Softw. **27**(3), 35–41 (2010)
14. Lee, J., Muthig, D., Naab, M.: An approach for developing service oriented product lines. In: 12th International Software Product Line Conference, Limerick, Ireland, pp. 275–284. IEEE Computer Society (2008)
15. Medeiros, F.M., de Almeida, E.S., de Lemos Meira, S.R.: Towards an approach for service-oriented product line architectures. In: Workshop on Service-oriented Architectures and Software Product Lines, San Francisco, CA, pp. 1–7. Software Engineering Institute (2009)
16. Robak, S., Franczyk, B.: Modeling web services variability with feature diagrams. In: Revised Papers from the NODe 2002 Web and Database-Related Workshops on Web, Web-Services, and Database Systems, pp. 120–128. Springer (2003)
17. Segura, S., Benavides, D., Ruiz-Cortes, A., Trinidad, P.: A taxonomy of variability in web service flows. In: First Workshop on Service-oriented Architectures and Product Lines, Kyoto, Japan, pp. 1–5. SEI (2007)
18. Sun, C., Rossing, R., Sinnema, M., Bulanov, P., Aiello, M.: Modeling and managing the variability of web-service-based systems. J. Syst. Softw. **83**(3), 502–516 (2010)
19. ter Beek, M., Gnesi, S., Fantechi, A., Zavattaro, G.: Modelling variability, evolvability, and adaptability in service computing. In: First International Workshop on Automated Configurations and Tailoring of Applications, Antwerp, Belgium, pp. 14–19. CEUR (2010)
20. Topaloglu, Y., Capilla, R.: Modeling the variability of web services from a pattern point of view. In: European Conference on Web Services, Erfurt, Germany, pp. 128–138. Springer (2004)

Chapter 19
Service-Oriented Product Lines

Jaejoon Lee and Gerald Kotonya

What you will learn in this chapter

- *What are the views on features and services as engineering concepts*
- *How third-party service providers are involved in service-oriented product lines*
- *What to consider to design service-oriented product lines*

1 Introduction

Dynamic reconfiguration approaches in the literature have focused on specific problems of each application area (e.g., context awareness [1], autonomous software component version control, etc.), and development of reusable and dynamically reconfigurable core assets has not been fully investigated. As such, a research theme that addresses development issues for reusable and dynamically reconfigurable core assets has emerged and it is called dynamic software product lines (DSPLs) [2].

When an application domain of DSPL is built upon services and service-oriented architecture, we call it a service-oriented product line (SOPL) [3–5]. For example, an application area for SOPL approaches is that of virtual offices [3]. Virtual Offices (VO) are equipped with many business peripherals that have various services that interact with each other and respond to their environment in order to assist office workers. In this chapter, we identify some challenges that we experienced during the development of a SOPL.

First of all, we need to understand the different engineering goals between software product line engineering (SPLE) and service-orientation (SO). The main engineering goal of SPLE is the development of core assets that enable systematic

J. Lee (✉) • G. Kotonya
School of Computing and Communications, Lancaster University, Lancaster, UK
e-mail: j.lee3@lancaster.ac.uk; gerald@comp.lancs.ac.uk

R. Capilla et al. (eds.), *Systems and Software Variability Management*,
DOI 10.1007/978-3-642-36583-6_19,

reuse. In the analysis phase, SPLE attempts to identify the common and variable aspects of systems and determine those requirements common to the entire product line and those specific to product line products. Product line architecture and components are designed to provide an infrastructure, by which various products of the product line can be instantiated efficiently. For the deployment and maintenance of a product, ensuring valid and operating environment relevant product configuration is the foremost concern, as invalid configuration may result in a system crash or malfunction.

On the other hand, SO aims at achieving system agility to cope with rapidly changing business environments by providing runtime flexibility. The key idea of SO is to provide an agile and flexible way for developing systems through a dynamic runtime architecture that allows services to be added on demand. The service-oriented architecture is the conceptual structure for realizing this vision [6]. The process of identifying and defining how services are orchestrated into an application is the major focus of the analysis phase in SO. A service broker provides runtime support for service discovery and selection. As such, key development issues include design considerations and constraints for the efficient, dependable, and correct matching between service consumers and providers.

Due to these differences, we ask ourselves the following questions when we develop a SOPL:

(i) Is it about developing reusable services to increase reusability of a service-based system?
(ii) Is it about using a service-oriented architecture style to enhance runtime flexibility of a product line?
(iii) Is it about adapting feature analysis to supplement service identification techniques (e.g., ontology)?
(iv) Is it about incorporating variation points into services to control service configuration more explicitly?

In the next section we identify and discuss a number of challenges related to these questions.

2 Challenges for Building a SOPL

Among various difficulties we experienced when developing the aforementioned SOPL, we discuss four challenges in the following to answer the above questions by first contrasting the concepts from the two different literatures (i.e., SPLE and SO) and then introducing our experiences.

2.1 Features Versus Services

As these two different notions (i.e., features and services) are used as key engineering drivers for SPLE and SO, respectively, we first need to understand their definitions and how they are analyzed. This challenge is related to the questions (i) and (iii). Features are "abstractions" of user or developer visible characteristics of a product line [7]. The idea of feature orientation for analyzing commonality and variability of a product line appeals to many product line engineers as features are an effective "media" for supporting communication among diverse stakeholders of a product line [7]. Products are typically discussed and described in terms of features gathered from market surveys, individual customers, research labs, or technology roadmaps. Therefore, it is natural and intuitive for people to express commonality and variability of product lines in terms of features and a feature model is used to provide a basis for a later development, parameterization, and configuration of various reusable assets (e.g., product line requirement models, reference architectural models, and reusable code components).

On the other hand, a service in SO is described as a collection of capabilities, grouped together with the functional context of the service [6]. The service contains the logic required to carry out these capabilities and provides a service contract that describes which of these capabilities is available for invocation. The advertisement and discovery of services is a key principle of SO and an integral part of the service-oriented architecture model. In the model, service providers publish descriptions of the services they provide to a registry. These services are then advertised by the registry for service consumers to discover.

Recently, service ontologies are increasingly being used to automate the advertisement and discovery of services [8]. Ontology allows the consumer and providers to share a common set of terms for describing service qualities and constraints. Also, service descriptions include a service-level agreement (SLA). An SLA concerns the terms and conditions of service provision and use, i.e., what a consumer can expect from a provider and restrictions on what a consumer can demand from a provider.

In summary, features are used to identify commonality and variability of a product line and to configure reusable assets, whereas services are used to identify a collection of functionalities together with SLA of providers and specify ontology for automated service advertisement and discovery.

2.2 Dynamic Characteristics of Service-Based Systems

This challenge is related to the question (ii). Service-based systems are distributed and composed from numerous services that can be discovered and replaced at runtime. This dynamic characteristic of SO is closely related to terms Quality of Service (QoS) and dynamic service orchestrations.

Traditionally, QoS has been associated with telephony and computer networking. QoS may be required for certain applications, such as voice over IP (VoIP), which have requirements on the data flowing across the network (such as latency, jitter, number of dropped packets, etc.). In service-based systems, qualities are considered to be constraints over the functionality of a service and it is important that mechanisms are in place for ensuring expected system qualities at runtime [9].

In SPLE, quality issues are usually addressed statically during the design and implementation of a system. Static quality management approaches rely on predicting the properties of a system based on the properties of its constituent components [9]. By simply applying static prediction of resource usage for developing a service-based system, however, a product might not have the resources to function correctly at a certain level of system quality at runtime. To address this issue, we first statically define QoS in terms of features with a maximum limit of available resources for each product and use this information when the product starts negotiating with service providers for selection of available services at runtime.

In order to automate the advertisement, discovery, and negotiation of services, participants in a service-based system must share a common set of terms for describing service qualities and constraints [9]. A standard description method facilitates processes such as service advertisement, discovery, selection, composition, substitution, negotiation, and runtime service monitoring.

In summary, most SPLE approaches focus on configuring product line variations before deployment and do not consider dynamic service composition. One way to address the problem is to distinguish between statically configured services (i.e., static services) and dynamic services during the feature analysis phase. A static service means that its configuration is determined for each consumer before deployment, whereas a dynamic service is not included in a product configuration but must be searched and bound into the product at runtime using the service-oriented architecture.

2.3 Involvement of Third-Party Service Providers

This challenge is also related to question (ii). SPLE promotes a systematic reuse within an organization and did not usually consider external organizations for developing reusable assets. The most similar form of third-party involvement might be the use of Commercial Off-The-Shelf (COTS) components. In SO, however, this is one of the main drivers that make the approach attractive and leads to three initiatives (1) service negotiations, which refer to communication processes that further coordination and cooperation [10], (2) service monitoring, which is a further process used to detect service failures and SLA violations at runtime [11], and (3) service reputation, which is about collaborative mechanisms for addressing trust issues between such parties and help to distinguish between low and high-quality service providers [11]. To support these, we could use a QoS-aware framework [4] that provides automated runtime support for service discovery, negotiation,

monitoring, and service provider rating. The QoS awareness would allow the consumer to handle recovery from SLA violations, service failures, and runtime environment limitations by renegotiating and substituting problematic services.

These aspects of dynamic service acquisition from third-party service providers are not considered in traditional SPLE approaches and should be incorporated into SOPL methods. For instance, we could consider the marketing perspectives to decide which services should be left out for third-party service providers. That is, a service might be determined to be searched and bound into the system at runtime, but not developed as core assets, due to budget constraints, the time-to-market, and the availability of in-house expertise.

2.4 Design Considerations for SOPLs

This challenge is related to the question (iv). In SO, nonfunctional QoS properties are the main criteria for distinguishing providers in an emerging service marketplace where multiple providers supply functionally equivalent services (i.e., those that implement a common service type). Service providers therefore need a standard method for describing the nonfunctional characteristics of the services they offer. A number of ontology-based initiatives have been proposed to reduce the ambiguity of describing nonfunctional attributes and to allow for better service selection.

However, from the SPLE point of view, it is also important to be able to specify a product-specific configuration explicitly so that a consumer can have tailored product configuration. For example, if it is specified that a feature is not included in a product configuration, the associated services should not be bound into the product even though the service providers are available at runtime. To achieve this, we need to clarify the design goals for developing such systems. The goals include supports for (1) late binding of networked services, (2) third-party service providers, (3) dynamic product reconfiguration, and (4) product line variation control.

For the former two design goals, we should provide SLA enabled service-oriented architecture. The service-oriented architecture style can provide a basis for supporting the late binding (i.e., runtime) of services. Also the capability of reasoning about SLA is essential for selecting the most appropriate service provides at runtime. The latter two design goals require a "dynamic product reconfigurator" for the design of product line architecture. That is, we need a separate component that can change a product configuration at runtime while maintaining its integrity.

2.5 Design and Implementation Techniques

In addition to the dynamic changes supported by the service-oriented architecture (i.e., discovery and binding of services at runtime), we also need to design and develop product line components to support dynamic reconfiguration. While the selection of appropriate binding techniques depends both on binding time and

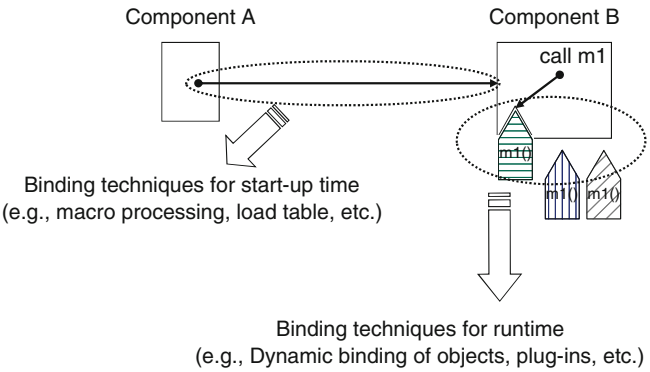


Fig. 19.1 Component binding techniques

quality attributes (e.g., flexibility) required for the SOPL, it is important that we are able to control the variations of a SOPL. Delaying binding time to a later phase of the life cycle (e.g., runtime) may provide more flexibility, but applicable implementation techniques are limited and they usually require more performance overheads. Therefore, it is important to consider such trade-offs when selecting appropriate component binding techniques.

For example, we can simply enable or disable accesses to certain services by using load tables and/or authentication-based access control techniques [12], if we can decide a product configuration of the SOPL at the start-up time. In the VO product line, for instance, the load table technique is used to determine the availability of user localization services: Access Point- or RFID-based. When the system starts to execute, it refers to the load table to determine which features should be made available to the user. If the user is a manager, she/he should be able to use the RFID-based user localization service, while a guest user can only use Access Point-based one. This means that the guest user cannot access some services (e.g., RFID-based user localization service) though they are available at runtime. In addition to those techniques, we can use dynamic binding of objects, menus, and plug-ins [13] techniques to bind components at runtime. For example, an appropriate printing proxy component is bound at runtime depending on the type of an available printing service provider at runtime. Figure 19.1 depicts this concept.

3 Outlook

Current product line approaches have focused on the development of statically configured products using core assets with static configuration of variation points. However, there's increasing demand for dynamic product reconfiguration in various application areas. An intuitive and elegant way to address the problem is to fuse service orientation with product line engineering. However, this poses several

problems. In this chapter, we have discussed the challenges that need to be addressed in order to develop effective service-oriented product lines.

To address the challenges, we should consider ways to (1) identify appropriate dynamic services, (2) adapt feature analysis to supplement service identification techniques, (3) adapt service orientation to enhance runtime flexibility of a product line, and (4) incorporate variation points into services to control product line configurations more explicitly. While the feature analysis can be used to identify and model product line variability, the resulting feature model can also be used to identify dynamic services. The system integration and deployment are different from statically configured products and should be supported by a service brokerage framework that provides runtime support for product line service discovery, negotiation, and QoS monitoring. Also we should further explore ways to decide the service granularity for enhancing reusability and to incorporate consumer context monitoring for improving quality assurance.

References

1. Yau, S.S., Karim, F., Wang, Y., Wang, B., Gupta, S.K.S.: Reconfigurable context-sensitive middleware for pervasive computing. Pervasive Computing July/September, 33–40 (2002)
2. Hallsteinsen, S., Hinchey, M., Park, S., Schmid, K.: Dynamic software product lines. IEEE Comput. **41**(4), 93–95 (2008)
3. Lee, J., Muthig, D., Naab, M.: An approach for developing service oriented product lines. In: Proceedings of the 12th International Software Product Line Conference (SPLC 2008), Limerick, Ireland, 8–12 Sept 2008, pp. 275–284
4. Kotonya, G., Lee, J., Robinson, D.: A consumer-centred approach for service-oriented product line development. In: Proceedings of WICSA2009, pp. 211–220 (2009)
5. Lee, J., Kotonya, G.: Combining service orientation with product line engineering. IEEE Software May/June, 35–41 (2010)
6. Erl, T.: Service-Oriented Architecture: Concepts, Technology, and Design. Prentice-Hall, Upper Saddle River, NJ (2005)
7. Lee, K., Kang, K., Lee, J.: Concepts and Guidelines of Feature Modeling for Product Line Software Engineering. In: LNCS, vol. 2319, pp. 62–77. Springer, Heidelberg (2002)
8. Dobson, G., Lock, R., Sommerville, I.: QoSOnt: a QoS ontology for service-centric systems. In: EUROMICRO-SEAA, pp. 80–87. IEEE Computer Society (2005)
9. Lunders, F., et al.: Using software component models and services in embedded real-time systems. In: Proceedings of 40th Annual Hawaii International Conference on System Sciences (HICSS'07) (2007)
10. Yan, J., Kowalczyk, R., Lin, J., Chhetri, M.B., Goh, S.K., Zhang, J.: Autonomous service level agreement negotiation for service composition provision. Future Generat. Comput. Syst. **23**(6), 748–759 (2007)
11. Benjamim, A.C., Sauvé, J., Cirne, W., Carelli, M.: Independently auditing service level agreements in the grid. In: Proceedings of the 11th HP OpenView University Association Workshop, HPOVUA (2004)
12. Sun Microsystems, Inc. Java authentication and authorization service (JAAS). http://www.oracle.com/technetwork/articles/javase/jaasv2-137221.html
13. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, Boston, MA (1995)

Chapter 20
Software Variability and Design Decisions

Rafael Capilla and Jan Bosch

What you will learn in this chapter
- *Understand variability models as decision models*
- *How design decisions can be entangled with variability models*

1 Introduction

In today's Software Product Line practice, variability models (often known as feature models) can be considered and understood as decision modes in which software engineers decide and select the best or more suitable design choices implemented in the form as configurable option to deliver the right products.

A variability model, as used today, describes the system options in terms of features that can be customized with valid values and selected to deploy different product configurations. Hence, software engineers need to think about the number and type of product configurations that will be offered to the customer. In other cases, late binding times and variability realization mechanism allow end users to make the final decisions, some of them at configuration and runtime, to configure the product to their own needs. Examples of such configurable design options are the language, the appearance of the interface, start-up options, different user profiles with different privileges, etc.

In this emerging research chapter we discuss two different perspectives. The first one that considers variability models as a decision model [1]. The second extends

R. Capilla (✉)
Rey Juan Carlos University, Móstoles, Madrid, Spain
e-mail: rafael.capilla@urjc.es

J. Bosch
Chalmers University of Technology, Gothenburg, Sweden
e-mail: jan@janbosch.com

R. Capilla et al. (eds.), *Systems and Software Variability Management*,
DOI 10.1007/978-3-642-36583-6_20,

variability models with design decisions that are able to explain the decision-making activity using feature models as input and in the context of SPLs.

2 Variability Model as a Decision Model

Variability models used in product lines, or more technically feature models, are in practice considered a kind of decision model where the structural variability of a set of related products is perceived as a decision model where the configurable options are the design alternatives for product configuration tasks. However, several types of decisions can be made when building and realizing the variability of a software system, such as we summarize in Table 20.1.

Using variability models as decision modes can be a complicated activity due to the large number of features and constraints. Also, scalability and traceability problems are common in complex feature models in order to realize the provided variability. As mentioned in [2] "*A decision model is defined as a model that captures variability in a product line in terms of open decisions and possible resolutions*." In addition, visualizing all the design choices of a variability model and their corresponding decisions as well is still challenging, as contemporary graphical representations are sometimes hard to manage and to display all the features at the same time. Hence, to ease the decision-making process of, for instance, feature selection will depend many times on how easy features can be offered and visualized to the user, as large feature model tree will hamper the decisions to be made, and taking into account the interdependencies between decisions and features. One way to simplify the decision-making process using variability model is to filter that part of the feature model in which we need to make decisions, i.e., introduce different approaches to add hierarchy, and show to the user only that subset of features that will be selected or configured in that part of the decision tree. For instance, feature models can be split at the subsystem level and a software engineer with limited responsibility over a subset of a system will only manage that part of the variability model, and then make decisions in the scope of a subset of the system's functionality.

In addition, variability management and rationale management share some similarities, such as stated in [3], where rationale management is combined with variability management to enhance variability modeling in software product line engineering. Such similarities can be defined between variation points, variability dependencies, constraint dependencies, and orthogonal variability models with rationale management, and use this rationale to enhance variability management activities (e.g., variability identification, product derivation, etc.).

In [4], a decision-oriented variability modeling language (DoVML) is proposed to support the modeling aspects of variability using decisions and relate these decisions with the assets of the solution space. In this approach, dependencies between decisions are described using visibility and validity conditions. Visibility conditions distinguish the relevant decisions for the user and guide him/her in the

Table 20.1 Relationship between types of design decisions and variability

Decision type	Effects on feature model	Development stage	Description
Add-modify a variant/ variation point	Variants and the logical formula relating variants and variation points	Modeling/design the structural variability	Design decisions and their rationale are captured at design time to explain feature modeling activities
Add-modify a dependency/ constraint	Requires and excludes relationships and other if-then-else relations delimiting the type and scope of the variability model	Modeling/design the structural variability	Design decisions and their rationale are captured at design time to explain feature modeling activities
Define allowed values for features	Variants, variation points including those in already defined relationships and constraints	Implementation	Design decisions can be extended to incorporate implementation or technical decisions more closely to code
Feature selection	Variants	Configuration	Decisions affecting the configuration are used to explain how and why a product will be built
Feature realization	Instantiate the variants with right and allowed values or activate/ deactivate variants. Resolve the logical formula of variation points	Configuration and deployment	Decisions affecting the configuration and deployment activities are used to explain how, why, and where products are built and deployed
Feature reconfiguration	Because of a runtime change, features can be modified during system execution	Runtime reconfiguration and redeployment	Runtime decisions, often deferred to a late stage, are more difficult to capture and manage as they might vary during system execution. Hence, a limited set of runtime decisions describing the runtime choices must be defined to manage a certain degree of unexpected variability
Binding	Binding time implementation mechanism	From design to implementation	Decisions affecting the binding time might change this to leverage the flexibility of the SPL and allow rebinding late decisions

product derivation process. The aforementioned approach has been implemented in a variability modeling tool called DecisionKing.

3 Approaches Combining Variability with Design Decisions

To date, very few approaches have considered the inclusion of design decisions in variability models as a way to explain the decisions made for product configuration operations in a product line context and in feature modeling activities as well.

3.1 Design Decisions and Feature Models

Feature models are in practice decision model in which decisions are made for (1) model the structural variability of features and their dependencies and (2) select the right configurable options during product derivation. As described in [5], design decisions are entangled as XML descriptions in a *Feature Oriented Model Driven Development* (FOMDD), which is a blend of Feature-oriented Programming (FOP) and Model-Driven Development (MDD). In this approach, products in a software product line are synthesized using MDD and scripting by composing features to create models that are transformed into executables.

A stepwise refinement process selects and composes features incrementally from a base architecture to increase the functionality of a software product, and the design decisions extend the expressiveness of the model by recording key architectural knowledge (AK) such as the decision issue, decision description, a status, and constraints among other items. Also, relationships and dependencies to other decisions or software requirements can be included as a list of elements in the XML description of the decision that uses the *Graph eXchange Language* (GXL) meta-model to depict the synthesis architecture. The synthesis architecture is represented using GXL and XAK to specify the extensions and to support the refinement of XML documents for product-line extensibility. This composition is supported by the AHEAD approach [6].

In [7], the authors present a model to manage the variability of a SPL using design rationale and capture the rationale behind the variability in order to explain the reasons of a design step. They propose a variability design rationale view to encompass the cognitive aspects and the assumptions of the variability definition and to automate the verification of the design rationale of SPLs.

From our point of view, entangling the design decisions and the rationale in code descriptions may obscure the documentation of both code and decisions. Therefore, we prefer to keep both artifacts separate and manage them as different but related artifacts. In software architecture, feature models are often represented in a separate model from UML diagrams and it is hard to include the design decisions as textual information in feature models (except if we use XML-based representations like in

[5] to explain the stepwise refinements made during product configuration and/or derivation). Hence, as practical guideline, we suggest to use a tabular form of representation to relate design decisions with features, and manage these separately in order to describe better the underpinning decisions using features.

3.2 Design Decisions in a Product Line Context

Other approaches attempt to relate architectural knowledge (i.e., design decisions and its rationale among others) in Software Product Lines. Some approaches [8] compare the meta-models of two architectural decision management tools (i.e., ADDSS and PAKME) to provide the necessary extensions for supporting product line-specific requirements and the relationships between design decisions and variability models. As a consequence, the proposed meta-model links variation points, variants and the binding time description with the description of the design decisions made in the product line. Decisions for the entire SPL and for concrete products are discriminated when a particular variability model is resolved, and to estimate the impact of SPL features in the reasoning activity.

An interesting comparison of five representative decision modeling approaches in product lines is described in [9], which provides a nice summary of how these models combine variability issues of feature models with decision modeling in the context of software product lines. We can deduct that the design decisions made using variability models are based on various elements, such as cardinality of features, constraints and dependency rules between features, compatibility of data types and artifacts (e.g., features are combined using variation points and based on a certain similarity and compatibility degree according to the functionality of a specific system unit), product derivation information (e.g., decisions made based on a common binding time), visibility of features (e.g., features that are activated or not based on the provided variability in a given moment or for a concrete product), etc.

4 Outlook

Few approaches deal with design decisions for SPL and in particular linked to feature models. The complexity of large variability models to assist software engineers for making the right decisions for product derivation, configuration, and deployment task may hamper to enrich such models with knowledge about design decisions. Also, once the structural variability is defined, the majority of the design decisions affect to product instantiation issues, except in open variability models where the feature model could be modified, even at runtime.

However, these runtime decisions affecting the selection of variants and variation points are more difficult to manage and track, and the effort to introduce runtime decision-making mechanism is bigger. Hence, we believe that this

emerging research topic in the software variability management research area seems attractive as a way to introduce the knowledge about the design decisions that can be used to explain the selection and reconfiguration of variability models.

References

1. Bosch, J.: Software architecture: the next step. In: Proceedings of the First European Workshop on Software Architecture (EWSA 2004), Springer LNCS, May 2004
2. Forster, T., Muthig, D., Pech, D.: Understanding decision models – visualization and complexity reduction of software variability. In: VaMoS 2008, pp. 111–119 (2008)
3. Kumar Thurimella, A., Bruegge, B., Creighton, O.: Identifying and exploiting the similarities between rationale management and variability management. In: SPLC 2008, pp. 99–108 (2008)
4. Dhungana, D., Grünbacher, P.: Understanding decision-oriented variability modelling. In: Workshop on Analyses of Software Product Lines (ASPL), SPLC 2008, Limerick, Ireland, pp. 233–242 (2008)
5. Trujillo, S., Azanza, M., Diaz, O., Capilla, R.: Exploring extensibility of architectural design decisions. In: Proceedings of the Second Workshop on SHAring and Reusing Architectural Knowledge Architecture, Rationale, and Design Intent, SHARK-ADI'07. ACM DL (2007)
6. Batory, D., Neal Sarvela, J., Rauschmayer, A.: Scaling step-wise refinement. IEEE Trans. Softw. Eng. **30**(6), 355–371 (2004)
7. Galvao, I., Van der Broek, P., Aksit, M.: A model for variability design rationale in SPL. In: ECSA Companion Volume 2010, I Workshop on Variability in Software Product Line Architectures, pp. 332–335. ACM DL (2010)
8. Capilla, R., Ali Babar, M.: On the role of architectural design decisions in software product line engineering. In: ECSA 2008. LNCS, vol. 5292, pp. 241–255. Springer (2008)
9. Schmid, K., Rabiser, R., Grünbacher, P.: A comparison of decision modeling approaches in product lines. In: VaMoS 2011, pp. 119–126 (2011)

Chapter 21
Variability and Aspect Orientation

Kwanwoo Lee

What you will learn in this chapter

- *The relationship between variability and aspect orientation*
- *How variability is realized using aspect orientation*

1 Introduction

Variability is an inherent property of software product lines. In software product line engineering, variability must be systematically described and managed throughout all development activities. Variability in a software product line is often analyzed and modeled in terms of features. Optional or alternative features in a feature diagram [1, 2] represent units of variations in requirements. However, realizing a feature may affect several parts of core assets (e.g., architectures or implementation components) instead of being localized.

There are two reasons for this: the first reason is a unit of features does not always correspond to that of components, i.e., the code implementing a particular feature may be scattered across multiple components. Second, features are not independent entities. If dependencies or interactions among features are hard-coded in several components implementing their related features, variations of feature dependency caused by feature variation (i.e., addition or deletion) may cause significant changes to many components. The above reasons make it difficult to realize the variability of a software product line in terms of features.

Aspect-oriented programming (AOP) [3] is a good candidate for handling the crosscutting problem, as it provides effective mechanisms for encapsulating

K. Lee (✉)
Department of Information Systems Engineering, Hansung University, Seongbuk-gu, Seoul, Republic of Korea
e-mail: kwlee@hansung.ac.kr

R. Capilla et al. (eds.), *Systems and Software Variability Management*,
DOI 10.1007/978-3-642-36583-6_21,

crosscutting concerns into separate modular units called aspects. This chapter describes how aspect orientation can help realizing variability and presents areas of practice that are relevant to the topic with discussion of benefits and possible problems.

2 Relationship Between Variability and Aspect Orientation

Variability identified in terms of features can be classified into two categories: modular features and crosscutting features, depending on the impact on their implementation.

Definition 21.1. Modular feature
A feature is modular if its implementation can be confined to a single modular component.

Definition 21.2. Crosscutting feature
A feature is crosscutting if its implementation spans multiple modular components.

Modular features can be implemented as modular components such as classes in object-oriented programming. Suppose, for example, diesel and gasoline engines are alternative features of an automobile product line. Each of them can be implemented independently from the other. On the other hand, crosscutting features have widespread impacts on multiple modular components. For example, safety policies employed by automobile products can have widespread impact on multiple control components, such as *Engine*, *Brake*, and *Airbag* components.

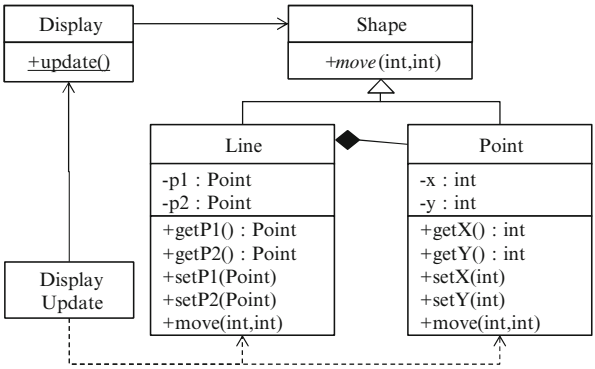
AOP supports separation of crosscutting features, whose implementation results in modification of several modular units (e.g., classes), from features that can well be encapsulated into modular units. This separation of concerns improves adaptability and configurability of product line assets, as the concerns that affect multiple modular units can be encapsulated into separate modular units, called aspects.

Definition 21.3. Aspect
An aspect is a separate modular unit encapsulating any crosscutting concern, which would otherwise be scattered across multiple components.

AOP languages, such as AspectJ [3] which is an aspect-oriented extension to Java, support the encapsulation of crosscutting features into new modular units—the aspects—through new composition mechanisms, such as pointcut advice. The *pointcut* mechanism is used to capture points where crosscutting concerns need to be inserted. A crosscutting concern to be inserted is defined through the *advice* mechanism.

Example 21.1. Aspectual implementation of a crosscutting feature
Suppose for example a simple drawing tool has a crosscutting feature, i.e., *UpdateDisplay*—the *Display* in the figure editor must be updated whenever the

state of each *Shape* instance changes. Implementing this feature in an object-oriented style leads to scattered *Display.update*() calls throughout the *set* and *move* methods of the *Line* and *Point* classes. Using AspectJ, the crosscutting concern can be effectively modularized into a single modular unit. That is, the *DisplayUpdate* aspect modularizes the scattered *Display.update*() calls using the pointcut-advice mechanism of AspectJ. The pointcut *ShapeUpdated* (lines 2 and 3) captures the call to methods of *Shape* subclasses (i.e., the *Line* and *Point* classes), where the method name starts with "set" or is "move." Whereas, the after advice (lines 4–6) inserts *Display.update*() after the join points specified at the pointcut *ShapeUpdated*.




```
1. public aspect DisplayUpdate {
2.     pointcut ShapeUpdated(Shape s):
3.         target(s) && (call(* Shape+.set*(..)) || call(* Shape+.move(..)));
4.     after(Shape s): ShapeUpdated(s) {
5.         Display.update(s);
6.     }
7. }
```

Realizing crosscutting features using AOP makes it easy to trace between features and their implementation units. If features are independent of each other, their variations (i.e., inclusion or exclusion) do not cause problems. However, if they are not, their variation may cause changes to the implementation or side effects in the behavior of other features.

Definition 21.4. Operational feature dependency

Operational feature dependencies are implicitly or explicitly created relationships between features in such a way that the behavior or implementation of one feature affects that of other features.

Operational feature dependencies [4] have significant implications on variability. If the code for dependencies between features is embedded into feature implementation modules, a feature variation will affect the modules implementing

other features. This problem, known as optional feature problem [5], mainly comes from a lack of understanding of operational feature dependencies and scattering dependency-related code across feature implementation modules.

Example 21.2. Operational feature dependency between *ShapeColor* and *UpdateDisplay*

Suppose for example the optional feature *ShapeColor* in the figure editor example extends the *Shape* class with a *color* attribute and corresponding getter and setter methods. However, introducing this feature affects the existing *DisplayUpdate* aspect that implements the *UpdateDisplay* feature to reflect the proper update of a display when a Shape's color changes. Line 9 indicates the code realizing the dependency between *ShapeColor* and *UpdateDisplay*. This implies the *DisplayUpdate* aspect has to be changed according to the selection of the *ShapeColor* feature.

```
1. public aspect ShapeColor {
2.       private Color Shape.color;
3.       public Color Shape.getColor() {return color;}
4.       public void Shape.changeColor(Color c) {color=c;}
5. }
6. public aspect DisplayUpdate {
7.       pointcut ShapeUpdated():
8.           target(s) && (call(* Shape+.set*(..)) || call(* Shape+.move(..)) ||
9.           call(* Shape+.changeColor(..)));
10.    after(Shape s): ShapeUpdated(s) {
11.       Display.update(s);
12.    }
13. }
```

One effective way of handling the variability issue related to operational feature dependency is separating dependency aspects from the implementation of features. AOP can help isolating dependency aspects between feature implementation modules as it provides effective mechanisms for extending a noninvasive way of crosscutting issues.

Example 21.3. Aspectual separation of the operational feature dependency between *ShapeColor* and *UpdateDisplay*

The code snippet below shows how the operational dependency between *ShapeColor* and *UpdateDisplay* can be separated from the *DisplayUpdate* aspect. The *DisplayUpdate* aspect (lines 1–6) defines only the core functionality (line 4) of the *UpdateDisplay* feature, which will be inserted at the join-points specified by the abstract pointcut *ShapeUpdated* (line 2). The *DependencyWithShapeColor* aspect (lines 7–11) defines the operational dependency between *ShapeColor* and *UpdateDisplay* by overriding the abstract pointcut. The *NoDependencyWith-ShapeColor* aspect implements the default dependency between *DispayUpdate*

and *Shape* instances. During the application engineering phase, one of the aspects can be configured according to the selection of the *ShapeColor* feature.

```
1. public abstract aspect DisplayUpdate {
2.       protected abstract pointcut ShapeUpdated(Shape s);
3.       after(Shape s): ShapeUpdated(s) {
4.          Display.update(s);
5.       }
6. }

7. public aspect DependencyWithShapeColor extends DisplayUpdate {
8.       protected pointcut ShapeUpdated(Shape s):
9.          target(s) && (call(* Shape+.set*(..)) || call(* Shape+.move(..)) ||
10.         call(* Shape+.changeColor(..)));
11. }

12. public aspect NoDependencyWithShapeColor extends DisplayUpdate {
13.       protected pointcut ShapeUpdated():
14.          target(s) && (call(* Shape+.set*(..)) || call(* Shape+.move(..)));
15. }
```

With the understanding of operational feature dependencies and AOP mechanisms, variability of a product line can be effectively handled.

3 Recommended Areas of Practice

This section describes two practice areas applying AOP to improve feature modularity and independence.

3.1 Modularization of Crosscutting Features

As described earlier, AOP by nature provides powerful mechanisms for encapsulating crosscutting concerns. With the help of AOP mechanisms, crosscutting features can be effectively modularized into aspectual components.

There have been several attempts to apply AOP in the development of industrial or non-trivial problems. Alves et al. [6] apply AOP in the development of mobile game product lines. They evaluate their approach in the context of an industrial-strength mobile game product line. Kästner et al. [7] refactor the embedded database system Berkeley DB into a software product line and evaluate the suitability of AspectJ for modularizing feature implementations. They report several limitations on the modularization of features when using the AspectJ language,

such as the increasing of coupling between aspects and classes due to the strong dependency of aspect pointcuts and implementation details of the base code. Zhang and Jacobsen conducted aspect-oriented refactoring of CORBA implementations [8]. Their results indicate that they were able to significantly reduce the complexity of the CORBA architecture with negligible performance overhead.

Quality attributes are the crosscutting concerns that have application-wide impact across modular components. Since separating and encapsulating them can help program understanding and improve adaptability, several efforts have been made to modularize quality attributes using AOP. Viega et al. [9] built an aspect-oriented extension to the C programming language to separate security policies from C programs. This approach allows developers to write the core functionality of the application, while a security expert focuses on specifying security properties. Szentiványi and Nadjm-Tehrani [10] proposed an approach to improve performance of fault-tolerant services using AspectJ. In this approach, an application-specific synchronization mechanism is defined as the separate aspects, which are alternatives of a general synchronization mechanism directly supported by the middleware. By using application-specific synchronization aspects, around 40 % of original overhead caused by a general synchronization mechanism could be reduced.

3.2 Separation of Feature Dependencies

Modularizing crosscutting features does not guarantee that feature implementation modules are independent. The optional feature problem may occur when optional features are not independent.

Kästner et al. [5] elaborated the impact of the optional feature problem in two case studies (i.e., Berkely DB and FAME-DBMS) and surveyed different solutions to the problem and their trade-offs. One effective way of handling the problem is to remove code implementing feature dependencies from the modular implementations of related features. The idea is to extract the code responsible for the dependency into a separate module, called derivative module [11]. The derivative module is included in the generation process to restore the original behavior if and only if both original implementation modules are selected.

Lee et al. [12] also addressed the problem by separating feature dependencies from feature implementations using AOP techniques. Specifically, they proposed aspect-oriented implementation patterns for feature dependencies, which are repeatable well-known patterns for the implementation of feature dependencies.

The optional feature problem is closely related to research in the field of feature interactions [13]. Feature interactions can cause unexpected behavior when two optional features are combined. For example, in a home integration system (HIS) product line, the Fire Control feature opens the water main and turns sprinklers on when a fire is detected. If the Flood Control feature, which shuts off the water main to a home in the event of a flood, is added to the HIS with the Fire Control feature,

the Flood Control and Fire Control features may cause an undesirable side effect (e.g., the Flood Control feature disturbs the Fire Control feature by shutting off the water main before the fire is under control). Handling feature interactions may cause significant changes to product line assets if interaction-related code is scattered across many implementation modules. Therefore, interaction related code should be separated from feature implementation for flexible feature composition.

4 Outlook

Variability may have crosscutting concerns. This chapter explained the relationships between variability and crosscutting concerns, and described how variability having crosscutting concerns can be effectively modularized using aspect-orientation mechanisms. Overall, aspect orientation becomes a valuable instrument in modularizing crosscutting variability.

However, AOP has several limitations, some of which include pointcut fragility and code readability. As pointed out in [7], the pointcut language of AspectJ has language limitations, such as the statement extension problem and pointcut fragility, which constrain the identification and definition of interaction points between class modules and aspect modules. These limitations can be alleviated by making the interaction points explicit in an abstract way. For example, crosscutting programming interfaces [14] can be used to clarify the separation of base and extensions.

In addition, aspects may be too small modular units. A fine-grained fragmentation of product line assets increases the complexity of managing variability. Nevertheless, substantial research addressing the above-mentioned limitations still is necessary before this relatively new paradigm can be applied on a broad scale in various industrial domains.

References

1. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (FODA) feasibility study. Technical report, SEI, Carnegie Mellon University, Pittsburgh, PA (1990)
2. Czarnecki, K., Helsen, S., Eisenecker, U.: Staged configuration using feature models. In: Proc. SPLC 2004, pp. 266–283 (2004)
3. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C. Lopes, C., Loingtier, J.-M., Irwin, J.: Aspect-oriented programming. In: Proc. ECOOP 1997, pp. 220–242 (1997)
4. Lee, K., Kang, K.C.: Feature dependency analysis for product line component design. In: Proc. ICSR 2004, Madrid, Spain, pp. 69–85. Springer (2004)
5. Kästner, C., Apel, S., ur Rahman, S., Rosenmüller, M., Batory, D., Saake, G.: On the impact of the optional feature problem: analysis and case studies. In: Proc. SPLC 2009, pp. 181–190 (2009)

6. Alves, V., Matos, Jr., P., Cole, L., Borba, P., Ramalho, G.: Extracting and evolving mobile games product lines. In: Proc. SPLC 2005, pp. 70–81 (2005)
7. Kästner, C., Apel, S., Batory, D.: A case study implementing features using AspectJ. In: Proc. SPLC 2007, pp. 223–232 (2007)
8. Zhang, C., Jacobsen, H.-A.: Refactoring middleware with aspects. IEEE Trans. Parallel Distr. Syst. **14**(11), 1–16 (2003)
9. Viega, J., Bloch, J.T., Chandra, P.: Applying aspect-oriented programming to security. Cutter IT Journal **14**(2):31–39, February 2001
10. Szentiványi, D., Nadjm-Tehrani, S.: Aspects for improvement of performance in fault-tolerant software. In: Proc. PRDC 2004, pp. 283–291 (2004)
11. Liu, J., Batory, D., Lengauer, C.: Feature-oriented refactoring of legacy applications. In: Proc. ICSE 2006, pp. 112–121 (2006)
12. Lee, K., Botterweck, G., Thiel, S.: Aspectual separation of feature dependencies for flexible feature composition. In: Proc. COMPSAC 2009, pp. 45–52 (2009)
13. Calder, M., Kolberg, M., Magill, E.H., Reiff-Marganiec, S.: Feature interaction – a critical review and considered forecast. Comput. Netw. **41**(1), 115–141 (2003)
14. William, G., Shonie, M., Sullivan, K., Song, Y., Tewari, N., Cai, Y., Rajan, H.: Modular software design with crosscutting interfaces. IEEE Softw. **23**(1), 51–60 (2006)

Biography of the Authors

Paris Avgeriou
Department of Computer Science, University of Groningen, The Netherlands
Paris Avgeriou is a professor of software engineering at the University of Groningen, the Netherlands. His research interests concern software architecture with a strong emphasis on architecture modeling, knowledge, evolution, and patterns. Avgeriou received his Ph.D. in software engineering from the National Technical University of Athens, Greece.
paris@cs.rug.nl

David Benavides
University of Seville, Seville, Spain
David Benavides is an Associate Professor at the University of Seville. He is an active member of the software product line community where he has been program cochair of VaMoS'09 and SPLC'12. He, together with his group, is well recognized in the automated analysis of feature models field. They develop and maintain FaMa, a framework for automated analysis of variability models.
benavides@us.es

Danilo Beuche
pure-systems GmbH, Magdeburg, Germany
Danilo Beuche works for pure-systems GmbH, a specialist provider of tools and services for the application of variant management and product line technology. As well as managing the company, Danilo also consults extensively on product line engineering, mainly for clients in embedded industries. He received a Ph.D. on embedded software families from the University of Magdeburg.
danilo.beuche@pure-systems.com

Jan Bosch
Chalmers University of Technology, Gothenburg, Sweden
Jan Bosch is professor of software engineering and codirector of the software research center at Chalmers University Technology in Gothenburg, Sweden. Earlier, he worked as Vice President Engineering Process at Intuit Inc where he

R. Capilla et al. (eds.), *Systems and Software Variability Management*,
DOI 10.1007/978-3-642-36583-6,

also led Intuit's Open Innovation efforts and headed the central mobile technologies team. Before Intuit, he was head of the Software and Application Technologies Laboratory at Nokia Research Center, Finland. Before joining Nokia, he headed the software engineering research group at the University of Groningen, The Netherlands, where he holds a professorship in software engineering. He received a M.Sc. degree from the University of Twente, The Netherlands, and a Ph.D. degree from Lund University, Sweden. His research activities include open innovation, innovation experiment systems, compositional software engineering, software ecosystems, software architecture, software product families, and software variability management. He is the author of a book "Design and Use of Software Architectures: Adopting and Evolving a Product Line Approach" published by Pearson Education (Addison-Wesley & ACM Press), (co-)editor of several books and volumes in, among others, the Springer LNCS series, and (co-)author of a significant number of research articles. He is editor for Science of Computer Programming, has been guest editor for journal issues, chaired several conferences as general and program chair, served on many program committees, and organized numerous workshops.

In the startup space, Jan serves on the advisory board of Assia Inc. in Redwood City, CA, as well as the advisory board of Burt in Gothenburg, Sweden. He is chairman of the board of Evisto in Gothenburg, Sweden. Also, he acts as an external business advisor for the School of Entrepreneurship at Chalmers University of Technology, Gothenburg, Sweden. As a consultant, as a professor, and as an employee, Jan has worked with and for many companies on innovation and R&D efficiency including Philips, Thales Naval Netherlands, Robert Bosch GmbH, Siemens, Nokia, Ericsson, Grundfos, Tellabs, Avaya, Tieto Enator, and Det Norska Veritas. More information about his background can be found at his website: http://www.janbosch.com. When not working, Jan divides his time between his family, a spouse and three young boys, reading science fiction, and sports, preferably long-distance running, swimming, biking, and horseback riding.
jan@janbosch.com

Goetz Botterweck
Lero-The Irish Software Engineering Research Centre, University of Limerick, Limerick, Ireland
Goetz Botterweck is a Senior Research Fellow at Lero-The Irish Software Engineering Research Centre, University of Limerick, Ireland. He leads Lero's research on Software Product Lines and Model-driven Software Evolution. His research interests include model-driven software engineering, software product lines, software evolution, variability in embedded systems, software visualization, and user interface engineering.
goetz.botterweck@lero.ie

Rafael Capilla
Rey Juan Carlos University, Móstoles, Madrid, Spain
Dr. Rafael Capilla received a Ph.D. in Computer Science from the Rey Juan Carlos University of Madrid (Spain) in 2004. He worked for a telecommunications company from 1989 to 2001, and after that, he worked more than 9 years as System

and Network administrator in the Faculty of Informatics at the University of Seville (Spain). In 1999 he moved to Madrid as researcher in the European Esprit project FLEX, and in 2000 he joined the Polytechnic University of Madrid as assistant professor in the Technical School of Informatics and Technical School of Telecommunications Engineering.

Today, he is associate professor in the Rey Juan Carlos University of Madrid since 2001. His first research area was software reuse, more specifically domain analysis and domain engineering methods, but currently his research interests and activities include software architecture, architectural knowledge, product line engineering, software variability, and more recently dynamic software products lines and technical debt. He is coauthor of a book on Web programming and more than 50 research conference and journal articles. He has chaired several workshops and served as General Chair of the 14th European Conference on Software Maintenance and Reengineering (CSMR) in 2010, is guest coeditor for journal issues, served on many program committees, and is reviewer in prestigious international journals. At present, he leads the Software Architecture & Internet Technologies (SAIT) research group balancing research and software development for companies, in particular mobile software applications. During his free time, Rafa likes traveling to Seville and any interesting place in the world, practice ski, paddle, ride motorbikes, and watch basketball and F1.
rafael.capilla@urjc.es

Ciaran Cawley
Dublin Institute of Technology, Dublin, Ireland
Ciaran Cawley is a lecturer at the School of Computing in the Dublin Institute of Technology with a focus on Software Engineering. Before taking up a role in academia, he worked extensively as a Software Engineer in industry over a 12-year period. His research interests are in the area of Software Engineering, particularly Visualization.
ciaran.cawley@dit.ie

Carlos Cetina
Universidad de San Jorge, Villanueva de Gállego, Zaragoza, Spain
Carlos Cetina. Ph.D. in Computer Science from the Technical University of Valencia. His research interests include Model-Driven Engineering and Software Product Lines, Pervasive Systems and Autonomic Computing, as well as technologies for smart environments such as OSGi or KNX. Some of his recent work has been published in both IEEE Computer and IEEE Pervasive Computing journals and the Software Product Line Conference.
ccetina@usj.es

Paul Clements
BigLever Software, Austin, TX, USA
Dr. Paul Clements is the V.P. of Customer Success at BigLever Software, where he works to spread the adoption of systems and software product line engineering. He was previously at Carnegie Mellon's Software Engineering Institute, where he

worked in software product line (SPL) engineering and software architecture documentation and analysis. Clements is coauthor of three software architecture books as well as the field's leading SPL text.
pclements@biglever.com

Sybren Deelstra
Océ Technologies B.V., Venlo, The Netherlands
Sybren Deelstra is R&D project manager for a production printing product line. He received his Ph.D. in Computer Science from the University of Groningen, The Netherlands.
sybren.deelstra@oce.com

Rick Flores
General Motors, Detroit, MI, USA
Rick Flores is a Staff Engineer at General Motors, working in the Model/System/Software space in GM's Software Architecture group, and is a Strategy Lead in the definition of GM Electrical's next generation process and tools. Throughout his career, he has been a key developer and advocate of General Motors' model-driven software development process, used to develop core components in GM's latest electrical architectures deployed on millions of vehicles annually.
rick.r.flores@gm.com

Makoto Fujii
Fuchu Complex, TOSHIBA Corporation, Fuchu-shi, Tokyo, Japan
Makoto Fujii works with TOSHIBA Corporation as a system and software engineer for Electric Power Generation Plant Monitoring and Control (EPG M&C) Systems for more than 30 years. He performed important roles for TOSHIBA EPG M&C Systems to get the honor of Product Line Hall of Fame.
makoto2.fujii@toshiba.co.jp

Matthias Galster
Department of Computer Science and Software Engineering, University of Canterbury, Christchurch, New Zealand
Matthias Galster is a Lecturer at the University of Canterbury in Christchurch, New Zealand. Previously, he worked as a senior researcher in the Software Engineering and Architecture group at the University of Groningen, The Netherlands. His research interests include requirements engineering and software architecture.
matthias.galster@canterbury.ac.nz

Sven Hallsteinsen
SINTEF ICT, Trondheim, Norway
Svein Hallsteinsen is a senior research scientist at SINTEF ICT, Trondheim, Norway, where he leads a research group focusing on software architecture. He received an M.S. in applied physics and mathematics from the Norwegian University of Science and Technology.
svein.hallsteinsen@sintef.no

Mike Hinchey
Lero-The Irish Software Engineering Research Centre, University of Limerick, Limerick, Ireland
Mike Hinchey is Director of Lero-the Irish Software Engineering Research Centre, and Professor of Software Engineering at University of Limerick, Ireland. Prior to joining Lero, Hinchey was Director of the NASA Software Engineering Laboratory and has previously been either Full or Visiting Professor in the USA, UK, Sweden, Australia, and Japan. Hinchey received a B.Sc. in Computer Systems from University of Limerick, an M.Sc. in Computation (Mathematics) from University of Oxford, UK, and a Ph.D. in Computer Science from University of Cambridge, UK. He is a Chartered Engineer, Chartered Mathematician, and a Chartered Professional Engineer. He is Vice President of IFIP and Chair of the IFIP Technical Assembly.
mike.hinchey@lero.ie

Kyo Chul Kang
Pohang University of Science and Technology (POSTECH), Pohang, Republic of Korea
Dr. Kyo Chul Kang received his Ph.D. from the University of Michigan in 1982. Since then he has worked as a visiting professor at the University of Michigan and as a member of technical staff at Bell Communications Research and AT&T Bell Laboratories before joining the Software Engineering Institute, Carnegie Mellon University, as a senior member in 1987. He is currently a professor at the Pohang University of Science and Technology (POSTECH) in Korea. He served as Director of the Software Engineering Center at Korea Information Technology Promotion Agency (KIPA) from 2001 to 2003. Also, he served as General Chair for the 8th International Conference on Software Reuse (ICSR) held in Madrid, Spain, in 2004, and also as General Chair for the 11th and 14th International Software Product Line Conference held, respectively, in Kyoto, Japan, in September 2007 and in Jeju, Korea, in September 2010.

While at the University of Michigan, he was involved in the development of PSL/PSA, a requirement engineering tool system, and a Meta modeling technique. Since then his research has focused on software reuse. The Feature-Oriented Domain Analysis (FODA) method he participated in the development at SEI has been widely accepted in the product line engineering community. While on leave to KIPA, he promoted the use of CMM in Korea. His current research areas include software reuse and product line engineering, requirements engineering, and computer-aided software engineering.
kck@postech.ac.kr

Gerald Kotonya
School of Computing and Communications, Lancaster University, Lancaster, UK
Gerald Kotonya is a senior lecturer in the School of Computing and Communications at Lancaster University. His research interests include software architecture, and service-oriented and component-based software engineering, especially novel ways of architecting, visualizing, and evolving self-managing,

hybrid, service-oriented systems. Kotonya has a Ph.D. in computer science from Lancaster University.
gerald@comp.lancs.ac.uk

Charles Krueger
BigLever Software, Austin, TX, USA
Dr. Charles Krueger, CEO of BigLever Software, is a thought leader in the product line engineering (PLE) field with 25 years of software engineering experience and 60 articles, columns, book chapters, and conference presentations. Krueger has proven expertise leading PLE teams and helping establish PLE practices for General Motors, General Dynamics, Lockheed Martin, Ikerlan/Alstom, and three Software Product Line Hall of Fame inductees.
ckrueger@biglever.com

Hyesun Lee
Pohang University of Science and Technology (POSTECH), Pohang, Republic of Korea
Hyesun Lee is a Ph.D. student in Computer Science and Engineering at Pohang University of Science and Technology (POSTECH) and a member of Software Engineering Laboratory at POSTECH. She received a B.S. in Computer Science and Engineering from POSTECH in 2009. Her research interests are in software product line engineering, focusing on change impact analysis of product line assets.
compial@postech.ac.kr

Jaejoon Lee
School of Computing and Communications, Lancaster University, Lancaster, UK
Jaejoon Lee is a lecturer in the School of Computing and Communications at Lancaster University. His research interests include software product line engineering, software architecture, and service-based software engineering. Lee has a Ph.D. in computer science and engineering from Pohang University of Science and Technology (POSTECH).
j.lee3@lancaster.ac.uk

Kwanwoo Lee
Department of Information Systems Engineering, Hansung University, Seongbuk-gu, Seoul, Republic of Korea
He received the B.S., M.S., and Ph.D. degrees in Computer Science and Engineering from Pohang University of Science and Technology in 1994, 1996, and 2003, respectively. He is an associate professor at the Hansung University. His research interests include product line architecture design, aspect-oriented development, real-time embedded systems, etc.
kwlee@hansung.ac.kr

Frank van der Linden
Philips Healthcare, Eindhoven, The Netherlands
Frank van der Linden did his Ph.D. in mathematics in 1984 at the University of Amsterdam. Thereafter he went to Philips research and was involved in several subjects in software engineering. Since 1991 he is involved in software product

lines within Philips. In 1999 he went to Philips Healthcare. There he was responsible for several funded partnership projects on product line engineering and related fields.
frank.van.der.linden@philips.com

Cristina López
Fundación TECNALIA Research & Innovation, Derio, Bizkaia, Spain
Cristina López obtained the Degree in Computer Science in 2006 at the University of Deusto and in 2008 she joined Tecnalia. Since then, she has been working in modeling, following the Model-driven Engineering (MDE) approach. These activities have been part of projects such as Mosis (Itea2) and the implementation of an internal tool for the management of Software Product Lines (SPLs) known as PLUM (http://www.tecnalia.com/plum). Additionally, she has also participated in the projects Modelplex (FP6) and CESAR (Artemis), with the goal of defining new modeling methodologies and enforce the integration and interoperability of tools.
cristina.lopez@tecnalia.com

Jason Mansell
Fundación TECNALIA Research & Innovation, Derio, Bizkaia, Spain
Jason Mansell leads the Software Productivity team in the IT Competitiveness unit within TENCALIA. He has obtained a Master in Business and Administration in October 2011 and his Ph.D. degree in Computer science from Deusto University (Spain) on the amplification of system family engineering for CMMI, in September 2006. He has been working since 1998 in research and development projects related to software product families, software process improvement, component-based platforms, and Model-Driven Development, such as ESAPS/CAF/FAMILIES/MoSiS (ITEA), MASTER/MODELWARE/MODELPLEX (FP6_FP7), and iFEST/CESAR/VARIES (ARTEMIS), among others.
jason.mansell@tecnalia.com

Yoshihiro Matsumoto
The ASTEM Research Institute of Kyoto, Shimogyo-ku, Kyoto City, Kyoto, Japan
Yoshihiro Matsumoto is an honorary advisor at the Advanced Software Technology and Mechatronics Research Institute of Kyoto (ASTEM-RI). He received his Dr. Eng. Degree from the University of Tokyo. He was a principal engineer at Toshiba Corporation from 1954 to 1989, where he started the Toshiba Software Factory. He was a professor at Kyoto University, Osaka Institute of Technology, and Tokyo City University from 1989 to 2002. He is an IEEE Life Fellow.
yhm@mvg.biglobe.ne.jp

Masami Okamoto
Fuchu Complex, TOSHIBA Corporation, Fuchu-shi, Tokyo, Japan
Masami Okamoto works with TOSHIBA Corporation as a system and software engineer for Electric Power Generation Plant Monitoring and Control (EPG M&C) Systems for more than 15 years. He designed the architecture of EPG software product line and integrated the software development environment for EPG software.
masami.okamoto@toshiba.co.jp

Sooyong Park
Sogang University, Mapo-gu, Seoul, South Republic of Korea
Sooyong Park is a Professor of Computer Science Department at Sogang University and President of National IT Promotion Agency. He is actively involved in academic activities including President of Korean Software Engineering Society, Steering Committee Member of Asian-Pacific Software Engineering Conference, and Guest-Editor of several journals including CACM, IEEE Computer, and IST. His research interests include Requirements engineering, Self-Adaptive software for Unmanned Systems, and Dynamic Software product line engineering.
sypark@sogang.ac.kr

Vicente Pelechano
Universidad Politécnica de Valencia, Valencia, Spain
Vicente Pelechano is Associate Professor at the Research Center on Software Production Methods (PROS) in the Universidad Politécnica de Valencia (UPV) and leads the technical supervision of MOSKitt, an open source CASE tool (http://www.moskitt.org) that gives full support to the Model-Driven Software Development. His research interests include Model-Driven Software Development, Service Engineering, Mobile and Ubiquitous Computing, Software Product Lines, Autonomous Computing, Human–Computer Interaction, and Business Process Modeling. Pelechano is Ph.D. from the UPV since 2001, he has published articles in well-known Scientific Journals and International Conferences. He has been member of Scientific and Organization Committees of prestigious International Conferences and Workshops.
pele@pros.upv.es

Antonio Ruiz-Cortés
University of Seville, Seville, Spain
Antonio Ruiz-Cortés is Associate Professor and Head of the Applied Software Engineering Group (ISA, http://www.isa.us.es) at University of Seville, Spain. He obtained his Ph.D. (with honors) in Computer Science from this University. His current research lines include service oriented computing, software product lines, and business process management. Previously, he worked on requirements engineering and multiparty interaction.
aruiz@us.es

Klaus Schmid
University of Hildesheim, Hildesheim, Germany
Klaus Schmid is a professor of software engineering at University of Hildesheim. His main areas of research are product line engineering, adaptive systems, and requirements engineering. Prior to University of Hildesheim, he was department head at Fraunhofer IESE. He authored more than 60 peer-reviewed papers and served as program chair and general chair for various major international conferences. He is member of the IEEE, ACM, and the GI.
schmid@sse.uni-hildesheim.de

Sergio Segura
University of Seville, Seville, Spain
Sergio Segura holds a Ph.D. Degree in Software Engineering from the University of Seville where he works as a full-time lecturer since 2006. He is a member of the Applied Software Engineering research group, in which he explores a variety of topics including software product lines, variability modeling, and testing. Dr. Sergio Segura also serves regularly as a (co)-referee for journals and conferences.
sergiosegura@us.es

Marco Sinnema
Q-Free ASA, Beilen, The Netherlands
Marco received his Ph.D. at the University of Groningen, where he investigated and validated strategies to effectively deal with the complexity behind variability management. After finishing his Ph.D. and having various R&D roles in industry, Marco is since 2009 working at Q-Free ASA in the role of product manager for its Video and ALPR portfolio.
marco.sinnema@q-free.com

Steffen Thiel
Furtwangen University of Applied Sciences, Furtwangen, Germany
Steffen Thiel is a Professor of Software Engineering and Associate Dean in the Department of Computer Science at Furtwangen University of Applied Sciences, Germany. His research interests are in the area of Software Engineering and include Software Product Lines, Software and Systems Architecture, Software Visualization, and Software Evolution.
steffen.thiel@hs-furtwangen.de

Pablo Trinidad
University of Seville, Seville, Spain
Pablo Trinidad is a senior lecturer at the University of Seville, where he received his Ph.D. and M.Sc. in computer science (with honors). He worked for several companies and worked as a freelance engineer before joining the academia in 2004. Dr. Trinidad is a member of the Applied Software Engineering research group working on software product lines with a special attention to the automated analysis of variability models and the automated deployment and maintenance of dynamic products.
ptrinidad@us.es

Index

R. Capilla et al. (eds.), *Systems and Software Variability Management*,
DOI 10.1007/978-3-642-36583-6, © Springer-Verlag Berlin Heidelberg 2013