informatik
Schnittstelle Zukunft

INF

A Model-driven Approach to Flexible and Adaptable Software Variability Management

PhD Thesis

Submitted in fulfillment of the requirements for the degree
Doctor of Technical Sciences (Dr. tech.)

By
Dipl.-Ing. Deepak Dhungana

Completed at
Christian Doppler Laboratory for Automated Software Engineering
Institute for Systems Engineering and Automation

Supervised by
a. Univ. Prof. Dr. Paul Grünbacher
o. Univ. Prof. Dr. Hanspeter Mössenböck

Linz, January, 2009

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Dissertation selbständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Deepak Dhungana   Linz, January 2009

Abstract

Traditional ways of building software, i.e., considering software projects to be independent of each other, are no longer sufficient to meet the challenges faced by the software industry. It is necessary to relate different products of an organization in such a way that their commonalities (marketing, technical or end-user considerations) can be shared between various products and projects. Both researchers and practitioners agree that reuse is the key to competitive advantage and variability is the key to successful reuse. In this thesis, we elaborate on the benefits of having variable software systems, and propose a model-based approach to capture the knowledge about flexibility and adaptability of systems. The novelty of our research lies in the combination of approaches for modeling stakeholder needs (requirements), product characteristics (features), architectural elements and other resources in the light of variability. Our research was guided by an analysis of the variability of our industry partner's software for continuous casting. We have tested and applied the approach in several other case studies in completely different domains. In this thesis, we describe three of the case studies to demonstrate the flexibility of the approach and extensibility/adaptability of the tools.

We chose a decision-oriented approach to variability modeling, which allows developers to systematically describe the variability of arbitrary domain specific assets and their dependencies (e.g, a component requires another component's functionally). Instead of solving the variability modeling problem for one particular domain, we developed a meta-approach, which can be configured to the specifics of different domains as required. This thesis describes the flexibility and adaptability of the approach using examples from several domains. We also provide formal semantics of the approach.

Our variability modeling tool $\mathcal{DecisionKing}$ provides a high-level view of a product line's reference architecture and also maintains the complex relationships between user needs, features, and components. We have developed a domain-specific language for capturing dependencies in product line models and integrated the off-the-shelf rule engine JBoss Drools into our tool. Our approach goes beyond available approaches for modeling features and product line architectures as it supports the modeling of arbitrary PL artifacts in an integrated manner, i.e., with support for traceability among assets, among decisions, as well as between assets and decisions. $\mathcal{DecisionKing}$ is flexible and adaptable to deal with specifics of different domains and can be easily extended with new functionality and integrated in foreign tool environments too. We have also implemented tools and techniques to support organization and structuring of the modeling space. Supporting modularization was a critical success factor for our approach. The approach is based on a simple assumption: a small model is easier to maintain than a large one. Instead of creating a single large product line variability model we use model fragments to describe the variability of selected parts of the system.

Zusammenfassung

Softwareprodukte werden typischerweise individuell betrachtet und unabhängig voneinander entwickelt. Gemeinsamkeiten zwischen ähnlichen Systemen können so nicht ausgenützt werden. Um den aktuellen Herausforderungen der heutigen Softwareindustrie (kurze Entwicklungszeiten, hohe Qualität, geringe Entwicklungskosten) besser entsprechen zu können, müssen Gemeinsamkeiten zwischen ähnlichen Softwareprodukten (z.B. im Hinblick auf Marketing, Verwendung von Komponenten, oder Produktmerkmale für Endnutzer (Features)) besser genutzt werden. Die Wiederverwendbarkeit von Software-Artefakten und deren Variabilität spielt dabei eine große Rolle. Diese Dissertation untersucht im Detail verschiedene Aspekte der Variabilität von Softwaresystemen. Wir beschreiben, wie mittels modellbasierter Ansätze Wissen über Variabilität explizit dokumentiert werden kann. Wir präsentieren einen integrierten Ansatz, der zur Modellierung verschiedener Artefakte wie etwa Anforderungen, Produktmerkmale oder Architekturelemente geeignet ist. Die Arbeit wurde auf Basis einer Analyse aktueller Herausforderungen der Industrie durchgeführt, im konkreten untersuchten wir Problemstellungen unseres Industrie-Partners Siemens VAI bei der Entwicklung von Automationssoftware. Wir haben den entwickelten Ansatz in mehreren Fallstudien in unterschiedlichen Bereichen angewandt, um dessen Flexibilität und Erweiterbarkeit zu demonstrieren.

Wir präsentieren einen entscheidungsorientierten Ansatz zur Modellierung von Variabilität. Entwickler können systematisch die Variabilität beliebiger Artefakte einer Domäne und deren Abhängigkeiten beschreiben. Anstelle einer konkreten Sprache für einen bestimmten Anwendungsfall, haben wir eine generische Lösung angestrebt. Der Ansatz beruht auf Metamodellierung und Metawerkzeugen um den Besonderheiten verschiedener Domänen gerecht zu werden. Die Arbeit beschreibt die Flexibilität und Anpassungsfähigkeit des Ansatzes anhand von Beispielen aus verschiedenen Bereichen und definiert die formale Semantik des Konzepts.

Im Rahmen der Dissertation wurde das Modellierungswerkzeug DecisionKing entwickelt. DecisionKing ermöglicht die Modellierung komplexer Beziehungen zwischen Entscheidungen des Nutzers und bildet so die Basis für die nutzergesteuerte Softwarekonfiguration. Um die komplexen Abhängigkeiten einfach beschreiben und automatisch auflösen zu können, haben wir eine domänenspezifische Sprache entwickelt, die auf JBoss Drools aufsetzt. DecisionKing ist flexibel und anpassungsfähig, um Besonderheiten der verschiedener Domänen zu unterstützen und kann durch Plugin-ins leicht erweitert werden. Die Werkzeuge unterstützen auch die Organisation und Strukturierung großer Modelle. Modularisierung ist ein entscheidender Erfolgsfaktor, da kleine Modelle leichter zu pflegen sind als große. In unserem Ansatz ermöglichen Modellfragmente die Beschreibung der Variabilität ausgewählter Teile des Systems. Werkzeuge unterstüzten dann die Komposition der einzelnen Modellteile zum gesamten Produktlinienmodell.

Prelude

Charles Darwin published a book entitled "The Voyage of the Beagle" in 1839. On his journey to the Galapagos Islands, he discovered among others, fourteen closely related but different species of Finches[1]. Although the birds belonged to the same family and were all about the same size (10–20 cm), they showed some peculiar differences (as depicted in figure 1, the size and shape of their beaks was different). This kind of variability was important for the finches as their survival depended on their capability to adapt to the surrounding– their beaks were highly adapted to make the best use of the available food sources in their habitat islands. Depending on where they lived (on trees, on the ground, etc.) and their primary food source (seeds, flowers, insects, etc.) they had specialized themselves to best fit the environment they lived in. Had this not been the case, they would not have survived. Darwin explained this principle later in his book "*On the Origin of Species*", as the principle of *natural selection*.
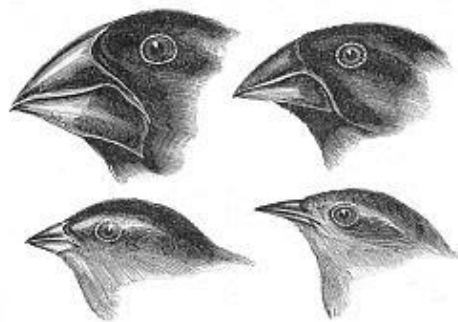

Figure source: Wikipedia

Figure 1: "Darwin's Finches" from Galapagos Islands and natural genetic variability.

Variability is mainly about adaptability. As we will see in this thesis, it is not only important for the survival of living beings, it is equally crucial for software companies to survive in the global competitive market. Conventional single-system software engineering is often insufficient to meet the tight budget and schedule constraints faced by the software industry. Companies therefore aim at understanding the relationships between similar products to exploit commonalities regarding marketing, technical, or end-user aspects. This is achieved by modeling techniques for capturing the variability of reusable core assets such as requirements, architecture, code, processes, documents, or models.

Variability in software architecture can be seen as different decisions which have been taken (or which can be taken) by a software architect when designing a system. The software architecture level of

---

[1]Finches are passerine birds, found chiefly in the northern hemisphere and Africa.

design deals with the specification for the overall system architecture, thereby abstracting from details such as algorithms, data structures or computation models.

Variability helps software architects and developers to react to changing requirements, runtime environments, special wishes of customers and to better deal with software evolution. Furthermore, availability of variability in software is a decisive factor for its reusability. Just like genetic variability helps individuals to adapt to changing living conditions, software variability is crucial for providing the flexibility required in today's software intensive systems.

In the case of living beings, variability is "modeled" in their genes. The decisions about which genes to include or which genes should actually dominate an individual's characteristics is "taken" by nature. Software developers and engineers need to perform the tedious work manually– create variability models as the basis for further automation (comparable to genes of the software system). Such models can then be used by software developers, customers and sales people as a reference model. In this thesis, we shall find out how!

Table of Contents

Part I

Introduction

Introduction and Motivation

Summary *This chapter sheds light on the specific research problems, goals and contributions of this thesis. We analyze the challenges faced by industry and elaborate on the need for variability in software systems. We present our vision of applying model-driven approaches to deal with variability and guide the reader through the rest of the work.*

Traditional ways of building software, i.e., considering software projects to be independent of each other, are no longer sufficient to meet the challenges faced by the software industry. It is necessary to relate different products of an organization in such a way that their commonalities (marketing, technical or end-user considerations) can be shared between various products and projects. Researchers and practitioners agree that *software reuse* is the key to gaining competitive advantage and *variability* is the key to software reuse.

1.1   Software Reuse and Variability

Software reuse is not a new concept. As depicted in Figure 1.1, software reuse has been practiced from the early days of software development. It is however noteworthy, that the level of abstraction on reuse has continuously shifted from low-level program constructs (sub-routines to objects) towards components. Increasing the degree of reuse in software engineering has attracted the interest of both researchers and practitioners for a long time and numerous methods and tools have been proposed. Among these software product lines are particularly interesting [Estublier & Vega, 2005]. There exist several definitions for software product lines. One of the widely accepted definitions is "*a product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the needs of a particular market segment or mission and that are developed from a common set of core software assets in a prescribed way*" [Clements & Northrop, 2001].

Software product line engineering (PLE) is the discipline of creating and managing software product lines. PLE aims at reducing cost and increasing productivity and reliability through leveraging reuse of artifacts and processes in particular domains [Pohl *et al.*, 2005]. It has been demonstrated that PLE is a successful approach to realize the potential of software reuse [van der Linden *et al.*, 2007, Estublier & Vega, 2005]. PLE is increasingly seen as a key strategic approach to attain and maintain

unique competitive positions and to satisfy the time-to-market, cost, productivity, and quality needs in many business environments [Clements & Northrop, 2001, Pohl *et al.*, 2005].
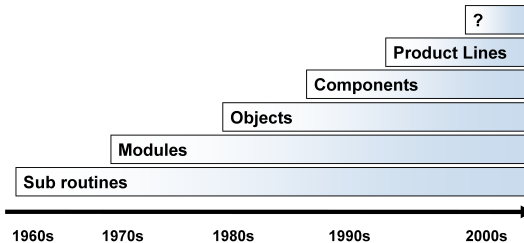
Figure 1.1: History of software reuse.

Many software product lines today are developed and maintained using a model-based approach. Numerous approaches are available for product line modeling, for example feature-oriented modeling languages [Czarnecki & Eisenecker, 2000, Kang *et al.*, 1990], decision-oriented variability modeling approaches [Campbell *et al.*, 1990, Schmid & John, 2004], UML-based variability modeling approaches [Gomaa, 2005], architecture modeling languages [Dashofy *et al.*, 2002], or orthogonal approaches [Pohl *et al.*, 2005]. Numerous tools have been developed to automate domain and application engineering activities based on these models. Despite many success stories (e.g., [Steger *et al.*, 2004, Thiel & Hein, 2002, Estublier & Vega, 2005, Verlage & Kiesgen, 2005]) and a clear trend (away from single-systems) towards variability-driven approaches in industry there are still several obstacles inhibiting widespread adoption. A reason for these problems lies in the inflexibility of existing variability modeling approaches and tools which often do not support the diverse needs of different organizations. While there is a strong consensus on the benefits of variability modeling, it remains challenging for organizations to (i) identify methods and techniques applicable for their particular context, (ii) adapt these methods and techniques to address the specific needs and (iii) integrate them with their current practices, tools, and standards.

Different organizations have different goals, expectations and needs, which is why it is difficult to define a variability modeling language/process that can be applied by all organizations. The need for a flexible variability modeling approach becomes evident when considering different languages, modeling notations, or architectural styles used by different organizations: different stakeholders have different views of the system, speak different languages (use different notations), have knowledge about different aspects of the system and have different expectations and goals. This makes communication between them difficult–which can in turn lead to misinterpretation of information and misunderstandings. It is therefore important to provide well-defined concepts to foster a common understanding of variability and its impacts.

Despite all the advancements in technology, software generation tools, efficient programming languages and progress in other related fields, one of the basic problems which prohibits the industrialization and full automation of software development is the fact that software development

is based on tacit knowledge, i.e., the quality of software usually depends on the creativity, experience and individual knowhow of the developers and architects. Tacit knowledge is often subconscious [O'Dell & Ostro, 1998], internalized, and individuals may or may not be aware of what she knows and how she accomplishes particular tasks. At the opposite end of the spectrum is conscious or explicit knowledge–knowledge that the individual holds explicitly and consciously in mental focus, and may communicate to others. Decisions taken by software engineers have an increasing impact on system quality as software plays an important role in many systems. For example, decisions of software engineers increasingly constrain decisions of other stakeholders in systems engineering, e.g., mechanical and electrical engineers, or sales staff to name but a few. Finding new ways of capturing and sharing the architectural knowledge of the technical software solution therefore has a large potential to improve the development process. This is particularly the case in reuse-driven software development, e.g., due to the strong need for communication between the developer and the "reuser" of software components.

1.2 Model-driven Engineering

A *model* is a simplification of a system built with an intended goal in mind, which answers questions in place of the actual system [Mellor *et al.*, 2004]. Model-driven engineering (MDE) refers to a range of development approaches that use *models* as a primary means of communication, documentation and automation. Model-driven computing paradigms make use of formal notations for capturing the desired abstractions of systems in models. Similar to the basic principle in object-oriented languages, where *everything is an object*, MDE follows the paradigm *everything is a model* [Bézivin & Gerbé, 2001]. A model is described/written in a well-defined language. A well-defined language is a language with well-defined form (syntax), and meaning (semantics), which is suitable for automated interpretation by a computer [Kleppe *et al.*, 2003]. Domain-specific modeling is the use of special languages to answer the questions of interest directly and are more intuitive to capture the intentions of involved stakeholders [Wile & Ramming, 1999]. Because of the use of domain-specific notions, modelers have pre-defined abstractions which directly represent concepts from the application domain. As described by [Bentley, 1986], DSLs can be seen as "specialized languages" that are designed to solve problems in particular domains. DSLs tend to support higher-level abstractions than general-purpose modeling languages, so they require less effort and fewer low-level details to specify a given system.

There are several known advantages of using a DSL over a general purpose language (GPL). The syntax of a concrete DSL usually makes use of natural notations for a domain avoiding syntactic clutter (such clutters are more often when using GPLs). Apart from that, the use of DSLs allows for better error checking [Wile & Ramming, 1999] as problem-specific analyzers can be used to find more errors than similar analyzers for GPLs. Errors can be reported in a language familiar to the domain expert. Since only a particular organization's requirements need be taken into account, it is easier for the modeling language to evolve in response to changes in the domain. DSLs hide lower level programming language details such as complex data structures, complicated algorithms, and tedious GPL syntax from programmers.

This thesis attempts to narrow the gap between developers, architects, sales staff, and customers through capturing and sharing architectural and sales knowledge of software systems in models. Capturing and sharing architectural knowledge is already a complex endeavor when dealing with conventional single-customer software systems. In variable software, the situation is even more difficult due to architectural flexibility and complex relationships between features and technical solution components.

We investigate how and where variability models are suitable as a common knowledge base, i.e., as a means of communication between developers and customers. This is not trivial because people view software from different perspectives. On this regard, we can differentiate between two levels of expertise of individual stakeholders involved in a software project–people with knowledge about the *solution space* and the *problem space* [Czarnecki & Eisenecker, 2000]. Individuals involved in the solution space deal with the technical issues behind how a product is to be implemented. Individuals in the problem space on the other hand focus on trying to understand the customers' problem. Mostly these are two different groups of people–the consequence is obvious: either the problem is wrongly understood or the wrong problem is solved.

The discrepancy between the two groups occurs because of the different perspective from which they view the software solution. Software engineers understand the implementation of the product. They understand the implications of how the product is organized (its architecture or design), and the kinds of issues which were considered when designing or implementing. Customers and sales people tend to be expert system users. They understand the problems the customers have, and why solving those problems will benefit the customer. To narrow the gap (developer–user), we analyze software systems from two perspectives (cf. Figure 1.2):
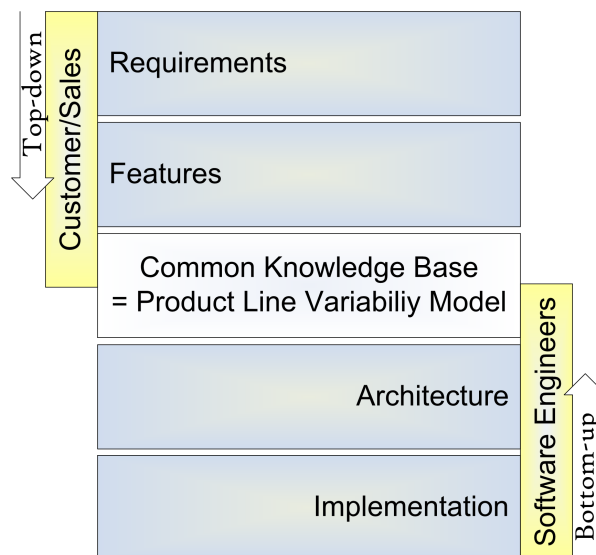
Figure 1.2: Variability models as a means of bridging the gap between customers and developers.

Bottom-up perspective: Developers usually view software from the perspective of the technical solution. They perceive the software as a set of components which they configure, adapt and extend in a predefined way. Software architects have a more abstract view on the underlying concrete technical solution and are mainly concerned with the dependencies and collaborations between components and adapters. The architects' responsibility is to ensure that the required functionality and level of service can be provided and that special requirements can be realized with minimal effort.

Top-down perspective: Sales staff perceive the software as a means to fulfill customer requirements. They emphasize on the distinction between standard features provided by the existing set of components and custom requirements realized through often elaborate extensions and adaptations. A typical issue is that sales people usually are unaware of the technical implications of custom requirements. It is difficult (and also not necessary) for the sales team to understand all the architectural constraints.

In this thesis, we combine top-down and bottom-up perspectives on the software system and capture the tacit knowledge of different stakeholders using models. Customer decision options represent the foundation for the top-down approach. The goal is to formalize the customer perception of the product such that their requirements and wishes can be formally educed. For this purpose, it is essential to understand the different decisions that can be taken by customers and dependencies among decisions. The goal of bottom-up analysis is to capture the already existing architectural variability and to derive the technically feasible decisions that can be taken by customers in product configuration. Decisions that have to be taken by the customer are therefore explicitly modeled. The variability model extracted in such a way can be combined with customer properties thereby allowing the derivation of a set of customer choices.

1.3   Research Motivation

The importance of variability in software systems and the necessity of making knowledge about variability explicit in models have already been identified as important research areas in software engineering. However, due to the broad spectrum of application areas and the diversity of implementation practices in different domains, a "standard" approach for dealing with this problem will probably never exist. There exist many "island solutions" for variability modeling which either focus on one particular level of abstraction (features/requirements or architecture) or are monolithic and fixed to a certain grammar, with set of predefined features. This hinders the widespread use of the existing approaches in different domains and application contexts. Despite the importance of variability modeling and the usage of such models in a wide array of contexts, researchers and practitioners are still struggling to find tools and techniques that best suit their modeling needs.

1.3.1   Industry Problems

This thesis was motivated by challenging problems faced by the industry. The research was conducted at the Christian Doppler Laboratory for Automated Software Engineering[1], in close collaboration with Siemens VAI[2], the world's leading engineering and plant building company for the iron, steel, and aluminum industries. Here we summarize *typical industry problems* which triggered this work:

- The industry is facing difficulties in understanding and modeling variability at different levels, i.e., features, architecture, documentation etc. The integration of variability at different levels is difficult, which results in isolated view on variability, effects of decisions are not visible to different user groups.

- Variability modeling requires know-how and experience, which is concentrated in few people. In most companies, these people are bottlenecks in the development process.

- Due to the inadequate coordination between Engineering and Sales, stakeholders do not understand the relationships between project decisions and technical configuration.

- Dealing with real-world complexity of constraints in formal notations is a big challenge, there is almost no support available for non-technicians, e.g., Sales staff vs. formal feature models.

- Variability models are seldom integrated with other models, which often causes redundant data-entry.

- Due to inflexibility of existing approaches it is hard to deal with new types of artifacts, e.g., new types of files.

Our vision is to develop tools and techniques to support the following software sales paradigm: The customer decides system functionality based on existing configurable components and receives a well proven product, which is robust, quickly deliverable, extensible and takes hidden requirements into consideration. The software is correct and complete right from the beginning. Flexibility, maintainability and testability are guaranteed over the complete lifecycle. The necessary process optimization features are present in the form of a highly developed process model. In order to address the typical problems faced by industry and come closer to realizing our vision, we develop a framework for rapidly building variability modeling languages (and generating modeling tools) to meet the needs of different domains.

1.3.2   Research Issues

In the course of this research project, we have identified several research issues, which are typical problems faced by both research and industry. These issues which lead to the research questions driving

---

[1]http://ase.jku.at
[2]http://www.industry.siemens.com/metals-mining/EN/

this work have been identified through a thorough literature review and workshops (meetings and discussions) with experts from the industry.

- There is a lack of integrated variability modeling approaches that work well with arbitrary and heterogeneous types of assets in different development environments.

- There is a lack of flexible and extensible tools that can be tailored to support a particular organization's needs.

- There is a lack of model modularization techniques to simplify evolution management of models, which is a crucial requirement for the success of model-based approaches.

The value of this thesis is not as much in answering the questions individually and in depth, but rather in combination and integration. The purpose of this research is thus to determine what improvements can be made to the current body-of-knowledge for variability modeling and to present an approach for doing so.

Research Issue: Integrated variability modeling

Managing variations at different levels of abstraction and across all generic development artifacts is a daunting task [Berg *et al.*, 2005], especially when the systems supporting various products are very large, as is common in an industrial setting. Each organization is unique; it has its own development practices, tools and techniques. A variability modeling approach should be flexible, so that variability at different abstraction layers (requirements, architecture or implementation) can be dealt "under one umbrella". Furthermore, it should be easily extensible to support the modeling of arbitrary artifacts in different domains. From these requirements regarding the required traceability facilities, we derive our first research question.

> **RQ1:** How can we model the variability of arbitrary artifacts and their dependencies?

Research Issue: Flexibility and extensibility of modeling tools

The complexity of today's software systems cannot be handled manually. Tools must be provided, wherever possible, to guide the creation and utilization of variability models. Using a domain-specific approach for modeling requires domain-specific tools. It is therefore important that the tools are as flexible as the approach itself to cope with different requirements in diverse application domains.

> **RQ2:** How can we provide effective tool support for variability modeling?

Research Issue: Structuring the modeling space

The scale and complexity of real-world product lines makes it practically infeasible to develop a single model of the entire system, regardless of the languages or notations used. Product line engineers need to apply different strategies for structuring the modeling space to support the creation and maintenance of the models. The high number of features and components in real-world systems means that modelers need strategies and mechanisms to organize the modeling space. *Divide and conquer* is a useful principle but the question remains which concrete strategies can be applied to divide and structure the modeling space.

> **RQ3:**   How can we support structuring of the modeling space?

1.4   Research Agenda

Not all of the research questions presented here shall be dealt with equal priority in this thesis. The primary goal is to design a domain-specific variability modeling framework supporting modeling and maintenance of variability models, thereby allowing the modeler to structure the modeling space. The modeling approach should incorporate both technical variability and business variability models. Apart from that it should be possible to map the structure of existing software system into the modeling paradigm.

1.4.1   Research objectives

The goal for this thesis is to create methods and tools to respond to the challenges outlined in RQ1, RQ2 and RQ3. More specifically we plan to:

- Create a **method for modeling variability**, which can incorporate different layers of abstraction in one model. The method should be independent of practices in different organizational settings and development practices.

- Demonstrate the **applicability of the approach** in different domains. It should be clear from different case studies, that the approach is independent of implementation practices in different domains and it can be used for different purposes, i.e., configuration, runtime adaptation, etc.

- Identify required capabilities for suitable tool support and use **prototypic implementation of tools** for the proof of concept. The tools should be flexible, adaptable and extensible to support the different modeling and evolution scenarios.

- Devise an approach for **structuring and evolving variability models**. The approach should be able to deal with large-scale software systems and allow structuring of the modeling space. For

example, typical situations in the industry should be supported, where development teams are actually "teams of teams".

By combining the flexibility provided by meta-tools and the intuitiveness that comes from domain-specific languages, we envision a highly customizable and easily extensible framework for domain-specific variability modeling.

1.4.2   Iterative Research Method Driven by Industry Needs

The research presented in this thesis has been performed at the Christian Doppler Laboratory for Automated Software Engineering[3] , together with the industry partner Siemens VAI[4]. In the course of the project, we used the industry as our laboratory and real world projects as our test bed (both for our approach and for supporting tools). The feedback from the experts in the industry therefore had a big impact on the research approach. The use of real world software as the test bed for our approach had many advantages, but also posed a few challenges for us.

Advantages: The biggest and perhaps the most obvious advantage was that we could take real large-scale systems into account and so we were aware of real problems of the architects and developers at work. This helped in understanding the problem better and enabled high quality research work not only limited to "toy examples". The industrial background of our approach and the real working context of our tools were appreciated by many peer reviewers and participants of several international conferences.

Challenges: Some of the challenges we faced were with regard to the validation of the approach. Experiments as a means of validation, as suggested by Wohlin [Wohlin *et al.*, 2000] and Basili [Basili, 1993] were difficult to perform as these require by definition at least two groups where different approaches are followed and at the end, the results of the different groups are compared. When working with large real-life systems, such an experiment is a huge financial burden. It is almost impossible to change context in which stakeholders from the industry partner work only for the sake of experiments.

Due to the given circumstances, under which the research was carried out, we applied a research method, which was highly driven by the needs of the industry. Our approach is iterative, meaning that the results are continuously revised and approved by our industry partner. The tool prototypes built for this purpose used feedback from our industry partner for early validation of research ideas. Furthermore, the value (or acceptance) of our research (just like in most software engineering related research) is not only dependent on the purely technical merits, but also on how it can fit in the overall context of business, architecture, process and organization. By and large, the following steps were pursued- the results were tested and verified in multiple iterations (see Figure 1.3).

---

[3]http://ase.jku.at/
[4]http://www.industry.siemens.com/metals/
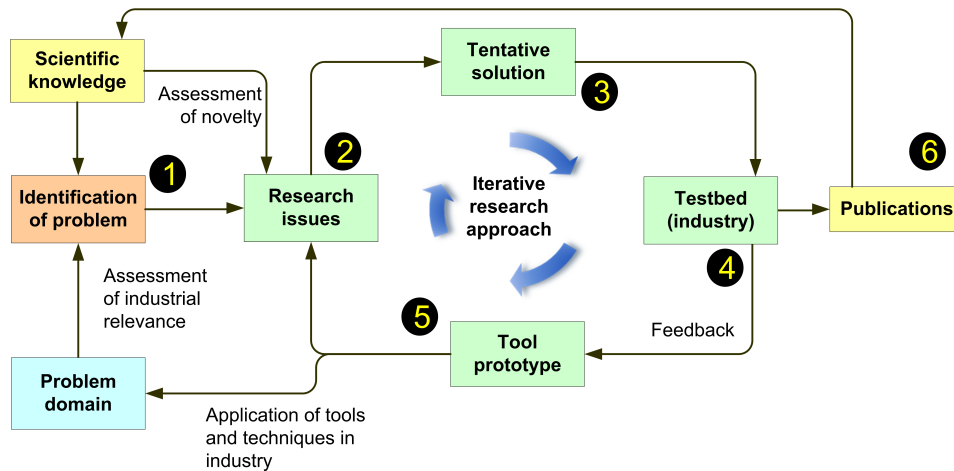
Figure 1.3: Overview of research approach.

1. *Identification of the industrial problem*: As this research was carried out with an existing industrial background, the selection of the problem domain was not an issue (the industry partner was fixed). So the first step was the analysis of our industry partner's current practices, thereby trying to spot the bottlenecks in the process and difficulties faced by developers and architects.

2. *Identification of research issues*: After a thorough analysis of the industrial problem and a detailed literature review, we were able to identify the research issues underlying the given problems. This was not always easy as sometimes it was necessary to find the right trade-offs between the requirements from the industry and the novelty in research.

3. *Development of tentative solution*: With the identified research issues, we proceeded to develop tentative solutions.

4. *Feedback from industry partner*: The tentative solutions were presented to the engineers and developers from our industry partner.

5. *Tool prototype development and application*: In order to convince the industry partner of the applicability of the approach, it was inevitable for us to develop tool prototypes, that demonstrated the approach. After countless iterations and tool demonstrations, we could enthuse our industry partners to actually use the tools for modeling purposes.

6. *Publication of research results and feedback*: It was equally important for us to enthuse the research community and present them our results. We primarily focused on software engineering conferences like ASE, SPLC, EUROMICRO, RE, WICSA etc, and published a total of 23 peer reviewed papers (selected list of publications depicted in Table 1.1).

1.5   Contributions

The novelty of our research lies in the combination of approaches for modeling stakeholder needs (requirements), product characteristics (features), architectural elements and other resources in an integrated manner. By using variability models as a means to capture tacit knowledge required for configuration, adaptation and monitoring of software systems, we provide a novel approach for domain modeling that goes beyond the capabilities of existing tools and techniques. This research will care for the crucial issue of evolving and maintaining product lines and their data and models. New maintenance processes and respective tools will be developed which are tightly integrated into PLE and other software development processes.

The approach presented in this thesis has been validated in several case studies, where we demonstrate the applicability of the tools and techniques in different domains. Apart from that, several refereed publications within the context of this thesis also provide proof for the acceptance and recognition from the research community. Here we present a brief summary of relevant publications and their contributions to the research questions.

Table 1.1: Selected publications and their contribution to the research questions.

	Publication	RQ1	RQ2	RQ3	Related
1.	*Structuring the Product Line Modeling Space: Strategies and Examples* 3rd International Variability Modeling Workshop (VAMOS 2009). *Authors*: Grünbacher P., Rabiser, R. Dhungana, D. Lehofer, M.	−	−	$\sqrt{}$	chapter 6
2.	*Supporting Evolution in Model-based Product Line Engineering.* 12th International Software Product Line Conference (SPLC 2008). *Authors*: Dhungana D., Neumayer T., Grünbacher P., Rabiser, R.	−	−	$\sqrt{}$	chapter 6
3.	*Understanding decision-oriented variability modeling.* Workshop on Analyses of Software Product Lines, in collocation with the 12th International Software Product Line Conference (SPLC 2008). *Authors*: Dhungana, D. and Grünbacher P.	$\sqrt{}$	−	−	chapter 4
4.	*Runtime Adaptation of IEC 61499 Applications Using Domain-specific Variability Models.* Workshop on Dynamic Software Product Lines, in collocation with the 12th International Software Product Line Conference (SPLC 2008). *Authors*: Froschauer, R., Dhungana, D., Grünbacher P.	$\sqrt{}$	−	$\sqrt{}$	chapter 8
5.	*Product line tools are product lines too: Lessons learned from developing a tool suite.* IEEE/ACM International Conference on Automated Software Engineering (ASE 2008). *Authors*: Grünbacher P., Rabiser. R., Dhungana, D.	$\sqrt{}$	$\sqrt{}$	−	chapter 7
6.	*Managing the Life-cycle of Industrial Automation Systems with Product Line Variability Models.* 34th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA 2008). *Authors*: Froshchauer, R., Dhungana, D., Grünbacher P.	$\sqrt{}$	−	$\sqrt{}$	chapter 8

7.	*Supporting Runtime System Adaptation through Product Line Engineering and Plug-in Techniques.* 7th IEEE International Conference on Composition-Based Software Systems (ICCBSS 2008). *Authors*: Wolfinger R., Reiter S., Dhungana D., Grünbacher P., and Prähofer, H.	$\sqrt{}$	$\sqrt{}$	–	chapter 7
8.	*Supporting Evolution of Product Line Architectures With Variability Model Fragments.* Working IEEE/IFIP Conference on Software Architecture (WICSA 2008). *Authors*: Dhungana, D., Neumayer, T., Grünbacher, P., Rabiser, R.	–	–	$\sqrt{}$	chapter 6
9.	*Value-Based Elicitation of Product Line Variability: An Experience Report.* Second International Workshop on Variability Modeling of Software-intensive Systems (VAMOS 2008). *Authors*: Rabiser, R., Dhungana, D., Grünbacher, P., Burgstaller, B.	$\sqrt{}$	–	–	chapter 7
10.	*Dealing with Changes in Service-Oriented Computing Through Integrated Goal and Variability Modeling.* Second International Workshop on Variability Modeling of Software-intensive Systems (VAMOS 2008) *Authors*: Clotet, R., Dhungana, D., Franch, X., Grünbacher, P., Lopez, L., Marco, J., Seyff, N.	$\sqrt{}$	–	$\sqrt{}$	chapter 9
11.	*Domain-specific Adaptations of Product Line Variability Modeling.* IFIP WG 8.1 Working Conference on Situational Method Engineering (SME 2007). *Authors*: Dhungana D., Grünbacher P., Rabiser R.	$\sqrt{}$	–	–	chapter 4, 5
12.	*Integrated Tool Support for Software Product Line Engineering.* 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007). *Authors*: Dhungana, D., Rabiser, R., Grünbacher, P., Neumayer, T.	–	$\sqrt{}$	–	chapter 5
13.	DOPLER: *An Adaptable Tool Suite for Product Line Engineering.* 11th International Software Product Line Conference (SPLC 2007) *Authors*: Dhungana, D., Rabiser, R., Grünbacher, P., Lehner, K., Federspiel, C.	–	$\sqrt{}$	–	chapter 5
14.	*Decision-Oriented Modeling of Product Line Architectures.* Working IEEE/IFIP Conference on Software Architecture (WICSA 2007). *Authors*: Dhungana, D., Rabiser, R., Grünbacher, P.	$\sqrt{}$	–	–	chapter 4
15.	*Goal and Variability Modeling for Service-oriented System: Integrating i* with Decision Models.* Software and Services Variability Management Workshop - Concepts Models and Tools (SSVM 2007). *Authors*: Grünbacher P., Dhungana D., Seyff N., Quintus M., Clotet R., Franch X., Lopez L., Marco J.	$\sqrt{}$	–	$\sqrt{}$	chapter 9

16.	*DecisionKing: A Flexible and Extensible Tool for Integrated Variability Modeling.* International Workshop on Variability Modeling of Software-intensive Systems (VAMOS 2007). *Authors*: Dhungana, D., Grünbacher, P., Rabiser, R.	–	√	–	chapter 9
17.	*Coordinating Multi-Team Variability Modeling in Product Line Engineering.* Workshop on Supporting Knowledge Collaboration in Software Development (KCSD 2006). *Authors*: Dhungana, D., Rabiser, R., Grünbacher, P.	–	–	√	chapter 6
18.	*Integrated Variability Modeling of Features and Architecture in Software Product Line Engineering.* Doctoral Symposium, 21st IEEE International Conference on Automated Software Engineering (ASE 2006). *Authors*: Dhungana, D.	√	√	√	chapter 1
19.	*Architectural Knowledge in Product Line Engineering: An Industrial Case Study.* 32nd EUROMICRO Conference on Software Engineering and Advanced Applications ((SEAA 2006). *Authors*: Dhungana, D., Rabiser, R., Grünbacher, P., Prähofer, H., Federspiel, C., Lehner, K.	√	–	–	chapter 1

1.6   Reader's Guide

Part I. Introduction

The first part sheds light on the research carried out, which ultimately resulted into this thesis. By analyzing the research issues and challenges from a software engineer's perspective, we guide the reader through the rest of the work.

Chapter 1–describes the research problems underlying this thesis. It elaborates on the research issues and is intended to motivate the reader.

Chapter 2–introduces software variability by providing examples of variability and identifies different application areas for variability models.

Chapter 3–provides an overview of the state of the art in variability modeling and an analysis of strengths and weaknesses.

Part II. Approach

The second part is about the approach developed in our research project and the tools supporting our approach.

Chapter 4–describes the decision-oriented approach for modeling variability including language definition (syntax and semantics) and domain-specific adaptations.

Chapter 5–presents a prototypic implementation of tools to support the modeling and evolution approach.

Chapter 6–presents an approach for dealing with evolution of variability models in multi-team environments and tool-support.

Part III. Evaluation

This part deals with the application of the approach and tools in different projects. Investigation of the applicability of variability models in different domains, for different purposes.

Chapter 7–describes the evaluation plan. By relating the evaluation studies to the research questions and by elaborating on the facts which are investigated by the studies, we guide the reader through the different case studies.

Chapter 8–describes one of the large-scale case studies carried out with our industry partner Siemens VAI. We use our decision-oriented approach for modeling and managing variability of continuous casting steel plants.

Chapter 9–describes a case study which sheds light on the usage of variability models at system runtime. Here we describe how industrial automation systems based on IEC 61499 benefit from such an approach.

Chapter 10–describes a case study, where we used our modeling approach and tools to monitor service oriented systems at runtime. By complementing goal modeling techniques with variability modeling, we also define rules for identifying variability in i* models.

Part IV. Final Remarks

Chapter 11–This chapter concludes the thesis with a summary and an outlook on further work. We also reflect on the lessons learned while carrying out the research work.

> ❝ There is nothing more difficult to take in hand, more perilous to conduct or more uncertain in its success than to take the lead in the introduction of a new order of things. ❞ —*Niccolo Machiavelli "The Prince", 1532*

Research on Software Variability

Summary *In this chapter we introduce the research problems underlying this thesis. By providing lively examples of variability in software systems and their usage in different contexts, we work out the different research areas related to software variability.*

Implementing a software system is about translating the needs of potential customers into an executable software solution. As more and more customer requirements are taken into consideration, the software system becomes more and more specific to the needs of one particular customer. In order to be able to react to changing requirements (either at later development phases or in the course of software evolution), software systems are increasingly designed to be variable. Van Gurp *et al.* describe variability in software as "delayed design decisions" [van Gurp *et al.*, 2001], where the software engineer allows for different choices during later phases of software development, rather than specifying all the details at the beginning. However, not all design decisions are related to software variability.

2.1   What is Variability?

Variability is the result of different design decisions which are wired into the assumptions/knowledge about the different kinds of contexts that need to be supported by the system. It is an important pre-requisite for successful application of reuse-based approaches. A typical reuse-driven software development process consists of three different activities:

1. *Selection* helps in determining the suitability of a component for use within the intended final system. It is based on comparing one component against others and evaluating its fitness of reuse.

2. *Adaptation* helps in preparing the components to match their assumptions about the context in which they are deployed. The purpose of adaptation is to ensure that conflicts among components are minimized.

3. *Assembly* is the integration of components through a well defined infrastructure, which provides the binding that forms a system from individual components.

When a system is built by *selecting*, *adapting* and *assembling* existing components, one can differentiate between variable and common parts between the different products built from the same product line. Let's consider a set of components which has been used to build three different products. As depicted in Figure 2.1, each of the three products has some distinct product-specific functionality and there exists some overlapping functionality between the products. Apart from that, the products also consist of different variations of the same conceptual functionality, a result of the adaptation of the components in use. Assembling such a system without proper documentation and guidance is a tedious and error-prone task. Most of the reuse-centered approaches seem to silently ignore the fact that reusing a component in a different context is often a non-trivial task, especially when it has to be adapted to match slightly different requirements.
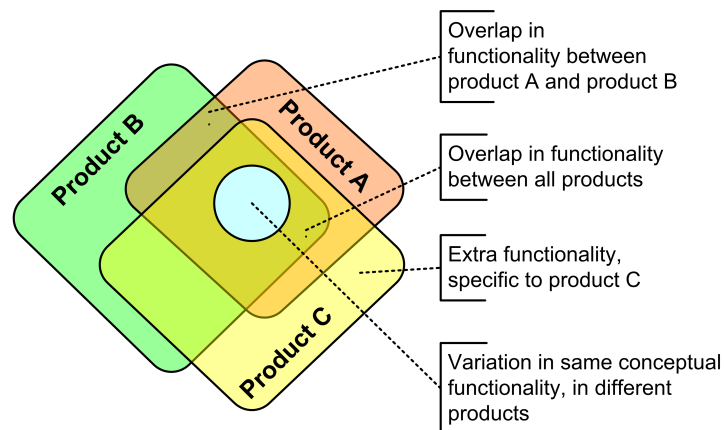
Figure 2.1: Commonality and variability between different products.

Configuring a software system to the specifics of new domains and environments is challenging. In many software systems (without variability in design), this is technically impossible. In others (where variability was planned), with the increase in reconfigurability of the system, the knowledge required for the management of the complexity also increases tremendously. This leads to situations, where special tools and techniques are needed to support individual stakeholders to share their knowledge about the system with other stakeholders.

2.1.1 Variability Occurrence

Dealing with variability (implementing, understanding and managing variable systems) is not trivial, as it depends on the software development practices in a particular environment. Variability can either be an emergent or a planned property of software systems. It results from different decisions taken by architects and developers to address requirements and contexts from different users. Experience from existing large-scale systems shows that knowledge about variability is mostly tacit in

nature and manifests itself in many different kinds of artifacts (documents, software components, test cases, configuration parameters, etc.) and different mechanisms supported by programming languages, architectural styles, design patterns, etc. When planning new software systems, variability is reflected in different variants of software architecture for fulfilling customer requirements and alternative mechanisms of implementing the architecture.

According to Pohl *et al.* [Pohl *et al.*, 2005] the discussion on variability is basically an attempt to find answers to two questions regarding reusable "assets" or "artifacts":

What varies? The answer to this question is also known as the "variability subject". A variability subject is a variable item of the real world or a variable property of such an item, identifying precisely the variable item or property of the real world.

How and where does it vary? A variability object is a particular instance of a variability subject. The variability object is used to identify the different "shapes" of a variability subject.

2.1.2 Impacts of Variability

Different sorts of variability have different negative and positive impacts. In theory it is good to have as much variability as possible. The principal danger is that of *excessive variability*, i.e., variability which goes beyond that needed for a positive interoperability trade-off, and which unnecessarily complicates the development process [Hazael-Massieux & Rosenthal, 2005]. Developers and engineers need to carefully consider and justify any variability allowed and its affect on final products. This can be done by explicitly documenting the choices made.

Variability is important for all phases of software development, e.g., variability at design time allows for creating a customized solution to meet the needs of individual customers or user groups. This is a common practice in product line engineering, where variability in the available software commodities (core assets) is exploited to create a family of products. Variability of software applications at runtime is important in long running applications like web servers or industrial automation systems. Dealing with variability at runtime is important, when a system-halt for reconfiguration is either economically or technically not feasible.

Variability has to be understood at different levels (e.g., requirements, architecture, or implementation) and for diverse domain-specific artifacts. The traceability between variation points, i.e., decision points describing possible choices about assets' functions or qualities, and the management of variability mechanisms implementing these points are important aspects. Variability models cover the problem space (stakeholder needs and desired features) and its solution space (architecture and components of the technical solution). They capture different variants of features and solution components and their valid combinations, i.e., the possible variants together with constraints and dependencies. Variability models also document fundamental system-wide decisions for the configuration and derivation of a product and the rationale for these decisions.

2.2   Dimensions of Variability

Different perspectives from which variability can be categorized/analyzed are referred to as dimensions of variability. Different dimensions are not necessarily orthogonal to one another. There are many possible associations, dependencies, and interrelationships.

2.2.1   Temporal and Spatial Variability

Several authors [Bosch *et al.*, 2002, Coplien *et al.*, 1998, Pohl *et al.*, 2005] differentiate between variability in space (spatial variability) and time (temporal variability). The presence of variability in space means that the same set of development artifacts is used to derive multiple applications with different features. Variability in time is given if the existence of different versions of an artifact that are valid at different times. The time dimension covers the change of a variable artifact over time. The space dimension covers the simultaneous use of a variable artifact in different shapes by different products.

2.2.2   Internal and External Variability

Different stakeholders perceive the variability of the system in two ways- (i) problem space oriented stakeholders focus on trying to understand the customers' problem and (ii) solution space oriented stakeholders deal with the technical issues behind how a product is to be implemented. The role of the stakeholder therefore represents another dimension of variability. Table 2.1 provides some examples.

Pohl *et al.* [Pohl *et al.*, 2005] differentiate between internal and external variability of software systems. External variability is the variability of domain artifacts that is visible to customers. Internal variability is the variability of domain artifacts that is hidden from customers, i.e., only known to developers and engineers.

Table 2.1: Example of problem space and solution space variability.

Problem space variability	Solution space variability
Support file transfer: Yes | No.	File transfer protocol to be used, e.g., Simple FTP, Secure FTP.
Maximum number of concurrent users to be supported: 100 | 1000 | 10000.	Type of database to be used, e.g., MySQL, Oracle Standard, Oracle Enterprise.
Communication with external services: Yes | No.	Type of communication protocol to be used, e.g., Binary, ASCII, XML.
Support instant messaging: Yes | No.	Chat client to be used, e.g., Skype, ICQ.

2.2.3   Artifact-level Variability

One can also differentiate between different layers of variability based on the type of artifact involved. Component-level variability is for example relevant and interesting for software architects, whereas the sales personnel rather deals with variability of requirements. Software developers on the other hand are interested in code-level variability.

Let us consider the different functionalities of a weather station in terms of variability in the requirements. As shown in Table 2.2, the variability of a system can be described using the terms which denote the availability of options (e.g., "can", "may", "might", etc). The dependencies among the different possible choices are usually described using conditional and prescriptive terms (e.g., "if ... then", "must", "have to" etc.). This clearly shows that with the increasing number of differences among the products that can be built, tools and techniques are needed to manage the complexity of the collected information.

Table 2.2: Example of requirements variability in a weather station control software.

Variability Type	Functionality
Mandatory	All weather stations consist of sensors to report the current temperature.
Optional	Some whether stations might report on the direction of wind.
Implied mandatory	If a weather station reports on the direction of wind, it must also report the speed of wind.
Conditional optional	If a weather station reports on direction of wind, it might additionally consist of a hurricane warning system.
Optional	Some weather stations report can measure the humidity of air.
Alternative	A weather station is either specialized for marine weather phenomena or alpine weather phenomena.

2.3   Variability Modeling

Variability models have been proposed as a means of communication to deal with explicit documentation of tacit knowledge and better utilization of the flexibility and adaptability provided by a system. In many cases, even a highly variable system cannot be efficiently adapted to a new environment (set of requirements) because of the lack of the knowledge about the available variability. To overcome such situations, sharing of knowledge among members of a team is inevitable. Due to the large size of software systems today, it is not practicable for individual stakeholders to act as "knowledge carriers", as the number of such carriers is limited.

The process of documenting and defining the variability of a system, such that the tacit knowledge in the heads of different stakeholders is made available is known as variability modeling. The amount and the kind of information that is contained in a variability model depend on the motives behind

variability modeling. As for example, the feature oriented domain analysis method [Kang *et al.*, 1990] supports the identification of prominent or distinctive user-visible features within a class of related software systems. Other approaches such as (Kumbang [Männistö *et al.*, 2001] and staged configuration of feature models [Czarnecki *et al.*, 2004]) focus on the usage of variability models for configuration purposes.

A thorough analysis of the state-of-the-art and workshops with experts from the industry have revealed that basically 6 different aspects of variability need to be considered (see Figure 2.2).
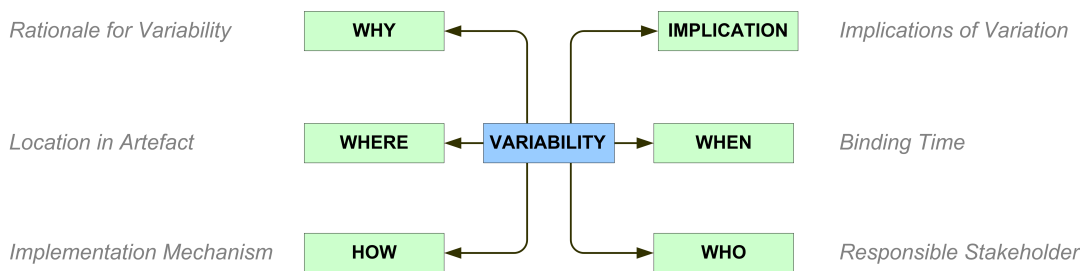
Figure 2.2: Different aspects of variability which need to be covered by a variability model.

WHY– The rationale for variability is probably the most important aspect of variability modeling because elicitation of this kind of knowledge can hardly be automated. For example, the manufacturer of a mobile telephone can decide whether to install Bluetooth or Infrared communication protocol in a certain model. The rationale behind this could be the need to address different user groups with different working environments. A variability model should be able to answer the question: "why is there a need for a certain variation?" [Thurimella, 2008].

WHERE– In addition to why the system was designed to be variable, it is important to know which development artifacts are related with the variability. Depending on how the variability is implemented, there can be several artifacts which realize one particular variability issue. The type of these artifacts differs in different domains. The point in the artifact space, where the variability occurs is referred to as the *variation point*.

HOW– Related to the different kinds of domain-specific artifacts are the mechanisms that are used to realize the required variability (see Section 2.4). These mechanisms range from low-level constructs such as configuration parameters, reflective programming techniques or conditional compilation–to higher design level techniques such as plug-in frameworks, meta-tools and application frameworks.

IMPLICATION– The consequence of implementing variability can be seen as fulfilling the rationale behind that particular variation point. This is however not always clear as the implication of making decisions about variability (e.g., changing configuration parameters or adapting a conditional

compilation template) is not explicitly documented. A variability model should therefore also answer the question, what is the consequence of making decisions related to the variability of the system?

WHEN– By defining the stage in the development process, one can specify the time which is suitable for taking configuration decisions. For example, decisions related to the high level functionality of the system (e.g.,. will a DVD player be integrated in a car?) can be taken at design time, some other decisions (e.g., which network protocol is used for communication?) are relevant at implementation time and yet others can only be taken at runtime and so on. The documentation of binding times [Schmid & John, 2004] for variability is therefore also an integral part of a variability model.

WHO– It is not enough to know why something can be changed and what the implications of the change are, if nobody makes the change. It is therefore equally important to know who (e.g., on the basis of stakeholder roles such as software architect, developer or sales person) can, may or must take the decision to decide among the available options. Sometimes such decisions can also be automatically taken by the system (on behalf of the user based on other decisions).

2.4   Variability Implementation Mechanisms

As exemplified in Table 2.2, the basic idea behind variability in software is easy to understand. However, creating a variable system that can be efficiently adapted to the new situation is a creative exercise, which requires a deep understanding of the domain, a sustainable architecture and appropriate assets for the context of reuse. Developers need to focus on what objects are supposed to do, not only on how to implement them. It is important to consider what is variable in the design and then isolate it, so that it is exchangeable. By putting layers between things that independently change and by programming to an interface, not the implementation, one can design software for optimal variability.

Today numerous variability implementation mechanisms such as configuration parameters, plug-in frameworks, meta-tools (generative techniques), application frameworks, reflective programming techniques or conditional compilation are adopted to support variability [Anastasopoulos & Gacek, 2001, Fritsch *et al.*, 2002].

2.4.1   Programmatic Practices

The use of programmatic variability implementation practices allows for more flexibility. Variants are programmed on the basis of existing code. It is necessary to have a good software architecture, which allows for such changes, extensions and adaptations. Examples of such techniques are inheritance, factory patterns, etc. Aspect-oriented techniques enable the explicit expression and modularization of crosscutting variability on model, code, and generator level [Voelter & Groher, 2007, Anastasopoulos & Muthig, 2004].

Plug-in techniques allow the extension of tools with new capabilities implemented as components by "plugging" them into the application core. Eclipse[1] is a prominent plug-in platform. It is built upon a small core and all needed functionality is provided via plug-ins. Eclipse plug-ins are written in Java and delivered as *jar*-libraries. The NetBeans[2] platform has also introduced a plug-in concept based on a virtual file system representing the hierarchical structure of the application. Plug-ins can extend Net-Beans by contributing to this virtual file system. The customization of plug-in-based systems is done by installing new components whenever required. It is obvious that such an approach increases the complexity of plug-in systems over time and their full exploitation becomes more difficult and error-prone without support for documenting the implications of adding, updating, and removing plug-ins. Wolfinger *et al.* have demonstrated the use of the .NET Plugin framework Plux.NET [Wolfinger *et al.*, 2008] in the context of a product line to achieve the desired flexibility in ERP applications.

Application frameworks like the Open Service Gateway Initiative (OSGi) [OSGi Alliance, 2003] also enable flexibility of tools. OSGi is a Java-based technology for deploying and managing components (referred to as bundles). It defines several mechanisms that support flexibility in the life cycle management of components. Technically, the OSGi framework provides a custom, dynamic Java class-loader and a service registry that is globally accessible within a single Java virtual machine. Another example is the Spring framework [Walls & Breidenbach, 2007], a full-stack Java/J2EE application framework. It supports adaptation by providing extensive support for assembling components via dependency injection and configuration files.

2.4.2   Descriptive Practices

Whenever the source code does not need to be changed to adapt the functionality of the system, we refer to such mechanisms as descriptive variability implementation practices. This is usually realized using property files, configuration constants and other kinds of parameterization techniques. Descriptive variability is usually adopted to implement foreseen variability. There are a few limitations of such techniques: one cannot add any completely new functionality without changing the core asset components. Parameterization only allows a "reuser" to change the values of the attributes in a core asset component or to chose from a list of predefined options on the core asset component. Use of configuration constants is a common implementation practice for factoring constant values into symbolic constants.

Configuration parameters and user preference settings support customization by changing preset parameters typically encoded as key-value pairs. They provide basic flexibility in tools (e.g., language settings, appearance, layout of the user interface, etc.). Variability is presented to users via configuration files and user preference dialogs. Tools make extensive use of this mechanism and typically provide a huge set of parameters supporting their adaptation. However, the impact of parameters and their dependencies are rarely documented which makes it hard for users to achieve specific cus-

---

[1]http://www.eclipse.org
[2]http://www.netbeans.org

tomization goals. In addition, configuration parameters alone do not provide the required degree of flexibility and need to be complemented with approaches supporting architectural variability.

```
#ifdef CONFIG_M68K
    #include <asm/setup.h>
#endif

#if defined(CONFIG_MIPS) || defined(CONFIG_MIPS64)
    #include <asm/bootinfo.h>
#endif

const struct linux_logo *fb_find_logo(int depth){
    const struct linux_logo *logo = 0;
    #ifdef CONFIG_LOGO_LINUX_MONO
        /* Generic Linux logo */
        logo = &logo_linux_mono;
    #endif
    #ifdef CONFIG_LOGO_SUPERH_MONO
        /* SuperH Linux logo */
        logo = &logo_superh_mono;
    #endif
    . . . .
    . . . . . .
}
```

Listing 2.1: Logo configuration using ifdef in Gentoo Linux Boot-up sequence.

2.4.3   Model-based Practices

Meta-tools such as MetaEdit+ and Pounamu support customization by generating domain-specific tools based on a specification. MetaEdit+ [Tolvanen & Rossi, 2003] is a tool for designing a modeling language, its concepts, rules, notations, and generators. The language definition is stored as a meta-model in the MetaEdit+ repository. MetaEdit+ provides a fully-fledged modeling tool with diagramming editors, browsers, generators, and multi-user support based on the defined modeling language. Pounamu [Grundy *et al.*, 2006, Zhu *et al.*, 2007] is a meta-tool for the specification and generation of multiple-view visual tools. The tool permits the rapid specification of visual notational elements, the underlying information model, visual editors, the relationships between notational and model elements, and the elements' behavior.

The Eclipse Modeling Framework (EMF)[3] supports the development of custom-made tools based on structured data models. Based on a model specification EMF provides a basic editor together with tools and runtime support for viewing and modifying models. IMP [Charles *et al.*, 2007] is an IDE meta-tooling platform that aims to reduce the burden of IDE development in Eclipse. The approach supports the customization of IDE appearance and behavior and aims at reusing code during IDE development.

In the area of product line engineering Voelter and Groher [Voelter & Groher, 2007] suggest a staged approach based on meta-models: In the first stage the tool for defining the product line is configured to the needs of the problem domain. Using a case study of a home automation system the

---

[3]http://www.eclipse.org/emf

authors demonstrate that the meta-model and domain-specific language for describing systems in the problem domain needs to be configurable.

2.5   Summary and Critical Analysis

Software development relies heavily on the use of variability to manage the differences between products by delaying design decisions to later stages of the development and usage of the constructed software systems. Depending on different programming languages, existing infrastructure and architectural conventions these techniques range from purely descriptive (configuration parameters, XML files), to purely programmatic (exploitation of object-oriented technology and the use of aspects). Therefore, implementation of variability is not a trivial task [Svahnberg *et al.*, 2005].

The benefits of a modeling approach and the value of capturing tacit knowledge explicitly in models is perceivable only when the information in the models can be used either by the involved stakeholders or by automated tools. In general, models help in a better understanding of the system by raising the level of abstraction. In the first place, in the long term software developers have less work to do, because variability enables reuse of the developed assets in a lot of different contexts. The second improvement comes from the fact that the developers can shift focus from code to models, thus paying more attention to solving the business problem at hand. This results in systems that fit much better with the needs of the end users. The end users get better custom-made software systems in less time.

There are also some disadvantages of using a model-based approach. For example, models are mostly only used to generate standard solutions. Using a model means introducing a new source of error and therefore, in addition to software evolution issues, model maintenance and synchronization also need to be addressed.

2.5.1   Benefits of Variability Modeling

Apart from general benefits of model based approaches, we elaborate more specifically on the advantages of a variability modeling approach in different phases of software development. By introducing variability in the design phase, implementation, or runtime environments, variable systems can be efficiently extended, adapted, customized, configured or tailored for use in different contexts. Several mechanisms are available for implementing variable software systems [Anastasopoulos & Gacek, 2001]. They range from code-level techniques (e.g., conditional compilation, reflective programming) to design-level flexibility paradigms (e.g., plug-in frameworks and meta-tools [Grundy *et al.*, 2006]). Svahnberg *et al.* [Svahnberg *et al.*, 2005] present a taxonomy of the technical solutions to implement variability, which can be assigned to phases of the development life cycle.

Variability-driven Development

Variability models can be used as a means of specifying a system. By documenting the different features that need to be supported and the required configurability of the system, variability models support developers in planning and implementing the system in the light of required variability. This helps in making the developed software components suitable for different context, which increases its reusability, at the same time increasing maintainability of the whole system. Just like test-driven development, where one writes test cases before coding, one can think of *variability driven development*, where one creates variability models before coding the system.

Furthermore variability models also play an important role in domain analysis. The FODA (Feature-oriented domain analysis) method [Kang *et al.*, 1990] applies the aggregation and generalization primitives to capture the commonalities of the applications in the domain in terms of abstractions. By supporting the identification, collection, organization and representation of the relevant information in a domain, FODA enables the analysis and modeling of domains. The process is based on the study of existing systems and their development histories, knowledge captured from domain experts, underlying theory, and emerging technology within the domain.

Model-driven Product Configuration

Documentation of systems architecture in the light of its variability is crucial when the system has to be extended or adapted to specific needs of different customers. Using a formal model for variability, various tedious and error prone tasks like configuration can be automated. For example, product derivation is the process of constructing products from core assets in a product line. This process can be guided by the utilization of variability models by allowing the users to take configuration decisions.

Runtime Exploitation of Variability Models

For systems, which are subject to change at runtime, it is crucial to have a smooth technical upgrade. The users must be aware of the consequences of the changes they make to the system at runtime. To enable such changes, it is important that the users are aware of available options for a change with regards to runtime adaptation needs. Variability models can help in this regard by presenting users with the decisions they need to make and hiding the technical complexity behind the decisions.

Documentation of the system only at code level is often insufficient to deal with maintenance challenges [Grubb & Takang, 2005]. Variability models can help here by helping to document the architecture. This can later guide the user while making changes to the system and making her aware of the implications of such changes.

2.5.2   Challenges and Research Issues

This chapter showed that variability research is manifold and a lot of progress has already been made in different areas. In this thesis, we focus on three aspects of variability, which are related to our research issues (see also Section 1.3.2):

- A variability modeling approach should be independent of implementation practices in different domains but be adaptable to the different variability mechanisms and artifact levels.

- There is a lack of flexible and extensible tools that can be tailored to support a particular organization's needs.

- The different dimensions of variability (Section 2.2) should be considered, when structuring the modeling space. Currently there are approaches which deal with this aspect of modeling.

> ❝ It would be possible to describe everything scientifically, but it would be without meaning if you described a Beethoven symphony as a variation of wave pressure. ❞ —*Albert Einstein*

Chapter 3

State of the Art in Variability Modeling

Summary *In this chapter, we review and analyze the existing body of scientific knowledge in the light of our research issues and objectives. We first present available variability modeling approaches at different levels of abstraction. Then we proceed with flexibility mechanisms for tool development. We conclude with an critical analysis of the modeling approaches.*

Researchers have been proposing numerous approaches supporting variability modeling on the level of features [Kang *et al.*, 1990, Czarnecki & Pietroszek, 2006] and architecture [Dashofy *et al.*, 2001, van Ommering *et al.*, 2000]. Some other approaches are orthogonal to the level of abstraction, such as OVM [Pohl *et al.*, 2005] and decision-oriented approaches [Atkinson *et al.*, 2002, Mansell & Sellier, 2004, Schmid & John, 2004]. In this chapter, we present an overview and critical discussion of these modeling approaches. We present problem space modeling approaches based on features and decisions; solution space modeling approaches using architecture description languages, and one orthogonal modeling approach.

3.1 Feature-oriented Approaches

Feature modeling is the most prominent candidate approach for modeling variability. Literature on product line engineering also shows that this is the most intensively researched method for variability modeling. Starting from FODA (Feature Oriented Domain Analysis [Kang *et al.*, 1990]), the feature-oriented view of product lines has already gone far beyond variability modeling and system documentation. Today numerous variants [Asikainen *et al.*, 2006, Czarnecki *et al.*, 2005, Czarnecki *et al.*, 2006, Czarnecki & Pietroszek, 2006] of feature-based variability modeling tools and techniques are available and several authors have purposed different formal interpretations of feature models [Batory, 2005, Schobbens *et al.*, 2007, Heymans *et al.*, 2007, Schobbens *et al.*, 2006]. In general, a feature model captures stakeholder-visible characteristics and aspects of the product line, such as functional features of individual products, software quality attributes of both the product line and the individual products to provide an overview of the systems capabilities.

To demonstrate, how feature models are used for variability modeling and how they are structured, we present a small example of a feature model of a car. A car consists of a motor, a transmission system and style. A car company `xyz` currently supports the following features. The types of available

motors are "diesel" and "electric". The customer has to choose at least one motor type. A hybrid car with both motor types is also possible. The types of available *transmission* systems are "manual" and "automatic". The customer can choose only one type of transmission system. A combination of both is not possible. The company also supports 3 *styles*– cabrio, combi and limousine. The customer can choose only one style. Upon customer wish, a car can be equipped with an air conditioner and/or a trailer. However, there are a few restrictions on the choices that can be made. For technical reasons, manual transmission is available only with an electric motor (C1) and if the customer chooses to have an air conditioner, then only limousine style cars are available (C2). Based on this description, we created a feature model as depicted in Figure 3.1.
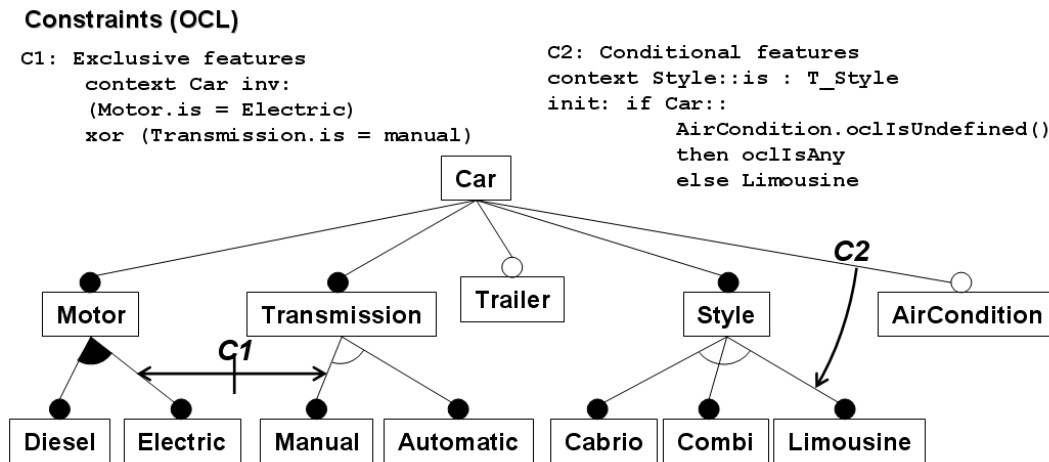
Figure 3.1: Example of a feature model of a car, for notations see Table 3.1.

3.1.1   Feature-oriented Domain Analysis

Kyo-Chul Kang and his group from the Software Engineering Institute at the Carnegie Mellon University published a technical report [Kang *et al.*, 1990] of the *Domain Analysis Project* in 1990, which describes a method for discovering and representing commonalities among related software systems. By capturing the knowledge of experts, the FODA method attempts to codify the "thinking processes" used to develop software systems in a related class or domain. FODA was initially not developed for product line variability modeling, rather for domain analysis purposes. Domain analysis supports software reuse by capturing domain expertise, which is used to support communication, training, tool development, and software specification and design [Kang *et al.*, 1990].

FODA treats features as the attributes of a system that directly affect end-users. End-users have to make decisions regarding the availability of features in the system, and they have to understand the meaning of the features in order to use the system. Documentation of a feature model includes: a

structure diagram showing a hierarchical decomposition of features indicating optional and alternative features, definition of features, and composition rules of the features. In order to specify the relationships between features, FODA makes use of a structural relationship "consists of", which represents a logical grouping of features. Alternative or optional features of each grouping are indicated in the feature model by small arcs and circles, respectively. Alternative features are regarded to be specializations of a more general category feature [Kang *et al.*, 1990]. For example, the automatic and manual transmission features (cf. Figure 3.1) are specializations of the general "transmission" feature. The term "alternative features" is used (rather than "specialization features") to indicate that no more than one specialization can be made for a system.

In addition to the relationships specified in the feature diagram, there exist composition rules defining more complex semantics of relationships between features. All optional and alternative features that cannot be selected when the named feature is selected are stated using the "mutually exclusive with" statement. All optional and alternative features that must be selected when the named feature is selected are defined using the "requires" statement.

3.1.2   Cardinality-based Feature Modeling

Cardinality-based feature modeling was introduced by Czarnecki *et al.* in 2005 and integrates a number of extensions to the original FODA notation (cf. Table 3.1). Constraints on features are expressed using cardinalities, as in Entity-Relation modeling or UML modeling [Chen, 1976]. A cardinality-based feature model [Czarnecki & Kim, 2005] is a hierarchy of features, where each feature has a feature cardinality. The authors argue that the use of *cardinalities* are a means of expressing additional constraints for feature modeling, and can help by providing constraint-satisfaction facilities for feature modeling and feature-based configuration tools.
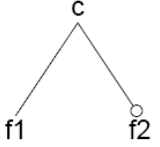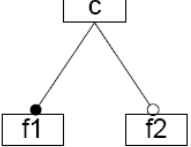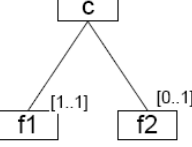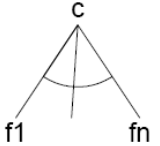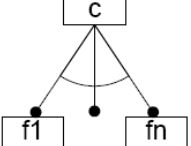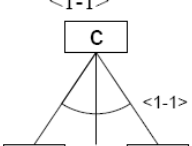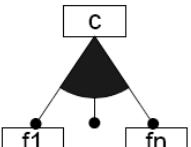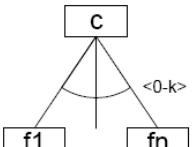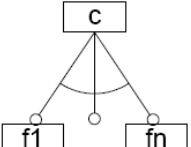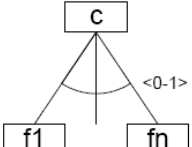
In cardinality-based feature modeling, a feature diagram is translated to a UML class diagram. In the class diagram, entities correspond to features and a multiplicity at the aggregate end of every composition is 1, while the multiplicity at the other end is the same as the corresponding cardinality in the feature diagram. The formalism of cardinality-based feature modeling allows for at most one attribute per feature and the type of an attribute is either a primitive type or a reference to another feature [Janota & Kiniry, 2007]. Riebisch *et al.* claim that cardinalities are already partially represented in previous notations. Moreover, they argue that "combinations of `mandatory` and `optional` features with `alternatives`, `or` and `xor` relations could lead to ambiguities [Riebisch *et al.*, 2002].

Czarnecki *et al.* [Czarnecki & Kim, 2005] describe the use of Object Constraint Language (OCL[1]) in the context of feature models to express constraints that are not expressible by the diagram. The authors argue that OCL is adequate for expressing such constraints and support their claim with a number of sample constraints. They also identify a set of facilities based on constraint satisfaction that can be provided by feature modeling and feature-based configuration tools. The authors also purpose the verification of feature-based model templates [Czarnecki & Pietroszek, 2006] against OCL

---

[1]OMG. UML 2.0 OCL Specification, 2003. `http://www.omg.org/docs/ptc/03-10-14.pdf`.

Table 3.1: Graphical representation of different notations used in feature modeling.

well-formedness constraints.

3.1.3   Other Approaches based on Features

The FeatuRSEB method [Griss *et al.*, 1998] is a combination of FODA and the RSEB (Reuse-Driven Software Engineering Business) method. RSEB is based on Jacobson's OO Software Engineering [Jacobson, 1994b] and OO Business Engineering [Jacobson, 1994a], applied to an organization engaged in building sets of related applications from sets of reusable components. FeatuRSEB uses UML extensions to model and specify application systems, reusable component systems and layered architectures, and to express system variability in terms of variation points and attached variants.

Later, Van Gurp *et al.* [van Gurp *et al.*, 2001] extended FeatuRSEB to deal with binding times, indicating when features can be selected, and external features, which are technical possibilities offered by the target platform of the system. The PLUSS approach [Eriksson *et al.*, 2005b, Eriksson *et al.*, 2005a] is based on FeatuRSEB and combines feature diagram and use case diagrams to depict the high-level view of a product family.

Kang *et al.* presented FORM (feature-oriented reuse method [Kang *et al.*, 1998]) in 1998, where the focus lies on the systematic discovery and exploitation of commonality across related software systems. FORM extends FODA to the software design phase and prescribes how feature models can be used to develop domain architectures and components for reuse. The underlying philosophy for this extension is that the features of a domain characterize each variant product in the domain, and the code that implements the characterizing features should be packaged, managed, and reused as software modules [Kang *et al.*, 1998]. Cechticky *et al.* [Cechticky *et al.*, 2004] present an XML-based feature modeling technique.

Some researchers have also presented the idea of feature templates. A feature template consists of a feature model and an annotated model expressed in some general modeling language such as UML or a domain-specific modeling language [Czarnecki & Antkiewicz, 2005]. Based on a particular configuration of features, a template processor creates an instance of the template by evaluating the presence conditions in the model and removing elements whose presence conditions evaluate to false. A key strength of model templates is that they allow us to maintain several model variants, such as variants of business or design models for different product-line members, in a superimposed form within a single artifact [Czarnecki & Pietroszek, 2006]. Furthermore, the authors argue that the model annotations establish the traceability between features and their realizations in the model.

3.1.4 Formal Semantics of Feature Models

There are several variants of feature modeling languages whose precise semantics has not yet been defined. The imprecision and ambiguity have therefore lead to confusions and non-standard interpretations of the modeling constructs. Schobbens *et al.* have investigated the generic semantics of feature models and argue that different variants of feature modeling/diagramming techniques mainly concern concrete syntax and have no influence on feature combinations [Schobbens *et al.*, 2007]. By generalizing the syntax of different feature diagramming techniques, the authors present a parametric construction that generalizes the syntax.

Recently several publications have dealt with a comparative study of the different variants of feature modeling techniques [Asikainen *et al.*, 2006, Heymans *et al.*, 2007, Schobbens *et al.*, 2006]. There have also been several attempts to formalize the correspondence between feature diagrams, grammars [Czarnecki *et al.*, 2005], and propositional formulas [Batory, 2005]. Some of the basic problems with the available formalizations are that the propositional formulas are not well-suited for a metamodel that enables cloning of features, especially in the work of Czarnecki et al [Czarnecki & Kim, 2005] and on the other hand grammar-based approaches are not suitable for capturing cross-cutting dependencies,

such as excludes and requires. Most importantly, however, it is not clear how dependencies between attributes should be modeled [Schobbens *et al.*, 2006].

Czarnecki *et al.* [Czarnecki *et al.*, 2006] present the idea that feature models are views on ontologies. By describing the parallels between feature models and ontologies (with respect to their role of being conceptual models), the authors argue that feature models form a notational subset of ontologies, which is why feature models are more likely to describe concepts more specialized than those of ontologies. This idea has been further refined by Wang *et al.* [Wang *et al.*, 2007] where the authors purpose the verification of feature models using the W3C Web Ontology Language (OWL)[2].

Different authors have focused on transforming feature models into different mathematical constructs for automated verification [Batory *et al.*, 2006] and model checking purposes. For example, Benavides *et al.* [Benavides *et al.*, 2005] propose modeling a feature model as a CSP (Constraint Satisfaction Problem) in order to automatically answer questions regarding the number of potential products, results of applying filters to a model, detecting dead features, etc.

Harry *et al.* [Harry *et al.*, 2005] have presented a model of interfaces that supports automated, compositional, feature-oriented model checking. Their approach supports modular verification of features using a model checking process consisting of three steps: Firstly, the CTL (Computation Tree Logic) property of individual features is proved, then constraints are automatically derived which should be preserved at composition time. In the third step the preservation constraints are tested on all features and product combinations. Their approach helps in detecting feature interactions compositionally.

3.2   Decision-oriented Approaches

The idea of decision modeling in product lines was introduced as a part of the Synthesis Project by Campbell *et al.* [Consortium, 1991, Campbell *et al.*, 1990] in the early 1990s, where decisions were "*actions which can be taken by application engineers to resolve the variations for a work product of a system in the domain*" [Campbell *et al.*, 1990]. Many other researchers like Forster *et al.* [Forster *et al.*, 2008], Schmid *et al.* [Schmid & John, 2004], Mansell *et al.* [Mansell & Sellier, 2004] have also been actively publishing their research results in this area. Surprisingly, researchers have not yet found a common definition for decisions. Some researchers follow decision modeling on a rather informal basis (e.g., using tables [Schmid & John, 2004]), while others (e.g., [Dhungana *et al.*, 2007c]) have already automated the decision making procedure by using executable descriptions and formal approaches.

3.2.1   Synthesis

A decision model defines the set of requirements and engineering decisions that an application engineer must resolve to describe and construct a deliverable application engineering work product [Consortium, 1991]. Decisions are used as elaborations to the domain's variability assumptions

---

[2]. http://www.w3.org/TR/owl-features.

Table 3.2: Example of a decision model in Synthesis [Consortium, 1991].

```
Traffic_Light_Controller: composed of
      Schedule: one of (Fixed_Schedule, Programmable)    {designates the type of traffic light sequence scheduling
                                                              the TLC system must accommodate}
      Geometry: one of                                   {intersection geometry}
            X: list length 4 of Street                   {street characteristics for an X intersection}
            T: list length 3 of Street                   {street characteristics for a T intersection}

Fixed_Schedule: composed of
      Start_Time: (0:00 .. 23:59)                        {start time for this traffic light sequence schedule}
      Stop_Time: (0:00 .. 23:59)                         {stop time for this traffic light sequence schedule}
      ....

Street: composed of
      Name: identifier                                   { street name}
      Right_Turn_Lanes: Lane_Group                       {characteristics for the right-hand turn lanes}
      Left_Turn_Lanes: Lane_Group                        {characteristics for the left-hand turn lanes}
      Through_Lanes: Lane_Group                          {characteristics for the through lanes}
      Pedestrian_Crosswalk: one of (Xwalk, NO_Xwalk)     {designates the presence of a pedestrian crosswalk
                                                              for this street}
      Crosswalk_Button: one of (CB, NO_CB)               {designates the presence of a pedestrian crosswalk
                                                         pushbutton for this street}
      ....

Lane_Group: composed of
      Number_of_Lanes: numeric(1..2)                     {number of traffic lanes in this Lane_Group}
      Sensor: one of (Sensor, NO_Sensor)                 {indicates whether there is a traffic monitoring
                                                              device for each lane in this Lane_Group}
      ....

Project_Information: composed of
      Name: identifier                                   {name for the TLC system}
      Mnemonic: identifier                               {TLC system mnemonic}
      ....
— — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — —
Constraints
      —   The number of through lanes for Street(1) must be the same as for Street(3).
      —   There can be at most 4 different schedules in the Fixed_Schedule.
      —   A Through_Lanes group must be specified for each Street.
      —   ....
```

and are defined in abstract form of the application modeling notation [Campbell *et al.*, 1990]. To construct a product, these decisions must be sufficient to distinguish the product from all other members of the family.

A decision model can either be represented as a list of questions or in a tabular format and consists of three parts: *Decision specification* (questions, descriptions, set of valid answers) suffice to distinguish among systems in the domain. *Decision groups* are used to structure decision specifications in logical groups. *Decision constraints* are rules that restrict the resolution of interdependent decisions. Each

decision is phrased as a question and a non-empty set of valid answers. If a set of related decisions is always resolved an a unit, it is also possible to define the set to be a composite decision. Decision constraints may be either structural or dependency [Consortium, 1991]. A *structural constraint* limits the number of instances of a decision group. A *dependency constraint* specifies how decisions made by an application engineer affect subsequent decisions.

Figure 3.2 depicts a decision-model used in the synthesis handbook, which portrays decision groups (e.g., Street, Lane_Group) and their corresponding decisions, along with appropriate constraints. Decision models are rather used as informal descriptions of the options available for identifying different products in the domain. They are written in plain text and there is no tool support available to create, manage, model-check or execute the decision models. For example, the constraints in Table 3.2 are written in plain English and therefore can be ambiguous.

3.2.2   PuLSE

PuLSE (Product Line Software Engineering) is a method which enables the conception and deployment of software product lines within a large variety of enterprise contexts [Bayer *et al.*, 1999]. This complex task is decomposed into three subtasks, each of them performed by a technical component:

PuLSE-Eco helps to determine an economic viable scope for the product line. A core idea of PuLSE-Eco is to explicitly base the definition of the product line scope on business objectives that are identified by product line stakeholders.

PuLSE-CDA is used to elicit and articulate product line concepts and their interrelationships. To derive product line member specifications from a product line model, a decision model is created that contains a structured set of decisions. Each decision corresponds to a variability in a workproduct together with the set of possible resolutions. To build the specification of a product line member, the decisions are resolved.

PuLSE-DSSA is applied to define a software reference architecture for the product line. During the creation of the reference architecture, implementation-specific decisions are collected that will have to be resolved during reference instantiation. These decisions and their possible resolutions are captured in the configuration model that extends the decision model.

The basic ideas of Decision Modeling in PuLSE have been further extended by Schmid *et al.* in [Schmid & John, 2004], where the notion of *decision variables* is used which are then referenced at the specific variation points using the decision evaluation primitives. Each of the decision variables that is defined in the decision model (cf. Table 3.3) is in turn described by the following information:

Name is an unique identifier and represents the name of the decision variable.

Relevancy of a decision variable specifies whether the information carried by the variable is important during product derivation (instantiation), e.g., the decision variable describing the memory size is only valid if the decision variable describing the existence of memory is true.

Table 3.3: Example of a decision model presented by Schmid and John [Schmid & John, 2004].

Name	Relevance	Description	Range	Selection	Constraints	Binding Times
Memory	System_Mem = True	Does the sytem have memory?	TRUE, FALSE	1		Compile Time
Memory_Size		The amount of memory the system has (KB)	0..100.000	1	Memory=TRUE => Memory_Size > 0	Installation, System Initialisation
Time_Measurement		How is time measurement done?	Hardware, Software	1		Compile Time

Description is a block of text, elaborating the meaning of the variable.

Range is the set of values, which can be assigned to the decision variable. This can be basically any of the typical data types used in programming languages. The most common type is the enumeration, as the relevant values are often domain-dependent.

Cardinality is a selection criterion, defining how many of the values of a decision variable can be assumed by it.

Constraints are used to describe interrelations among different decision variables. The authors use constraints to describe the requires relationship, as this can be treated as a special case in their framework.

Binding times specify when the decision can be bound. This can be source time, compile time, installation time, etc. Additional binding times may exist, and can be product line specific.

3.2.3   KobrA

KobrA (Komponentenbasierte Anwendungsentwicklung) is a method for modeling architectures developed by Fraunhofer IESE. KobrA can be viewed as a "ready-to- use" customization of PuLSE. The KobrA method [Atkinson *et al.*, 2000] supports a model-driven UML-based representation of components and a product line approach to their development and evolution. This enables the benefits of component-based development to be realized throughout the software life-cycle and allows the reusability of components to be significantly enhanced [Atkinson *et al.*, 2002].

Decision models are used to establish the relationships between "user visible options" and the "system features". KobrA differentiates between *simple decisions* and *high-level decisions*. Simple decisions are directly related to the variation points within assets and high-level decisions can possibly affect the answer set of other decisions. In KobrA decisions are used to capture the following information (cf. Table 3.4):

1. A textual, domain-related question that represents the decision to be made.

Table 3.4: Example of a decision model in KobrA approach, Atkinsion *et al.* [Atkinson *et al.*, 2002].

Id	Customer Interaction	Resolution	Diagram	Effect
1	Does the loan of items cost anything?	y	Activity Diagram	Activity „Loan Cost Determination" is present
			„Item Check In"	
		n	Activity Diagram	Activities „Loan Cost Determination" and „Loan Cost Collection" are not present .
			„Item Check In"	
2	What is the type of payment?	per item	Activity Diagram	Activity „Loan Cost Collection" is present
			„Item Check In"	
		membership fee	Activity Diagram	Activity „Loan Cost Collection" is not present .
			„Item Check In"	

2. The set of possible answers to that question.

3. References to the affected assets and variation points (simple decisions) and references to the affected decisions (high-level decisions).

4. The description of the effect on the assets (simple decisions) and to the affected decisions(high-level decisions).

3.2.4   ESI- VManage

Mansell and Sellier present an XML-based decision modeling process in [Mansell & Sellier, 2004], where a decision model represents all possible user requirements defined during the domain analysis and the set of rules and constraints associated with them. Just like in Synthesis [Consortium, 1991], the authors define a decision model to be ". . . a document defining the decisions that must be made to specify a member of a domain". Each decision presented within the decision model is defined with a set containing the following information [Mansell & Sellier, 2004]:

1. A name identifying the decision with a unique identifier.

2. A description giving the necessary information to support the decision making.

3. The type expressing the possible values supported by the decision.

4. The default value indicating the value automatically affecting the decision if the decision is not taken explicitly.

5. The validity indicating the criteria to make or not to make the decision.

6. The dependencies which explicitly allow the specification of the relationships between different decisions.

The authors explicitly differentiate between *restricted* and *unrestricted* decisions. Unrestricted decisions do not support constraints other than their data type restrictions. Restricted decisions have other restriction specifications making their specification and their implementation within the XML documents more complex. One of the unique features about this approach (compared to our approach) is the concept of *collection of decisions*, which are instances of a decision or a set of decisions. For example, when two instances of a certain component are required, the process of taking decisions to configure these components has to be repeated for each required component, i.e., decisions need to be duplicated. The number of instances can be specified as a collection restriction. Two types of relationships between decisions are supported: (i) The value of a decision can impact on the range or value of another decision, and (ii) The value of a decision can impact whether another decision should be made.

3.3   Architecture-level Variability

Architecture description languages (ADLs) are formal notations for describing software systems. ADLs lie at the conceptual intersection between requirement, programming and modeling languages but they are distinct from all three. In contrast to requirements modeling languages, which model the problem space, ADLs model the solution space. General purpose modeling languages usually focus on modeling the internal structure of components whereas ADLs usually describe the interplay of components. Compared to programming languages, where machine code is generated for a specific platform or technology, ADLs are designed to be independent of technological platforms.

A number of ADLs have been proposed for modeling architectures, both within a particular domain and as general-purpose architecture modeling languages, e.g., Darwin [Magee & Kramer, 1995], Aesop [Garlan *et al.*, 1994], Koala [van Ommering *et al.*, 2000], xADL [Dashofy *et al.*, 2001], etc. There are so many existing languages, that it is not always clear, which of several possible ADLs is best suited for a particular problem. A systematic survey [Medvidovic & Taylor, 2000] of architecture description languages reveals that most ADLs share a set of fundamental modeling constructs and concepts, including components, connectors, interfaces, and architectural configurations.

In this thesis, we look at ADLs in the light of their variability modeling capabilities and extensibility. Most of the ADLs available today are monolithic. Their feature sets and grammar are fixed, and adding new constructs to a monolithic ADL is not possible without modifications to the tool set supporting that ADL [Dashofy *et al.*, 2002]. Among several available ADLS, we discuss the capabilities of two

representative ADLs- Koala [van Ommering *et al.*, 2000] and xADL [Dashofy *et al.*, 2001]. Koala was chosen because of its ability to model product line architectures with optional and variant elements. xADL is a good example of an extensible modeling language, that can be adapted/extended to support domain specific needs. An xADL extension to model variability is also already available.

3.3.1   xADL 2.0

xADL 2.0 is a software architecture description language (ADL) developed by the University of California[3], Irvine for modeling the architecture of software systems. Unlike many other ADLs, xADL 2.0 is defined as a set of XML schemas. This gives xADL 2.0 unprecedented extensibility and flexibility, as well as basic support from the many available commercial XML tools. The xADL 2.0 language itself is not bound to any particular architectural style, tool set, or methodology. xADL 2.0 and the complementary tools [Dashofy *et al.*, 2007] like Apigen and the xArch. The data binding library can be used by themselves, independent of any particular development environment or domain. The current set of xADL 2.0 schemas includes modeling support for: *(i)* run-time and design-time elements of a system, *(ii)* support for architectural types, *(iii)* architecture diffing and *(iv)* product line architectures.
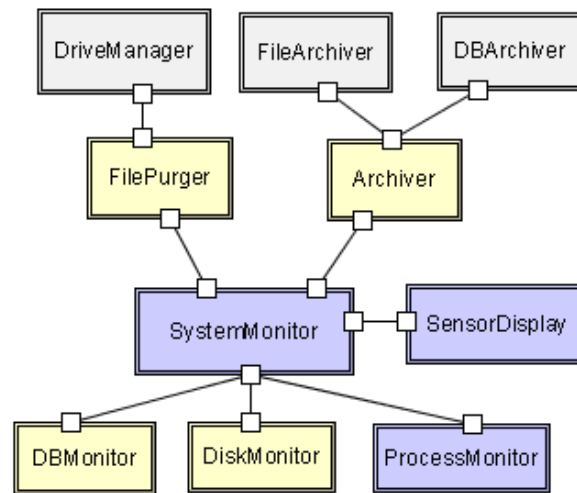
Figure 3.2: Example of a xADL architecture model.

xADL defines basic architectural elements such as components, connectors, links, or interfaces to model the architecture. Via extensions xADL supports modeling components or connectors of the architecture as optional or variant. In xADL options indicate points of variation in an architecture where the structure may vary by the inclusion or exclusion of an element or a group of elements. Variants indicate points in an architecture where one of several alternatives may be substituted for an

---

[3]http://www.isr.uci.edu/projects/xarchuci/

element or a group of elements. Relationships and interdependencies between different architectural elements are modeled using guards which are boolean expressions. These guards consist of variables, whose values determine whether a certain optional component will be included in the final system or not, based on decisions in product configuration. Guards also monitor which variant of a variable element is to be included in the final product.

Table 3.5: Examples of some typical constraints for modeling architecture variability of xADL model depicted in Figure 3.2.

#	Description
C1	FilePurger, Archiver, DBMonitor and DiskMonitor are optional.
C2	You can only choose one type of archiver.
C3	FilePurger can be chosen only if DiskMonitor exists.
C4	Without DBArchiver, DBMonitor makes no sense.
C5	Without FileArchiver, DiskMonitor makes no sense.
C6	If DBArchiver is chosen, you must also choose DBMonitor.
C7	If FileArchiver is chosen, DiskMonitor is also needed.

Table 3.6: Formal representation of constraints listed in Table 3.5., showing the boolean guards which need to be modeled for each component to express the constraints in plain text.

#	File Purger	File Archiver	DB Archiver	DB Monitor	Disk Monitor
C1	FP	FA	DBA	DBM	DM
C2	FP	FA∧¬DBA	DBA∧¬FA	DBM	DM
C3	FP∧DM	FA∧¬DBA	DBA∧¬FA	DBM	DM
C4	FP∧DM	FA∧¬DBA	DBA∧¬FA	(DBA∧¬FA)∧DBM	DM
C5	FP∧DM	FA∧¬DBA	DBA∧¬FA	(DBA∧¬FA)∧DBM	(FA∧¬DBA)∧DM
C6	FP∧DM	FA∧¬DBA	DBA∧¬FA	¬(DBA∧¬FA) ∨((DBA∧¬FA)∧DBM)	(FA∧¬DBA)∧DM
C7	FP∧DM	FA∧¬DBA	DBA∧¬FA	¬(DBA∧¬FA) ∨((DBA∧¬FA)∧DBM)	¬(FA∧¬DBA) ∨((FA∧¬DBA)∧DM)

Here, we present a small example of a subsystem, which we modeled using xADL (Figure 3.2). Table 3.6 depicts the complexity of formalizing constraints using the xADL's guard language. The boolean expressions to be used as guards of the components get more complex as the number of constraints increases. We used the tool ArchStudio [Dashofy *et al.*, 2007] to create the model. In order to capture the variability, we performed three steps:

Architecture modeling: xADL models describe the architecture in terms of sub-architectures, components, connectors, interfaces, and links.

Capturing variability: xADL supports variability modeling through options, variants, and versions. We identified optional and variant architectural elements by analyzing the technical solution. The relationships between the modelled components and connectors are listed in Table 3.5

Modeling variability dependencies In xADL constraints are modeled using a boolean guard language. Such guards specify a condition under which, an architectural element will be included in a derived product. In this modeling step, we translated the variability constraints from previous step into formal xADL guards is shown in Table 3.6.

3.3.2   Koala

Koala is a mix of a component model and architectural description language in building product families in the consumer electronics domain [van Ommering *et al.*, 2000]. It was initially designed by Philips to model embedded software in television sets. Koala's component model is inspired (and implemented) by MicrosoftCOM[4]. It has Darwin-like components [Allen & Garlan, 1997] with provided and required interfaces. The binding of components is specified by compound components. There is a strict separation between component and configuration development, i.e., component builders make no assumptions about configurations and the configuration designers do not change components. Koala, however, does not take into account non-functional requirements such as timing and memory consumption and lacks a formal execution model.

Koala models are built using the following modeling constructs:

Components are units of design, development and reuse. They communicate with each other through interfaces. Koala provides a graphical notation, where components look like IC chips and configurations look like electronic circuits.

Interfaces are units of specification and binding. They can be compared with Java or .Net interfaces. A component may (and usually does) provide more than one interface. Components access functions in their environment through explicit requires interfaces. Once made available for use, interface definitions are immutable but can be superseded by extended interfaces.

Switches are used to enable function binding with conditional expressions to route function calls to appropriate components. Switches are particularly important in order to handle structural diversity in the connections between components. With partial evaluation these connections can even be turned into normal function calls.

Diversity Interfaces are interfaces that denote which properties are required by a component. Such interfaces play an important role in Koala models as reusable components should not contain configuration-specific information. But moving all configuration-specific code out of the component results in almost empty components, which are not very useful. Therefore non-trivial

---

[4]http://www.microsoft.com/COM

reusable components are parameterized over all configuration-specific information through diversity interfaces.
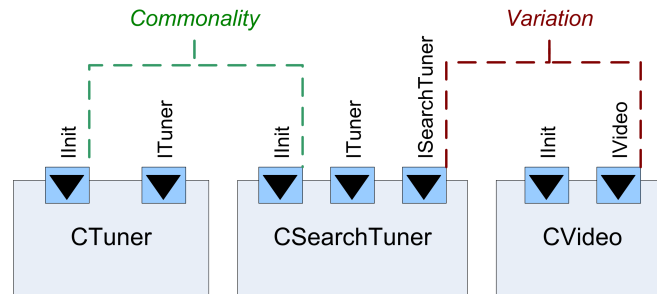
Figure 3.3: Example of a Koala Model showing commonality and variability among components in a repository.

As depicted in Figure 3.3, Koala deals with variability of components on the basis of their components. There may be multiple components that offer the same interface, but have different non-functional characteristics; e.g., different tuners, micro-controllers, runtime kernels, etc.

3.4   Orthogonal Variability Modeling

An orthogonal variability model (OVM) provides a cross-sectional view of the variability across all software development artifacts [Pohl *et al.*, 2005]. It defines the variability by relating different artifacts, e.g., software development models such as feature models, use case models, design models, component models, and test models to each other. The basic elements of an orthogonal variability model are *variation points*, *variants*, and *variability dependencies*. There are two types of "variation points": an "internal variation point" has associated variants that are only visible to developers but not to customers and an "external variation point" has associated variants that are visible to developers and customers.

The core concepts of the OVM language are variation points ("what varies") and variants ("where does it vary"). Each variation point has to offer at least one variant (offers-association). Additionally, the constrains-associations between these elements describe dependencies between variable elements.

OVM makes use of graphical notation(cf. Figure 3.4), which was first proposed by Halmans *et al.* in [Halmans & Pohl, 2003]. Variation points are represented as triangles. A variation point is associated with one, or more than one use case using the "include" relationship. The variants are associated with the variation points using the different relationships as depicted in Figure 3.4.

The OVM approach can be used to document the variability of arbitrary development artifacts, ranging from textual requirements to design models and even test cases. Also, the documentation of variability is separated from the technical realization of variability in the development artifacts and
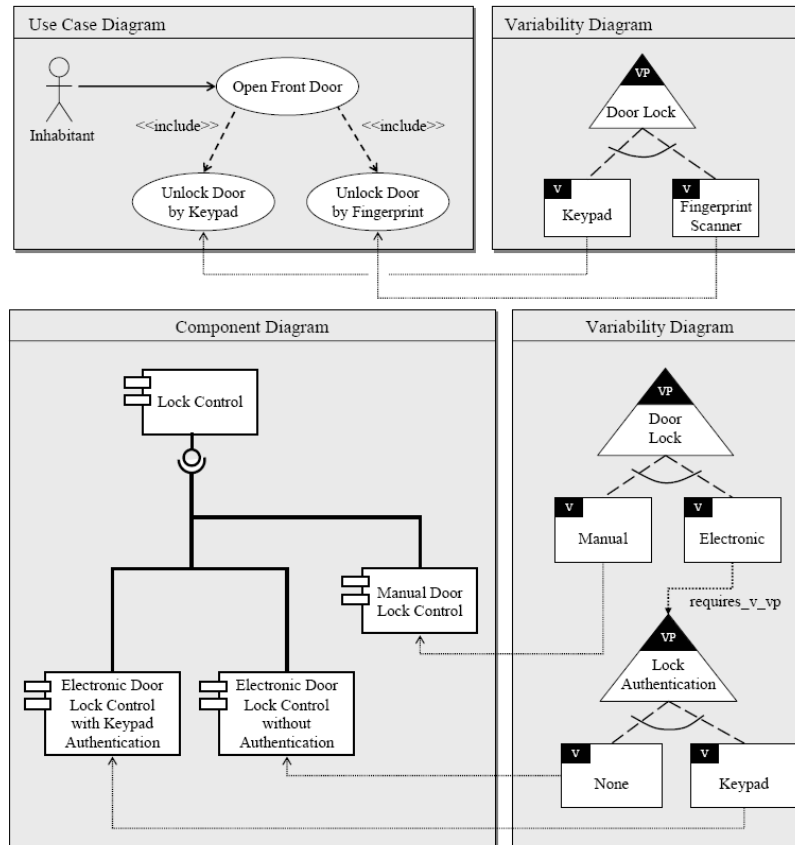
Figure 3.4: Examples of OVM models, depicting assets at different levels of abstraction, thereby demonstrating the orthogonality of the approach. *Figure source [Pohl et al., 2005]*.

as such provides a further level of abstraction [Pohl *et al.*, 2005] that aids developers in managing complexity. OVM also results in a significant reduction of model size and complexity can be achieved, because only the variable aspects of a product line are documented in the OVM models.

3.5 Critical Analysis of Modeling Approaches

3.5.1 Feature Modeling

Feature modeling proved to be a very suitable approach for getting an overview of the systems functionality. Provided that the reader is familiar with the notations, it was relatively easy to figure out the different combinations of the product which is described using the model. However, we had several

difficulties while creating feature models for large real-world industrial systems.

What is a feature: This was the first and the most important question, which we asked ourselves before creating the feature model. As the term *feature* is not uniquely defined in the literature, it is difficult to define the individual features when creating a feature model. In our example in Figure 3.1, the feature "Style" is just an abstract concept, not something the customer can buy.

How to "build the tree": Most feature modeling approaches use feature trees as the representation of the feature model. We had difficulties in building the tree, as our experience showed that a feature model is usually not a tree, but rather a graph-like structure. In our example in Figure 3.1, the tree structure is spoiled by the constraints C1 and C2.

Sub-feature vs. Constraints: Very often, we came across situations, where it was not clear whether a sub-feature or a constraint is a more suitable construct to model dependencies. It is possible to eliminate constraints by duplicating the features [Broek *et al.*, 2008], but then the tree becomes very large very soon. In our example in Figure 3.1, the constraint C2 can be removed by arranging the feature "Limousine" under "AirCondition".

Traceability to the solution space: Most of the feature modeling approaches do not provide explicit support to model the solution space. This aspect is either completely ignored or supported only through fine granular features. As both the problem space and the solution space are described using features, it is difficult to visualize the traceability between the two.

Dealing with real-world complexity: Constraint specification languages such as OCL provide a good fit to describe the dependency between the features in small models. However in real-world software systems, feature models are pretty large and the models can hardly be created or managed by anybody. In our example in Figure 3.1, the constraints C1 and C2 are specified in OCL, which demonstrates how the feature model complexity regarding maintenance efforts rise with the increase in the number of constraints.

Selection/No-Selection: Feature models do not allow us to distinguish between a feature that has not yet been decided by a user and a feature which the user decided not to include in the derived product. In both cases the feature is marked as not selected. It can be necessary, however, to model the dependencies with respect to the fact if a decision has been taken, regardless of its value.

Hidden features: Not all decisions needed for deriving a product will be taken by a user directly. Some can be inferred from other decisions that already have been taken. In feature modeling approaches it is not possible to define hidden features which capture important variability information without being visible externally in application engineering. In our example in Figure 3.1, one can infer the minimum motor power by considering the selection of all other features.

3.5.2   Decision Modeling

Most of the drawbacks of feature modeling approaches can be compensated by decision modeling approaches. However, existing decision-oriented variability modeling approaches are not ideal for all cases either.

Informal description: Existing approaches consider decision models to be rather informal way of capturing variability. Most of researchers use simple natural language based tables to specify decisions and dependencies among them. This inhibits the usage of such models in automated processes.

Consistency checking: Specifications based on natural languages have two basic problems. They can be ambiguous, as the interpretation of the specification is solely a concern of the reader. They can also be inconsistent, the user may specify contradicting model elements without being warned.

Model utilization: Having a variability model alone is not enough. These models have to be used, such that the information contained in them guides the process of generating, configuring or developing the required software solution. Existing approaches (because of the lack of formal descriptions) do not provide automated tools for model utilization.

Traceability to software artifacts: Existing approaches either do not consider the traceability of decisions to software artifacts at all or are not generic enough to deal with the diversity of implementation practices in different domains.

3.5.3   Architecture Modeling

Modeling architectural variability is not supported by most ADLs. We experimented with variability modeling capabilities of xADL2.0 and were clobbered over the head by the complexity of the approach.

Dependencies among problem space concepts: With xADL 2.0 it is not possible to model the dependencies among the different customization options, i.e., the available options are represented in a flat list. The decisions that can be taken during product derivation are completely independent of each other, which does not reflect the reality in real-world systems.

Complexity of the guard language: Modeling dependencies between architectural elements using guards in xADL is a good start, but a higher level of abstraction is necessary to properly express constraints and to deal with the complexity. The examples in Table 3.5 and Table 3.6 show that translating fairly simple constraints into guards for one small sub-system already turns out to be quite challenging. Guards get more complicated as the number of constraints increases. In case of changes to the architecture, many guards are potentially affected and need to be updated accordingly. In real-world examples, a significant number of constraints need to be modeled, which makes xADL hard to use in this respect.

Part II

Approach

A Decision-oriented Approach for Domain-specific Variability Modeling

Summary *In this chapter we argue that decisions should be treated as first-class citizens in modeling variability. We describe the process of creating domain-specific variability modeling languages, which are based on decisions and can be easily adapted to problem settings in different application domains. We further describe the formal semantics of decision-oriented variability models in detail and present examples.*

The cost of extending and adapting general purpose languages and tools for variability modeling is significantly high, and may be comparable to developing a new language from scratch. For this reason, different application domains tend to create their own domain-specific languages and tools for variability modeling, i.e., they are *re-inventing the wheel*. The disadvantage of such practices is obvious: researchers and practitioners need to solve the basic modeling problem from scratch over and over again. This leads to unnecessary duplication of efforts. Here we therefore propose a solution addressing this problem. We envision a variability modeling language, that is concrete enough (so that the effort of creating a new language is minimized) as well as flexible enough (to deal with diverse implementation practices in different domains). Our approach is based on the simple assumption that despite the many different implementation practices in different domains, the basic problem when describing variability involves modeling the *problem space* (i.e., stakeholder needs or desired features), the *solution space* (i.e., the architecture and the components of the technical solution) and traceability between the two.

Modeling the solution space requires capabilities to capture the variability of diverse reusable assets such as architecture, code, test cases, processes, documents, and models. Managing variations at different levels of abstraction and across all generic development artifacts is a daunting task, especially when the systems supporting various products are very large, as is common in industrial settings [Berg *et al.*, 2005]. A language for modeling the solution space should be flexible and adaptable to implementation practices in different domains. It should be independent of the implementation practices of a particular domain.

Modeling the problem space requires an *"apparatus"* which allows the specification of objectives. For example, decisions that can be taken by different stakeholders during product derivation in PLE are essential for understanding the problem space. The objectives should be clearly defined, and

stated in a manner that will allow automated assessment and processing for appropriate outcomes (e.g., final product in a product derivation process). Variability modeling in the problem space therefore is about modeling the available set of choices and the relationships among these.

4.1   Approach

We propose a decision-oriented variability modeling language (DoVML), which supports the modeling of the problem space using *decisions* and the solution space using *assets*. Figure 4.1 depicts a high-level meta-model of a modeling language, where the key modeling elements are *decisions* and *assets*. Decisions and assets are linked together by *inclusion conditions*.
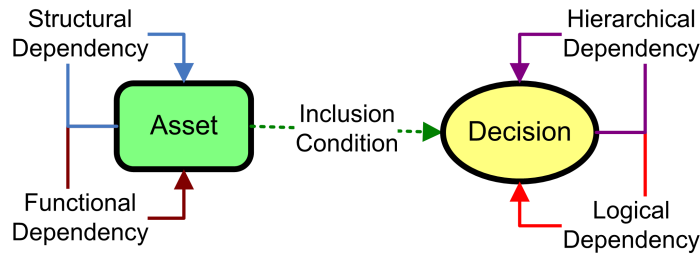
Figure 4.1: Core meta-model depicting the key modeling elements.

The core meta-model depicted in Figure 4.1 is generic and adaptable to address domain-specific concepts. These are introduced in the form of new asset types together with their attributes and relationships between them (cf. Figure 4.2). This configurability of the language allows us to adapt the approach to the needs of different variability implementation practices. Figure 4.3 depicts a simple variability model consisting of decisions and assets as the key modeling elements.

4.1.1   The Notion of a Decision

Decision taking is a process of judging the merits of multiple options and selecting one of them for action. The outcome of a decision making process leads to the selection of a course of actions among several available alternatives. In other terms, a decision is a set of choices available at a certain point in time. Variability is also about alternatives and choices. For example, the selection of a certain product from a product line is done by taking a set of configuration decisions. As we are used to thinking in terms of decisions in real life situations, it is quite intuitive to think of variability in terms of decisions as well.

A *decision* arises whenever for a given goal there exist two or more ways of achieving it. Decisions can be used to represent the variation points in a product line model. The process of taking a decision involves judging the merits of multiple options and selecting one of them for action (e.g., based on a
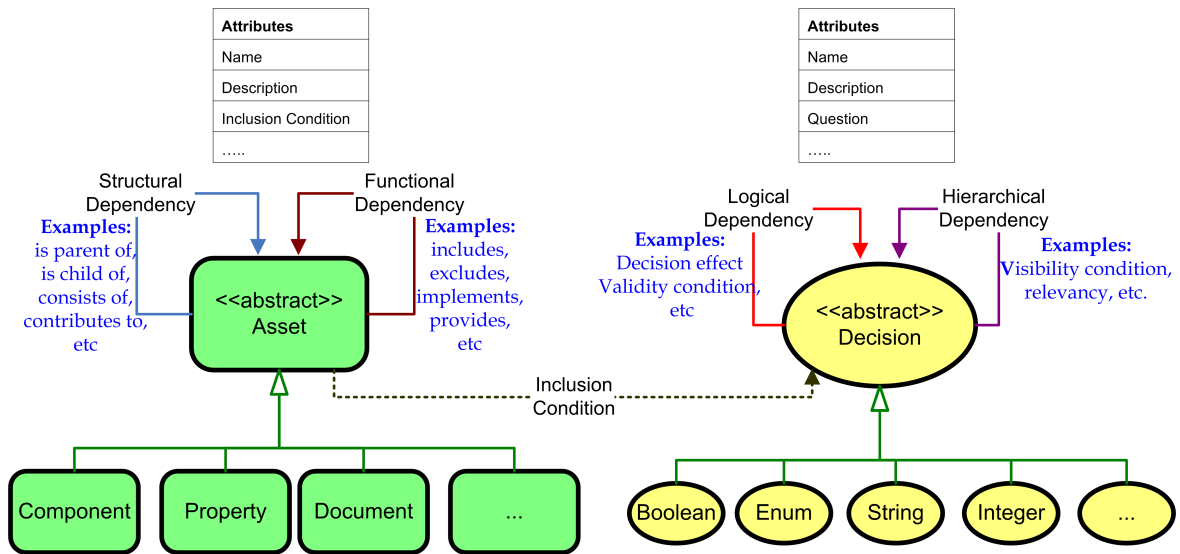
Figure 4.2: Example of domain-specific refinements of the core meta-model and adaptation of the modeling language.

consideration of customer requirements). In other terms, taking decisions leads to the selection of a course of actions among several available alternatives. Decisions are not independent of each other and cannot be made in isolation. Due to the dependencies surrounding a given decision, decisions made earlier can lead to new decisions. Many decisions are limited (constrained) depending on the context of already taken decisions. In our modeling approach, we take care of two kinds of dependencies among decisions: firstly, not all decisions are equally important or relevant at a certain time. We therefore need constructs to model the hierarchy of decisions. Secondly, taking a certain decision may have implications on other decisions which also need to be considered. We therefore need to take care of factors that influence the process of taking decisions itself. The core meta-model (Figure 4.1) shows *hierarchical dependencies* specifying how the decisions are organized and *logical dependencies* specifying the relationship between the decision-values.

Hierarchical dependencies are used to specify when a particular decision is visible to the user. A hierarchical arrangement of decisions adds context to the decisions. For example, it would make no sense to ask the user about the capacity of a database system, if she does not intend to use a database.

Logical dependencies are actions that need to be executed after a decision has been taken. Typically, these are rules that need to be checked before and after a decision is taken. For example, the type of the database to be used can be logically induced from the system size.
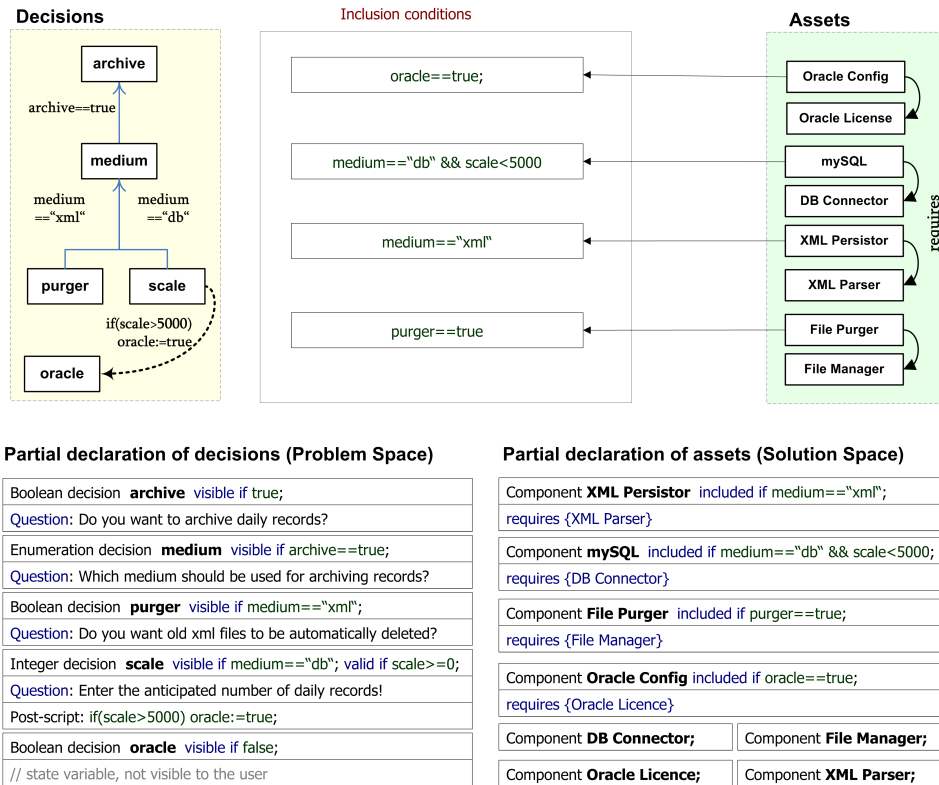
Figure 4.3: Example of a variability model.

4.1.2   The Notion of an Asset

Assets describe the artifacts and their dependencies that are available in a certain development environment. The term asset was selected as a generic term to represent all kinds of artifacts, whose variability needs to be modeled. This is required, as different variability mechanisms help in achieving variation at different artifact levels, like i.e., the requirements, architecture, or implementation level. A great challenge lies in linking variability mechanisms across different artifacts. Using a generic term allows for domain-specific refinements/interpretation of the term "asset", as the process of modeling variability is dependent on the process of software development. At the meta-level we support two basic types of dependencies among assets: *structural dependencies* are used to specify the organization of the assets and *functional dependencies* are used to specify how the system is implemented.

Structural dependencies are used to describe the "physical organization" of the assets. This includes how the assets are packaged or divided into sub-systems. In our modeling language, structural dependencies are represented by relationship links "consists of", "contributes to", "is predecessor

of", "is successor of" etc.

Functional dependencies are used to describe the "logical organization" of the assets. This includes how the assets are implemented or are functionally dependent to each other. In our modeling language, functional dependencies are represented by relationship links "includes", "excludes" etc.

In our modeling approach, assets are linked to decisions via *inclusion conditions*, representing the context and situation when a certain asset is required in the desired product. In other words, assets are "*aware*" of the decisions, which influence their selection for a product. The assets can also be included because of their dependencies to required assets, for example, a configuration parameter (also an asset) can be included if a component which requires this parameter is included. The value of the parameter can vary according to the taken decisions.

4.2   Structure of Decision Models

We informally describe the structure of decision models and the intuitive meaning of the different constructs informally, so that the reader can familiarize with the different concepts, before reading the section on the formal semantics. For the sake of simplicity in the examples, we assume the syntax of decision models to be similar to Pascal like programming languages.

Every decision corresponds to a decision variable (comparable to a typed variable in programming languages), whose value is set by taking the decisions. The names have no formal meaning but they have huge practical importance for the readability of a decision model (just like the use of mnemonic names in traditional programming). Decisions are specified in a decision model by providing the following details (cf. Figure 4.4):

- The set of possible values (defined by the decision type) and value constraints (defined by validity conditions),

- The specification of a decision's position in the decision hierarchy in relation to other available decisions (defined by the visibility condition),

- The specification of the implications of taking the decision or effects on other decisions (defined by decision effects),

- The organizational structure of decisions in groups and tasks, and

- Labels and annotations providing information for the user to better understand the decision (defined by decision attributes).

In order to explain the meaning of these constructs, we use a very simple decision model depicted in Figure 4.5, where four decisions and dependencies among them have been depicted.
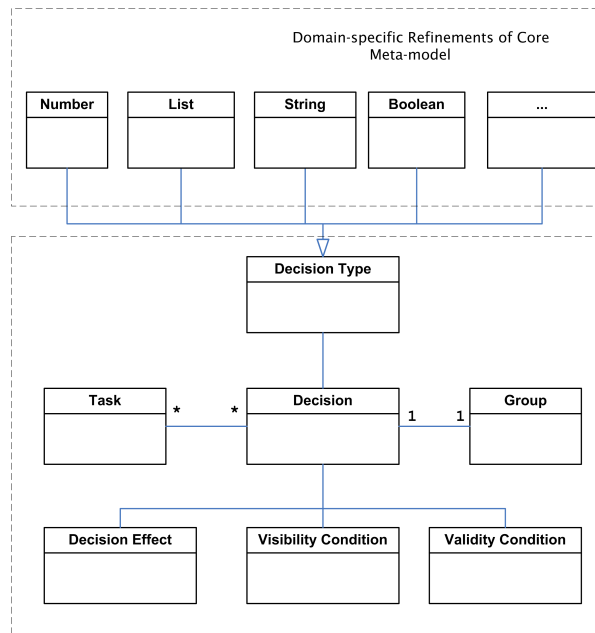
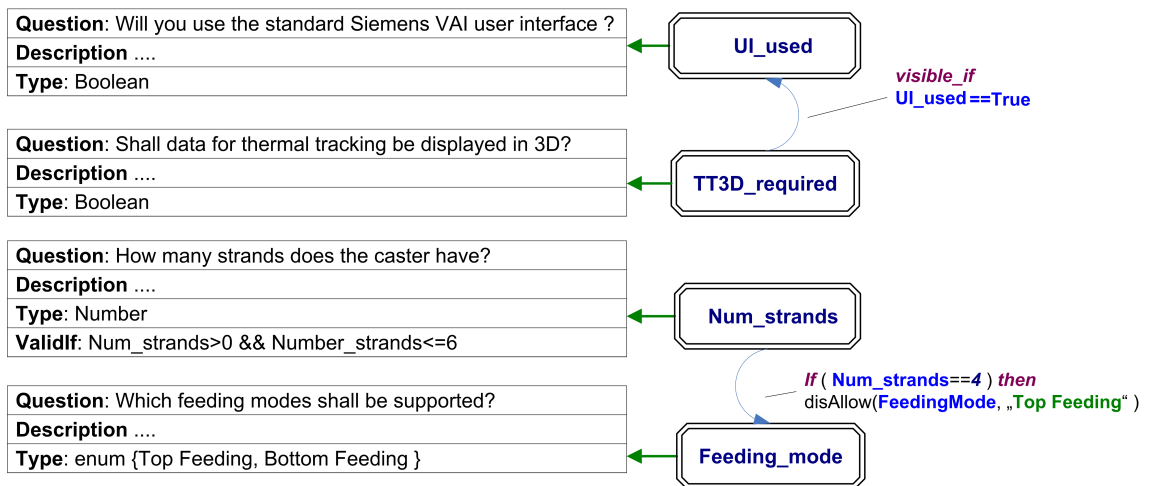**Figure 4.4:** Meta-model for Decision Models.

Figure 4.5: Example decision model showing different modeling constructs.

4.2.1 Decision Type

A decision can be compared to a variable of a given decision type. The type of the decision restricts the range of values which can be assigned to the decision. There are three predefined decision types in our modeling language, which represent Boolean, Strings and Numbers. Besides the predefined types, the user can define her own enumeration types as required for each model. Here are some examples:

```
Boolean dbRequired;
```

This denotes a decision, where the user has the option of choosing `True` or `False` for the decision `dbRequired`. If there is a decision regarding the color of a product, then its type could be defined as an enumeration, specifying all the available colors. This would mean that the user can choose between the four given colors and assign a subset of the given set as the value of the decision `color`.

```
ColorType = { red, blue, black, orange };
ColorType color;
```

In Figure 4.5, we can see three decision types:

```
Boolean UI_used, TT3D_required;
Number Num_strands;
FeedType = { Top Feeding, Bottom Feeding };
FeedType FeedingMode;
```

Decision attributes are annotations on decisions, which provide detailed information about decisions. Annotations have no formal meaning- but are helpful in understanding the model. Examples of such labels are- descriptions, images and URLs, the question which the user is asked etc. Use of labeling functions (as compared to unstructured text-tags) helps with the better interpretation of tags.

4.2.2 Validity Condition

The set of possible values of a decision specified by the type of the decision is often too broad. As an example let us consider a number decision `Speed_min`. Per definition of the type of `Speed_min`, all real numbers $\mathbb{R}$ are the possible values. In order to support the modeler to restrict the values of the decision, a validity condition can be specified for `Speed_min`.

The validity condition of a decision can be seen as a post-condition which has to be fulfilled after the decision has been taken, which means this condition is asserted before the value is assigned to the decision variable. Here are some examples:

- In order to restrict the value of `Speed_min` to only positive numbers, one could define the validity condition as follows, which means that the user can assign only positive rational numbers to `Speed_min`.:

```
Speed_min.validIf(Speed_min > 0);
```

- To denote that the user must choose the color red, when taking the decision on the color, one can specify:

```
color.validIf(color==ColorType.red);
```

- Using validity conditions, it is also possible to specify multiple ranges for decisions. For example,

```
decision1.validIf((decision1 >= n1 && decision1 <= n2) || (decision1 >= n3 && decision1 <= n4))
```

- In Figure 4.5, we can see that there is a validity condition specified for the decision `Num_Strands`,

```
Num_Strands.validIf ( Num_Strands > 0 && Num_Strands <= 6);
```

4.2.3   Visibility Condition

The visibility condition of a decision specifies when a certain decision can be taken by the user. It is used to specify the hierarchical dependency among the decisions. Visibility conditions are mostly used to specify dependencies among general decisions and more detailed decisions on already chosen options. Here are some examples:

- When configuring a car, it makes sense to first decide whether to have a cabrio or a limousine before deciding on the color of the roof. The visibility of the decision `color` depends on the type of the car.

```
CTP = {cabrio, limousine};
CTP carType;
Color carColor;
carColor.visibleIf( carType == CTP.limousine);
```

This would mean that the user needs to decide on the color of the roof only after choosing `limousine` as the `carType`.

- In Figure 4.5, we can see that there is a visibility condition specified for the decision `TT3D_required`,

```
TT3D_required.visibleIf(UI_Used);
```

meaning that it makes no sense to ask about 3D visualization, if the user does not want any user interface at all.

The order of taking decisions is partly specified by the visibility conditions, because if there is a visibility condition associated with a decision, the user has to take the decisions appearing in the visibility condition first. To elaborate on the effects of visibility conditions, we define a relationship $\diamond$ between decisions with respect to their visibility conditions. A decision v_2 is said to have a $\diamond$ relationship to another decision v_1, if the decision v_1 appears in the visibility condition of v_2. This kind of relationship between decisions, which is written as $v_2 \diamond v_1$ (read as v_2's visibility depends on v_1) is non-reflexive (the visibility condition of a variable cannot depend on itself), strictly anti-symmetric (visibility of decisions cannot depend on each other) and transitive.

It is also possible that some decisions are never visible to the user. In this case, their visibility condition is simply specified to be `false`. We refer to such decisions as state decisions.

4.2.4   State Decisions

Decisions which are never visible to the user are referred to as *state decisions* and can be bound to their values only as a result of derivation rules (decision-effects of other decisions). Such rules help in aggregating values of decisions which have already been taken and allow us to simplify complex expressions in models. State decisions can be used to keep track of different execution states of the model. Here is one example:

- The decision determining whether an Oracle database is needed for a final system may be bound to a certain value automatically after the user decides on the size of the final system. The user is never asked the question, whether she wants an Oracle database.

```
Boolean oracle_needed;
oracle_needed.visibleIf(false);
int size;
size.visibleIf(true);
size.effect{
     if(size>5000) then oracle_needed:=true;
}
```

State decisions also help the modeler when creating the models. She can use the state decisions instead of complex expressions, when building different expressions. In the above example, `(size>5000)` is equivalent to `oracle_needed`.

4.2.5   Decision effects

The logical dependencies among decisions are modeled using a set of rules. Rules can be used basically for *Assertion, Binding, Update and Information*. The semantics of rules used for assertion and binding is identical to constraints specified using boolean expressions in constraint satisfaction problems (CSPs). However, by using rules to update the model and to communicate to users at runtime, one can go beyond traditional constraints (as this is not the focus of constraints in CSPs). This also shows that variability models based on DoVML are created with the focus of an interactive product derivation process. The decision-effects are specified in the form:

$$\text{if } \langle\text{condition}\rangle \text{ then } \langle\text{action}\rangle,$$

where `condition` is a boolean expression built using decision variables
and `action` is a function changing decision variables.

A rule is activated or triggered when its `condition` evaluates to `true`. Here we present a few examples of rules and their application.

Assertion: Dependencies among decisions, where certain conditions always need to hold, e.g.,

```
// a constraint in the form
(v1==n1) implies (v2==n2)
// could be specified using the rule
if(v1==n1) then assert (v2==n2);
// or simply
assert (!(v1==n1)||(v2==n2))
```

The assert action is a read-only action. It does not change the value of the variables, but only makes sure that the condition holds.

Binding: Whenever there is a need to change the values of the variables we make use of binding actions. For example,

```
if (v1 == n1) then v2 = n2;
```

In general a binding action is comparable to a constraint as in CSPs, i.e., a condition implies a binding. In contrast to the assertion action, binding actions change the actual value of the decisions (i.e., they take decisions on behalf of the user). Here `setValue` is used as an example of a binding action (the actual syntax and semantics of all the actions is fixed when defining the rule language for the domain).

Update: Not only the values but also different attributes of decisions can be updated/manipulated using rules. As for example, depending on the value of one decision, the validity condition of another decision might change. Such an update action can be used to change the specification of the model at runtime. The modification of the decision model itself as an implication of the decisions taken by the user can however also lead to problems regarding the determinability of the decision making procedure.

Information: Rules can also be used for informative purposes. By defining actions like `inform`, or `display` one can capture knowledge which is required for the user during product derivation. Such rules have no formal semantics, but can be very helpful to the user to improve guidance during derivation. Example usage scenarios for this would be the presentation of relevant information to the user based on variability models [Rabiser *et al.*, 2007]. For example,

```
if (v1 == n1) then inform(n1 + `` is lower than the recommended..., would you like to change?'');
```

4.3   Structure of Asset Models

The structure and organization of the solution space is specific to the domain/industrial context at hand, therefore the core of DoVML can be parameterized and adapted to the domain using an asset-meta model. Our approach does not assume fixed types of assets for modeling variability. By providing an abstract conceptual representation of structured data (comparable to entity-relationship models in relational databases), the modeler defines the "modeling language" for the solution space. Building such a model requires knowledge about the domain and the organization's implementation practices. The meta-model defines the types of assets to be included in the product line (for instance in Figure 4.6, the asset types are `Components`, `Resources` and `Properties`) and the possible relationships between the different asset types (in Figure 4.6, these relationships are `contributesTo` and `requires`).

Asset models are instances of the asset meta model describing the structure of the solution space. For example, the asset model in Figure 4.7 is an instance of the meta-model depicted in Figure 4.6.
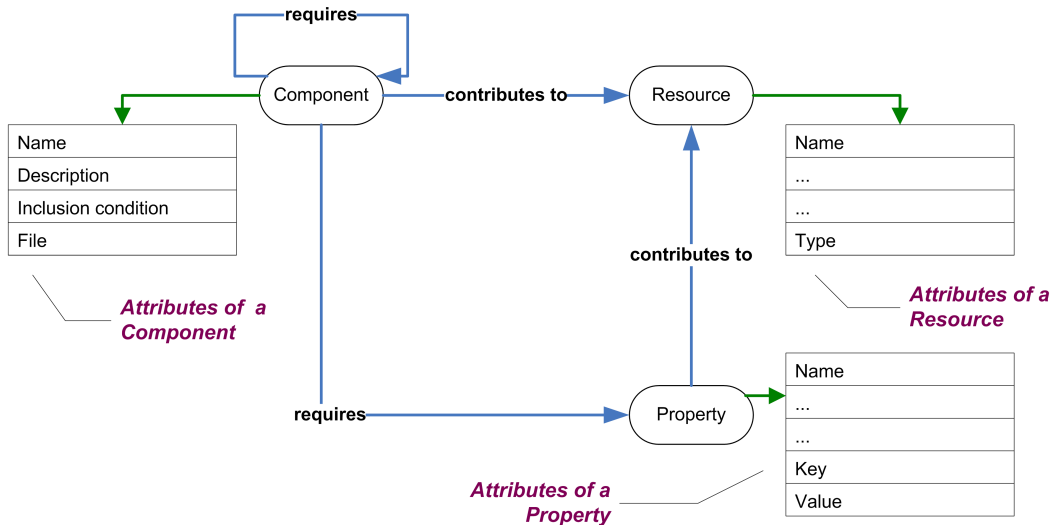
Figure 4.6: Example of a (partial) domain-specific meta model, specifying the kinds of assets, their attributes and relationships between them.

When building an asset meta-model the types of assets to be used, their attributes and dependencies among them can be defined.
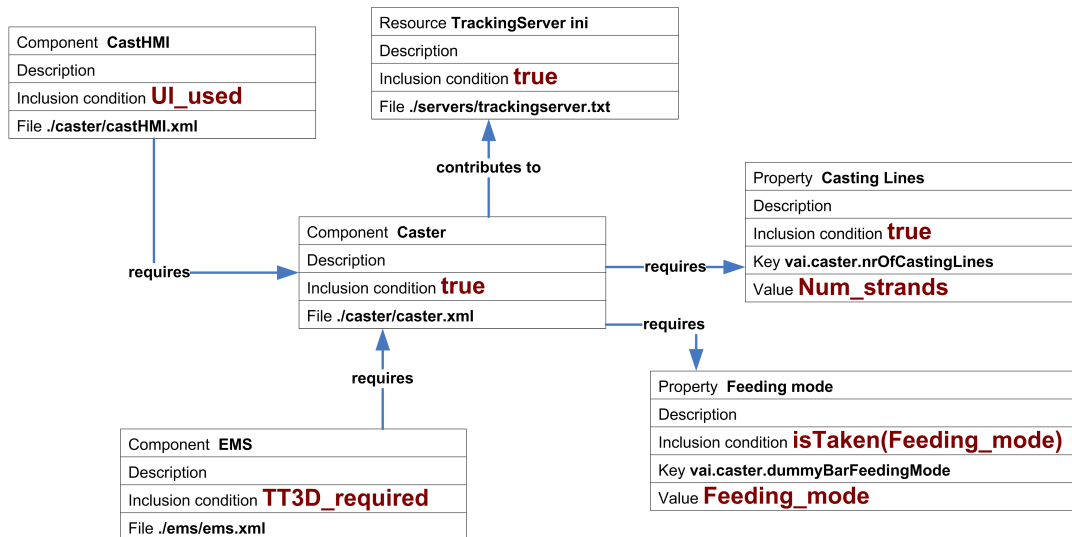
Figure 4.7: Example of a (partial) asset model, depicting a set of available assets, their attribute values and relationships between them.

59

We associate a Boolean expression called inclusion condition to every asset in the asset model. Such an expression specifies the condition under which the asset will be included in the final product. If an asset is always included in the system (e.g., utility classes, common libraries) then its inclusion condition is simply `true`. Considering the example presented in Figure 4.7, the inclusion condition of the component `EMS` is defined as `TT3D_required`. This means that the component `EMS` is included in the final product, if the decision `TT3D_required` is set to `true`. The inclusion condition can be arbitrarily complex and can involve any number of variables, thus supporting arbitrarily complex dependencies between decisions and assets.

Often assets are not included or excluded from the final product directly because of decisions taken by the user but rather because of technical dependencies resulting from their implementation. For example (cf. Figure 4.7), the property `Casting Lines` could be included in the final product because it is required by the component `Caster`.

4.4   Intuitive Interpretation of DoVML

The operational semantics of decision-oriented variability models can be explained intuitively using an algorithm, which can interpret such variability models. The result of executing such a variability model is a set of taken decisions (binding of decision variables) and a set of assets required for the desired product (and the attribute values of the included assets).

Variability models built using DoVML are constructed such that they can be used for highly automated product derivation processes. For example,

1. Visibility conditions are used to distinguish between decisions which are relevant for the user and the ones which are not. This guides the user through a product derivation process.

2. Decision attributes like questions, descriptions and images are used to communicate decisions to the user.

3. Decision effects are propagated automatically to ensure the consistency of the decision taking procedure.

4.4.1   Algorithms for Executing Models

Decision-making based on variability models (e.g., as a part product derivation/configuration) is an interactive process. Decisions can either be visible or invisible to the user. The transition between these states is regulated by the evaluation of the visibility condition, which is triggered whenever a new variable binding takes place. All visible decisions are presented to the user. The variable binding takes place either as a result of a user interaction or as a result of rules which are evaluated as required after a decision is taken. An asset can either be included in or excluded from the desired final product. The transition between these states occurs as a result of the evaluation of the inclusion condition of the assets. Psedocode of an algorithm for executing a variability model is presented in Listing 4.1.

```
// algorithm for taking decisions
_____
Initialize a hash table of taken decisions <decision , value >: taken_decisions
Initialize list of required assets: required_assets

repeat {
   foreach (decision d in variability model) {
      if ( d is visible ) { // evaluation of the visibility condition
         display d.question to the user
         value val = read input from user
         if ( val is valid value of d) { // evaluation of validity condition
            add <d, value> to the table taken_decisions
            propagate the effects of decision d  // M1(d): call decision effect propagation algorithm
            calculate list of required assets // M2: call asset inclusion evaluation algorithm
         }
      }
   }
} until (all visible decisions are taken)


//M1(decision d): algorithm for propagation of decision effects
_____
foreach (rule r in d){
   if(r.condition evaluates to TRUE){
      execute r.script
      list l = decisions changed when executing r.script
      // propagate the effect of decision of all affected decisions
      for(each decision dx in l){
         propagate the effects of decision dx  // M1(dx): call decision effect propagation algorithm
      }
   }
}

//M2: algorithm for evaluation of asset inclusion
_____
foreach (Asset a in variability model){
   if(a.inclusioncondition evaluates to TRUE){
      add a to the list required_assets
      add all assets required (modeled through asset relationship) to the list required_assets
   }
}
```

Listing 4.1: Sample algorithm for executing variability models

Firstly, the visibility condition of each decision variable is evaluated. If the condition holds, then a question is presented to the user (possibly with other labels of the decision variable) so that the variable is better understood when taking the decision. The input from the user is evaluated against the validity condition. If the input was valid, then the variable is bound to the input value. Such a binding has two implications:

1. *It propagates the effects of all decisions*, by evaluating all the rules and executing them as necessary. Such rules can also cause a variable binding, which leads to a recursive call of the rule engine. So the execution of the action specified in the rule requires that the condition evaluates to `true`. The rule engine does not use a brute force algorithm to evaluate all the expressions after every change.

Only the expressions, which contain the currently taken decision are evaluated. The execution of the action can change the set of already bound variables; can however also only be informative. As the rule engine can trigger the evaluation of the rules again, it is important that there are no cyclic dependencies in the model. Cycles in the rules are detected using standard cycle detection algorithms for graph like data structures.

2. *It triggers the evaluation of asset inclusion*, which is the process of figuring out which assets need to be included in the final product. The process consists of two phases (i) evaluation of the inclusion condition and (ii) evaluation of asset dependencies. The set of included assets can then be used by domain-specific application generators simulators and deployment tools for further processing.

4.4.2   Example: Model Execution

In this section, we describe using the example depicted in Figure 4.3, how a variability model is executed by listing one possible order of taking the decisions:

1. The first decision, which can be taken by the user is `archive`. If the user answers `FALSE`, the process is completed. Otherwise go to step 2.

2. The user is asked to take the decision `medium`, the two possible options being `xml` and `db`. If the user chooses `xml` then go to step 3. If the user chooses `db` then go to step 4.

3. The user decides whether a `purger` is required.

4. The user decides on the `scale` of the system. If the scale is larger than 5000, then the decision `oracle` is automatically set to `TRUE`.

Depending on how the decisions were taken, the list of required (included) assets is calculated by evaluating the inclusion conditions and resolving the asset dependencies.

4.5   Formal Semantics of DoVML

To describe the formal meaning of DoVML, we follow the formalization principles as introduced in [Harel & Rumpe, 2000, Harel & Rumpe, 2004], where a language consists of a syntactic notation (syntax[1]) which is a possibly infinite set of elements that can be used in the communication together with their meaning (semantics). We use the term syntax whenever we refer to the notation of the language and focus purely on the notational aspects of the language completely disregarding any meaning. The meaning of a language is described by its semantics.

---

[1]Formally, syntax of a language is defined as a Tuple(T, N, P, S), where T is the set of terminal symbols, N is the set of non-terminal symbols, P is a set of productions and S is the start symbol.

The semantic definition for a language $\mathcal{L}$ consists of two parts: a semantic domain which we denote generically by $\mathcal{S}$ and a semantic mapping from the syntax to the semantic domain denoted by $\mathcal{M}$. Often the mapping $\mathcal{M}$ is explained informally, by examples and in plain English. But regardless of the degree of formality of its exposition, the semantic mapping itself must be a rigorously defined function from $\mathcal{L}$ to $\mathcal{S}$, written $\mathcal{M} : \mathcal{L} \rightarrow \mathcal{S}$ (cf. Figure 4.8). A language is (formally) unambiguous when $\mathcal{L}$, $\mathcal{S}$ and $\mathcal{M}$ all receive a mathematical definitions [Harel & Rumpe, 2000]. It is important to note that $\mathcal{M}$ is a complete function, i.e., defined $\forall m \in \mathcal{L}$.
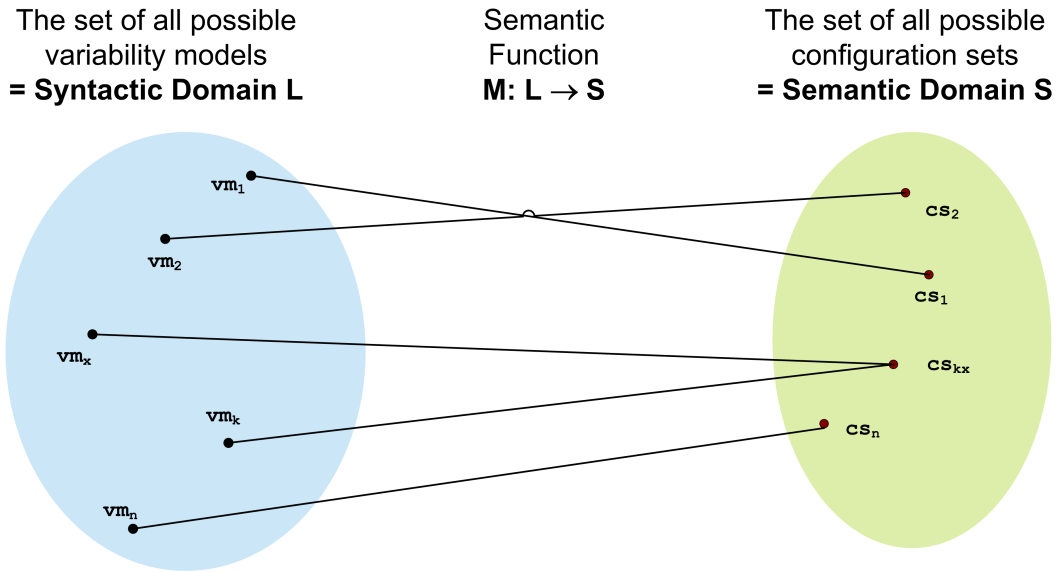
Figure 4.8: Syntactic and semantic domains for DoVML.

4.5.1 The Syntactic Domain $\mathcal{L}$

The syntactic domain for DoVML is defined as the set of all possible variability models which can be written in the language. It is formally defined as,

$$\mathcal{L} = \{vm | vm = \langle \mathcal{DM}, \mathcal{AM} \rangle\}$$

where $\mathcal{DM}$ is a decision model, and $\mathcal{AM}$ is an asset model. $\mathcal{DM}$ and $\mathcal{AM}$ are described in detail in definitions 3 and 4 respectively. In this section, we first present the mathematical structures that will be used to define the formal semantics of DoVML.

- Since tuples are frequently used in this formalization, we define a notation to handle them conveniently. Let t be a tuple defined by

$$t = \langle c_1, c_2, \ldots, c_n \rangle$$

where $c_1, c_2, \ldots, c_n$ are the tuple member's names. Let Π_i be a function that returns the the ith component of t.

- Let us consider the following to be given $\mathcal{AT}$, $\mathcal{ATA}$, $\mathcal{ATR}$, τ, where

  - $\mathcal{AT}$ is a finite set of asset types, which are defined for a given development context. This set is given by the modeler and can consist of arbitrary artifact types, which are supposed to be reused in the context of a product line. Example– $\mathcal{AT}$ = { `Component`, `Testcase`, `Service`}

  - $\mathcal{ATA}$ is a finite set of asset type attributes. These attributes are used to specify detailed properties of the assets in the model. Example– $\mathcal{ATA}$ = {Name, Description, URL, Price, MaxRTime}

  - $\mathcal{ATR}$ is a function which associates each asset type to a set of attributes, defined formally as

    $$\mathcal{ATR} \subseteq \mathcal{AT} \times \mathcal{ATA}.$$

    $\mathcal{ATR}$ is only a subset of the given Cartesian product because not all asset types are related to all the available asset attributes. Example– This example shows that the asset type `Component` has three attributes: `Name`, `Price`, `Description`.
    `Service` has two attributes: `URL, MaxRTime` and
    `Testcase` has one attribute: `Description`.

    $$
    \begin{aligned}
    \mathcal{ATR} = \quad \{ \quad & \langle Component, Name \rangle, \langle Component, Price \rangle, \\
    & \langle Component, Description \rangle, \langle Service, URL \rangle, \\
    & \langle Testcase, Description \rangle, \langle Service, MaxRTime \rangle \\
    \}
    \end{aligned}
    $$

  - τ is a function which associates each each asset attribute with a data type, which is formally described as

    $$\tau : \mathcal{ATR} \rightarrow \mathcal{DT} \cup \{\mathbb{E}xpr\}.$$

    where,

    * $\mathcal{DT}$ is a set of data types as defined in definition 1.

    * $\mathbb{E}xpr$ is a special given data type used for attributes.

Example– This example only shows a partial definition of the function τ, but the function is defined for all elements in $\mathcal{ATR}$.

$$
\begin{aligned}
\tau = \{ \quad & \langle Component, Name \rangle \mapsto \mathbb{S}, \\
& \langle Component, Price \rangle \mapsto \mathbb{Expr}, \\
& \langle Service, MaxRTime \rangle \mapsto \mathbb{Q}, \\
& \dots \\
\}
\end{aligned}
$$

Definition 1 (Data type) *A data type θ is a couple*

$$
\theta = \langle \mathsf{Id}_\theta, \mathcal{D}om_\theta \rangle
$$

where

- *Id_θ is the type identifier, which corresponds to its local name.*

- *$\mathcal{D}om_\theta$ is a set called the interpretation domain of θ. It represents the set of possible values belonging to the type.*

For the formalization of DoVML, we define $\mathcal{D}T$ to be the set of all types defined in the context.

$$
\mathcal{D}T = \mathcal{D}T_p \cup \mathcal{D}T_u
$$

- $\mathcal{D}T_p$ is a finite given set of predefined data types

$$
\mathcal{D}T_p = \{ \langle Number, \mathbb{Q} \rangle, \langle Boolean, \mathbb{B} \rangle, \langle String, \mathbb{S} \rangle \}
$$

- $\mathcal{D}T_u$ is a finite set of data types provided by the modeler. The specification of $\mathcal{D}T_u$ includes the set of possible values for each type, which are usually enumerated.

Example– A set of user defined data types $\mathcal{D}T_u' \subset \mathcal{D}T_u$

$$
\begin{aligned}
\mathcal{D}T_u' = \{ \quad & \langle Quality, \{high, medium, low\} \rangle, \\
& \langle Color, \{black, red, blue, green, orange\} \rangle, \\
& \langle FileType, \{xml, txt, exe, dll\} \rangle, \\
& \langle ShoeType, \{casual, business, sports\} \rangle \\
\}
\end{aligned}
$$

The interpretation function for types is a function $\llbracket \bullet_1 \rrbracket : \mathcal{DT} \to \mathcal{D}om$ which returns the interpretation domain corresponding to the type provided. It is formally defined by:

$$\llbracket \theta \rrbracket \overset{\text{def}}{=} \mathcal{D}om_\theta$$

Further we define $\mathcal{D}om$ to be the interpretation domain of all data types.

$$\mathcal{D}om = \bigcup_{\theta \in \mathcal{DT}} \mathcal{D}om_\theta$$

Definition 2 (Boolean Formulae) *In DOVML complex formulae are built from decisions and simpler sub-formulae, by means of functions and operations. To give an abstract definition of decision models, it is not necessary to fix the concrete syntax in which the modeler writes the expressions, and thus we shall assume that such a syntax exists (together with well-defined semantics).*

$\mathcal{T}erms$ in DoVML are defined recursively by

- *All constants are terms, i.e.,*
  let T be a value from type θ ($T \in \llbracket \theta \rrbracket$), then T is a term.

- *All decisions are terms, i.e.,*
  let T be a decision identifier ($T \in \mathcal{D}$) then T is a term.

- *All expressions built using decisions, constants and operators are terms, i.e.,*
  let $<op>$ be an operator from $\mathcal{O}per$ with arity n, and $T_1, T_2, \ldots, T_n$ be n terms then $<op>(T_1, \ldots, T_n)$ is also a term, where $\mathcal{O}per = \{+, -, *, /, \ldots\}$.

Example– A set of terms $Terms_x \subset \mathcal{T}erms$
$Terms_x = \{1.4, 293, high, \{high, low\}, x+20, x*4+2, "local"\}$
$\mathbb{BF}$ (Boolean formulae) are defined recursively by

- *If T_1, T_2 are terms, then the following are **atoms**:*

$$T_1=T_2 , T_1<T_2 , T_1 \in T_2 , T_1 \subset T_2$$

  Plus the usual syntactic short-hands:

$$T_1 \neq T_2 , T_1 \leq T_2 , T_1 > T_2 , T_1 \geq T_2 , T_1 \notin T_2 , T_1 \subseteq T_2 , T_1 \supset T_2 , T_1 \supseteq T_2$$

- *All atoms are boolean formulae;*

- *If ϕ, φ are Boolean formulae, then the following are Boolean formulae:*

$$\neg\phi , \phi \vee \varphi$$

  Plus the usual syntactic short-hands:

$$\phi \wedge \varphi , \phi \Rightarrow \varphi , \phi \iff \varphi$$

Example–
$Atoms : d_1 = 5, d_2 \in \{a, b, c\}, d_3 \subseteq \{a, b\}, d_4 = x + 7$
$Terms : 2 \in \{a, b, c\} \vee d_4 = x + 7, d_3 \subseteq \{a, b\} \Rightarrow (d = 5)$

Definition 3 (Decision Model) *Given $\mathcal{DT} = \mathcal{DT}_p \cup \mathcal{DT}_u$ and $\mathbb{BF}$, a decision model $\mathcal{DM}$ can be defined as a tuple*

$$\mathcal{DM} = \langle \mathcal{D}, \tau, f_{vis}, f_{val}, f_{pos} \rangle$$

where

- *$\mathcal{D}$ is a given finite set of decisions provided by the modeler. We differentiate between two kinds of decisions, i.e., the ones that are directly taken by the user (user decisions = $\mathcal{UD}$) and the ones, which get their values derived from already taken decisions (state decisions = $\mathcal{SD}$).*

$$\mathcal{D} = \mathcal{UD} \cup \mathcal{SD}$$

- *τ is a typing function labeling each decision to its corresponding data type defined as $\tau : \mathcal{D} \rightarrow \mathcal{DT}$.*

- *f_{vis} is a function specifying the visibility condition for each decision defined as $f_{vis} : \mathcal{D} \rightarrow \mathbb{BF}$, where $\mathbb{BF}$ is a set of Boolean Formulae defined as in definition 2. There is one additional constraint to f_{vis}, which requires $\forall d \in \mathcal{D} : f_{vis}(d)$ does not contain d.*
  Example of f_{vis} –

  - *$f_{vis}(d) = x > 20$ would mean that the decision d can be taken only if x is already taken and has the value greater than 20.*

  - *$f_{vis}(d) = d > 20$ would mean that the decision d can be taken only if d is already taken. This is not possible, therefore not allowed.*

- *f_{val} is a function specifying the validity condition for each decision defined as $f_{val} : \mathcal{D} \rightarrow \mathbb{BF}$, where $\mathbb{BF}$ is a set of Boolean Formulae defined as in definition 2.*

- *f_{pos} is a set of rules for deriving the value of decisions. There are two kinds of such rules– value derivation rules (f_{der}) and type redefinition functions (τ').*

$$f_{pos} = f_{der} \cup \tau'$$

  - *f_{der} is a rule defining how the values of certain decisions are calculated depending on whether the specified activation condition is fulfilled. f_{der} is formally defined as a subset, because not all decisions have value derivation rules to determine their values.*

$$f_{der} \subseteq \mathcal{D} \rightarrow \mathbb{BF} \times Terms$$

  Example– $f_{der}(d) = \langle a > 20, b + c * 10 \rangle$ would mean in an intuitive Java-like syntax: `if(a>20) setValue(d, b+c*20);`

  $f_{der}(d) = \langle a > 20, null \rangle$ is equivalent to `if(a>20) reset(d);`

  There are some additional constraints to f_{der}

∗ *A derivation rule is always present for all state decisions, and only some user decisions have a derivation rule,*

$$f_{der} : (\mathcal{SD} \rightarrow \mathbb{BF} \times \mathcal{T}erm) \cup (\mathcal{UD} \nrightarrow \mathbb{BF} \times \mathcal{T}erms)$$

∗ *The decision with such a derivation rule should not appear either in the condition or the term returned by f_{der},*

$$\forall d.f_{der}(d) : \langle \varphi, T \rangle .$$

d neither appears in φ nor in T.

Example– Some invalid f_{der} according to the this constraint are:
$f_{der}(d) = \langle d > 20, 43 \rangle$, $f_{der}(d) = \langle a > 20, d + 1 \rangle$.

– *τ' is a conditional type redefinition function, specifying the condition under which the type of a decision is changed.*

$$\tau' \subseteq \mathcal{D} \rightarrow \mathbb{BF} \times \mathcal{DT}$$

For decisions of pre-defined types, the type redefinition function is defined as:

$$\forall d \in \mathcal{D}.\tau(d) \in \mathcal{DT}_p \Rightarrow \tau'(d) = \langle TRUE, \tau(d) \rangle$$

which means, there is no type redefinition for decisions of type $\mathcal{DT}_p$.

$\tau'(d) = \langle \varphi, \theta \rangle$ means, if the condition φ is fulfilled, the type of the decision is changed to θ.

Example– Consider a decision d, such that

$$\tau(d) = \langle Color, \{black, red, blue, green, orange\} \rangle$$

By using a type redefinition function, the set of colors allowed for the decision can be changed, e.g.,

$$\tau'(d) = \langle a > 20, \langle Color', \{blue, green, orange\} \rangle \rangle$$

which means, if a>20, then only three colors (instead of the 5 colors at the beginning) are available to select from.

Definition 4 (Asset Model) *An asset model $\mathcal{AM}$ can be defined as a tuple*

$$\mathcal{AM} = \langle \mathcal{A}, f_{av}, R_{inc}, R_{exc}, f_{inc} \rangle$$

where

• *$\mathcal{A}$ is a finite set of assets. It is required that*

– *the type of the asset is defined in $\mathcal{AT}$.*

$$\forall a \in \mathcal{A}.\tau(a) \in \mathcal{AT}$$

– *the values of the attributes (belonging to this asset) are of the type specified in the asset type.*
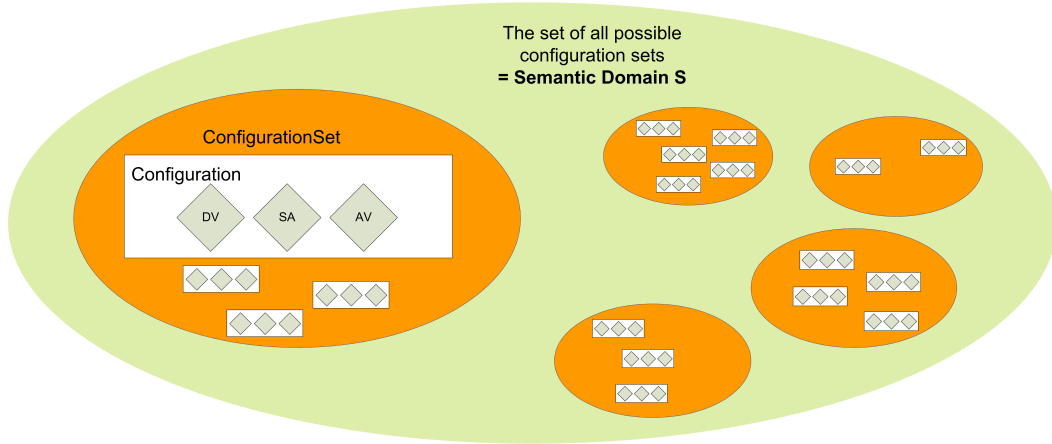
$$\forall \langle at, \alpha \rangle \in \mathcal{ATR}\cdot$$
$$\forall a \in \mathcal{A}, \text{ such that } \tau(a) = at \cdot$$
$$f_{av}(a, \langle at, \alpha \rangle) \in \mathcal{D}om_{\tau(\langle at, \alpha \rangle)}$$
$$\text{where } \mathcal{D}om_{\tau(\mathbb{E}xpr)} = Terms.$$

- $R_{inc} \subseteq \mathcal{A} \times \mathcal{A}$, *represents the "inclusion" relationship among the assets.*
- $R_{exc} \subseteq \mathcal{A} \times \mathcal{A}$, *represents the "exclusion" relationship among the assets.*

4.5.2 The Semantic Domain $\mathcal{S}$

Figure 4.9: The semantic domain

Definition 5 (Semantic Domain) *Every decision-oriented variability model (vm) represents a set of configurations, which is defined as follows:*

$$\mathcal{S} = \{ConfigurationSet\}$$

Definition 6 (Configuration set)

$$ConfigurationSet = \mathcal{P}(Configuration)$$

Definition 7 (Configuration set)

$$Configuration = \langle DecVals, SelectedAssets, AttrVals \rangle$$

where,

- $DecVals : \mathcal{D} \to (\mathcal{D}om \cup \{null\}) \times \{vis, hid\}$
  $DecVals$ represents is a function which is defined for every decision in the decision model. The first element of the tuple returned by the function represents the value of the decision. If the value of the decision is $null$, then it means that the decision has not been taken yet. The second element of the tuple is either vis or hid, which refers to whether the decision is visible or hidden after the configuration. In a valid configuration c, it is not possible for a decision to be visible but not yet taken.

$$\forall cs \in \mathcal{S}.\forall c \in cs.\forall dv \in \Pi_1(c) : x \mapsto \{null, vis\} \notin dv.$$

- $SelectedAssets \subseteq \mathcal{A}$
  $SelectedAssets$ is the set of assets that have been selected to be included in the final product. There can be two causes for the inclusion of assets:

  - *The inclusion condition of the asset evaluates to true with the given set of $DecVals$.*

  - *The asset is in a R_{inc} relationship with a already included asset.*

- $AttrVals \subseteq SelectedAssets \times \mathcal{ATR} \to \mathcal{D}om$
  For all the assets that are included in the final product, it is required that the asset attributes have been assigned their values.

4.5.3   The Semantic Function

Given a syntactic domain $\mathcal{L}$ and a semantic domain $\mathcal{S}$, the final and main step in defining a semantics is to relate the syntactic expression to the elements of the semantic domain, so that each syntactic creature is mapped to its meaning [Harel & Rumpe, 2004].

Semantic Mapping $\mathcal{M}$

The semantic mapping function $\mathcal{M} : \mathcal{L} \to \mathcal{S}$ for is defined as as follows-

$$\forall m \in \mathcal{L}.\mathcal{M}(m) = \{cs \in \mathcal{S} | cs = \langle dv, sa, av \rangle\}$$

satisfying the following rules

1. The values of Decisions fulfill the criteria regarding their validity, visibility, value derivation and type redefinition. Formally,

$$\forall d \in \mathcal{D} \quad . \quad [\![f_{val}(d)]\!]_{dv}$$
$$\wedge([\![f_{vis}(d)]\!]_{dv} \Leftrightarrow \Pi_2(dv(d)) = vis)$$
$$\wedge[\![f_{der}(d)]\!]_{dv}$$
$$\wedge[\![\tau'(d)]\!]_{dv}$$

70

- The semantic interpretation for validity condition (f_{val}) and visibility condition (f_{vis}) are functions defined in the semantic in the semantic interpretation of Boolean Formula.

- The semantic interpretation for derivation rules (f_{der}) is a function

$$\llbracket \bullet_1 \rrbracket_{\bullet_2} : f_{der} \to DecVals \to \mathbb{B}$$

which is defined as

$$\llbracket d \mapsto \langle \varphi, T \rangle \rrbracket_{dv} \quad \equiv \quad \llbracket \varphi \Rightarrow d = T \rrbracket_{dv}$$

- The semantic interpretation for Type Redefinition Function τ' is a function

$$\llbracket \bullet_1 \rrbracket_{\bullet_2} : \tau' \to DecVals \to \mathbb{B}$$

which is defined as $\tau' \subseteq \mathcal{D} \to \mathbb{BF} \times \mathcal{DT}$, where

$$
\begin{aligned}
\llbracket d \mapsto \langle \varphi, \theta \rangle \rrbracket_{dv} \quad \equiv \quad & \Pi_1(dv(d)) \in \Pi_2(\tau(d)) \\
& \vee (\llbracket \Pi_1(\varphi) \rrbracket_{dv} \wedge \Pi_1(dv(d)) \in \Pi_2(\theta))
\end{aligned}
$$

2. The set of selected assets is chosen based on whether the inclusion condition is fulfilled and whether the asset is required by already included asset.

$$sa = sa' \cup \{a| \langle b, a \rangle \in R^+_{inc} \wedge b \in sa'\}$$

where

$$sa' = \{a | a \in \mathcal{A} \wedge \llbracket f_{inc}(a) \rrbracket_{dv}\}$$

with an additional constraint:

$$sa \cup ua = \oslash$$

where

$$ua = \{a | a \in \mathcal{A} \wedge b \in sa \wedge \langle b, a \rangle \in R_{exc}\}$$

3. The attributes of the selected assets are either just the values entered by the modeler, or if the type of attribute is $\mathbb{E}xpr$ then the value returned by the evaluation of the Term associated with the attribute value.

$$
\begin{aligned}
\forall a \in sa, s.t.\tau(a) = at. \quad & let\mu = (a, \langle at, \alpha \rangle). \\
av(\mu) = \llbracket f_{av}(\mu) \rrbracket_{dv} \quad \vee \quad & (\tau(\langle at, \alpha \rangle) \in \mathcal{DT} \\
& \wedge av(\mu) = f_{av}(\mu))
\end{aligned}
$$

Semantics of Terms and Boolean Formulae

The semantic interpretation of terms is a function

$$\llbracket \bullet_1 \rrbracket_{\bullet_2} : \mathcal{T}erm \to DecVals \to \mathcal{D}om$$

which returns the value of a term for a given valuation function and at a given set of decision values (dv). It is defined by:

$$
\begin{aligned}
\forall T \in \llbracket \theta \rrbracket : \llbracket T \rrbracket_{dv} &= T \\
\forall T \in \mathcal{D} : \llbracket T \rrbracket_{dv} &= \Pi_1(dv(T)) \\
\llbracket {<}op{>}(T_1, \ldots, T_n) \rrbracket_{dv} &= \llbracket {<}op{>} \rrbracket (\llbracket T_1 \rrbracket_{dv}, \ldots, \llbracket T_n \rrbracket_{dv})
\end{aligned}
$$

- The semantic interpretation for atoms is a function

$$\llbracket \bullet_1 \rrbracket_{\bullet_2} : \mathcal{A}tom \to DecVals \to \mathbb{B}$$

which takes an atom, a set of decision values dv, and returns $\mathbb{T}$ or $\mathbb{F}$ depending if the values of the decisions in dv verifies that formula. It is formally defined by:

$$\llbracket T_1 {<}op{>} T_2 \rrbracket_{dv} = \llbracket {<}op{>} \rrbracket \llbracket T_1 \rrbracket_{dv} \llbracket T_2 \rrbracket_{dv}$$

where the operators semantic interpretation is a function

$$\llbracket \bullet_1 \rrbracket : operator \to \llbracket \theta_1 \rrbracket \to \llbracket \theta_2 \rrbracket \to \mathbb{B}$$

defined by

$$
\begin{aligned}
\llbracket = \rrbracket &\equiv \lambda t_1 t_2 . t_1 = t_2 \\
\llbracket < \rrbracket &\equiv \lambda t_1 t_2 . t_1 < t_2 \\
\llbracket \in \rrbracket &\equiv \lambda t_1 t_2 . t_1 \in t_2 \\
\llbracket \subset \rrbracket &\equiv \lambda t_1 t_2 . t_1 \subset t_2
\end{aligned}
$$

and their syntactic short-hands:

$$
\begin{aligned}
\llbracket \neq \rrbracket &\equiv \lambda t_1 t_2 . \neg(t_1 = t_2) \\
\llbracket \leq \rrbracket &\equiv \lambda t_1 t_2 . (t_1 < t_2) \vee (t_1 = t_2) \\
\llbracket > \rrbracket &\equiv \lambda t_1 t_2 . t_2 < t_1 \\
\llbracket \geq \rrbracket &\equiv \lambda t_1 t_2 . (t_2 < t_1) \vee (t_1 = t_2) \\
\llbracket \notin \rrbracket &\equiv \lambda t_1 t_2 . \neg(t_1 \in t_2) \\
\llbracket \subseteq \rrbracket &\equiv \lambda t_1 t_2 . (t_1 \subset t_2) \vee (t_1 = t_2) \\
\llbracket \supset \rrbracket &\equiv \lambda t_1 t_2 . t_2 \subset t_1 \\
\llbracket \supseteq \rrbracket &\equiv \lambda t_1 t_2 . (t_2 \subset t_1) \vee (t_1 = t_2)
\end{aligned}
$$

- The semantic interpretation for Boolean formulae ($\mathbb{BF}$) is a function

$$\llbracket \bullet_1 \rrbracket_{\bullet_2} : \mathbb{BF} \rightarrow DecVals \rightarrow \mathbb{B}$$

which is defined by

$$
\begin{aligned}
\llbracket \neg \phi \rrbracket_{dv} &\equiv \neg \llbracket \phi \rrbracket_{dv} \\
\llbracket \phi <op> \varphi \rrbracket_{dv} &\equiv \llbracket <op> \rrbracket \llbracket \phi \rrbracket_{dv} \llbracket \varphi \rrbracket_{dv}
\end{aligned}
$$

The semantics of the operator $\vee$ is a function

$$\llbracket \bullet_1 \rrbracket : operator \rightarrow \mathbb{B} \rightarrow \mathbb{B} \rightarrow \mathbb{B}$$

defined by

$$\llbracket \vee \rrbracket \equiv \lambda \phi \varphi . \phi \vee \varphi$$

and the semantics of the syntactic short-hands operators:

$$
\begin{aligned}
\llbracket \wedge \rrbracket &\equiv \lambda \phi \varphi . \neg (\neg \phi \vee \neg \varphi) \\
\llbracket \Rightarrow \rrbracket &\equiv \lambda \phi \varphi . \neg \phi \vee \varphi \\
\llbracket \Longleftrightarrow \rrbracket &\equiv \lambda \phi \varphi . (\phi \wedge \varphi) \vee (\neg \phi \wedge \neg \varphi)
\end{aligned}
$$

4.5.4 Example

Example Syntactic domain

For illustrative purposes, let us consider a simple decision-oriented variability model defined as follows. Given sets are:

Set of defined asset types $\mathcal{AT}$ = { Component}
Set of defined asset attributes $\mathcal{ATA}$ = { Name, Price}
Set of asset attributes $\mathcal{ATR}$ = { $\langle Component, Name \rangle$, $\langle Component, Price \rangle$}
Set of pre-defined decision types $\mathcal{DT}_p = \{\langle Boolean, \mathbb{B} \rangle\}$
Set of user-defined decision types $\mathcal{DT}_u = \{\langle Color, \{red, green, blue\} \rangle\}$

Set of decisions $\mathcal{D}$ = { b1, b2, q1}
Definition of decision types

$$
\begin{aligned}
\tau = \{ &b1 \mapsto \langle Boolean, \mathbb{B} \rangle, \\
&b2 \mapsto \langle Boolean, \mathbb{B} \rangle, \\
&q1 \mapsto \langle Color, \{red, blue, green\} \rangle\}
\end{aligned}
$$

Definition of visibility conditions

$$
\begin{aligned}
f_{vis} \;=\; & \{ b1 \mapsto true, \\
& \; b2 \mapsto false, \\
& \; q1 \mapsto b1 \}
\end{aligned}
$$

Definition of validity conditions

$$
\begin{aligned}
f_{val} \;=\; & \{ b1 \mapsto true, \\
& \; b2 \mapsto true, \\
& \; q1 \mapsto (q1 \in \{ \, \{red, blue\}, \{red, green\} \}) \}
\end{aligned}
$$

Definition of decision value derivation rules

$$
f_{der} \;=\; \{ b2 \mapsto \langle q1 = \{red, blue\}, b2 = true \, \rangle \}
$$

Set of assets $\mathcal{A} = \{$ a1, a2, a3 $\}$
Definition of asset types

$$
\tau \;=\; \{ a1 \mapsto Component, a2 \mapsto Component, a3 \mapsto Component \}
$$

Definition of inclusion condition and requires (inclusion) relationships

$$
\begin{aligned}
f_{inc} \;&=\; \{ a1 \mapsto b1, a2 \mapsto b2 \} \\
R_{inc} \;&=\; \{ \langle a1, a3 \rangle \}
\end{aligned}
$$

Semantic Domain

The variability model presented in this example represents ONE configuration set (see definition 7). The semantic mapping function $\mathcal{M}$ maps the example variability model to the configuration set $CS_{example}$.

$$
CS_{example} = \{ C_1, C_3, C_3, C_4, C_5 \}
$$

where C_1, C_2, C_3, C_4 are the four possible configurations of the given example.

C_1 represents the start configuration, where no decision has yet been taken.

$$
C_1 = \langle \{ b_1 \mapsto \langle null, vis \rangle, b_2 \mapsto \langle null, hid \rangle, q_1 \mapsto \langle null, hid \rangle \}, \{\}, \{\} \rangle
$$

C_2 represents the configuration, where the decision b_1 has the value $false$.

$$
C_2 = \langle \{ b_1 \mapsto \langle false, vis \rangle, b_2 \mapsto \langle null, hid \rangle, q_1 \mapsto \langle null, hid \rangle \}, \{\}, \{\} \rangle
$$

C_3 represents the configuration, where the decision b_1 has the value *true*. Now the decision q_1 is visible.

$$C_3 = \langle \{b_1 \mapsto \langle true, vis \rangle , b_2 \mapsto \langle null, hid \rangle , q_1 \mapsto \langle null, vis \rangle \}, \{a_1, a_3\}, \{\ldots\} \rangle$$

C_4 represents the configuration, where the decision b_1 has the value *true*. Now the decision q_1 is taken, value = red, green.

$$C_4 = \langle \{b_1 \mapsto \langle true, vis \rangle , b_2 \mapsto \langle null, hid \rangle , q_1 \mapsto \langle \{red, green\}, vis \rangle \}, \{a_1, a_3\}, \{\ldots\} \rangle$$

C_5 represents the configuration, where the decision b_1 has the value *true*. Now the decision q_1 is taken, value = red, blue. This automatically changes the decision b_2.

$$C_5 = \langle \{b_1 \mapsto \langle true, vis \rangle , b_2 \mapsto \langle true, hid \rangle , q_1 \mapsto \langle \{red, blue\}, vis \rangle \}, \{a_1, a_2, a_3\}, \{\ldots\} \rangle$$

4.6 Summary

In this chapter we presented a decision-oriented approach to modeling variability of software systems. We illustrated the approach with several examples and presented a formal semantics of the modeling approach. Variability is modeled through decisions and artifacts are modeled through the assets. Arbitrary artifact types are supported by our approach, as the core meta model is refined for each domain at hand separately. These artifacts can be at different levels of granularity and abstraction. There are two ways in which the traceability information among arbitrary artifacts can be maintained.

Solution-space dependencies: Our approach allows the modeler to define arbitrary dependency types in the domain-specific meta-model. This provides flexible facilities to define dependencies among any kind of solution space artifacts, known as assets in our approach.

Problem-space dependencies: It is also possible to establish traceability links between the assets through the decisions they are related to. Dependencies among decisions (problem-space dependencies) usually reflect the dependencies among the assets dependent on the decisions.

In order to demonstrate that the approach can be applied to different modeling contexts, we present three case studies in the evaluation part of this thesis.

66 When it is not necessary to make a decision, it is necessary not to make a decision. **99** —*Lord Falkland*

$\mathcal{D}ecision\mathcal{K}ing$: A Flexible and Extensible Tool for Integrated Variability Modeling

Summary *This chapter describes $\mathcal{D}ecision\mathcal{K}ing$, a meta-tool implementing our modeling approach. It is adaptable to new domains, extensible with new capabilities and offers automation to ease tedious tasks. $\mathcal{D}ecision\mathcal{K}ing$ implements the modeling approach described in the previous chapter.*

$\mathcal{D}ecision\mathcal{K}ing$ is an integral part of a larger tool suite called DOPLER (**D**ecision-**o**riented **P**roduct **L**ine **E**ngineering for effective **R**euse). In this thesis, we shall concentrate on $\mathcal{D}ecision\mathcal{K}ing$ and its architecture, but for the sake of completeness, we present an overview of the DOPLER tool-suite and describe how $\mathcal{D}ecision\mathcal{K}ing$ fits in the big picture.

5.1 The DOPLER Tool Suite

DOPLER consists of three integrated tools. An overview of the functionality provided by the three tools and their relationships to each other is depicted in Figure 5.1. The communication and transfer of information between the tools occurs through models, i.e., there is a data-dependency between the tools. ConfigurationWizard requires the output of ProjectKing which requires the output from $\mathcal{D}ecision$-$\mathcal{K}ing$. $\mathcal{D}ecision\mathcal{K}ing$ is therefore the basis platform for modeling, ProjectKing and ConfigurationWizard provide a front-end for utilizing the variability models.

$\mathcal{D}ecision\mathcal{K}ing$ is a tool for variability modeling and management. The tool provides meta-modeling features to define a domain-specific meta-model with asset types and interdependencies between them. This meta-model can then be used to create domain-specific variability models. Details about $\mathcal{D}ecision\mathcal{K}ing$ follow in the next section.

ProjectKing is a tool for product and sales managers to pre-configure the variability model for product derivation. Different roles and responsibilities of stakeholders in product derivation can be defined, irrelevant variability can be pruned (e.g., by setting defaults), and additional sales and project knowledge can be added (e.g., multimedia decision-support information). Details about ProjectKing can be found in [Rabiser, 2009].

ConfigurationWizard is a tool for decision takers (e.g., sales people or application engineers) to actually take decisions during product derivation. The tool provides capabilities for product customization based on variability models. Different roles of stakeholders and added knowledge created with ProjectKing are taken into consideration. The output of ConfigurationWizard is a product derived from the product line based on the taken derivation decisions. Details about ConfigurationWizard can be found in [Rabiser, 2009].
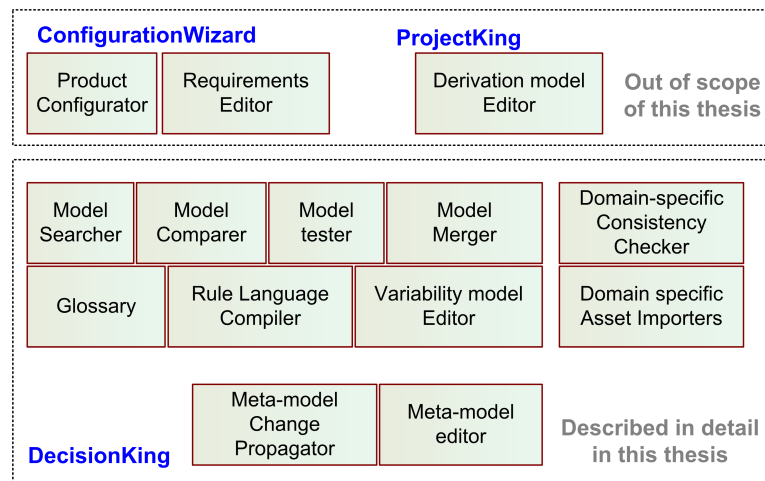
Figure 5.1: Overview of Dopler Tools.

5.2   Domain-specific Variability Modeling

5.2.1   Meta-model editor

Refinements of the core meta-model depicted in Figure 4.1 can be created using the meta-model editor provided by *DecisionKing*. The editor is based on Eclipse Forms[1] and provides a tree/table based user interface to create meta-models. By providing such a user interface (as compared to textual meta-modeling), *DecisionKing* "guides" (by allowing only what is valid) the user during the creation of the meta-model.

Default asset types: The tool provides one default asset type called `Asset`, which must be refined for a new domain. The default asset type consists of three default attributes:`Name`, `Description`, and `IncludedIF`.

Default attribute types: The attributes of an asset type have associated data types. It is important to assign data types to the attributes to generate corresponding user interface elements for the variability
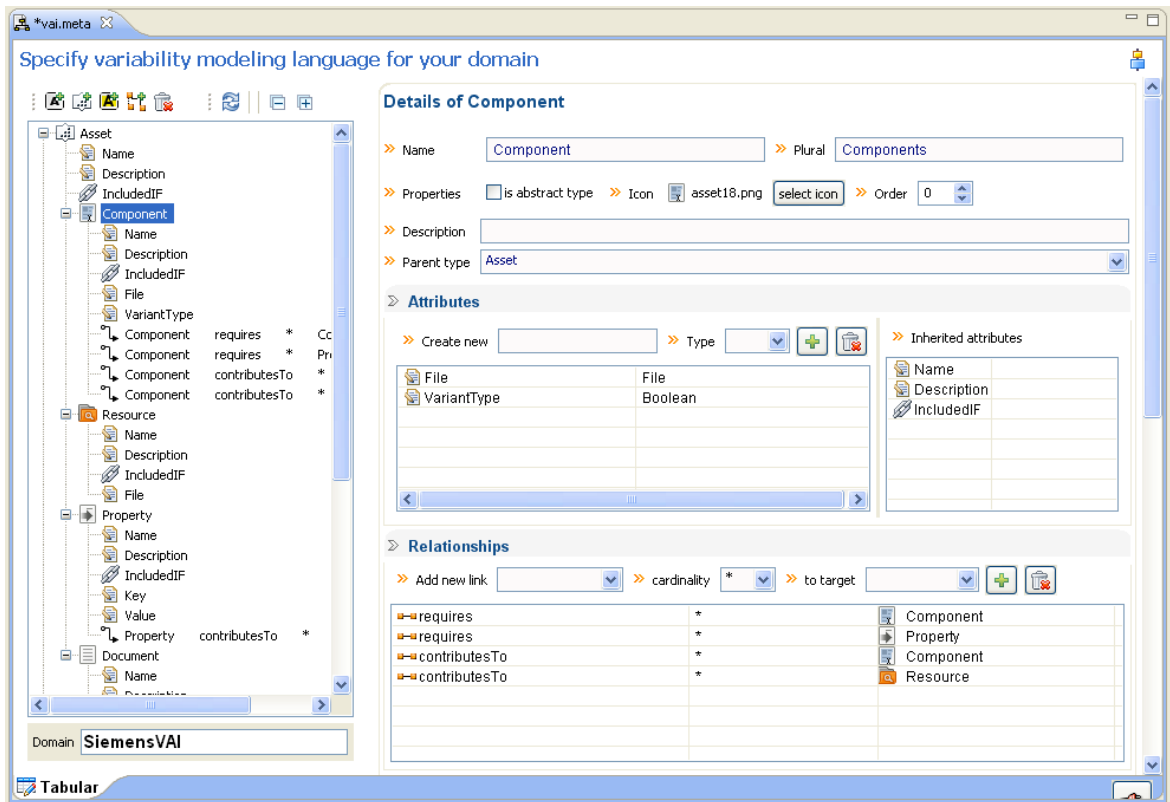
---

[1]http://www.eclipse.org/articles/Article-Forms/article.html

Figure 5.2: *DecisionKing* Meta-model editor.

model editor. *DecisionKing* currently implements the following types: `Boolean`, `Expression`, `File`, `List`, `Number`, `String`, `Paragraph` and `URL`. This list can be easily extended by providing new attribute types and corresponding code to generate appropriate UI elements and associated behavior. For example, we defined a new attribute type `Javascript` for one of our case studies.

Figure 5.2 shows the meta-model editor in *DecisionKing*. It depicts a new asset type `Component` defined as the subtype of the default type `Asset`. It inherits the default attributes from its parent type and defines two new attributes (File and VariantType). One can also see the different relationships between the type Component and other asset types, e.g., requires * property and contributesTo * resource. The meaning of all the asset types depicted in Figure 5.2 is explained in our case study on Siemens VAIs steel plant product line.

The output of the meta-model editor is a .meta file, which serves as the input for configuring the variability model editor. The variability model editor then allows the creation of variability models based on this meta-model. Similar ideas for generating domain-specific tools was also followed by Grundy *et al.* in [Grundy *et al.*, 2006], where they propose meta-tools capable of generating domain-

specific visual language editors from high-level tool specifications.

5.2.2   Variability Modeling Editor

The variability model editor is a meta-editor for variability models, which is "instantiated" to model the specifics of the domain using the .meta file created using the meta-model editor. The screenshot depicted in Figure 5.3 was configured using the meta model depicted in Figure 5.2. The variability model editor provides a seperate modeling page for each of asset type.

The asset variability modeling pages allow the creation of assets of the corresponding asset type, their attributes and relationships. The information contained in the .meta file is thereby used to generate user interface elements as required to model different data types.
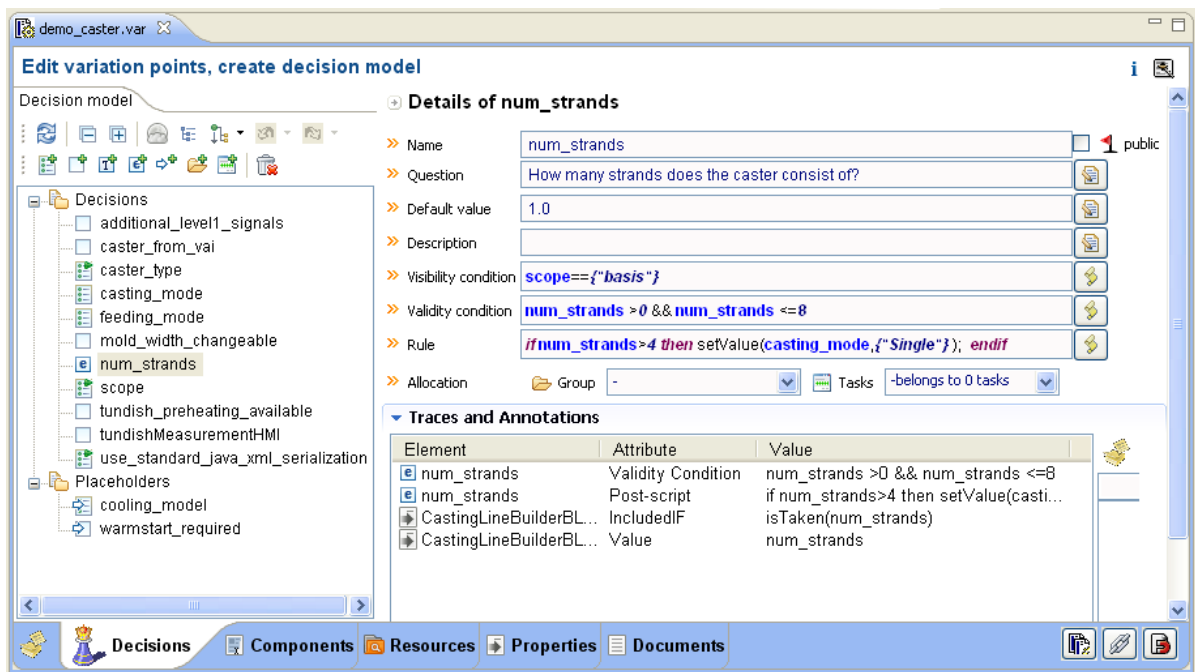

Figure 5.3: *DecisionKing* variability model editor.

As depicted in Figure 5.3, the variability model editor consists of a fixed page for modeling the decisions. Currently we support three decision attributes to communicate the meaning of a decision to the user: `Descriptions` are blocks of text (e.g., in HTML) and are used to clarify the meaning of a decison. HTML also allows the easy integration of images, videos and animations to improve the guidance of the product derivation process. `Questions` are formulated in a concise way in the user's problem space language, such that the answer to that question implies the value of the decision. By

making use of `annotations` (cf. Figure 5.12) one can attach arbitrary information (in textual form) to a decision.

Other attributes such as visibility condition, validity condition and decision effects, which need to be formally evaluated are modeled using a rule and expression language (provided as a plugin to $\mathcal{DecisionKing}$).

5.2.3 Checking Consistency of Models

"Building Software" refers to the process of converting source code files into executable code. When building software, one of the first steps is the compilation of source code–thereby a software tool (compiler/builder) checks for syntax errors in the code. Today automated builders are very popular in software development environments. For example, Eclipse provides advanced means of building applications using *incremental project builders*, which are programs that manipulate/check the resources in a project (models, code, etc) on the fly. $\mathcal{DecisionKing}$ has made use of the "builder infrastructure" (provided by Eclipse) and provides a "variability model builder". The builder checks variability models for errors and displays the detected inconsistencies in the problem viewer (also an integrated part of the Eclipse IDE), as depicted in figure 6.10. This makes the users aware of different modeling problems in their model. Here we list the types of errors, which are currently detected by the tool:

Simple well-formedness rules are used to detect some trivial mistakes the modeler might make. For example,

- Enumeration decisions, which have less than two possible values.

- Mandatory asset attributes, which have not been assigned a value.

- References to files (as asset attributes), which no longer exist.

Syntax errors in expressions and rules are detected using the rule-language-plugin. The logic necessary for detecting syntax errors in the expressions and rules depends on the rule language and is thus not part the of $\mathcal{DecisionKing}$ core. This applies to the syntax of:

- Visibility conditions, validity conditions and decision-effects.

- Inclusion conditions and all other asset-attributes (of type Expression) defined in the domain-specific meta-model.

Cyclic dependencies between decisions are detected by transforming the decision model in a graph-like data structure. Cyclic dependencies can occur in three ways,

- The effect of a decision $d1$ affects another decision $d2$, which in turn (directly or indirectly) affects $d1$.

- The visibility condition of a decision $d1$ is dependent on a decision $d2$, whose visibility condition directly or indirectly depends on the decision $d1$.

- The validity condition of a decision $d1$ is dependent on a decision $d2$, whose visibility condition directly or indirectly depends on the decision $d1$.

5.3 Support for Executing Models

DecisionKing provides an extension point for the rule language infrastructure, which consists of the definition of the language, a compiler, an execution engine and an editor [Wallner, 2008]. We are currently using a rule engine based on JBoss Drools[2]. All rules are written in an intuitive language and are translated into corresponding representation in Drools[3] notation.

5.3.1 Rule Language

The definition of the rule language consists of the definition of the supported decision types (data types) and functions for manipulating the data. *DecisionKing* currently supports four basic types of decisions:

`Boolean` decisions are used to represent yes/no questions. In contrast to the data type `Boolean` in programming languages, we support three possible states for variables of this type: `true`, `false` and `undefined`. The state undefined was important, as we needed to distinguish between the answer "no" to a certain question and "not yet decided".

`Number` decisions are used mostly for parameter values, where the user decides on a numerical value. These are comparable to the type "double" in programming languages. Other numerical types: integer, short etc can be simulated using number decisions.

`String` decisions are used for similar purposes as number decisions. They correspond to the data type "String" in programming languages.

`Enumeration` decisions can be seen as arrays of strings. Such decisions are used whenever different alternatives to the same variation point need to be modeled.

We are currently using an expression language, which shows high syntactic resemblance to Pascal. One can make use of standard operators (e.g.,$+, -, \div, *, =, \neq, \leq, \geq, <, >, \vee, \wedge$, etc.) to build expressions. *DecisionKing* provides an expression editor (with syntax highlighting and auto completion, cf. Figure 5.3) to ease the modeling process. Apart from the standard operators we provide the following actions to query the value of decisions and build more complex expressions (grammar depicted in Listing 5.1).

---

[2]http://www.jboss.com/products/rules/
[3]http://www.jboss.org/drools/

```
RuleLangCompiler      = { Rule }.
Rule                  = ( "if" Expression "then" { Action } "endif" ) | Action.
Action                = [ ActionFunctionName "("  Parameters ")" ] ";".
Parameters            = [ Expression { "," Expression } ].
Function              = "contains" | "isTaken" | "max" | "abs".
ActionFunctionName    = "setValue" | "reset" | "selectOption" |
                          "deSelectOption" | "allow" | "disAllow".
Expression            = AndExpr { "||" AndExpr }.
AndExpr               = EqlExpr { "&&" EqlExpr }.
EqlExpr               = RelExpr { ( "==" | "!=" ) RelExpr }.
RelExpr               = AddExpr { ( "<" | ">" | "<=" | ">=" ) AndExpr }.
AddExpr               = MulExpr { ( "+" | "−" ) MulExpr }.
MulExpr               = Unary { ( "*" | "/" | "%" ) Unary}.
Unary                 = {"+" | "−" | "!" } Primary.
Primary               = (Literal | ident | Function "(" Parameters ")".
Literal               = numberLiteral | stringLiteral | true | false | null.
(** END of Grammar **)
```

Listing 5.1: *DecisionKing*'s rule language grammar used to express dependencies among decisions [Wallner, 2008].

1. `setValue(d, p)` is an assignment function, which assigns the value p to decision d. The function setValue was used instead of the usual assignment operator "=" because the version of JBoss Rule Engine which we used (version 4.0) supported only function calls in the actions defined as a part of the rules.

2. `selectOption(d, p)`, `deselectOption(d, p)` are used to select/deselect the alternative p in a enumeration decision.

3. `contains(d, p)` is a set operator which can be used in enumeration decisions to perform $\subset, \subseteq, \in$ operations. p is the set to be compared with the value of the decision d.

4. `allow(d, p)`, `disallow(d, p)` are used to expand/restrict the set of possible values by the set in a enumeration decision d.

5. `isTaken(d)` is used to query, whether a decision has already been taken by the user.

6. `reset(d)` is used to retract a taken decision. Retracting a decision also resets all its implications modeled in the rules.

5.3.2 Rule Language Editor

A special rule editor guides the user while modeling dependencies among decisions. Code completion is one of the most important features that is crucial for user acceptance. Code completion provides the pop up menu with the list of the variables used in the script when the user starts typing new variable name. Hints show the arguments and returning value for a just typed function, as well as a short description for them. Such facilities help the user, as they do not require to know the detailed

syntax of the rule language. The rule language editor and support for code completion is depicted in Figure 5.4.

Furthermore, the rule language editor provides syntax highlighting feature, which increases the readability of the script. On the long run, this feature helps the developers who need to maintain the variability models. We also provide on-the-fly syntax checking features, making the user aware of typos and other preventable mistakes. Figure 5.4 shows a typical screenshot of the editor window, showing some of the features such as (1) comments (2) syntax errors (3) error markers and error messages (4) error viewer.
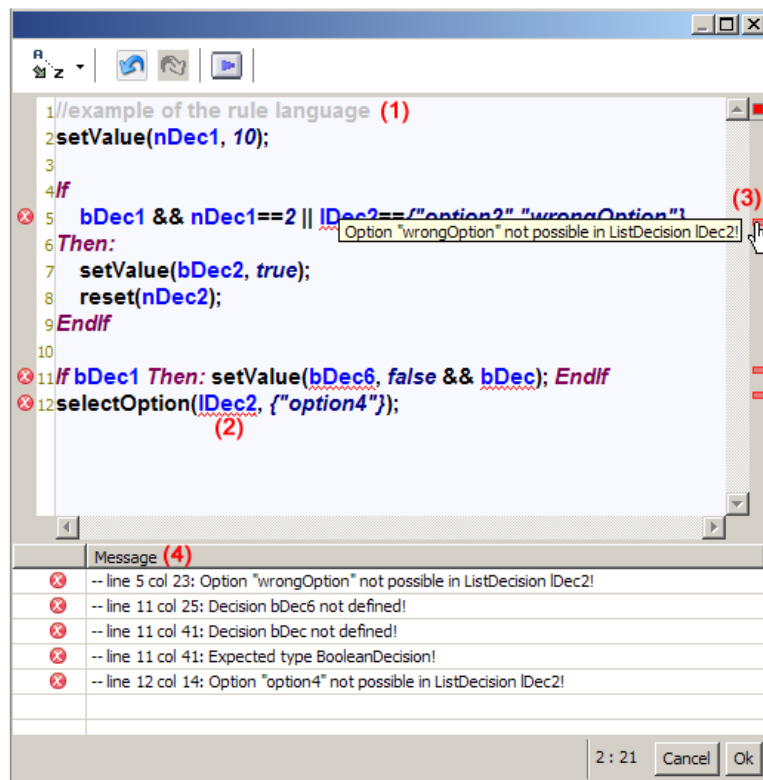
Figure 5.4: Rule language editor.

5.3.3   Compiler and Execution Engine

In order to create the compiler for our rule language we used the compiler compiler Coco/R[4], which is a compiler generator. Coco/R takes an attributed grammar (ATG) of a source language and

---

[4]http://www.ssw.uni-linz.ac.at/coco/

generates a scanner and a parser for this language [Mössenböck, 1991]. The user has to supply a main class that calls the parser as well as semantic classes (e.g., a symbol table handler or a code generator) that are used by semantic actions in the parser.

Using the compiler generated by Coco, the abstract syntax tree of the variability model is transformed into the structure of JBoss Drools, which is an open-source, object-oriented production rules engine. JBoss has become a popular business logic framework, used by Java developers to create complex rule-based applications by combining Java platform and business rule technology. Its basic architecture follows the structure of a classical expert system. Figure 5.5 depicts how the rule language compiler fits into the landscape of rule execution engine provided by JBoss Drools. The concept of an expert system is this: the knowledge of an expert is encoded into the rule set. When exposed to the same data, the expert system AI will perform in a similar manner to the expert.
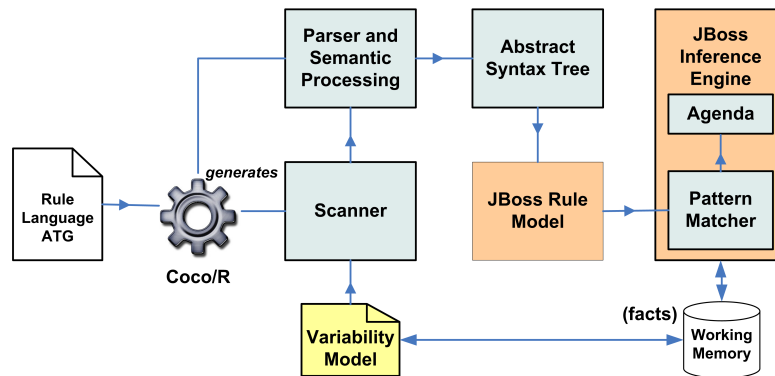
Figure 5.5: Generation of rule language compiler and evaluation of variability models using JBoss Rule Execution Engine.

JBoss Drools is a forward chaining rule engine. It starts with a rule base, which contains all of the appropriate knowledge encoded into If-Then rules, and a working memory, which may or may not initially contain any data, assertions or initially known information. In the case of *DecisionKing*, the rules are the decision effects modeled as decision attributes and the facts are the decisions taken by the user. The system examines all the rule conditions and determines a subset of the rules whose conditions are satisfied based on the working memory. This process is called "Pattern Matching" and it is based on the Rete [Forgy & Shepard, 1987] algorithm, which evaluates a declarative predicate against a changing set of rules in real time. JBoss Drools wraps the semantics of the normal relational Rete into a ReteOO (Object-Oriented) model that's more compatible with Java objects.

When the rule is fired, any actions specified in its THEN clause are carried out. These actions can modify the working memory, the rule-base itself, or do just about anything else the system programmer decides to include. This loop of firing rules and performing actions continues until one of two conditions are met: there are no more rules whose conditions are satisfied or a rule is fired whose action specifies the program should terminate.

To give an impression of the syntax of JBoss Drools, we provide an example of a rule written in *DecisionKing*'s rule language, and its automatic translation into the JBoss notation Drools[5].

```
//The following is a simple rule consisting of one if statement:
if (num_strands>4)then
  setValue(casting_mode, {"Single"});
endif

//is automatically translated into drools rule in the following form:
rule "0"
salience 0
no-loop true
when
  num_strands:RuleNumberDecision(
    name == "num_strands",
    active==true)and
    eval(num_strands.getPValue()>4)then
      ArrayList<String> drools_a;
      num_strands.identify();
      drools_a = new ArrayList<String>();
      drools_a.add("Single");
      num_strands.set(0,"casting_mode", drools_a);
  end
```

Listing 5.2: Example of a rule in *DecisionKing* and its conversion to Drools notation.

Here is a simple example of how Rete optimizes the network of rule and why it is faster than a batch (brute-force) method of executing rules. Given two rules as in Listing 5.3, there are different ways, how the rules can be evaluated. As depicted in Figure 5.6, the JBoss Rule Engine which is adopted by *DecisionKing* optimizes the Rete tree such that the evaluation of expressions is more efficient because only a selected subset of expressions need to be evaluted every time a decision is changed.

```
//rule 1:
if (a && b) then setValue(d, true);

//rule 2:
if (a && b && c) then setValue(e, true);
```

Listing 5.3: Examples of two simple rules in *DecisionKing*.

5.3.4   Model Testing

The use of builders is a means of carrying out static tests on the model, which are not enough to test the correctness of the model. Dynamic testing involves working with the model, giving input values and checking if the output is as expected. Dynamic testing of models has to do with model execution. These tests can either be predefined (batch testing by running scripts to take decisions on behalf of the user), random or manual "real life" tests which reflect the true intention of testing the model.

---

[5]Drools is a business rule management system (BRMS) and an enhanced Rules Engine implementation, ReteOO, based on Charles Forgy's Rete algorithm tailored for the Java language.
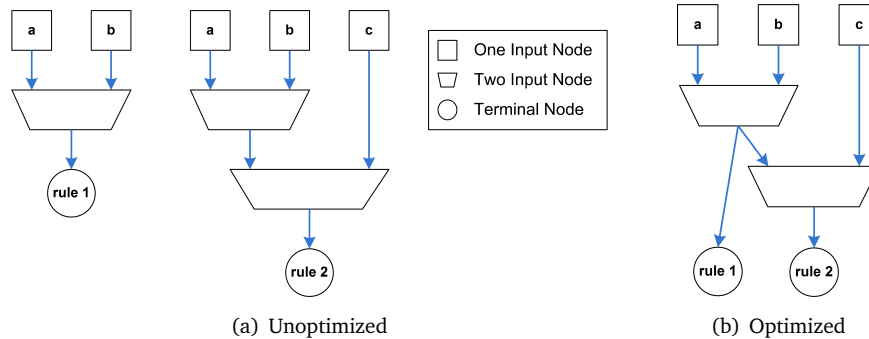
Figure 5.6: Rete Networks are optimized by the JBoss Rule Engine, in such a way that it is easy to find which expressions need to be evaluated, when a certain decision is changed.

The nature of the tool required for dynamic testing of variability models depends on the purpose of variability modeling. *DecisionKing* provides a default model testing tool, with which it is possible to execute the variability model, take different decisions and make sure that the list of required assets matches the expectations. *DecisionKing* therefore provides an extension point for model launchers, where different tools can be plugged in and used as model testing tools.

The default implementation of the model testing tool is the test dialog depicted in Figure 5.7. It is a miniature version of the ConfigurationWizard, which is used for product configuration using variability models in the context of product lines. The left column of the tool displays the decisions modeled in the variability model. The user is provided with checkboxes, combo boxes, radio buttons and other UI elements required for answering the question depending on the kind of decisions. The right column of the tool displays the set of required assets, calculated on the basis of the decisions taken by the user.

5.4 *DecisionKing* **Model Evolution Framework**

Software maintenance and evolution are among the most challenging and cost-intensive activities in software engineering. A software system MUST be maintained if it is to remain useful. Software change is inevitable as new requirements emerge when the software is used, the business environment changes or errors must be repaired. The issue of software aging [Parnas, 1994] and its implications for the development and maintenance process was described by Parnas in a well known paper in 1994. In the context of product lines, new customer requirements, technology changes and internal enhancements lead to the continuous evolution of the reusable assets. Software maintenance activities can be categorized in three types [Swanson, 1976]:

Corrective maintenance indicates the changes made in a system to solve processing, performance or implementation failures.
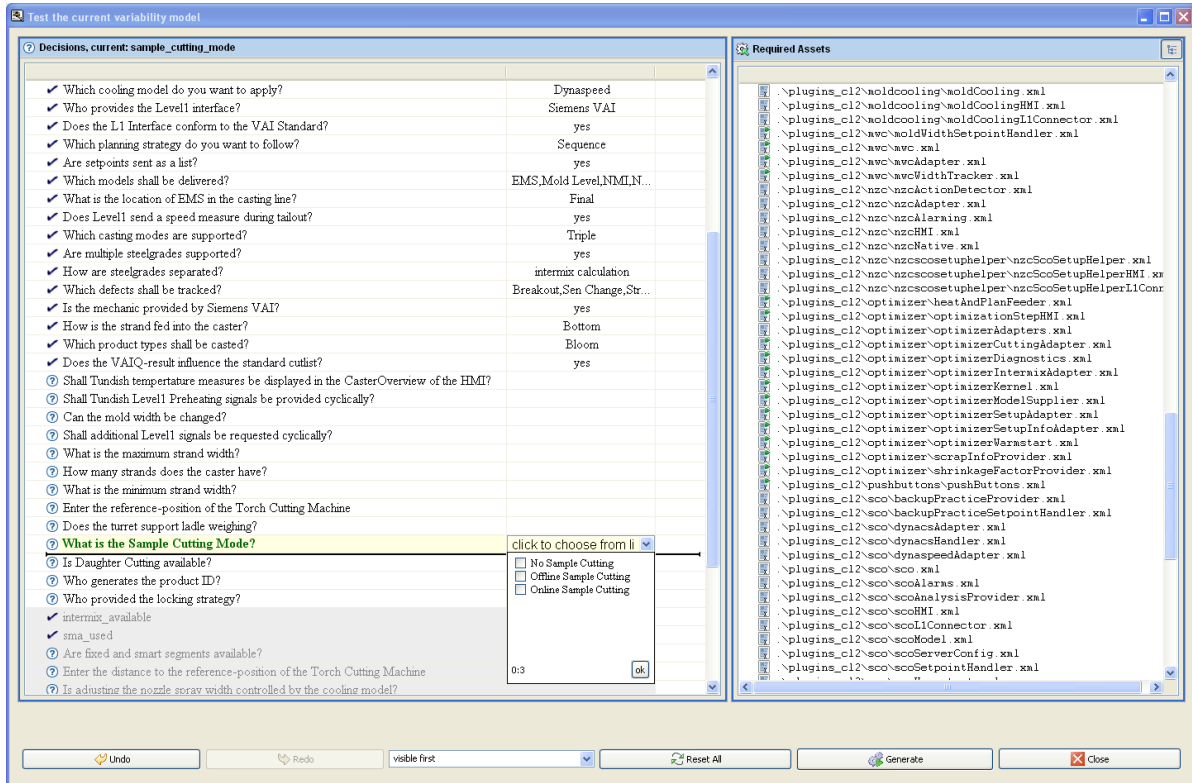
**Figure 5.7:** *DecisionKing* variability model execution dialog (dynamic testing).

Adaptive maintenance is the change of a software system triggered by changes in the business or technical environment.

Perfective maintenance is used to improve the quality (processing inefficiency, performance enhancement, etc.) and maintainability of a system.

Evolution management is a race against time. While software engineers try to understand and capture the variability of a complex existing system the reusable assets are frequently changed to meet the business needs. Evolving and maintaining a software system requires dealing with different types of changes caused by changing customer needs, evolving technology, or market developments. Despite its importance surprisingly few papers are available on product line evolution (e.g., [Bosch, 2000, McGregor, June 2003, Svahnberg & Bosch, 1999]). Many PLE approaches assume that activities in domain and application engineering can take a fairly stable product line for granted. However, PLE should thus treat evolution as the normal case and not as the exception [Dhungana *et al.*, 2008]. Evolution support becomes success-critical in a model-based development approach to ensure consistency after

changes to meta-models, models, and actual development artifacts.

$\mathcal{D}ecision\mathcal{K}ing$ variability models and meta-models are saved in an XML format, which allowed us to share the models and corresponding meta-models among different stakeholders using Concurrent Versions Systems (CVS). We used the version control system Subversion[6] for this purpose. It is possible to use "Eclipse Text Comparer" to compare variability models, but the comparison is unstructured and not comprehensible to the user (cf. Figure 5.8).
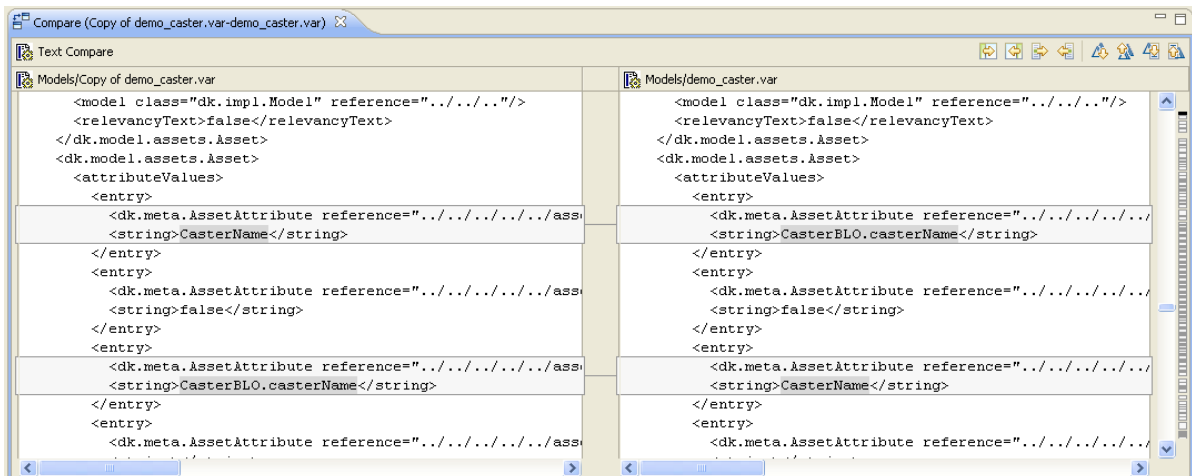
Figure 5.8: Eclipse Text Comparer used for comparing two versions of variability models

Whenever changes were made to the models, the models could be compared using the text file compare tool provided by Eclipse. As depicted in Figure 5.8, the two versions of a variability model can be compared using the "Eclipse Text Comparer" tool. However, the user has no idea at the model level, whether elements were added, deleted or whether the attributes changed. For this purpose, it was important to actually develop our own model comparison tool, which guided the user through the comparison process in an intuitive manner. Similar approach to developing a tool for comparing architecture models was pursued by [Abi-Antoun *et al.*, 2006].

5.5 Supporting Meta-model Evolution

The domain meta-models are subject to evolution similar to the product line's core assets. An effective model-driven development cycle relies on support for domain evolution: Modeling tools and techniques must be adaptable to changing requirements in the problem domain. For instance, the introduction of new asset types or the modification of existing assets (e.g., by changing an attribute) require updating existing models.

---

[6]http://subversion.tigris.org/
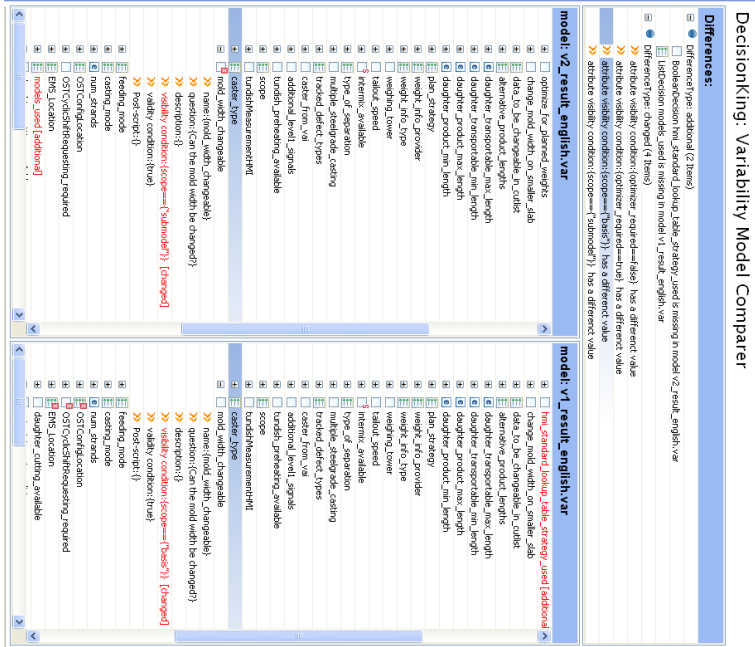
(a) Variability model comparer.

(b) Meta model comparer and updater.

Figure 5.9: Model comparison tools in *DecisionKing*.

Figure 5.9(b) shows the meta-model change propagator in our *DecisionKing* tool which allows updating existing variability models after changes were made to the meta-model. In the lower half of Figure 5.9(b), two different versions of the domain-specific meta-model are compared and the differences are shown in different colors. The tool presents suggestions for actions, which can be carried out in order to update the old meta-model (cf. upper half of Figure 5.9(b)). By carrying out the suggested actions, the meta-model of the variability models can be updated. The tool suggests actions for each difference between two meta-models which can be executed to synchronize them. Adding new elements (asset types, attributes, or relationships) to an existing meta-model is straightforward as they can just be added to the new model without affecting already existing model elements. *DecisionKing*'s meta-model change propagator also supports the deletion of meta-model elements. However, user confirmation is required in such cases as the update will result in the deletion of model elements, their attributes, and relationships. Whenever new model elements are detected, the tool relies on input from the user to distinguish between renaming and addition.

5.6   Extensibility of *DecisionKing*

DecisionKing is based on a plug-in architecture to allow the interaction with arbitrary external tools. There are two kinds of integration facilities:

Integration of external tools into DecisionKing: The plug-in concept allows users to develop and integrate company-specific functionality. We have made use of this concept when developing several extensions together with our industry partner: (i) Variability extraction plug-ins import information about existing assets and their relationships from existing configurations to populate the variability model. (ii) The language used to describe rules and constraints for relationships in variability models is provided as a plug-in. (iii) We have been developing domain-specific model maintenance and evolution capabilities. For instance, we provide model differencing and model synchronization capabilities as plug-ins that support the evolution of variability models.

Integration of DecisionKing with other tools: *DecisionKing* can also be used as an off-the-shelf variability management engine to provide variability management capabilities to existing applications. We have been testing this functionality in a number of case studies such as the runtime adaptation of a plug-in-based system realized in .NET [Wolfinger *et al.*, 2008] as well as the adaptation of service-oriented systems based on runtime monitoring [Clotet *et al.*, 2008].

5.7   Features for Comfortable Modeling

In our experience, user acceptance of modeling tools often depends upon features of the tool, which may not be relevant from a research perspective. Users of the tools look for standard features like

(undo-redo, cut-paste modeling elements, refactoring, searching, etc) which are common in most programming tools. We therefore enhanced *DecisionKing* with some nice-to-have features, which helped us a lot in achieving better response from the tools' users.

5.7.1 Searching

Suitable support for searching for model elements (within a certain model or in all the models in the workspace), which considers the structure of the models and the modeling notations used in *DecisionKing* is required for comfortable modeling. General purpose search utilities would not be able to interpret the modeling constructs, attributes and dependencies in a variability model. As depicted in Figure 5.10, we integrated the search functionality in Eclipse. The search dialog was extended such that it can refer to a domain-specific glossary and looks for synonyms and antonyms of the search queries. The search result page presents the results of the search operation in a structured manner, separated by the types of model elements.
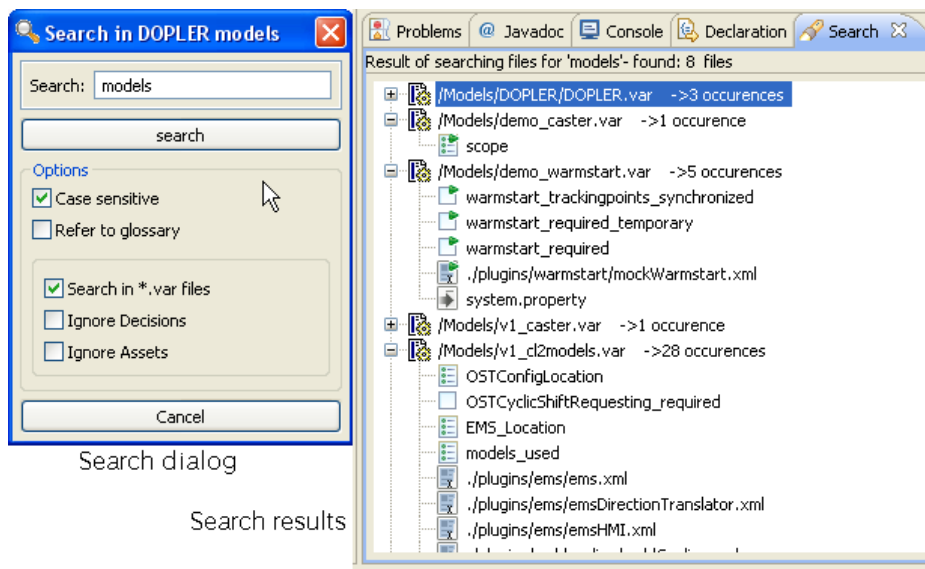

Figure 5.10: *DecisionKing* search dialog and search result page.

5.7.2 Refactoring

Refactoring is changing the structure of a program without changing its functionality. Refactoring within the model-driven software development process means to refactor the corresponding models. Refactoring can be used to restructure and optimize the model without altering the model's behavior.
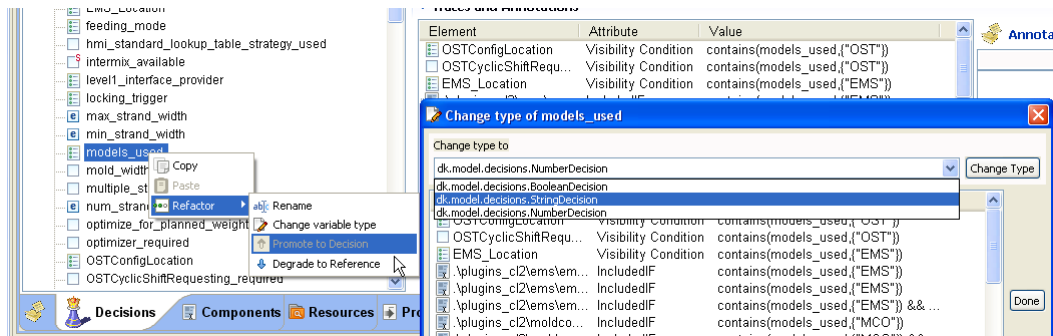
Figure 5.11: Refactoring support in $\mathcal{D}$ecision$\mathcal{K}$ing.

$\mathcal{D}$ecision$\mathcal{K}$ing's refactoring tools can be grouped into three broad categories: *Changing the name of decisions* involves detecting the effect of changing the variable name, and adapting all expressions and rules to the new name. *Changing the logical organization of a model at decision-type level* involves detecting semantic and syntax errors which can occur after such a change. Different functions are defined only for certain types of decisions (e.g., the function `contains(..)` is defined only for enumeration decisions), and changing the type of such a decision leads to syntax errors in the corresponding expressions. *Changing the modeling elements' status* involves turning variables and assets to placeholders and vice versa.

The refactoring utility in $\mathcal{D}$ecision$\mathcal{K}$ing (cf. Figure 5.11) allows the user to choose the refactoring and to pass in the information needed for the selected refactoring. The user then sees the "before" and "after" version of the model and has a chance to respond to any problems flagged by the tool.

5.7.3   Traces

Modelers define the dependencies among model elements in one direction (i.e., Component A *requires* Component B). It is usually helpful to also execute the inverse dependency queries (i.e., which other components require the component B?). If this kind of information is not directly visible to the user, she has to go through all components to check if they require the component B. $\mathcal{D}$ecision$\mathcal{K}$ing provides remedy to this situation by providing a trace viewer, which is an inversed relationship resolver for each model element. For example, if A *requires* B, C *requires* B and D *requires* B, the trace viewer calculates the list of elements which require B, i.e., B *is required by* A, C and D.

5.7.4   Annotations

Models are annotated to add extra information that are not defined in the metamodel. Annotations are textual tags (comparable to post-its), which can be attached to each model element (decisions and assets) and attach semantically rich metadata applicable to a particular application domain that

help further clarify the model elements. *DecisionKing* provides an annotation editor, annotation viewer and annotation-based search dialog to navigate through the model.
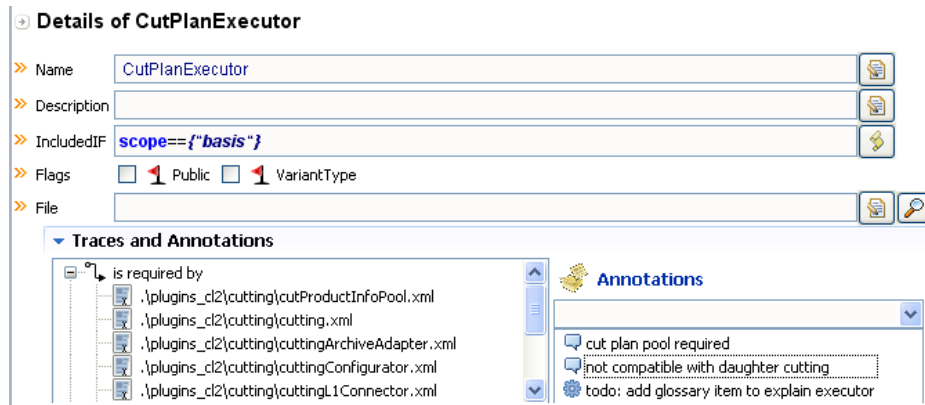
Figure 5.12: Traces and annotations viewer, showing some annotations required by the asset cut-PlanExecutor.

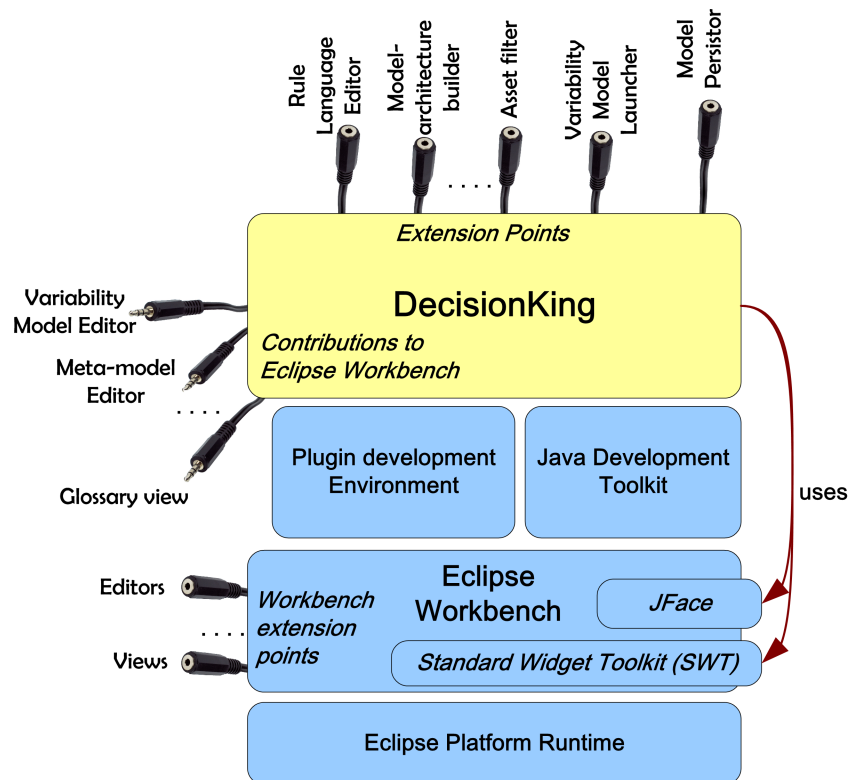## 5.8   Eclipse as the Base Platform for *DecisionKing*

DecisionKing was developed as a plugin for the Eclipse[7] platform. Eclipse is a Java-based, extensible open source development environment. It is a plugin framework and provides a set of services [Fayad *et al.*, 1999] upon which all plugin extensions are created as a basis for building different tools. It also provides the runtime in which plugins are loaded, integrated, and executed. The primary purpose of the platform is to enable other tool developers to easily build and deliver integrated tools. The Eclipse platform itself is a sort of universal tool platform - it is an IDE for anything and nothing in particular [McAffer & Lemieux, 2005].

The Eclipse platform, by itself, does not provide a great deal of end-user functionality. The real value comes from tool plugins for Eclipse that extend the platform to work with the different kinds of resources. This pluggable architecture allows a more seamless experience for the end user when moving between different tools [Clayberg & Rubel, 2006]. It can deal with any type of resources (Java files, C files, Word files, HTML files, JSP files, etc) in a generic manner but does not know how to do anything that is specific to a particular file type.

In addition, the Eclipse platform defines a workbench user interface and a set of common domain-independent user interaction paradigms that tool builders plug into to add new capabilities. The platform comes with a set of standard views which can be extended by tool builders. Tool builders can both add new views, and plug new domain-specific capabilities into existing views. By allowing easy

---

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

extension of basic building blocks such as editors, views, action sets, perspectives, wizards, preference pages, commands, key bindings, undo/redo support, presentations, themes, Eclipse highly reduces the costs of tool development [Gamma & Beck, 2003]. Eclipse exploits the advantages of object-oriented frameworks [Schmid, 1997] to a large extent.

Figure 5.13: *DecisionKing* contributes to workbench extension points and provides extension points.

The basic design of *DecisionKing* was influenced by the extension and extension point metaphor provided by Eclipse platform. An extension point is a declaration made by a plugin to specify that it is open to being extended with new functionality in a particular way. It is defined using a XML schema defining the structure of the meta-data that extensions are required to supply. An extension is the flipside of the coin. It is a declaration provided by a plugin to specify that it provides functionality to extend another plugin. It specifies an XML node, which complies with the schema specified by the extension point. Each plugin contributes to one or more extension points and optionally declares new extension points. Plugins can depend on a set of other plugins and contain Java code libraries and other files. They can also export Java-based APIs for downstream plug-ins, which would otherwise not be visible to other plugins (as each eclipse plugin "lives" in its own plug-in subdirectory).

The Eclipse platform itself is built up with a set of core plugins (*everything is a plugin in Eclipse [Gamma & Beck, 2003]*). The basic setup of an Eclipse application and the position of *DecisionKing* is depicted in Figure 5.13. *DecisionKing* contributes to the extension points of the Eclipse workbench and makes use of the base libraries JFace and SWT.

The Standard Widget Toolkit (SWT) is a platform-independent widget toolkit providing native widgets (button, tree, table, menu, etc). Currently different implementations for Win32, GTK, Motif and Mac are available. SWT integrates easily with other native application and is comparable to OLE under Win32. This library is usually meant to be used instead of Swing.

JFace is an application programming interface based on SWT, which implements the model-view-controller paradigm. Different UI constructs like application window, menu bar, tool bar, content area & status line, tree & table viewers, preference & wizard framework are based on JFace.

5.9   Summary

Product line engineering comprises many heterogeneous activities such as tailoring of the approach to the specifics of a domain, capturing variability of core assets or evolving the product line. The complexity of product lines implicates that tool support is inevitable to facilitate smooth performance and to avoid costly errors. In this chapter, we described *DecisionKing*, an integral part of the DOPLER tool suite which has been developed to provide such integrated support. *DecisionKing* is flexible and extensible to support domain-specific needs.

DecisionKing is a meta-tool for variability modeling. It provides variability modeling environments for a family of domains, rather than a single one. By configuring a meta-tool, one can significantly reduce the amount of time, effort, and resources required to develop and maintain variability modeling tools. *DecisionKing* is also a component framework for variability modeling. *DecisionKing* modularizes the required functionality into components, which encapsulate their internal states and provide services to other components or applications. By extending/completing the framework users can easily produce an application which is customized using application-specific components. A detailed description of the tools and the organization of the functionality follows later in this chapter. Here we briefly summarize the features of *DecisionKing*:

Domain-specific variability modeling Creation of domain-specific specialization of the core-meta model (cf. Figure 4.1) is supported by the *meta-model editor*, which is an integral part of *DecisionKing* and provides features for domain-specific refinements of the core meta-model.

The variability modeling functionality is a central feature of *DecisionKing*. It is supported by the *variability model editor* with standard functionality allowing users to edit, view and refactor variability models. The variability modeling editor is automatically adjusted to the needs of the new domain using the domain-specific meta-model, which gives the modeler all domain-specific modeling facilities–as if *DecisionKing* was particularly developed for their purpose.

Support for executing models: The *model launching environment* is used to "enact" a variability model. Different kinds of model utility tools maybe plugged as needed into this environment.

A *rule language compiler* is plugged into the variability modeling environment in order to allow the modeler to create dependencies among decisions using a rule language. We are currently using a simple, self-knitted rule language compiler for this purpose. This component also provides a rule language editor (including syntax highlighting, code completion, syntax check etc.) so that the user is taken-by-the-hand when modeling.

The *JBoss rule engine* is used to evaluate the rules specified in the model. This component is a third party library that provides an open source and standards-based business rules engine and business rules management system (BRMS). The JBoss Drools engine implements the full Rete algorithm [Forgy & Shepard, 1987] with high performance indexing and optimization.

The *model tester* component is one of the standard components of $\mathcal{D}$*ecision*$\mathcal{K}$*ing*. It presents the decisions contained in the variability model to the user in a suitable form and collects users' decisions. Intuitiveness of the decision taking procedure is improved by presenting the decisions in the form of questions to the users. The user feels comfortable even when working with large models, because the irrelevant decisions are automatically filtered out.

Model consistency checking: The *consistency checking framework* is used to detect inconsistencies between the model elements. This framework uses the rule language compiler to perform plausibility checks between the rules. It also detects cyclic dependencies among model elements.

Domain-specific consistency checkers are needed to deal with the consistency between the models and the software system described by the models. This component is also used to interpret the domain-specific meta-model and make plausibility checks in the variability models based on the meta-model.

Model-manipulation API: A *generic model API* is provided by the variability modeling environment, which allows arbitrary tools to interact with $\mathcal{D}$*ecision*$\mathcal{K}$*ing*, manipulate and use variability models programmatically. Such an API proved to be very useful for different kinds of extensions, which were not pre-planned. This mechanism also enables communication of $\mathcal{D}$*ecision*$\mathcal{K}$*ing* with third party tools, using custom adapters.

This is also useful, whenever the variability model itself needs to be changed at runtime. This is for example required for the management of asset instances at runtime. This component is therefore an extension to the model launching environment.

For example, a *web-based questionnaire* (appendix B) was developed to demonstrate the flexibility of the model launching environment provided by $\mathcal{D}$*ecision*$\mathcal{K}$*ing*. This component transforms a variability model into an interactive web-site. The web-site is only the front-end of the remote server based model launching environment.

Support for model evolution: *DecisionKing* also provides functionality for model differencing. Changes are inevitable and it is not always easy for developers to know what has changed between two versions of variability models. This feature provides relief by allowing the comparison of models. The meta-modeling environment also makes use of this framework to compare different versions of a meta-model. The meta-model comparer can also propagate changes to corresponding variability models automatically using this component.

> ❝ Men have become the tools of their tools. ❞ —*Henry David Thoreau (1817 - 1862)*

> ❝ We shall not fail or falter; we shall not weaken or tire...Give us the tools and we will finish the job. ❞
> —*Sir Winston Churchill (1874 - 1965), BBC radio broadcast, Feb 9, 1941*

Chapter 6

Structuring the Modeling Space and Modularizing Variability Models

Summary *This chapter presents an approach structuring the modeling space by organizing variability models as a set of interrelated model fragments. We discuss the needs for such modularization techniques and provide example scenarios to illustrate the application of model fragments.*

In contemporary software development organizations development teams require a mix of skills. For example, database administrators and security managers have their own languages and tools, as do the network engineers who control the underlying hardware. A typical development team is often quite fragmented. Such environments are inevitable in large-scale systems as single stakeholders can only maintain a small part of a system. This poses additional challenges for evolution.

The fundamental characteristic of many software systems is that they are very large and far beyond the ability of any individual or small group to create or even to understand in detail. If a software system were small, effective coordination could occur because a single individual or small group could direct its work and keep all the implementation details in focus [Kraut & Streeter, 1995]. The dependencies between the communication structure of a development team and the technical structure of a system have been addressed by Conway's law [Conway, 1968, Herbsleb & Grinter, 1999].

6.1 Structuring the Modeling Space

We propose a multi-model approach for structuring the modeling space, which allows working with smaller models, thus reducing complexity of creation, maintenance, and evolution. Changes can be made locally in specific model fragments by the assigned team of experts. The structure also supports the evolution of different subsystems at different speeds. An overview of the multi-model evolution approach is depicted in Figure 6.1.

Back in 1972, Parnas dealt with this by presenting [Parnas, 1972] the criteria of decomposing systems in to modules. He recognized that the process of decomposition was not only a technical division of the product but a division of labor among individuals. Despite the widespread use of modular decomposition, different studies of software developers suggest that they still spend over 50% of their

time communicating with others [Perry *et al.*, 1994]. We apply similar modularization concepts to variability models and also minimize the communication effort between developers by using automated tools.

Development teams of our industry partner have demanded a high degree of flexibility when creating and evolving variability models for not being constrained by other teams. This is for instance relevant when modeling new features or when making local changes to a subsystem. Product line modeling tools should provide capabilities for creating model fragments, defining inter-fragment relationships, and integrating the model fragments. From a high-level point of view, there are two possible mechanisms for specifying model fragments and their dependencies.

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

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

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

6.1.1   Approach Overview

We chose to implement the approach based on precise dependencies, as the engineers of our industry partner have demanded a higher degree of freedom when defining and evolving product line models. They requested more leeway and support for "lazy consistency", i.e., temporary inconsistency during modeling with support for identifying and resolving inconsistencies later. This is particularly important when evolving the product line e.g., by adding new functionality.

The key elements of the approach are as follows:

A *model fragment* describes the reusable assets and their variability of an arbitrary part of the product line (e.g., a set of features, a subsystem, or cross-cutting functionality). Model fragments serve as the basic unit of evolution and are created and maintained by individual stakeholders only loosely coupled with the activities of other stakeholders. Model fragments are never directly utilized in product derivation.
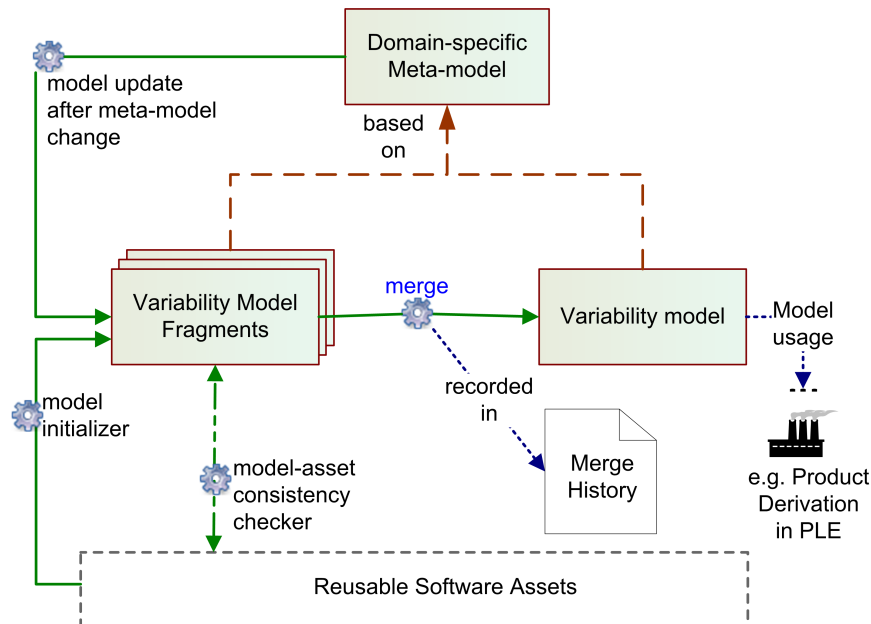
Figure 6.1: Overview of the multi-model approach based on model fragments.

A *variability model* is merged from a set of model fragments at certain points in the product line life-cycle (e.g., before starting product derivation). Unlike model fragments a variability model can be used in product derivation [Rabiser *et al.*, 2007] as all inconsistencies between the constituent fragments have been resolved during merging. The resulting model must not be changed. It is updated by re-merging model fragments.

The *merge history* establishes trace links between the model fragments and the result of the merge process. Model fragment owners use the merge history to revise their individual fragments based on the applied conflict resolution actions to expedite future merge processes.

Variability model fragments are based on a *domain-specific meta-model* which defines the specifics of an organization or domain by defining the types of assets to be reused in the product line together with their attributes and dependencies. The result model of the merge process is therefore also based on the same domain-specific meta-model. Variability models need to be updated automatically after changes to the meta-model.

6.1.2 Model Fragments

A model fragment consists of two kinds of modeling entities (lower half of Figure 6.2): *model elements* and *placeholder elements*. We have adopted concepts from object-oriented programming languages to define the visibility of model elements in model fragments. Similar to `private` and `public`

elements in classes, modelers can specify *public elements* of a fragment to make them visible outside the model (cf. Figure 6.3). Model elements are defined as *private elements* if they are not relevant to other parts of the system or must not be known outside for variability modeling, i.e., they are internal to a subsystem with no direct relationships to elements in other models.
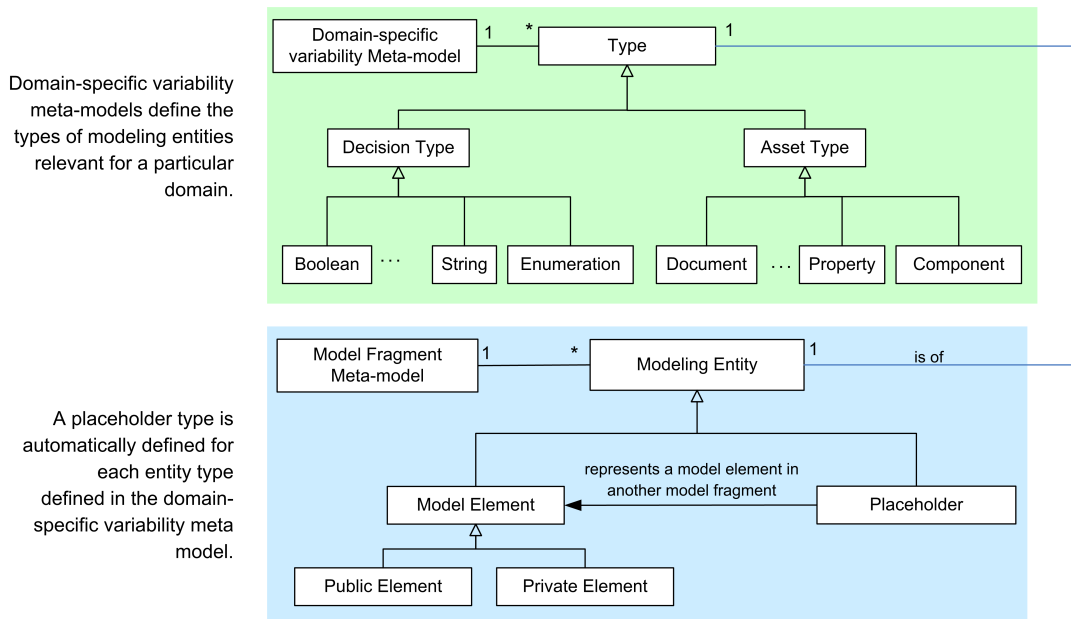
Figure 6.2: High level meta-model depicting different models and their dependencies.

Placeholder elements are comparable to `required` interfaces in a programming language. They have a data type defined in the meta-model and can be seen as tagged and typed variables which are replaced with model elements in other model fragments during merging. Placeholder elements are introduced in a model fragment whenever relationships to elements from other model fragments need to be defined. This is for instance necessary when specifying product composition rules between elements. The explicit location or the exact names of the referenced elements are not needed during modeling to allow loose connections between fragments [Dhungana *et al.*, 2008].

Example: Model fragment 1 in Figure 6.3 contains a placeholder for the decision `archive` that is used to define the dependency between `dbSupport` and `archive`. The decision `dbRequired` in model fragment 2 is a placeholder for a decision defined in another model fragment. To ease evolution the person creating fragment 2 does not need to know the real name of the element `dbSupport` she is referring to. During merging `dbRequired` is replaced with `dbSupport` from model fragment 1 to resolve this ambiguity.

An arbitrary number of model fragments are created for different parts of the system based on the domain-specific meta-model. The model fragments can evolve independently and at different speeds.

Tools semi-automatically merge fragments into a single variability model whenever required. In the example shown in Figure 6.5 the product line engineer creates a single model at time `t1` based on the versions of the model fragments available at this time. Project managers can use the merged model for deriving a product. The merged model must not be changed during derivation. After a set of changes to various fragments another merge process is initiated at time `t2`. The re-merge benefits from the conflict resolutions in the merge at `t1`. The merged model can be used for another product derivation. In the example, the domain-specific meta-model changes from `v1` to `v2`. This will require the update of existing fragments to ensure consistency with the new meta-model `v2`.
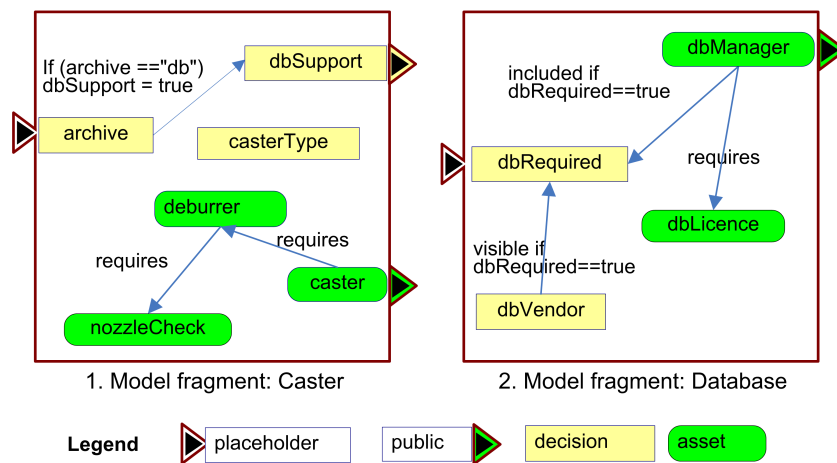
Figure 6.3: Example of model fragments.
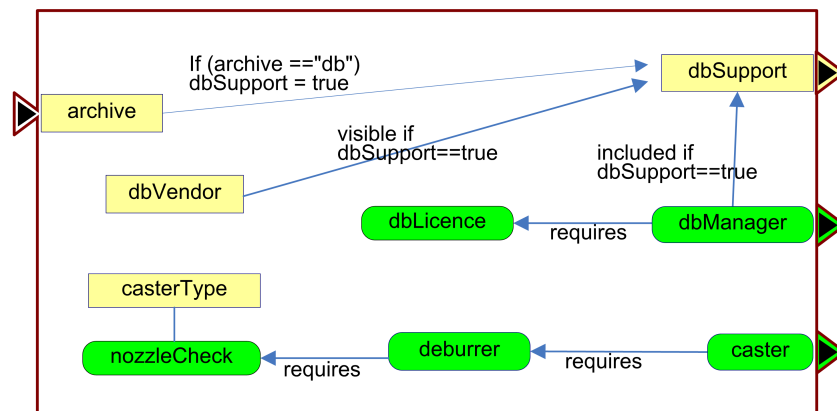
Figure 6.4: The result of merging the two fragments depicted in Figure 6.3.

103
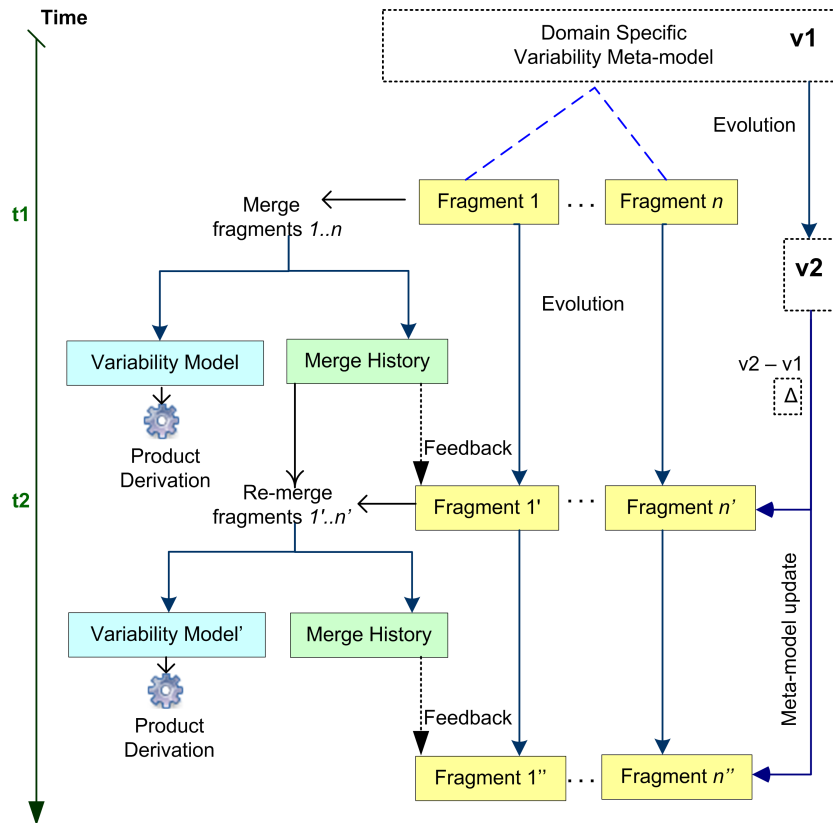
Figure 6.5: Model evolution occurs at variability model and corresponding meta-model level.

6.1.3   Fragment Merging

It is noteworthy to mention that decomposition implies recomposition [Grinter, 1998], which means working with small models requires techniques to actually create a large model from the small ones. Model fragments are incomplete as they represent only a partial view of the system. The links to other parts of the system – modeled using placeholders – need to be resolved before a single model can be generated for product derivation. During merging the elements of the constituent model fragments are collected in a new model and the placeholders are replaced with corresponding model elements from other model fragments. It is important to note that the source model fragments remain unchanged during merging.

Similar collaborative feature modeling approaches were presented by Tang *et al.* [Tang *et al.*, 2007], where the authors present a process for the non-locked multi-client collaborative feature modeling, consisting of a feature adjustment method to solve the conflicts and an enhanced naming mechanism to preserve the design intentions. Table 1 lists four types of merge conflicts together with a resolution

Table 6.1: Summary of different types of merge conflicts and possible resolution.

Merge conflict	Resolution strategy
Multiple occurrences of same identifier	Rename involved elements or drop all others but one
Name mismatches	Synonym check with glossary
	Rename one of the mismatching elements
Multiple definitions	User confirm semantic equality
	Delete one of the instances
No matching elements for place-holders	Automatically suggest a candidate resolution. If no matches are found, placeholders remain unchanged

strategy:
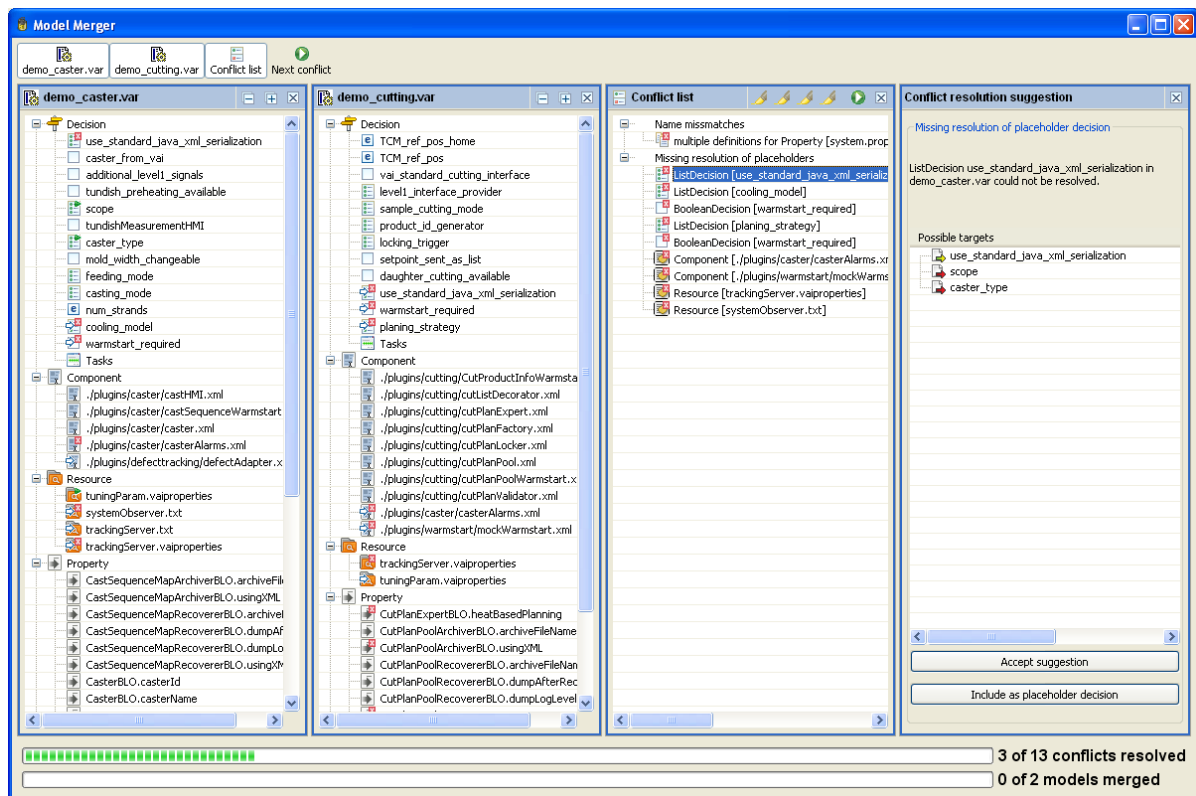
Figure 6.6: $\mathcal{D}ecision\mathcal{K}ing$ Variability Model Merger.

Multiple occurrence of the same identifier. It is essential that all the elements in a model have a
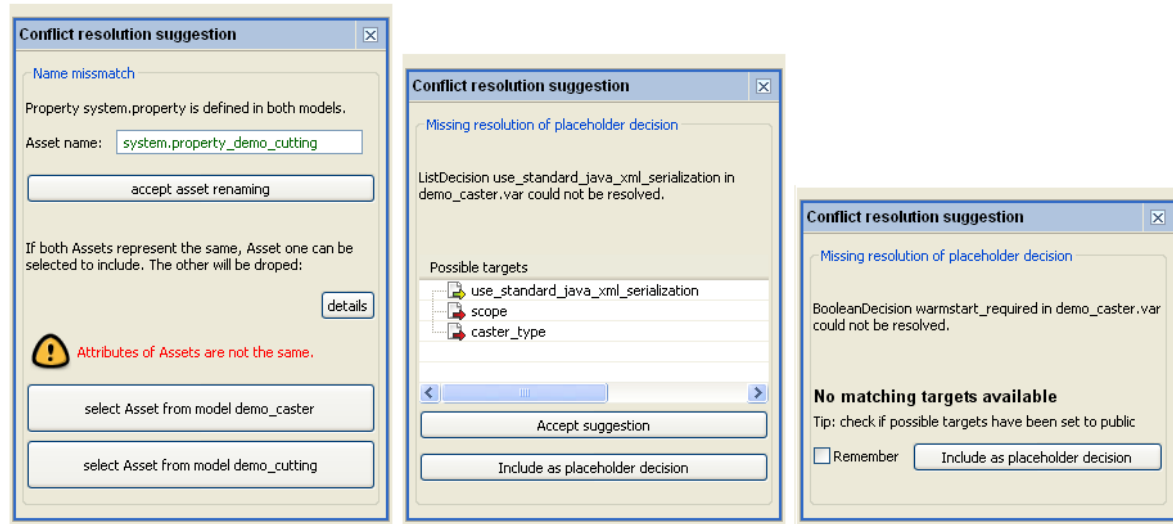
Figure 6.7: Merger suggestions for resolving conflicts.

unique identifier. However, as the modelers are not aware of other model fragments during modeling, elements in different model fragments can have the same name. This leads to a conflict during merging. At least one of the conflicting elements has to be renamed or dropped.

Name mismatches. The name of a placeholder might not match the name of the intended element. For example, a model fragment might contain a model element with the name `admin` and some other fragments may refer to the same element using the name `sysAdmin`. Such cases are difficult to resolve fully automatically and we rely on the human expert during merging to confirm the semantic equality of the used element names. However, our tools adopt domain-specific glossaries defining synonyms of the used names to ease merging in these cases.

Multiple definitions. Different model fragments may define the variability of a common part of a system. This can for example happen when shared components are used by more than one subsystem, and several subsystem owners decide to model the shared components' variability as a part of their subsystem. Our algorithm detects all element instances (based on naming conventions, types of elements, and relationships among elements) and includes only one instance in the merged model. For example, whenever a component with the same name is contained in more than one model fragment, the user either decides to (i) rename one of the components before merging; or (ii) to include the component only once in the merged model.

No matching elements for placeholders. It is also possible that no modeler feels responsible for a certain part of the system. As a result several model fragments might define placeholders for which no real model element exists. Again user intervention is required to resolve the problem. The user selects a binding element from a list of suggested candidate elements. If no binding element is available, the

resulting variability model will still contain unresolved placeholders.
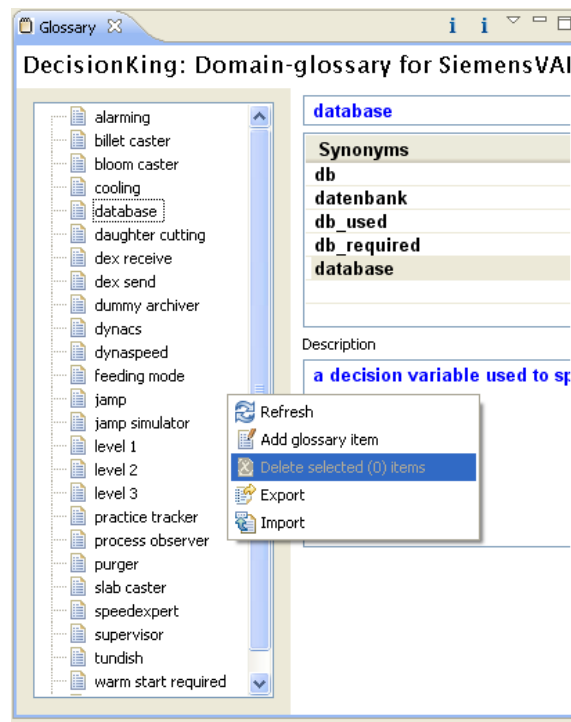
Figure 6.8: *DecisionKing*'s domain-glossary tool for synonym checkup during merging.

Example. Figure 6.4 depicts the result of merging the model fragments from Figure 6.3. Ideally, all reference elements match a public element in other models. However, as the model fragments are initially created without explicit coordination and are based on "loose references" various conflicts can occur during merging. Whenever model elements are renamed (in case of name conflicts), deleted or dropped (in case of multiple occurrences), their attributes, constraints and conditions also need to be updated accordingly. Because the placeholder element dbRequired was mapped to dbSupport, for example, the following condition (in model fragment 2 of Figure 6.3)

```
visible if dbRequired==true
```

was automatically changed to the following condition during the merge process (cf. Figure 6.4).

```
visible if dbSupport==true
```

6.1.4 Merge History

During the model merging process, we record the applied changes and bindings in a merge history. There can be two kinds of changes, i.e., changes made automatically by the merging tool and changes

made explicitly by the user. The merge history enables three important features:
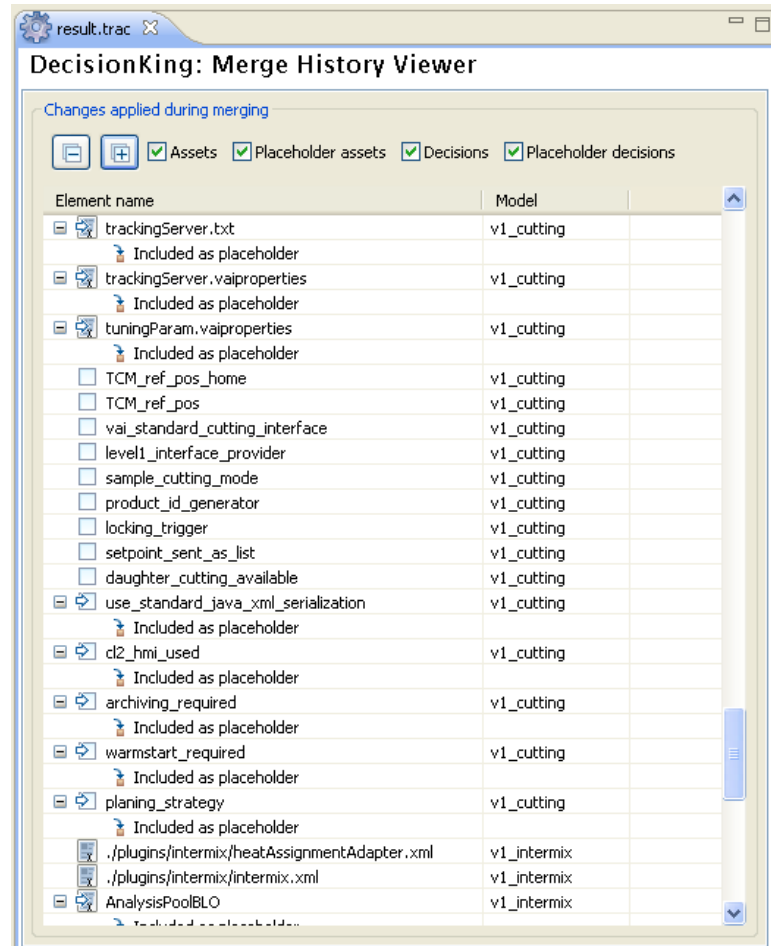
Figure 6.9: *DecisionKing*'s merge history viewer tool.

Forward and backward traceability. The merge history links the model fragments and the variability model to support forward traceability ("how is my model fragment used in the variability model?") and backward traceability ("from which model fragment does a certain element in the variability model originate?").

Feedback to model fragment owners. The original model fragments remain unchanged after merging. However, model fragment owners are informed about conflict resolution changes such as deleting elements, references, and relationships applied during merging to avoid that modelers slowly disconnect from each other. Modelers get information regarding the change actions that were necessary during merging (e.g., which elements had to be renamed). They may decide to revise their model fragments

based on this feedback (cf. table 6.2). This helps modelers to converge and agree on definitions in the model fragments.

Repeatability of merging. In case of frequent changes to the fragments and a high number of fragments repeating the merging process each time from scratch can be tedious. Whenever the merge process has to be repeated after changes to model fragments the merge history is used to replay the previously taken change actions with minimal user intervention (quick re-merge). Besides user choices made during merging, the merge history contains all change actions automatically performed by the merging tool (e.g., renames, reference-mappings, or changes in attributes or relationship links).

Table 6.2: Different types of merging strategies and possible feedback.

Merge strategy	Feedback to fragment owner
Element renamed	Rename element in the fragment to ensure positive match
Element deleted	Either drop element from fragment or change element to a placeholder
Missing resolutions of reference	Either remove the reference element or change from reference to definition type

6.2   Checking the Consistency of Model Fragments and Assets

The core assets of a product line evolve continuously to address changes such as new customer requirements, technology changes, or necessary refactoring. For example, a large component may be divided, a component may be moved to another subsystem, or new relationships between components may be established. It is therefore essential to understand, model, and maintain the links between the product line's variability models and the asset base.

The adoption of a product line approach involves the creation of an initial product line architecture model. The effort required for building this model can be minimized by tools analyzing the existing assets and creating an initial model automatically. Depending on the variability implementation mechanisms, there are different ways of identifying variation points and variable assets. For example, implementation-level variability constructs (such as inheritance or parameterization) can be identified by analyzing the existing implementation. We have created a tool that automatically generates an initial product line variability model for our industry partner by analyzing the Spring XML configuration files of the existing system.

Product line models have to be kept consistent with the architecture during maintenance and evolution. Engineers need to frequently change the architecture and the variability model fragments (e.g., when introducing new variants). Inconsistencies resulting from such changes need to be automatically detected and fixed to support the co-evolution of the architecture with the respective model fragments and vice versa. Changes to software components of the product line architecture have to be propagated

to the models. Similarly, models should not simply be changed without making corresponding changes to the architecture. We have developed tools for automatic change detection that give developers and product line engineers instant feedback about inconsistencies. We found that in many cases it is possible to keep track of the changes in the underlying core assets and to synchronize the models automatically because the elements in the variability models directly map to the existing assets. Our tool can currently detect two types of inconsistencies:

Orphaned model elements and links. A model may contain elements that have been deleted or renamed in the asset base. In order to resolve this inconsistency, either the obsolete model element is deleted or the asset base is changed to match the model. The model may also contain dependencies between elements that are no longer correct or available.

Missing model elements and links. It is also possible that not all parts of the system are already covered in the model. Such cases happen, e.g., if components are added to the asset based or modified to meet the needs of daily business. Our tools automatically detect such inconsistencies and make the users aware of them.
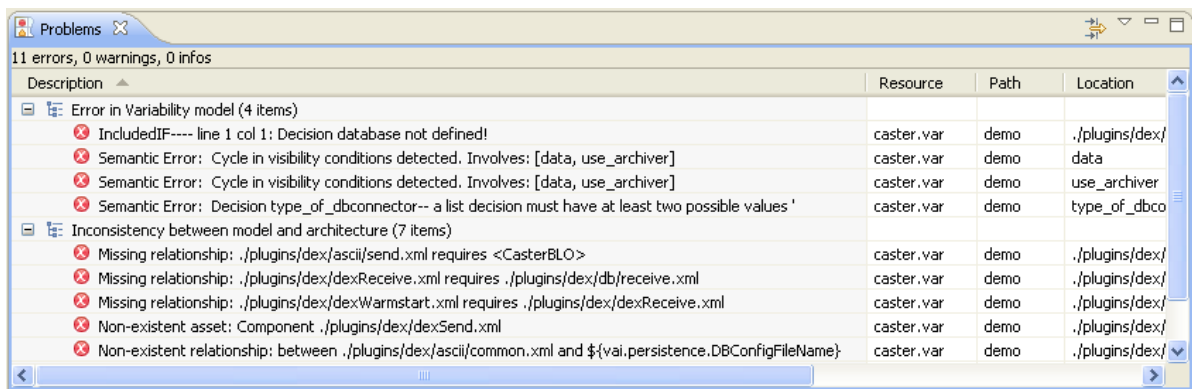
Figure 6.10: $\mathcal{D}ecision\mathcal{K}ing$ problem viewer, showing the different kinds of inconsistencies detected by model-architecture-synchronization tool.

Figure 6.10 shows the error viewer in our $\mathcal{D}ecision\mathcal{K}ing$ variability modeling tool, which displays inconsistencies between the model and the architecture. The tool uses already existing models and artifacts to find inconsistencies: Whenever models are changed, existing architectural elements are used as a reference for comparison. Whenever architectural elements are changed, the existing models serve as a lookup table. For example, when a new variant is introduced by changing the variability model, the tool ensures that there exists an artifact with the same name and structure (together with dependencies to other artifacts). Similarly, when a new component is added to the architecture, the tool automatically looks for its existence in the available variability models.

6.3   Application of Model Fragments

No matter which modeling approach is followed, developing a single model of a product line is practically infeasible due to the size and complexity of today's systems. The high number of features and components in real-world systems means that modelers need strategies and mechanisms to organize the modeling space. *Divide and conquer* is a useful principle but the question remains which concrete strategies can be applied to divide and structure the modeling space. There are many ways for modeling and managing variability but the basic challenges remain: Product line engineers need to define the variability of the *problem space*, i.e., stakeholder needs and desired features; the variability of the *solution space*, i.e., the architecture and the components of the technical solution; and the dependencies between these two. Regardless of the concrete modeling approach used there are several options to structure and organize the modeling space:

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

Decomposing into Multiple Product Lines. On a larger scale complex products are often organized using a multi-product line structure [Reiser & Weber, 2006]. For example, there may be separate product lines for different target customers, e.g., mobile phone product lines for senior citizens, teenagers, and business people [Jaaksi, 2002]. Other examples are complex software-intensive system such as cars or industrial plants with *system of systems* architectures, which may contain several smaller product lines as part of the larger system. Models have to be defined for each of these product lines and kept consistent during domain and application engineering. This strategy often means that different stakeholders create variability models for the product line they are responsible for.

Structuring by Asset Type. Another way of dealing with the scale of product line models is to structure the modeling space based on the asset types in the domain. Separate models can then be created for different types of product line assets. Examples are requirements variability models based on use cases [Halmans & Pohl, 2004], architecture variability models [Dashofy *et al.*, 2001], or documentation variability models for technical and user-specific documents [John, 2001]. This approach is in line with orthogonal approaches [Pohl *et al.*, 2005] that suggest using few variability models that are related with possibly many asset models. Structuring by asset type allows managing variability in a coherent manner. It is however important to consider the dependencies between the different types of artifacts which can cause additional complexity.

Following the Organizational Structure.   This strategy suggests to follow the structure of the organization when creating product line models. Different stakeholders are interested in different concerns of a product line [Dolan *et al.*, 1998]. In many organizations architectural knowledge is distributed across different stakeholders independent of their roles and responsibilities in the development process. Conway's Law [Conway, 1968] states that "...organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations". In a multi-team environment, individual teams collaborate closely on certain aspects of a product line. It can thus be a good strategy to structure the product line modeling space based on the team structure to reflect the modeling concerns of the involved stakeholder groups. However, creating product line models driven by stakeholders can easily increase the redundancy in models.

Considering Cross-cutting Concerns.   Using concepts from aspect-oriented development to structure product line models is helpful when many crosscutting features need to be described. Aspect-oriented product line modeling can be used to model both problem and solution space variability. For instance, Völter and Groher [Voelter & Groher, 2007] describe an approach that involves creating a model of the core features all products have in common and defining aspect variability models for product-specific features shared by only some products. Complex aspect dependencies can however lead to difficulties of managing their interaction.

Focusing on Market Needs.   Structuring the modeling space can also be driven by business and management considerations, e.g., from marketing [Kang *et al.*, 2002] or product management perspectives [Helferich *et al.*, 2006]. Focusing variability modeling on business considerations eases the communication with customers. If combined with other strategies this approach can support the communication between customers and sales people. If following this strategy in pure form, models often are unrelated with the technical solution thus leading to problems when trying to understand the actual realization of the variability.

6.4   Summary

Supporting modularization was a critical success factor for our approach. The approach is based on a simple assumption: a small model is easier to maintain than a large one. Instead of creating a single large product line variability model we use *model fragments* to describe the variability of selected parts of the system. These model fragments represent the units of evolution in our approach [Dhungana *et al.*, 2008]. Engineers in different teams maintain and evolve model fragments describing particular parts of the product line. The approach supports the demands of real-world development processes as different teams can work on variability model fragments describing the parts of the system they know best.

In our approach model fragments are thus created and evolved without explicit coordination to achieve the required leeway. These fragments need to be merged into one variability model before con-

crete products can be derived from the product line. The resulting model has the same basic structure as the constituent model fragments. Merging works even if the fragments are overlapping – i.e., they share certain elements – as no black box is assumed during merging.

DecisionKing allows the creation of variability model fragments. This is an unique feature of *DecisionKing*, which supports decentralized and unsynchronized creation of variability models structuring the modeling space. This provides the necessary leeway for modeling purposes.

A *domain-glossary* is used, so that the decentralized creation of model fragments can be coordinated with respect to the terms and notations used for modeling. The glossary consists of explanations for the different terms, and their synonyms.

Model merger component merges the different model fragments created by multiple teams into one complete variability model. This step is necessary as the model fragments consist of unresolved references and represent only a partial variability model. The model merger component is responsible for semi-automatic resolution of the conflicts which occur during merging.

A *merge log* is also created during the merging process, which records the merging process. This log is used for establishing traceability between the model fragments and the merged model. The merge log viewer component displays the traceability information to the user and also allows for propagating changes to the model fragments based on the user interaction during the merge process.

" Divide and rule, a sound motto. Unite and lead, a better one. " —*Johann Wolfgang von Goethe*

Part III

Evaluation

Chapter 7

Evaluation Plan

Summary *We applied our modeling approach and our tools in several research projects to evaluate our approach and to demonstrate the applicability of our tools. In this chapter, we present an overview of three major case studies and explain how the different aspects of these case studies are related to our research questions.*

Back in 1985, Redwine and Riddle [Redwine & Riddle, 1985] reviewed a number of software technologies to see how they develop and propagate. In their analysis, they elaborate on different phases (see Table 7.1) of research in software engineering (SE). The Redwine-Riddle data suggests that around 15-20 years of evolution are spent in concept formation. It is therefore obvious, that the time span of a PhD research project is not long enough to go through all the stages of SE research. We have therefore concentrated on the first four phases (cf. Table 7.1) of research within the context of this PhD project. Continuation of the project for the commercialization and popularization of the results is part of future work.

Table 7.1: Phases of software engineering research [Redwine & Riddle, 1985].

#	Phase	Description
1.	Basic research	Investigate basic ideas, frame critical research questions.
2.	Concept formulation	Circulate ideas informally, publish solutions to specific subproblems.
3.	Development and extension	Make preliminary use of the technology, generalize the approach.
4.	Internal enhancement	Extend approach to another domain, use technology for real problems.
5.	External enhancement	Similar to internal, but involving a broader community of people who were not developers.
6.	Popularization	Develop production-quality, and commercialize.

Although our work primarily focused on automating the process of creating customer-specific software systems at Siemens VAI, the modeling concepts [Dhungana *et al.*, 2008, Dhungana *et al.*, 2007a] and tools [Dhungana *et al.*, 2007c, Dhungana *et al.*, 2007d] introduced to this project soon attracted

the attention of many other companies and researchers. Several large companies like Bosch[1], Mälardalen Västeras University[2], KEBA[3] and BMD[4] have showed serious interest in our research results. We were also invited by several research institutions like the Irish software engineering research center LERO, Siemens CT and University of Namur to introduce and present our work. We have tested and applied the outcomes of this research project in several other companies in Austria [Dhungana *et al.*, 2007b, Froschauer *et al.*, 2008, Wolfinger *et al.*, 2008]. In these projects, we used our modeling approach to explicitly capture developers' decisions in models and use the models to automate configuration and runtime adaptation of systems [Clotet *et al.*, 2008, Froschauer *et al.*, 2008, Wolfinger *et al.*, 2008].

7.1 Evaluation Case Studies

In this thesis, we describe in detail three case studies where our approach and tools have been applied. For each case study we describe (i) the domain of interest, (ii) technical background, (iii) variability implementation mechanisms and (iv) the challenges for developers and architects. We then revisit the research questions and show in each case, how the research issue was dealt with in the case study. The three research questions are then dealt under three sections.

Modeling: We demonstrate the flexibility of our approach by describing how we modeled the variability of software in different domains.

Tool support: We describe how *DecisionKing* was exploited for variability modeling and give an overview of tool extensions required in each case.

Modularization: We describe how creation and maintenance of variability models was made easier by structuring the modeling space using model fragments.

An overview of the different case studies is presented in Table 7.2.

7.1.1 Modeling Variability of Continuous Casting Automation Software

Context: Siemens VAI[5] is the world leader in engineering and plant-building of iron, steel and aluminum industries. The company has developed and maintains a product line of steel plant automation software (CL2). About 30 software engineers are involved in developing and maintaining this system. The size of the software is about 1.5 million lines of code (mainly Java).

Objectives: The main objective of our collaboration with Siemens VAI is to provide tool support for modeling CL2's variability and to automate the product derivation process. Siemens VAI uses

---

[1] http://bosch.de
[2] http://www.mdh.se/
[3] http://keba.com
[4] http://bmd.com
[5] http://www.industry.siemens.com/metals/en/

Table 7.2: Overview of case studies and evalutation aspects.

	Case study 1	Case study 2	Case study 3
Domain	Steel Plant Level 2 Automation Software	IEC 61499 Industrial Automation Systems	Service-oriented Systems
Research partner	Siemens VAI, Linz, Austria.	Fachhochschule Oberösterreich, Wels, Austria.	Universitat Politècnica de Catalunya, Barcelona.
Technical background	XML component descriptions and Java properties, implemented using Spring Component Framework.	Function block based systems conformant to the standard IEC 61499.	Multi-stakeholder distributed systems: specified using i* models and implemented as services.
Variability mechanisms	Steel plant automation software is configured to match the needs of different customers. We deal with variability of components and their configuration parameters.	Industrial automation systems are reconfigured at runtime. We deal with variability of functionality that can be provided by a particular system.	Service-oriented systems are monitored and adapted at runtime. We deal with variability in specification (i* models) and the implementation (services).
Challenges	Configuration is tedious and error-prone due to the high number of components, configuration parameters and dependencies among them.	Reconfiguration at runtime requires the user to understand dependencies among function blocks in (usually) highly complex networks.	It is difficult for service providers to keep track of which services are needed under which condition (runtime parameters).

the Eclipse platform for the development of the CL2 system. Seamless integration of SPLE tools with their Eclipse-based development environment was one of the prerequisites to successfully introduce a product line approach. It was also essential to develop and integrate tool extensions supporting consistency checking and synchronization between product line models and the CL2 system components.

Relation to research questions:

RQ1 We model variability of Components (Spring XML files), Properties (Configuration parameters), Resources (Configuration files) and Documents (Technical specification).

RQ2 We developed *DecisionKing* extensions to automatically create variability models, to auto-

maticially check for consistency between the models and the architecture.

RQ3 The modeling space was structured by differentiating between different layers of architecture and types of decisions (business and technical). We also considered the organizational aspects of Siemens VAI and structured the modeling by considering multiple teams involved in the development.

7.1.2 Modeling Variability of IEC 61499 Industrial Automation Systems

Context: In cooperation with FH Oberösterreich Research[6] we investigate the usefulness of product line techniques in the domain of industrial automation systems (IAS). Such systems are usually based on a distributed architecture consisting of multiple physically and/or logically distributed components. More and more IAS are based on the emerging standard IEC 61499 which provides a component-based framework for automation systems and standardizes the use of function blocks in distributed industrial process measurement and control systems.

Objectives: Our project aims at defining a model-based reconfiguration process for IAS. IEC 61499 provides low-level support for reconfiguration of applications at run-time. This means that a lot of technical details need to be known by the user. Our objective is to explore the advantages of using product line variability models for run-time reconfiguration of IAS.

Relation to research questions:

RQ1 We model variability of function blocks and function block instances. We differentiate between design time and runtime entities.

RQ2 We developed $\mathcal{D}$ecision$\mathcal{K}$ing extensions to keep track of runtime instances of function blocks based on a platform specific meta model.

RQ3 Model fragments were created to differentiate design time assets and their runtime instances.

7.1.3 Modeling Variability of Service-oriented Systems based on i* Models

Context: In cooperation with Universitat Politècnica de Catalunya, Barcelona[7], we investigate the usefulness of product line techniques in the domain of service-oriented systems. We have been using the i* language [Yu, 1996] to model a service-oriented multi-stakeholder distributed system in the travel domain to validate the usefulness of i* for that purpose.

---

[6] http://www.fh-ooe.at/fh-oberoesterreich/fe/forschung/fe-wels.html
[7] http://www.upc.es/

Objectives: A major goal of the project was to enhance i* with capabilities for variability modeling in the context of our MSDS framework. In order to achieve this goal, we developed capabilities for monitoring service-oriented systems at runtime and made adaptations to the system based on the knowledge modeled in variability models.

Relation to research questions:

RQ1 We model variability of stakeholder goals, services fulfilling these goals and the instance of services.

RQ2 We developed $\mathcal{D}ecision\mathcal{K}ing$ extensions to monitor service oriented systems at runtime. We also introduced the concept of monitoring scripts in the form of Java-script.

RQ3 Different service providers create their own variability model fragments, which provides easier creation and better maintenance facilities than creating one large model.

7.2 Other Application Areas

We have also carried out three other case studies, which **we do not describe** in this thesis. Variability modeling of an enterprise resource planning system is described in detail in [Rabiser, 2009]. Dealing with variability of the domain-specific language MONACO is described in detail in [Wirth, 2008]. Variability modeling of eclipse-based applications is described in [Grünbacher *et al.*, 2008].

7.2.1 Variability Modeling of an Enterprise Resource Planning System

This case study was carried out together with BMD Systemhaus GmbH[8], at the Christian Doppler Laboratory for Automated Software Engineering[9]. BMD is a medium-sized company offering ERP software products mainly to SMEs in Austria, Germany, and Hungary. In cooperation with BMD we have developed a set of usage scenarios demonstrating the need for an integrated approach that uses variability models, dynamic plug-in extensibility, and architecture reconfiguration mechanisms. The scenarios are motivated by the ERP domain and BMD's market environment. In different papers [Wolfinger *et al.*, 2008, Rabiser *et al.*, 2009], we outline the scenarios and discuss the benefits of our runtime adaptation approach. We present an approach demonstrating the benefits of integrating those plug-in platforms and variability modeling techniques. The plug-in platform provides extensibility as well as runtime reconfiguration and adaptation mechanisms on the .NET platform. Automatic runtime adaptations are attained by using the knowledge documented in variability models.

---

[8]http://www.bmd.com
[9]http://ase.jku.at

7.2.2   Dealing with Variability of the Domain-specific Language MONACO

This case study is ongoing work at the Christian Doppler Laboratory for Automated Software Engineering[10], in cooperation with KEBA[11]. Here we are investigating the usefulness of variability modeling techniques, to create end-user specific visual editors for the domain specific language MONACO [Prähofer *et al.*, 2008]. Variability modeling techniques are used to define different variants of components, routines, and parameter settings. $\mathcal{DecisionKing}$ is employed for modeling dependencies and constraints between program variants as well as for representing higher-level configuration decisions together with their impacts. Finally, a model specifies how higher-level decisions, program variants, and elements of control programs are presented to different types of users in interactive interfaces. From those models highly customized end-user programming environments are generated.

7.2.3   Variability Modeling of Eclipse-based Applications

We decided to swallow our own medicine, i.e., to treat our product line tool suite as a product line by modeling its variability and to benefit from its capabilities for customization [Grünbacher *et al.*, 2008]. We applied the DOPLER tools to manage the variability of the tool suite and to automatically generate end-user tools for customizing DOPLER. For this we parameterized the DOPLER tools with a meta-model for modelling variability in Eclipse plugins. In the second step we developed a variability model for DOPLER. Finally, we used our enduser tool to create variants of the tool suite.

7.3   Validity and Limitations

In software engineering (SE), it is often not sufficient to just consider the technical side of a problem without considering how the solution affects existing organizational processes. This is mainly because of the presence of the human factor in SE, which makes research in SE somewhat different from research in other fields of computer science. It is therefore very important to consider the issues of how to integrate solutions in the current practices of different stakeholders how to get people to agree that a particular technical solution is beneficial for them. For this reason, research in software engineering often lacks quantitative experimental validation results and does not have well-understood guidance for researchers [Shaw, 2001]. In most cases Software engineering researchers don't write explicitly about their paradigms of research and their standards for judging quality of results [Shaw, 2003]. They are motivated by practical problems, and key objectives of the research are often quality, cost, and timeliness of software products [Shaw, 2003]. Software engineering researchers have criticized common practice in the field for failing to collect, analyze, and report experimental measurements in research reports [Tichy *et al.*, 1995, Shaw, 2001, Shaw, 2003, Tichy, 1998].

---

[10] http://ase.jku.at
[11] http://www.keba.com

We are aware of the need for a quantitative validation of the approach presented in this thesis. However due to practical reasons (circumstances not under our control), we have concentrated ourselves in qualitative validation. Qualitative studies are subjective, and therefore do not provide any proof for applicability of the approach. However, they help us in answering some of the important questions which are prerequisites for a proper validation. A necessary (but not sufficient) condition for success is the question whether the method is transferable to industry. Apart from that it is equally important to find out if the method can be used by other organizations or companies, and/or in other sub domains or domains.

Although industrial validation of new approaches is the best way to demonstrate their suitability, doing so is easier said than done. Industrial validation requires the cooperation of industrial partners, which poses inherent time and money constraints on a study.

> " Computer scientists publish relatively few papers with experimentally validated results. The low ratio of validated results appears to be a serious weakness in CS research. This weakness should be rectified. "
> —*Mary Shaw. The coming-of-age of software architecture research. (ICSE 2001).*

Case Study 1: Modeling Variability of Continuous Casting Automation Software

Summary *In this chapter we illustrate the use of our tools and techniques to model the variability of process automation software in continuous casting steel plants. The case study was carried out with Siemens VAI over a time span of 3 years, and includes details about the role played by DecisionKing in analysis, modeling and maintenance of variability models. We report on interesting aspects of variability model and corresponding meta-model evolution. The case study is concluded with a brief summary of our experiences and lessons learned.*

8.1 Introduction to Siemens VAI

Siemens VAI[1] is the world's leading engineering and plant-building company for the iron, steel and aluminum industries. The company offers the complete spectrum of related technological and automation solutions, backed by a full range of expert metallurgical services. With the recent acquisition by Siemens, VAI has been integrated in the Siemens Industrial Solutions and Services Group (I&S), a global player in the construction of industrial plants. Siemens VAI provides completely integrated solutions for the entire iron and steel production route. By providing modular-designed, expandable and upgradeable automation packages (both integrated and stand-alone solutions for all automation levels), the software solution is equipped with the required flexibility to deal with project or customer-specific modifications.

In our cooperation with Siemens VAI we focus on automation software for their continuous casting technology, in particular, the caster level 2 automation software (CL2). Today continuous casting accounts for approx. 95% of steel production worldwide and is the leading technology used to convert steel from a molten into a solid form. CL2 is the software layer between the level 1 automation, i.e., the machine-oriented basic automation, and the level 3 software, which handles enterprise resource planning, product planning, and other managerial aspects. The task of a typical CL2 system is to ensure that product quality and production are independent of human influences. Furthermore, data tracking and analysis capabilities facilitate the continuous improvement of process knowledge and therefore process optimization. CL2 provides capabilities such as material tracking, process supervision, and process

---

[1]http://www.industry.siemens.com/metals/en/

optimization [Federspiel *et al.*, 2005]. CL2 software is a crucial technology in Siemens VAI's continuous casting products. The advanced process automation and process optimization capabilities of CL2 allow Siemens VAI to guarantee steel quality and to maintain a competitive advantage.

An example of the CL2-client (operator view) is depicted in Figure 8.1. The two screenshots display noticeable differences on the GUI, although they are created using the same set of components, which have been configured differently for different customers. This is an example of simple variability in the software. There are several thousand setscrews in the whole system, that allow fine tuning of the software to exactly match the needs of potential customers. Our goal in the project is to largely automatically generate/configure customer-specific CL2 solutions based on customer decisions.
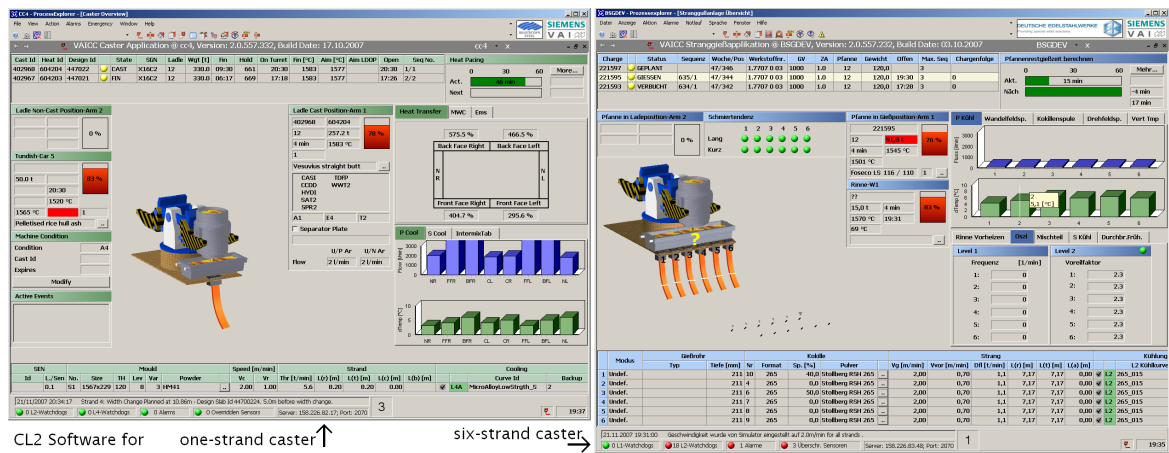
Figure 8.1: CL2 software configured for two different customers, showing the differences in the GUI.

Siemens VAI has developed a stable process optimization system that is based on the application of already well proven packages. Flexible process models allow off-line tuning and simulation prior to release to the production system. Therefore no software modifications are needed to adapt model behavior. In this way, the CL2 System guarantees minimum plant downtime through maximized utilization of pre-tested, pre-configured and proven components.

Due to the dramatic increase in demand for steel products, the main challenge for steel producers and plant builders is to ensure short project realization periods as well as fast and smooth startups for new and especially for revamped production units [Federspiel *et al.*, 2005]. The number of startup projects at Siemens VAI has increased drastically since the 1970s. There are basically two ways of dealing with such growth – either grow the size of the developers team with the growing market demand, or develop the product more intelligently with the same team. Siemens VAI has chosen the latter path and has applied the "plug & play" philosophy to continuous casting by developing VAI's "connect and cast" feature[2].

---

[2]"connect and cast" is a registered trademark of Siemens VAI

8.1.1 Architecture of CL2 System

CL2 is based on a state-of-the-art component based software architecture. It consists of about 650 components, of which most of them are interchangeable. Each component has a defined interface and most of the components can be tested individually. It builds upon Java J2SE servers which control the continuous casting process. The server functionality is split into more than 20 standalone processes, all of which run in their own Java Virtual Machine. For example, there is one supervisor process for managing other processes, one process for establishing communication between others, another one for monitoring the system and several processes for each main functionality, e.g., Material tracking, L1-Connection, L3-Connection, cooling models, ASTC etc. These processes can be updated on the fly as CL2 includes reconnect functionality for each process.

The code base consists of around 350 KLoC of Java, around 200 KLoC C++. Additional 5 KLoc to 130 KLoC of Java code is developed per project. Currently more than 1000 parameters are used to configure the components. Basically CL2 is a client-server application. At Siemens VAI, we distinguished between three kinds of components.

Passive Components are components that only react on input and therefore need some impulse from outside. They do not function on their own. Examples of such components are SCO, ASTC, EQX, VAIQ etc. Passive components can be tested using mock objects in a simulation environment.

Active Components are components that provide input and trigger the system. The CL2 system has only 4 active components: *Tracking* – cyclically updates the tracking information, *L1Connection*– polls the L1 interface periodically, *DexReceive* – handles incoming messages from other systems and *HMI-Models* react upon actions from the user.

Adapters are connectors between the components. They can be *data-transfer adapters* which consist of no business logic and merely fetch data from one object, convert its format if needed and pass the data on to the next object, e.g.,`L1ActionDetectors, L1SetpointHandlers` etc. They can also be *business-logic adapters* which are usually used to bridge functionality of utility classes as needed in foreign systems. The use of business-logic adapters allows the core components to remain unpolluted from customer-specific solutions, e.g.,`SpeedExpert`.

The Spring Framework[3] is used to describe software components and their dependencies. Spring is a glue framework that gives an easy way of configuring and resolving dependencies throughout the J2EE stack. It makes use of many abstractions like MVC, AOP, declarative management of beans and services, JDBC, JMS, etc, and provides declarative access to enterprise services. The added ease of development and maintenance make the value proposition presented with Spring very attractive to most companies. Extensive use of the Spring Framework has had a big impact on the architecture of the CL2 system. It has eliminated lookup code from within the application. Needless to say, it has increased the "pluggablity" and hot swapping facilities by promoting good OO design. Spring was also one of the biggest enablers for increased reusability and testability of the CL2 software.

---

[3]http://www.springframework.org

One of the key technique used in the Spring Framework is *dependency injection*, which decouples object creators and locators from application logic and thus improves loose coupling and testablilty [Walls & Breidenbach, 2007]– two prerequisites for proper exploitation of variability. In object-oriented languages, this is also known as inversion of control (IoC) and more informally IoC is described as the Hollywood principle–"don't call me I will call you". Loose Coupling is improved because you don't hard-code dependencies between layers and modules. Instead you configure them outside of the code. This makes it easy to swap in a new implementation of a service, or break off a module and reuse it elsewhere. Testability is improved because your Objects don't know or care what environment they're in as long as someone injects their dependencies. Hence you can deploy Objects into a test environment and inject Mock Objects for their dependencies with ease. The flexibility provided by such architecture has obvious advantages: customers can judge the system functionality on the basis of already existing configurable components, and different types of casters can operate with the same software.

8.1.2   Current Challenges for Developers and Engineers

Siemens VAI delivers between 20 and 30 new systems per year based on customer-specific requirements. Product customization and new development is mostly done using the copy & adapt paradigm (start with a project from the past, copy and adapt the code to the new requirements until no errors are obvious). The current software development process practiced by Siemens VAI seem to work pretty well; however, the size and complexity of CL2 software makes product customization a tedious and error-prone task. Some of the challenges are listed in Table 8.1.

It is impossible in a large-scale system for engineers or even a small team to create and maintain variability models for the complete system. Instead different teams are in charge of different parts of the product line. Support for the distributed and coordinated creation of variability models by different teams is thus essential. This includes features to resolve conflicts when merging multiple variability models.

Table 8.1: Challenges for developers and engineers at Siemens VAI.

Technical challenges	Market challenges	Personnel challenges
Highly complex application, hundreds of components and 1000s of paramters	Functionality has to be continuously improved/added for customer satisfaction	Only few key developers with an overview of the whole system
Needs to be adapted for each project (approx. 25 per year)	High demands on robustness, performance, quality, stability	Many experts for smaller parts (subsystems)
Flexibility required to react to wishes of customers	Reduce time-to-market and price	Reduction of time-to-market requires more developers

The complexity and variability of the CL2 software system has several effects which are well known to occur in all kinds of software systems:

Isolated variability mechanisms– Different stakeholders handle variability on different levels not taking variability on other levels into account. The knowledge about variability is not systematically used in product customization during sales processes which is weakly integrated with product configuration carried out by developers.

Slow and errant configuration process– Due to the large number of configuration parameters and no explicit documentation of the dependencies among them, the configuration process is error-prone. The knowledge about variability on the requirements/feature level is mostly available in the minds of sales people. Knowledge about variability on the architecture/ implementation level is only available in the minds of the developers. That makes it difficult for stakeholders to understand what implications the decisions they take on their level can have on the variability on other levels.

Long and flat learning curves– Personnel changes are quite common especially in the IT-branch. Whenever new developers or engineers join the company, it is essential that their learning curves are as steep as possible. This is however not the standard case in most companies, due to the massive amount of tacit knowledge involved in software development. Explicit documentation of architectural and sales knowledge can bring significant improvements in this aspect.

Sensitivity to knowledge gaps– Key engineers and developers with an overview of the whole system represent bottlenecks in the software development process. The number of such people is limited and they are required in almost all phases of the project. Most software development processes are therefore very sensitive so the knowledge gaps created in the absence of such knowledge carriers. This implies that the process can to some extent be improved if key engineers and developers can better pass their knowledge to other stakeholders.

One of the causes is the absence of explicit knowledge. For example, a developer intending to configure the system may find out the information regarding which components require which others by analyzing the source code (explicit knowledge). He may easily oversee detailed decisions, which were made by other developers previously (e.g., that a certain configuration requires Oracle database and won't work properly with mySQL because of the data-load).

8.2 Understanding Variability of CL2

The variability in CL2 software was analyzed, from two perspectives [Dhungana *et al.*, 2006]: bottom-up (using automated tools) and top-down (based on moderated workshops). An overview of the variability implementation techniques practices by Siemens VAI is depicted in Figure 8.2. The number of configuration entities decreases as we move from the center of the figure (1000s of configuration parameters and mathematical models) towards the outer border (20–40 processes and servers for one customer-specific CL2 system).

"Fine grained configuration" (e.g., changing colors, UI appearance and preset parameters like speed, amount and formats) is dealt with using configuration parameters. Different computation mod-
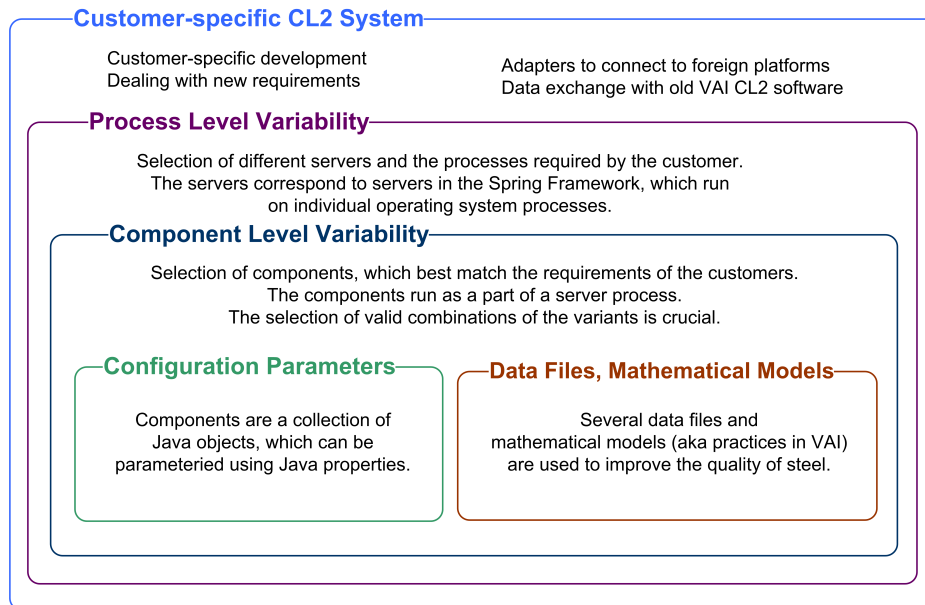
Figure 8.2: Variability implementation mechanisms currently adopted by Siemens VAI for easy configuration of CL2 software.

els and optimization components are parameterized using various chemical and data models for better performance. The next level variability occurs at the component level, i.e., various components can either be included or excluded from the final system. After the components have been selected from the existing pool, they are aggregated in a Java process, and run together depending on their data and logical dependencies among each other. Finally a customer-specific CL2 software system is built using the selected components (running in logical groups of processes) and customer-specific extensions.

8.2.1   Bottom-up Analysis Using Automated Tools

In order to better understand variability at the implementation level, we developed a Spring component analysis tool, which parsed Siemens VAI's component repository to detect existing variability. The spring analysis component is a plugin for *DecisionKing* (depicted in Figure 8.3) and is used to detect inconsistencies in the component repository (e.g., missing Java Beans definitions and violation of architecture conventions).

Using the Spring analyzer, we could detect many architectural violations in Siemens VAI's component repository. This helped us in winning the engineers' trust and reliance on the tool set. Sometimes, the problems detected by the tool were known to some engineers, at other times, the tool surprised them. We experienced that the acceptance of a software tool not only depends on the number of

Figure 8.3: Spring configuration file analyser, to review and detect failures in existing spring component definitions.

technical features provided by the tool, but rather on the personal benefits for the users of the tools.

8.2.2 Top-down Analysis Based on Moderated Workshops

In order to better understand the problems faced by developers, engineers and sales people at VAI, we conducted a series of workshops[4] with our industry partner. The goals of these workshops were to understand the variability of different parts of the system and to define a repeatable process for eliciting variability which can be used by software engineers in their daily practice. We started with a tentative process and tested it in an initial workshop. Based on experiences and feedback from participants we iteratively adapted and enhanced the process in further workshops. In total, three 3-hour workshops were conducted to capture the most relevant variability of the six largest and most complex subsystems

---

[4]Detailed description of the workshops can be found in [Rabiser *et al.*, 2008].

of the software system. Relevance in this context means that the variability addresses a development risk (high loss if a certain decision is not taken or if it is delayed during derivation) and an important business aspect (directly creating customer value in application engineering). More specifically the workshops aimed at the following goals.

1. Finding the most important differences between products previously developed.

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

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

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

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

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

The first workshop involved two groups with experience in two large and important subsystems of Siemens VAI's software system. Each group consisted of three engineers that had been involved in the development of the subsystem. The workshop was organized to collaboratively develop one flip chart per subsystem (cf. Figure 8.4), with yellow cards describing the variability, blue cards describing the rationale of the variability, and red cards describing the variation points in the form of questions representing decisions to be taken during product derivation. A moderator facilitated the process. One scribe took care of arranging the materials on the flip charts. Another scribe took notes about observations and lessons learned in the process.

Firstly, the participants collected the most important differences between previously developed products. Each difference was written on a separate yellow card. In the following moderated plenary discussion these differences were analyzed and rephrased or adapted where necessary to develop a shared understanding of variability and to improve clarity. When discussing the rationale for the collected variability (i.e., customer requirements or internal organizational decisions) participants requested documenting the consequences of this variability for development. Each group discussed the implementation of the variability in their subsystem and documented how it affects the product derivation process. In a moderated plenary discussion, the collected consequences were analyzed and adjusted, put on blue cards, and arranged with yellow cards describing the variability.

While discussing the consequences, participants explored possible risks and the relevancy of variability. This confirmed the need of value-based elements in the process. In a moderated plenary discussion, the team therefore defined whether a decision on the identified variability must be taken early in
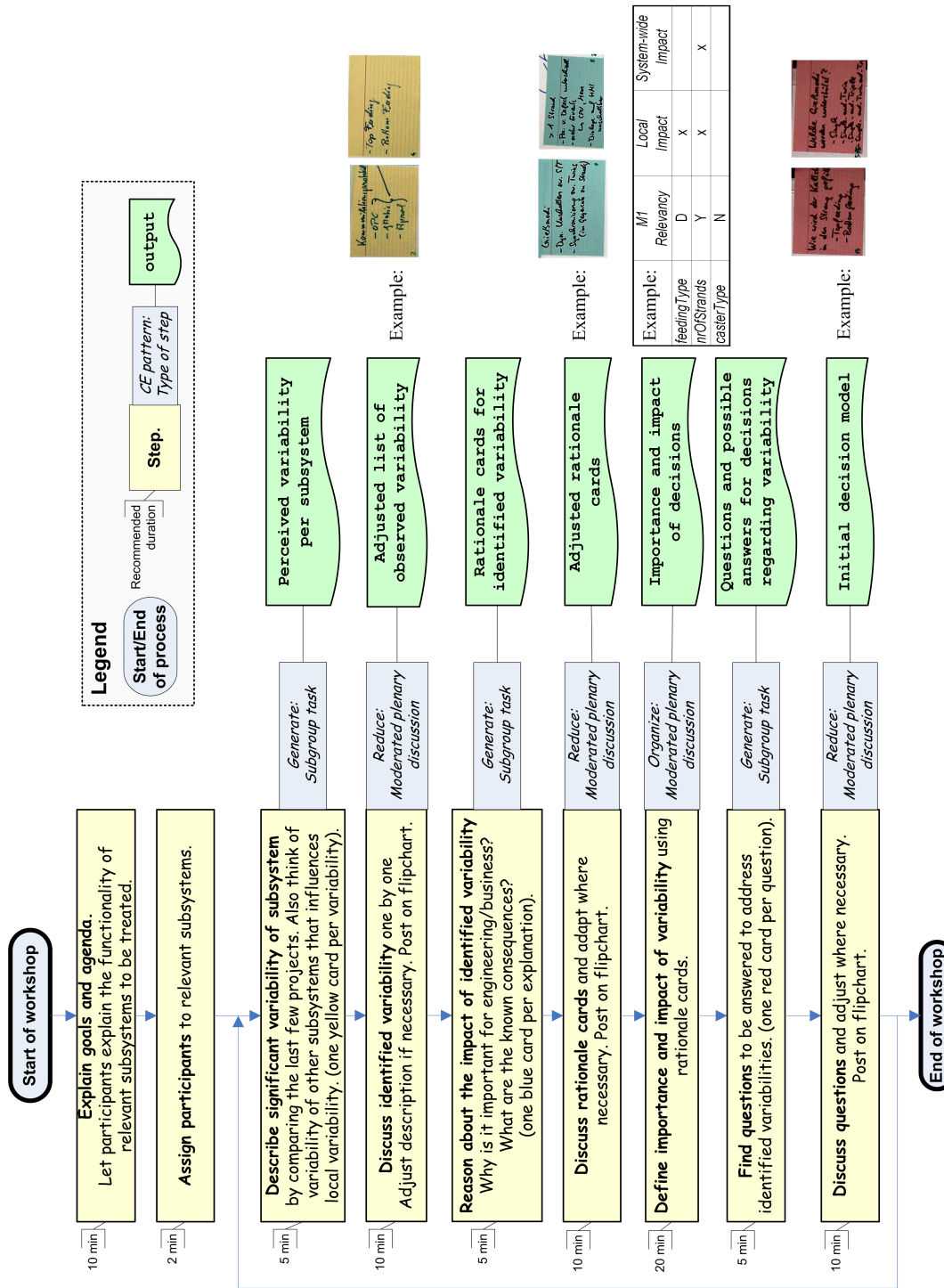
Figure 8.4: Overview of the variability elicitation workshop activities at Siemens VAI [Rabiser *et al.*, 2008].

product derivation (i.e., at the first milestone in application engineering) and whether the decision has a local impact within the subsystem and/or system-wide impact affecting the entire system.

Finally, variation points were elicited in the form of questions representing decisions to be taken during product derivation. These were put on red cards and arranged with existing cards (cf. Figure 8.4). In additional iterations variation points were re-prioritized, dropped if considered unimportant or rephrased if necessary to increase clarity. Also, the relevancy table was adjusted in some cases.

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

8.3   Using *DecisionKing* for CL2 Variability Modeling

DecisionKing was used for variability modeling of the CL2 software from the beginning of the project. The tool itself profited from this, as it was continuously being evaluated by the engineers at Siemens VAI. After having identified the different variability implementation practices at Siemens VAI (cf. Figure 8.2), the next step was to identify relevant parts and the required granularity for modeling variability.

Among the different levels of variability implementation, we chose to focus on modeling component-level variability and associated configuration parameters. *DecisionKing* was parameterized with a meta-model corresponding to the needs of Siemens VAI, the engineers had the impression that *DecisionKing* was developed particularly for them because of the Siemens VAI-specific abstractions in the tool. Figure 8.5 depicts the configured tool suite.

8.3.1   Domain Modeling and Tool Adaptation

In various workshops conducted with the engineers and sales experts of Siemens VAI, we identified different types of core assets that are intended to be reused in the product line (cf. Figure 8.6).

Components are Spring XML files, which are aggregated Java Beans and represent an encapsulated software component. Such a component can be replaced or exchanged by other variant components. In *DecisionKing*, we only model the components on the file level, thus enabling the component level variability depicted in Figure 8.2.

Properties are configuration parameters for components. These can range from simple properties, set in a .properties file to a whole set of lists, maps and hash tables. In *DecisionKing* Properties are treated as key value pairs, which is why this asset type has the two attributes: `key` and `value`.
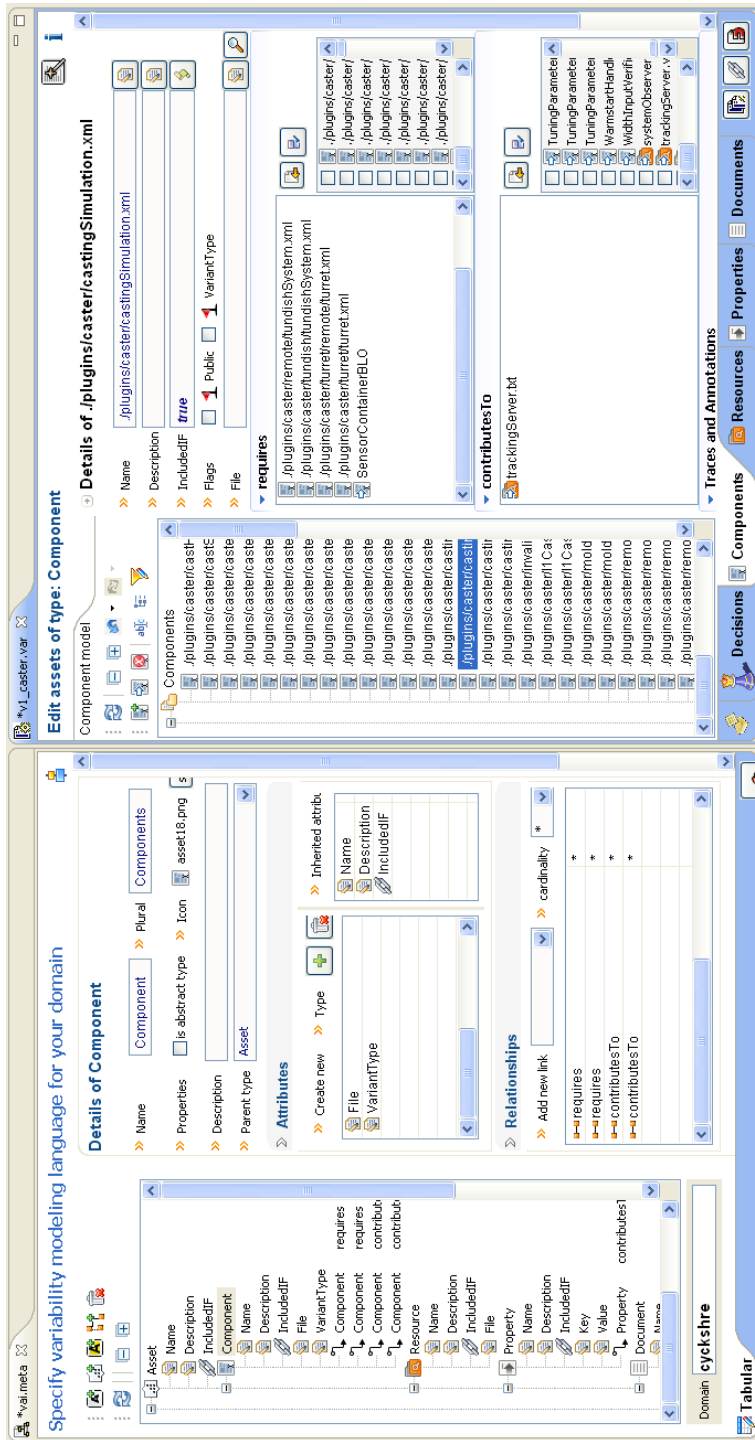
Figure 8.5: Configured variability modeling editor for Siemens VAI (right) and corresponding meta-model editor(left).

The `value` attribute is of type *Expression*, which allows arbitrary decisions to be used for setting the value of the property.

Resources represent legacy hard- or software elements of the CL2 system. We also used the asset type "Resource" to model the process level variability (cf. Figure 8.6) by modeling Process input files (list of components to be executed in one process).
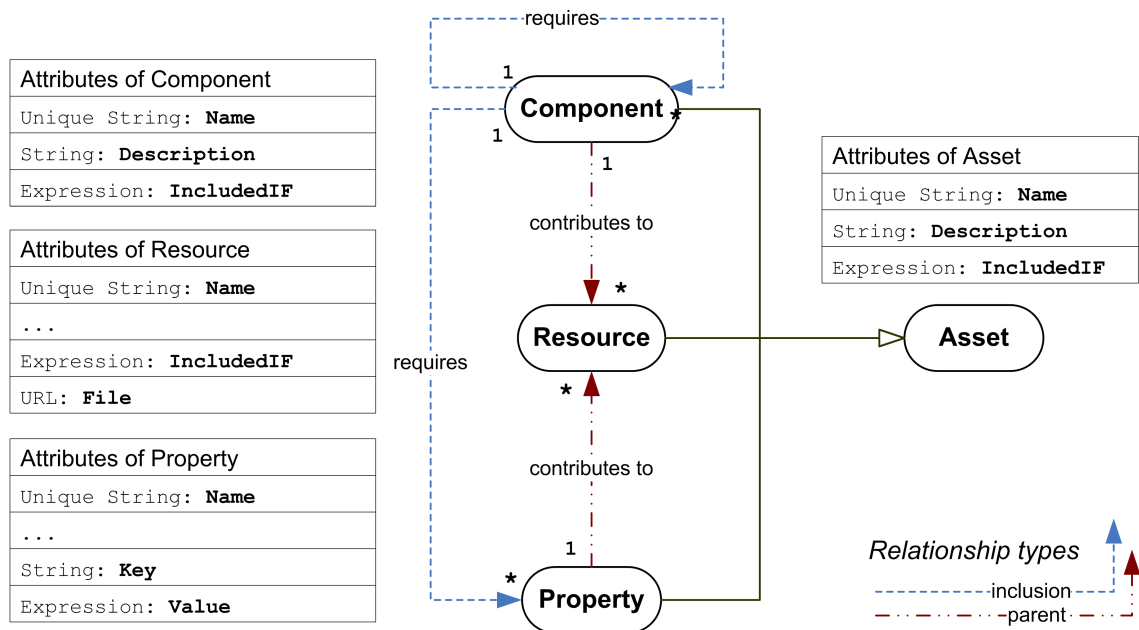
Figure 8.6: Asset meta-model for Siemens VAI.

We also identified the functional dependency `requires` between assets, e.g., a software component may rely on another component to function properly (similar modeling capabilities are available in architecture description languages such as xADL). A domain-specific resolver for the relationship `requires` adds all components required by a certain component as soon as the parent is added to the final system (i.e., by taking a decision during product derivation). Information about the deployment structure of the system is modeled using the relationship `contributesTo`. A simple example is a component that contributes to the sub-system it belongs to. From the variability modeling perspective, both these relationships lead to the inclusion of the related assets. The actual interpretation of the relationships occurs during the generation of the product, which is described in detail in [Rabiser, 2009].

8.3.2   Asset Modeling

Modeling the dependencies among the assets and identifying the technical variability of CL2 software was largely automated. Based on the Spring configuration analysis tool (depicted in Figure 8.3), we built a tool for creating initial variability models called the Spring Importer tool (depicted in Figure 8.7). To illustrate how the spring importer component works, we present two snippets of a Spring XML file, where two BLOs(Business Logic Objects) are defined. One can guess from the naming convention, that this object is responsible for converting messages which need to be saved in the database. The format of the converted message depends on the type of archiving medium used.

Snippet 1 shows the definition of an object called `MessageConverterBLO` (line 4) which is an instance of the class `vai.dex.impl.DefaultMessageConverter` (line 5). This variant is responsible for converting messages for archiving them in a database and is stored in an XML file called `./plugins/dex/db/send.xml`.

```
<!--Spring Bean Definition in file: ./plugins/dex/db/send.xml -->
<bean id="MessageConverterBLO"
     class="vai.dex.impl.DefaultMessageConverter" init-method="init">
     <property name="valueConverter" ref="ValueConverterBLO" />
</bean>
```

Listing 8.1: Snippet of Spring XML file showing the database variant of the component responsible for conversion of messages.

Snippet 2 is another variant of the same `MessageConverterBLO`, but in this case, it is an instance of the class `vai.dex.ascii.StringBufferMessageConverter` (line 5). This variant is responsible for converting messages for archiving them in a text file and is stored in a xml file called `./plugins/dex/ascii/send.xml`.

```
<!--Spring Bean Definition in file: ./plugins/dex/ascii/send.xml -->
<bean id="MessageConverterBLO"
     class="vai.dex.ascii.StringBufferMessageConverter" init-method="init">
     <property name="valueConverter" ref="ValueConverterBLO"/>
     <property name="defaultMessageFormatter" ref="MessageFormatterBLO"/>
     <property name="convertNullNumbersValuesToZero"> <value>false</value> </property>
     <property name="itemSeparator"> <value>\${vai.dex.itemSeperator}</value> </property>
</bean>
```

Listing 8.2: Snippet of Spring XML file showing the ascii variant of the component responsible for conversion of messages.

The automatically generated variability model (by analysing the two snippets), consists of two components (`ascii_send` and `db_send`) and one property (`dex_itemSeparator`). Two placeholder elements (`ValueConverterBLO` and `MessageFormatterBLO`) are created to capture the information that `ascii_send` requires two more components, which have not yet been analyzed/found. One property (`vai.dex.itemSeperator`) is also added to the variability model and linked to `db_send` through the `requires` relationship.

Depending on whether the first or the second component is included in the desired final system, the behavior of the component is changed (messages are either converted to a database format

or a text file format). This example seems to be trivial, however in real project situations, one has to include the right components in the final product depending on the decisions taken by the customers. It is not always clear what the implications of a certain decision are. For example, the inclusion of the first component requires the inclusion of another component, where `ValueConverterBLO` is located. Similarly, the inclusion of the second variant requires the inclusion of two other components(`ValueConverterBLO` and `MessageFormatterBLO`), and setting the value of two properties(`itemSeparator` and `convertNullNumbersValuesToZero`).
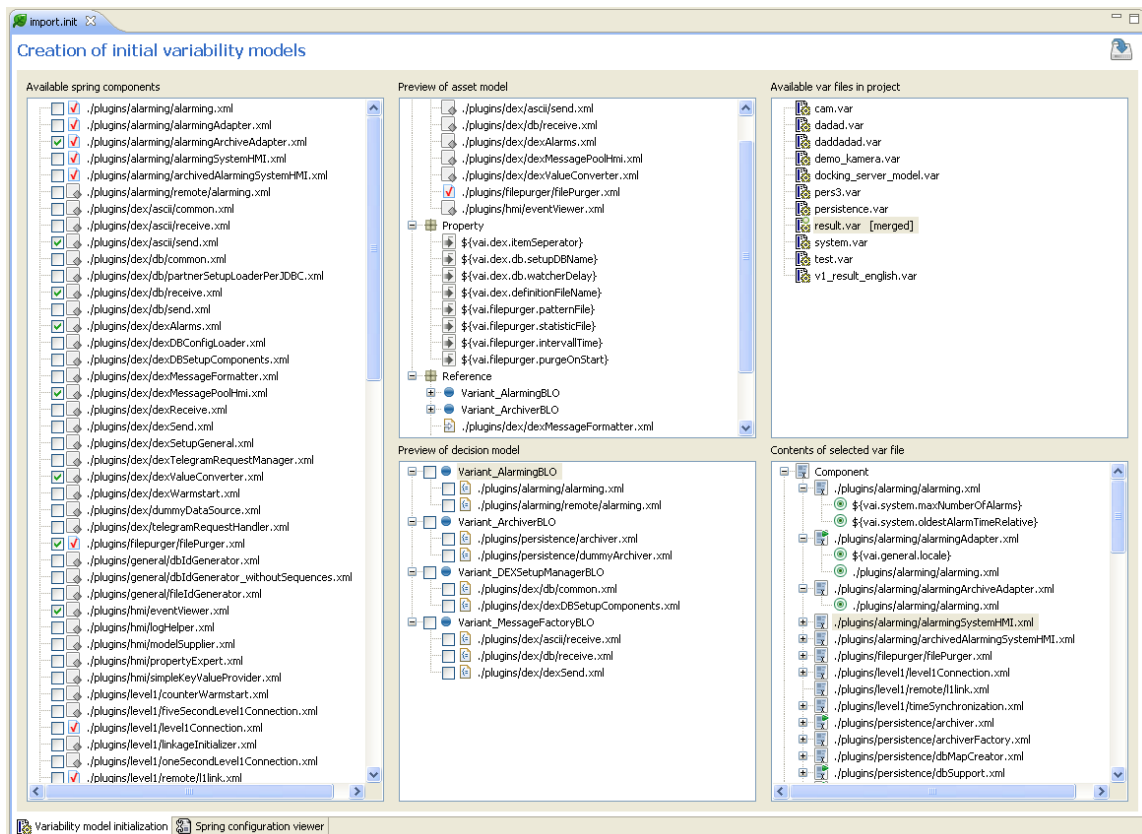
Figure 8.7: Spring component definition importer tool used to automatically create initial variability models.

8.3.3   Decision Modeling

At Siemens VAI, we modeled decisions required for initial phases of a project (i.e., the most relevant ones for largely automating the first milestone in the project (M1). These decisions came from

three sources:

1. Some decisions were automatically generated using the Spring importer tool (depicted in Figure 8.7), which detected multiple variants of components and initialized the variability model with a decision to decide among these variants. The tool therefore typically detected only decisions in purely technical terms, which are relevant only for developers. For example, the tool automatically generates the question "Choose the variant for IArchiveDataFormatter." which is later rephrased by the software engineer to "Should messages be converted in ascii format or database format for archiving?".

2. Some decisions were found and formulated through brainstorming sessions in moderated workshops (Section 8.2.2). We experienced that such workshops result in rather high level decisions (e.g., Will the VAI-Q Quality Control system be part of the final system?).

3. Some decisions were modeled by individual developers and engineers at Siemens VAI. We experienced that only a few individuals have the detailed knowledge required for creating a variability model. Identification of decisions by individual stakeholders was carried out on subsystem levels (e.g., decision related to the secondary cooling system Dynacs: Is adjusting the nozzle spray width controlled by the cooling model?).

8.3.4 Domain-specific Model Consistency Checker

DecisionKing extends the idea of the incremental project builder (provided by the Eclipse platform) and allows foreign tools to add their domain-specific knowledge to enable domain-specific model checking. For example, we extended the default model consistency checker to deal with inconsistencies in Spring configuration XML files in one of our case studies with Siemens VAI. There are four types of error messages detected by the VAI consistency checker, as depicted in Figure 8.8.

Missing relationships: Dependencies among assets, which were detected in the file system is not reflected in the model.

Missing assets: Assets, which were detected in the file system are not modeled.

Non-existent relationships: Dependencies are modeled in the variability model, but not present in the file system.

Non-existent assets: Assets, which are modeled in the variability model were not found in the file system.
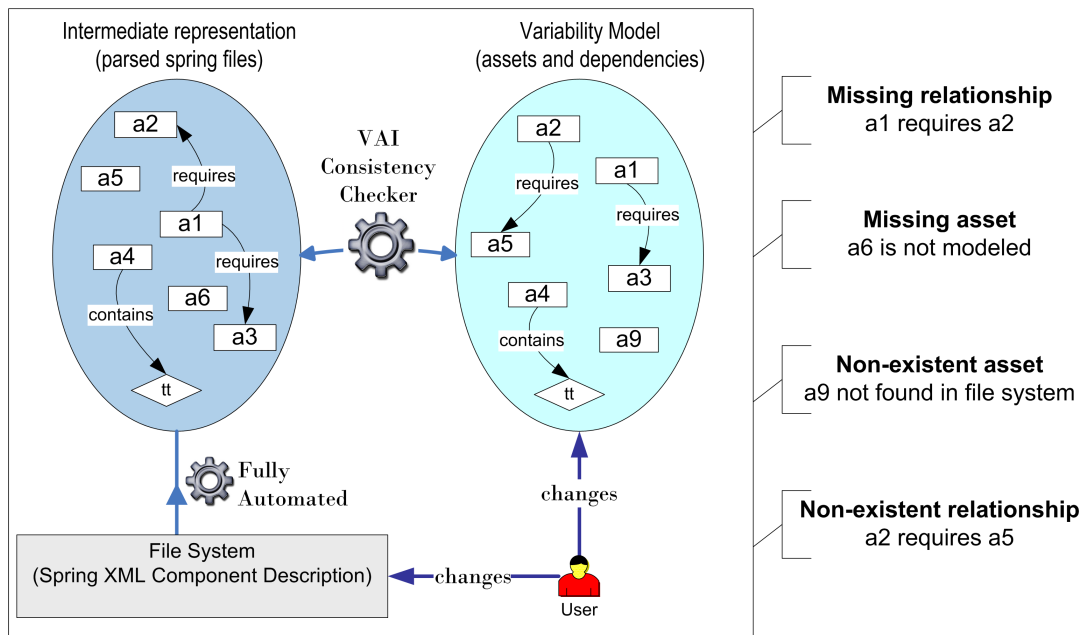
Figure 8.8: Types of inconsistencies detected by VAI model consistency checker.

8.4   Experiences

Despite its simplicity, the meta modeling core provided a good match to describe the variability for the different asset types. A key to accelerate the modeling process were automatic importers. Support for domain evolution turned out to be essential because the characteristics of the problem domain needed to stabilize in the initial stages of product line adoption. We were able to adapt our modeling paradigm to these often-changing requirements. The concepts of domain evolution can also be used to introduce product lines to new organizations by starting with a simple domain-model and adapting it over time as new modeling aspects have to be considered. Already existing models do not need to be created again and again as the meta-model changes; we provide the capabilities that allow updating them.

PLE is a process. Application of PLE practices requires adaptation of existing work processes. People need to change themselves too. The transition from a conventional software development approach to a product line (model based) approach is a drastic change of paradigm from "copy and adapt" to "derive and adapt". It is therefore important to note that the prerequisite for such an approach to work is the change in the developer's mindset. While introducing a product line approach at Siemens VAI, we experienced that the developers had a lot of concerns at the beginning. After having found different advantageous "side effects" of putting the effort in to get the approach working, most developers, engineers and sales personnel could be convinced.
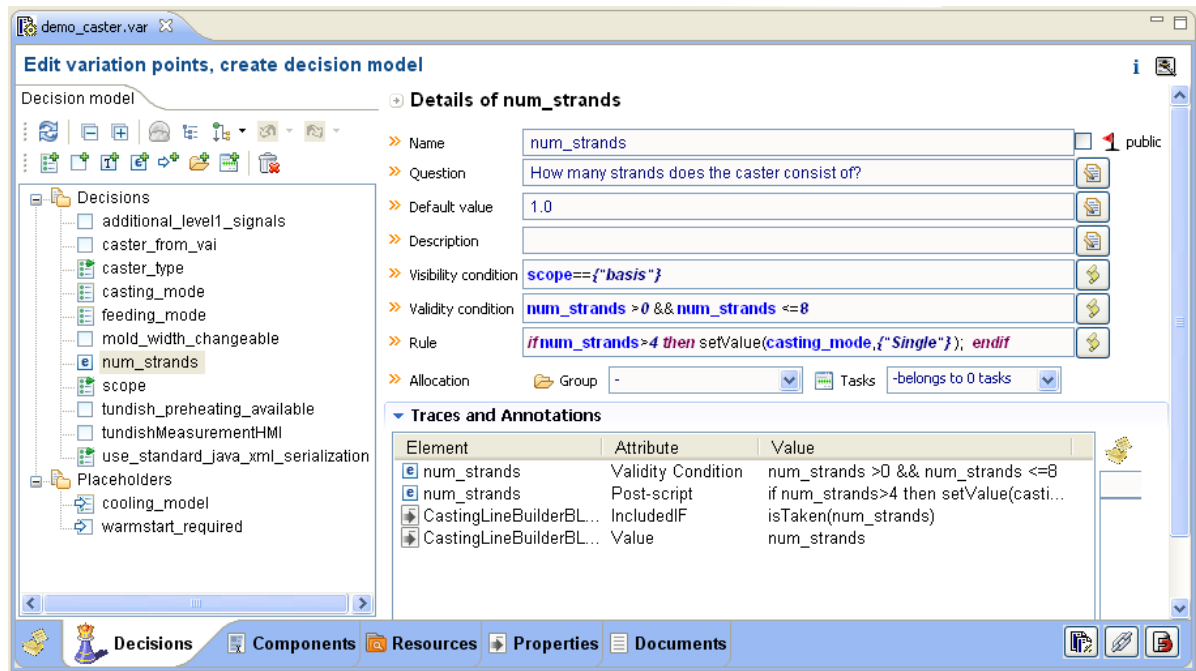
Figure 8.9: Using *DecisionKing* to model decision dependencies for Siemens VAI's CL2 Subsystem Caster.

Oh! Yet another "thing" to maintain, was the first reaction of most of the developers. At the beginning of the project, the tools were still in very early stages, and the we lacked convincing arguments why the developers would have to spend extra time in creating models of their system, rather than spend time in customer projects. "We don't time for modeling", "We need to put more effort in customer projects and to solve their problem", were common statements to escape the modeling task.

Apart from that, we had a hard time convincing both management, sales and developers why a new tool had to be developed to solve the variability modeling problem. Only the engineers who had been directly involved in the feature modeling workshops understood the drawbacks of feature modeling approaches and weaknesses of feature modeling tools.

In the course of time, we had a small team of engineers, who intensively worked with our tools for analyzing their software, creating variability models and advertising the benefits of the approach. Some of the advantages perceived by the users of our tools are:

Architecture documentation comes for free. Variability models, after they are created can also take the role of architecture documentation, where not only the overall structure is captured but also the rationale behind different architectural decisions. Such documentation can be used as tutorial for newcomers, advertisement for customers and guidance for developers.

Architectural violation and errosion is detected and avoided intuitively. Several times during the modeling process, we came across situations, where the engineers were themselves surprised about their systems' architecture. The modeling tools exposed several violations in the architectural conventions. When using variability models as a means of system specification (variability driven development), bad architecture is avoided per design.

Supposed variability and dead code is detected in the course of modeling. Developers and engineers supposedly believed that their system provided more variability than it actually did. This had occurred because they were not aware of different dependencies between components, which were supposed to be used independently but could not.

Existing development is explored, users are more aware of systems' features and constraints. It is important for the developers to actually understand the peripheries of their system. We experienced that creating a variability model "forces" the developers to scrutinize their own implementation in the light of variability and makes them aware of several implicit assumptions they had made (which mostly represented the constraints in the system).

We created variability model fragments for 11 sub-systems of the CL2 system. The variability of the system was elicited in two ways: (i) we used automated tools which helped us deriving the technical variability on the level of components by parsing the existing Spring XML configuration files; (ii) we conducted moderated workshops with engineers from Siemens VAI representing various subsystems to identify major differences of products delivered to customers and to formally describe these differences in models [Rabiser *et al.*, 2008]. It became apparent early on in the project that working with a single variability model is practically infeasible in the multi-team development environment at Siemens VAI which led to the development of the fragment-based approach.

The developed variability model fragments vary in size and complexity due to the different scope of the subsystems. The average model fragment has 32 components, 21 properties, 12 decisions and 23 reference elements to express inter-fragment dependencies. The model merger was used frequently to create an integrated variability model. Most of the merging process was done automatically using the conflict resolution features. In the initial merge process based on an early version of the variability model fragments intervention was required in 28 cases to resolve ambiguities. The current version of the merged model contains 324 components, 160 properties, and 78 decisions.

The underlying product line assets were changed frequently during modeling which again confirmed the need for evolution support. For instance, Spring XML configuration files were updated and new relationships between components were introduced frequently when refactoring components. The inconsistencies resulting from these changes were detected automatically by our consistency checking tool and the engineers fixed the variability models accordingly. We are currently working on automated support for fixing certain types of inconsistencies after changes.

Meta-model evolution capabilities were particularly important to support product line adoption at our industry partner as a static meta-model did not provide the necessary flexibility. We started with a relatively simple meta-model which was extended and modified to model further aspects of the

product line as the project progressed. The initial simple meta-model reflected the basic asset types `Component` (Spring XML files), `Property` (individual configuration parameters) and `Resource` (additional files such as third party licenses or hardware specs). Based on this simple meta-model we introduced new asset types, attributes and dependencies in the project. For instance, we added a new asset type `Document` to the meta-model for modeling arbitrary pieces of documentation (technical specifications, detailed designs, user manuals, etc). This iterative approach ensured continuous validation of the evolving meta-model and helped avoiding the introduction of unneeded concepts. Our meta-model evolution features were essential to automatically propagate the change to existing model fragments.

We found that the distinction between internal and external variability can be further refined and categorized on the basis of stakeholders involved in deciding on the variability. While conducting different workshops for the elicitation of variability with our industry partner we came across 5 levels of variability (depicted on Figure 8.10). These range from purely technical decisions for developers to high level decisions, which are typically taken by sales/customers.
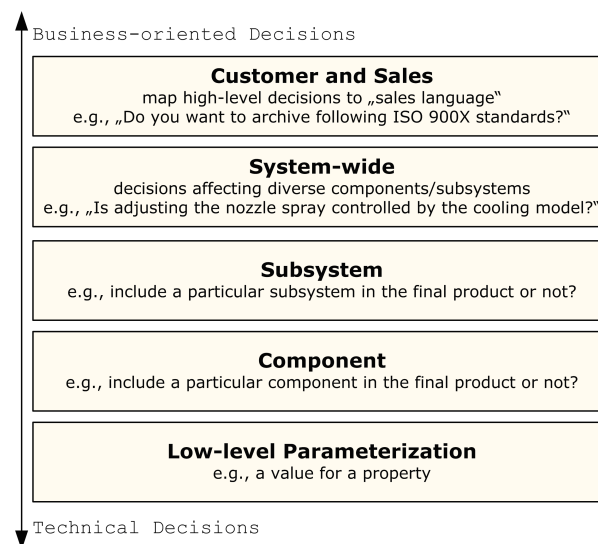
Figure 8.10: Layers of variability in a typical software system [Rabiser *et al.*, 2008] .

8.5   Ongoing and Future Work

In our future work, we shall build several extensions to the tools so that the approach can be used by other departments at Siemens VAI. Currently we are working towards adding extensions to model variability of user-documentation and to model test-cases as a part of the asset model. Furthermore, we

shall address some of the issues and concerns of developers:

1. How do you actually model, what is the right granularity of decisions?

2. When modeling a subsystem, how do you actually decide where (in which model fragment) a certain element needs to be modeled in?

3. Different users perceive the decisions in different ways, how can the kind of questions and graphical representations be efficiently managed for different users?

Chapter 9

Case Study 2: Modeling Variability of IEC 61499 Industrial Automation Systems

Summary *This case study presents an approach based on product line variability models to manage the life cycle of industrial automation systems (IAS) and to automate the maintenance and reconfiguration process. We complement the standard IEC 61499 with our variability modeling approach to support both initial deployment and runtime reconfiguration.*

This case study is part of an ongoing PhD research being carried out by Roman Froschauer at the Upper Austrian University of Applied Sciences in Wels, Austria. In this collaboration project, we have applied our modeling approach and extended our tools to deal with the variability of Function Block based systems. Here we discuss only the modeling aspects of our research, further details can be found in Roman Froschauer's PhD thesis [Froschauer, 2009].

9.1 Introduction to Industrial Automation Systems

Industrial Automation Systems (IAS) consist of numerous complex sensor units cooperating with actuators to perform measurement and control. Usually they are based on a distributed architecture consisting of multiple physically and/or logically distributed components fulfilling a set of defined requirements [Froschauer *et al.*, 2008]. Modern industrial automation systems execute applications distributed across heterogeneous networks. The current trend towards component-based software architectures has also influenced the development of industrial automation systems (IAS). Programmable logic controller(PLC) and PC-based soft controllers are beginning to use software components, such as object-oriented technology, to bring together the different worlds of factory automation and business systems [Lewis, 2001].

IAS are mostly continuous production systems and a system halt typically results in a huge financial burden. A special focus lies thus on the runtime reconfiguration of such systems. IAS are particularly vulnerable to changing market needs and volatile economies as they are mostly used for producing market-related goods [Froschauer *et al.*, 2008]. Therefore IAS are characterized by their dynamically changing environment which requires flexibility and adaptability at runtime. Additionally

the market has raised a demand for downtime-less operation and change of automation and control for such systems [Froschauer *et al.*, 2006].

9.2   Technical Background: IEC 61499 Standard

Today's industrial automation systems are mainly based on the standard IEC 61131-3 which is dedicated to systems based on programmable logic controllers (PLCs). Function blocks (FB) introduced by IEC 61131-3 are an established concept for industrial applications to define robust and reusable software components. A function block in the IEC 61131-3 standard is the basic building blocks from which entire applications may be built. There are two types of function blocks: basic function blocks and composite function blocks. A composite function block contains other composite function blocks and/or basic function blocks. A basic function block contains algorithms and an execution control chart (ECC).

Current IEC 61131-3 based engineering tools support basic low-level reconfiguration activities such as on-the-fly change of specific parts of code and variable values. They lack, however, support for reconfiguration at task level which can lead to indefinite switching points in time (due to cyclic execution policy), jitter effects (task reconfiguration influences other tasks), or the possibility of inconsistent states that often lead to deadlocks [Sünder *et al.*, 2006]. In an attempt to address these issues the International Electro-technical Commission has introduced the successor standard IEC 61499 which provides more flexibility both at initial design and runtime compared to IEC 61131-3. The IEC 61499 Standard defines an open architecture for the next generation of distributed control and automation.

In this section we describe the basic concepts of the IEC 61499 standard for the application of function blocks in distributed industrial-process measurement and control systems. We build upon the concepts introduced here and carry on to further description of enhancements to the modeling approach based on variability models in the next section.
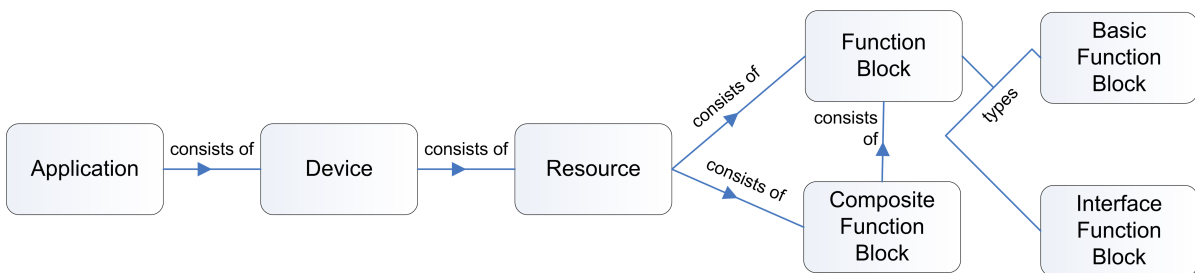
Figure 9.1: IEC 61499 Meta Model.

IEC 61499 can be seen as a reference architecture for distributed, modular, and flexible control systems. It specifies a highly generic architectural model for distributed applications in industrial process measurement and control systems (IPMCS). IEC 61499 extends the function block model of its

predecessor IEC 61131-3 and offers the following main features:

1. Component-oriented basic building blocks called Function Blocks (FBs) to encapsulate algorithms.

2. Event-driven execution modeled using event connections between function blocks.

3. Graphical and intuitive modeling of control flow on the basis of wiring FBs by logically applying event and data connections.

4. Explicit support for distribution to compose applications distributed across devices and their resources (e.g., computation tasks).

5. Definitions for the interaction between devices of different vendors by using standardized commands encoded in XML.

6. Basic support for reconfiguration by allowing additional functional blocks to be instantiated and dynamically integrated by rewiring the event and data connections during full operation.

7. Compliance with existing standards of the domain, especially the commonly used standard IEC 61131.

The IEC 61499 standard defines a distributed model for splitting different parts of an industrial automation process and complex machinery control into functional modules called function blocks (cf. figure 9.1). These function blocks can be distributed and interconnected across multiple controllers. The control software system is thus a collection of devices interconnected and communicating with each other by means of a communication network consisting of segments and links.

A *device* is an independent physical entity capable of performing one or more specified functions in a particular context and delimited by its interfaces.

A *resource* is a functional unit having independent control of its operation, and which provides various services to applications including scheduling and execution of algorithms.

An *application* is a software functional unit that is specific to the solution of a problem in industrial-process measurement and control. An application may be distributed among devices and may communicate with other applications.

A *function block* is a software functional unit that is the smallest element of a distributed control system. It utilizes an execution control chart (ECC) state machine to control the execution of its algorithms.

In addition to IEC 61131-3 or as a technique for implementing the system level software several companies use internally developed component or module frameworks. For example, ABB supports modules in addition to IEC 61131-3 programs. A module gives a higher level of re-use and can to some extent be compared with an object. Honeywell has an internal component-based systems called URT (unified real-time).

9.2.1   Examples of Variability in IAS

Variability is very important in IAS because of their long-running and zero-down-time nature. There are two basic aspects of variability:

Hardware variability is reflected in the set of available hardware components. Depending on which devices are available (e.g., conveyor, robot-grippers, sensors, etc.), and their specific attributes (e.g., width of conveyor, maximum span of robot grippers, types and number of sensors, etc.), an industrial automation system can produce a wide range of products (variability of the products that can be manufactured). The hardware devices which can be used to build the manufacturing system is decided on the basis of which products are to be manufactured in the system, i.e., decisions are the characteristics of the products.

There can also be several dependencies among the hardware components, for example, speed of conveyor must match the capabilities of the robot. Hardware variability is a decisive factor for software variability.

Software variability is reflected on the set of software controllers which are used to drive the hardware components. The software solution is determined by the available hardware, however, to drive the same set of selected hardware components, one can deploy different variants of the controller software. Some examples of control layer software variability are– control algorithm for robots (depends on the type of robot, number of axes, available sensors etc.) or the type of conveyor belt controller (smart belts can determine their own speed, dumb belt can just switch on/off). Furthermore, software variability can be dealt with at different levels of abstraction.

Here, we concentrate on software variability and present an example of a bottle sorting application, thereby focusing on the variability concerns. Figure 9.2 depicts a simple IAS application consisting of one process where all the functions of the application are run. The application consists of three devices – conveyor belt to transport the bottles, a robot grip for picking up the bottles from the belt, and a separator for identifying the different types of bottles. Different function blocks, which constitute the application are mapped to different resources through the devices in use. The resources represent different process interfaces that can be parameterized as needed to adjust the speed of the sorting system.

9.2.2   Challenges

Despite many advances, the life cycle management of large-scale, component-based IAS still remains a big challenge. The knowledge required for the maintenance and runtime reconfiguration is often tacit and relies on individual stakeholder's capabilities: an error-prone and risky strategy in safety-critical environments. State-of-the-art industrial control and automation systems do not sufficiently solve the problem of down-time-less reconfiguration of control applications [Hummer *et al.*, 2006]. Runtime reconfiguration is essential as IAS need to be monitored and fine-tuned at runtime to ensure
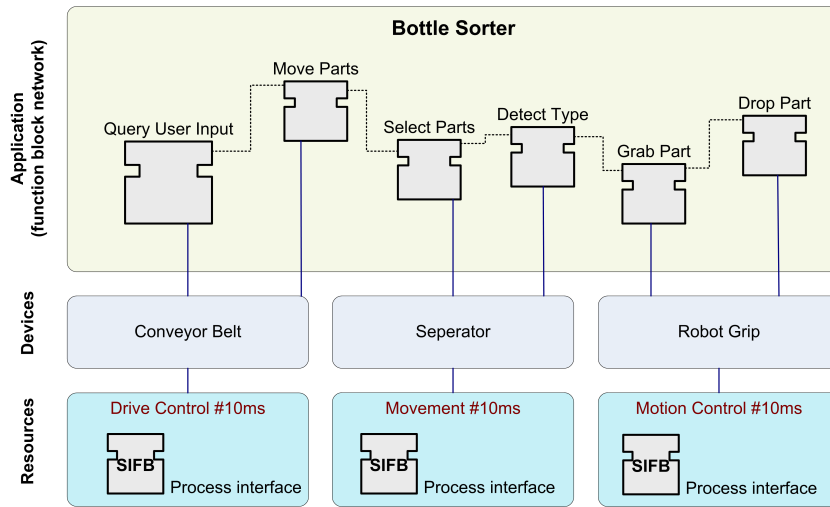
Figure 9.2: Bottle sorting application depicting different levels of the IEC-61499 meta-model.
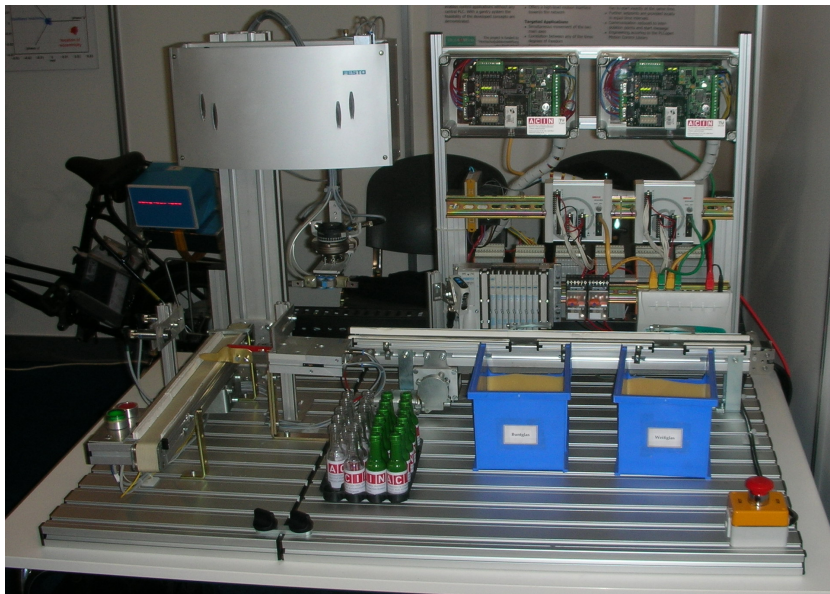
Figure 9.3: Demonstration setup of the bottle sorting application at AlpinaTec GmbH & Astrium GmbH.

continuous satisfaction of QoS constraints and to accommodate changing functional and non-functional requirements [Grabmair *et al.*, 2006].

For instance, in a continuous car spraying system an autonomous and dynamic product recon-

figuration of the system is important to ensure optimal use of the available resources at runtime. Reconfiguration is usually carried out manually based on intuition and tacit knowledge of the involved stakeholders. This is however risky in safety critical environments due to the complexity of today's and future IAS. Researchers have thus started adopting model-based approaches to manage the variability and configurability of IAS and to support their automated reconfiguration.

A control system based on IEC 61499 is configured using management functions which can be deployed in physically and/or logically distributed devices. The standard provides convenient mechanisms to reconfigure the devices by sending XML commands. The functionality of the system can also be changed at runtime by changing the flow of events and data between function blocks. Online reconfiguration is supported only at a very low level of abstraction. For example, it is possible to instantiate and terminate function blocks during full operation. In a real-world system with thousands of function blocks such a fine-grained reconfiguration process (e.g., at the level of function block rewiring) is very challenging and complex.

Unfortunately, IEC 61499 provides no support for managing the knowledge required for reconfiguration at the task level. Performing reconfiguration with respect to application specific constraints is therefore based purely on the tacit knowledge of the stakeholders- an error-prone and risky reconfiguration strategy. Consequently, there is a strong need for approaches managing the reconfiguration process at a higher level of abstraction. Such approaches rely on understanding the variability of the system at different levels of abstraction (e.g., requirements, architecture, or implementation level) and modeling diverse domain-specific devices and artifacts.

Let us consider a simple reconfiguration scenario of an application and follow the steps necessary for certain changes, as depicted in figure 9.4. The sample application consists of one function block (START) in the beginning. The following steps are necessary, if two new function blocks need to be added to the system.
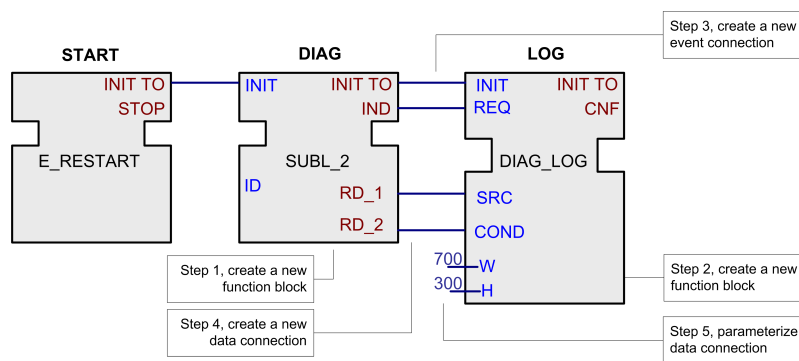
Figure 9.4: Different reconfiguration steps in IAS and corresponding XML commands depicting the complexity and error-prone nature.

Step 1, creation of a new function block is done by sending an XML command to the platform.

```
<!—Addition of a function block —>
<Request ID="3" Action = "CREATE"> <FB Name= "DIAG" Type= "SUBL_2"> </Request>
```

Step 2, Another function block is created using the command similar to the first one.

```
<!—Addition of a function block —>
<Request ID="4" Action="CREATE"> <FB Name="LOG" Type="DIAG_LOG"> </Request>
```

Step 3, creation of a new event connection is done by referencing the existing function blocks and specifying them as source and destination of a connection.

```
<!—Addition of a event connection —>
<Request ID="7" Action="CREATE"> <Connection Source="DIAG.IND" Destination="LOG.REQ">  </Request>
```

Step 4, creation of a new data connection is similar to the creation of event connections, the only difference being the type of ports in source and destination.

```
<!—Addition of a data connection —>
<Request ID="8" Action="CREATE"> <Connection Source="DIAG.RD_1" Destination="LOG.SRC"> </Request>
```

Step 5, parameterization of data connections is done by sending a write command to the corresponding data ports.

```
<!—Parameterize a data connection —>
<Request ID="10" Action="WRITE"> <Connection Source="700" Destination="LOG.W"> </Request>
```

This example clearly shows the challenges faced by an engineer when reconfiguring the application. The process of writing the necessary XML commands for addition or configuration of the function blocks can be automated. We therefore strive towards modeling the variability of the application on a higher level, thereby automating the code-level reconfiguration process by generating the XML commands.

9.3   Using *DecisionKing* for IAS Variability Modeling

In order to facilitate variability modeling of industrial automation systems, we first created a IAS-specific meta-model, which was used to configure *DecisionKing*. The proposed IAS meta-model can be used to define features and devices needed in a specific applications from the automation and control domain, such as transporting and sorting systems (e.g., Conveyors, Pick & Place Robots). We differentiate between the design time model elements and runtime instances (cf. figure 9.5).

9.3.1   IAS-specific Meta-model

Design time elements capture basic knowledge about the target application domain and specific constraints, which can be compared to other domain-specific meta-models [Dhungana *et al.*, 2007b]
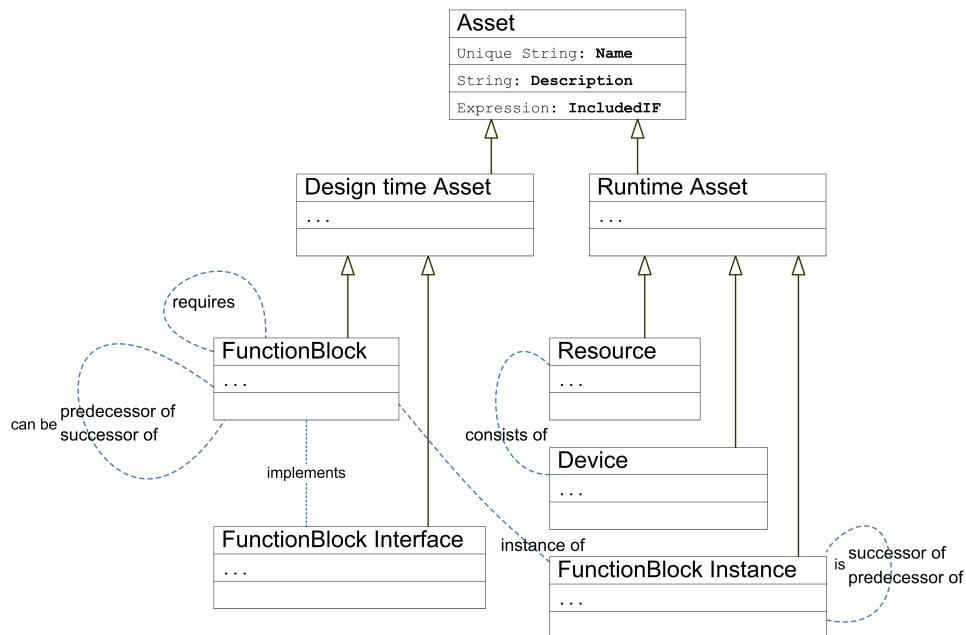
Figure 9.5: Meta-model for modeling variability of IAS [Froschauer, 2009].

for *DecisionKing*. The design time elements reflect the domain component instances of previously defined domain component types (e.g., `HorizontalConveyor` as a domain representation of `Movement`).

Run time elements reflect the desired deployment platform and available runtime component types, which define the execution environment for runtime components to be designed later (e.g., `TransportBottle` as a runtime instance of `HorizontalConveyor`).

Design time and runtime elements are interconnected using domain- and respectively runtime relationships. These relationships can both be established by defining component relationship types. The component relationship type defines (1) which component types can be interconnected, (2) the cardinality of the connection (1 to 1 or 1 to N) and (3) the direction of the relationship (bi- or unidirectional).

As described earlier in this thesis, variability is modeled through decisions. Each component specifies an inclusion condition based on the decisions. Using these decisions the deployment engineer can answer questions, such as "Do you need a conveyor in front of the Pick&Place robot?", and the required components are included into the deployed application. Stakeholder can thus generate an executable application by taking decisions that automatically lead to the selection of applications at runtime [Froschauer *et al.*, 2008].
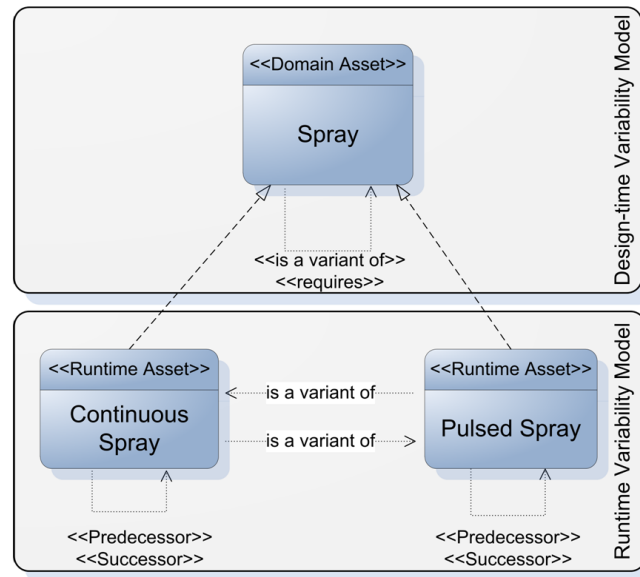
Figure 9.6: Example of a design-time and runtime variability model. [Froschauer *et al.*, 2008].

9.3.2 Tool Support

DecisionKing is currently used to model two variability meta-models: (i) the *platform meta-model* specifying the running infrastructure for IAS and (ii) the *domain meta-model* specifying the types of assets to be run on the platform. Based on these two meta-models, we support variability modeling of design-time entities (available components) and enable generation of runtime asset models out of running IEC 61499 applications. *DecisionKing* acts as a third party variability modeling component in a larger tool suite called ControlKing.

ControlKing is based on the open source IEC 61499 framework 4DIAC [Profactor, 2007]. It integrates the variability modeling capabilities provided by *DecisionKing* and IEC 61499 IDE functionality of 4DIAC. Figure 9.7 depicts the design time and runtime variability model editors in ControlKing. The design time variability model editor is contributed by *DecisionKing*, which is a good example of how *DecisionKing* can be used as an off-the-shelf variability modeling component.

Variability models created using ControlKing are manipulated at runtime and updated using the Model API provided by *DecisionKing*. The model execution engine from *DecisionKing* is an integral part of ControlKing and enables end-users to reconfigure IAS applications by letting them take decisions as they emerge at runtime.

Runtime View

Design View
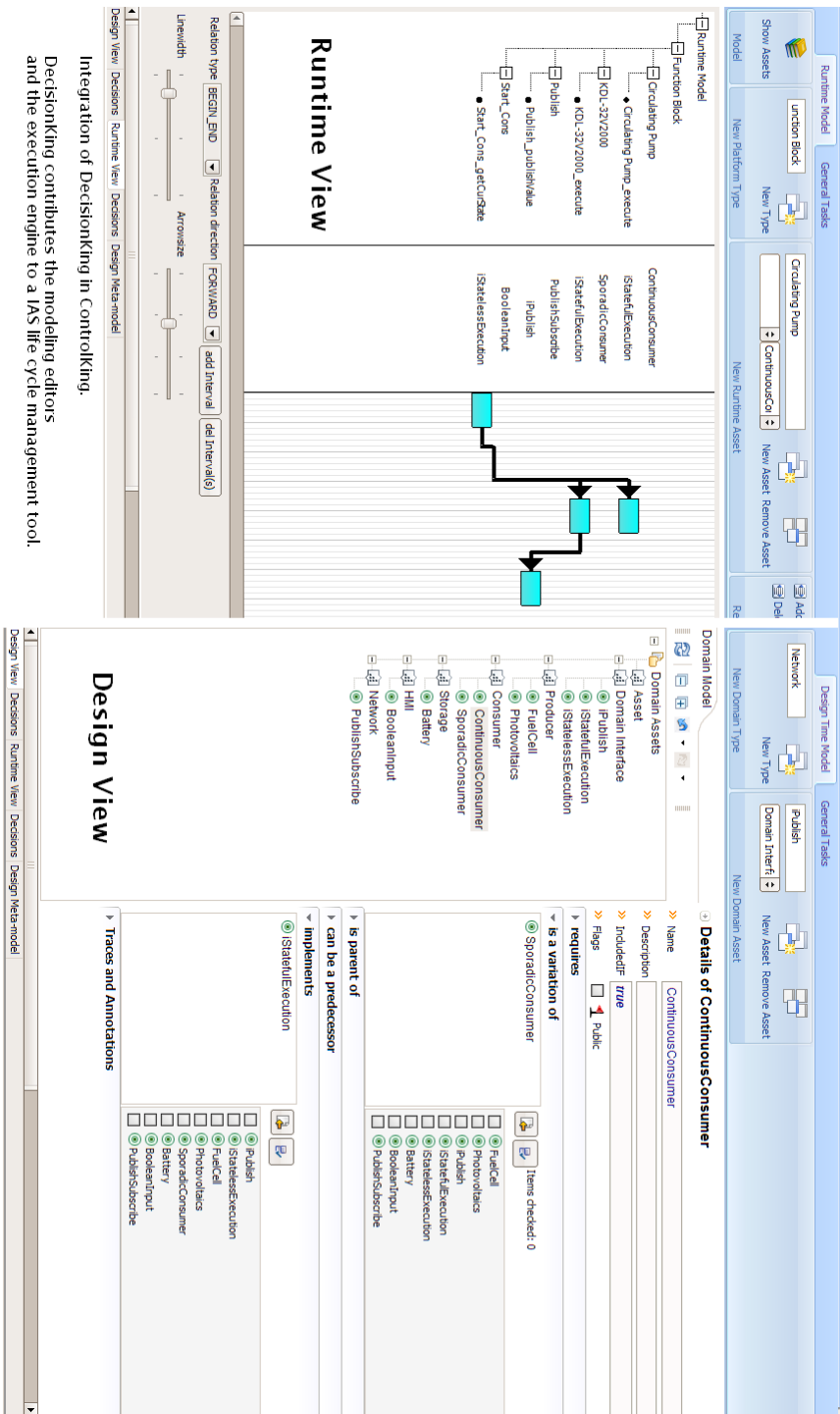
Figure 9.7: ControlKing [Froschauer, 2009]: A tool for managing the life cycle of IAS components, showing the variability modeling editor components contributed by *DecisionKing*.

9.4 Experiences

Our modeling approach was initially designed to model variability of static configurations of assets and had not considered support for modeling runtime instances of assets and runtime constraints, such as execution ordering and execution time of runtime assets. Such a static variability model is useful for deployment of the software system; however, it lacks different concepts for multiple runtime instances and multiple configurations. In this case study, we complemented our approach with the ability to integrate a platform meta-model and a runtime asset model into a runtime variability model. The enhanced approach provided the basis for a new modeling concept in frameworks for component-oriented IAS such as IEC 61499.

Case Study 3: Modeling Variability of Service-oriented Systems based on i* Models

Summary *This case study is an attempt to integrate variability modeling and goal modeling techniques. We complement the i* modeling framework with our modeling technique and create variability models based on i* models. We then use the variability models for monitoring service-oriented systems at runtime.*

Service-orientation has become an integral part of many buzzwords such as service-oriented computing, service-oriented architectures, or service-oriented software engineering and is promoted by a number of emerging standards for service-oriented development. Recently, researchers have started to explore the integration of service-oriented systems and variability modeling [Gruler *et al.*, 2007]. Variability modeling is increasingly seen as a mechanism to support run-time evolution and dynamism in different domains and to design, analyze, monitor, and adapt service-oriented systems [Penã *et al.*, 2006]. In service-oriented system decisions about the system architecture are increasingly shifted from system design to system operation. At the same time the modeling framework i* [Yu, 1996] is gaining popularity to model service-oriented and agent-based systems [Penserini *et al.*, 2006] and researchers are seeking new ways to enhance it with variability modeling capabilities [Liaskos *et al.*, 2006].

10.1 Introduction to Goal Modeling

Modeling service-oriented systems requires an understanding of stakeholder goals and different alternatives to fulfilling those goals. It is therefore important to consider variability at the level of stakeholders in terms of their goals, characteristics and contexts before sketching a solution, because variability in a solution must reflect the variability of the problem [Liaskos *et al.*, 2006]. This increases the chances that the resulting software system will feature the appropriate flexibility needed to respond to later changes.

Goal-oriented modeling techniques are particularly useful in early-phases of requirements engineering (RE). Early-phase requirements consider, e.g., how the intended system meets organizational goals, why the system is needed and how the stakeholders' interests may be addressed. Several techniques have been developed for modeling goals. Some of the most reputable are EEML (Extended

157

Enterprise Modeling Language), GRL (Goal-oriented Requirements Language), and i*. In this case study we use the i* framework.

The i* framework offers an agent-oriented approach to requirements engineering [Yu, 1996]. i* is used in early stages of RE to explore alternative configurations which are analyzed using a qualitative reasoning procedure. In order to enable such analysis, strategic relationships among multiple actors are modeled explicitly. *Actors* depend on each other for *goals* to be achieved, *tasks* to be performed, and *resources* to be furnished. A notion of *softgoal* is used to deal systematically with quality attributes, or non-functional requirements. Dependencies among actors give rise to opportunities as well as vulnerabilities [Yu, 1997]. The i* framework consists of two main modeling components:

The Strategic Dependency model (SD) describes a network of dependency relationships among various actors in an organizational context [Yu, 1996]. The actor is usually identified within the context of the model. This model shows who an actor is and who depends on the work of an actor. A SD model consist of a set of nodes and links connecting the actors. Nodes represent actors and each link represents a dependency between two actors. The depending actor is called *depender* and the actor who is depended upon is called the *dependee*.

The Strategic Rationale model (SR) allows modeling of the reasons associated with each actor and their dependencies, and provides information about how actors arrive at their goals and soft goals [Yu, 1996]. Compared with SD models, SR models provide a more detailed level of modeling by looking inside actors to model internal, intentional relationships. Intentional elements (goals, soft goals, tasks, resources) appear in the SR model not only as external dependencies, but also as internal elements linked by means-ends relationships and task-decompositions [Yu, 1997].

10.1.1   Variability in i* Models

Goal-oriented approaches for modeling service-oriented systems and their variability in an integrated manner are needed to address the needs of heterogeneous stakeholders and to develop and evolve these systems. The i* framework does not have specific constructs for modeling variability. Some authors have tackled this issue by extending i* with explicit constructs [Clotet *et al.*, 2007b]. Others consider variability as implicit in i* models depending on the types of modeling constructs used. We adhere to this perspective and aim at identifying candidate variation points by analyzing the very structure of the i* model. From our analysis, we have identified six different cases of candidate variation points. We present next the different cases classified by the type of i* construct [Clotet *et al.*, 2008].

Means-end Variability: The "means-end links" provide understanding about why an actor would engage in some tasks, pursue a goal, need a resource, or want a soft goal; thus it may be describing a variation point, usually related to external variability, i.e., the variability of artifacts that is visible to customers. Means-end links are composed as an OR, so at least one of the means should be attained to achieve the end. This OR can be interpreted as a variation point when the customer

(a) Means-end Variability.
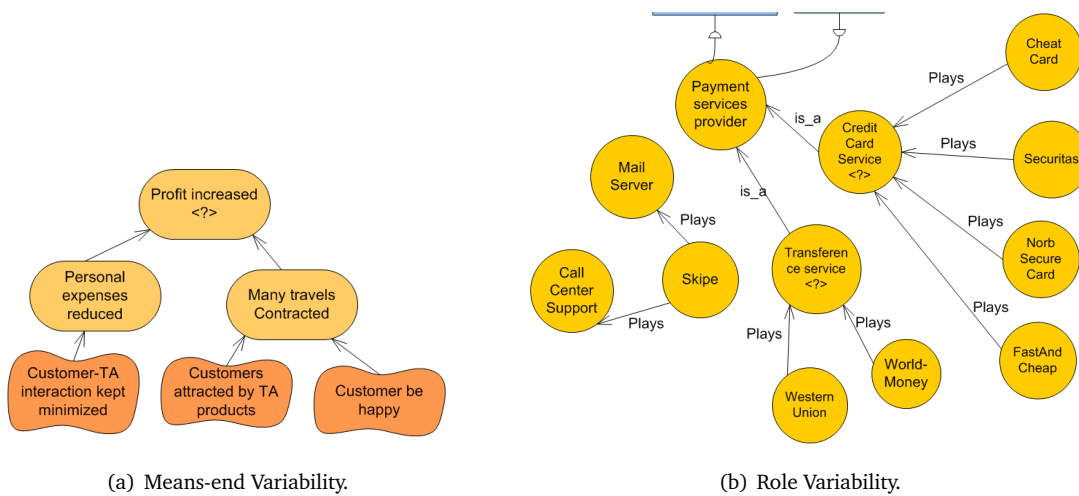
(b) Role Variability.

Figure 10.1: Example of means-end variability and role variability in i* models.

can decide the way she wants the system to achieve her goal. For instance Figure 10.1(a) depicts means-end variability. The goal of increasing profit can be achieved by two means (i) by increasing the number of contracts and (ii) by reducing the personnel expenses.

Role Variability: Agents allow modeling real services or components, and these agents play roles. At this point we can find some internal variation points(i.e., the variability of domain artifacts that remains hidden from customers), because the architect can decide among the different services (agents) that can play the role. Such variability is represented using the "plays" link. For example in Figure 10.1(b) we find two agents playing the role "Transference Service". This means that the architect has to choose between the two services (Western Union or World Money) when deploying the system.

Instance Variability: The i* framework also allows modeling which services instances can be deployed. The example shown in Figure 10.2 highlights that there are two agents (the Spanish and the Austrian Amadeus Server) as instances of the agent Amadeus. The variability is modeled using the link "instance".

Role Inheritance variability: Variability related with architectural features is found in the relationship between actors. In an i* model, actors can represent the different roles our system has to include. These roles can be classified as a hierarchy using the is_a link. For example Figure 10.2 shows a classification for Travel Payment role. The architect can choose to have the credit card payment or transference payment role in the final system.

Intentional Element Inheritance Variability: At a finer-grained scale, a super actor may have different intentional elements refined onto the subactor. This inheritance relationship may take differ-

ent forms (see [Clotet *et al.*, 2007a] for further details). In the case of having more than one heir, the intentional element in the super actor becomes a variation point.

Softgoal variability: Since the satisfaction of a softgoal is not uniquely defined, we may imagine several criteria acceptable at different moments or contexts.
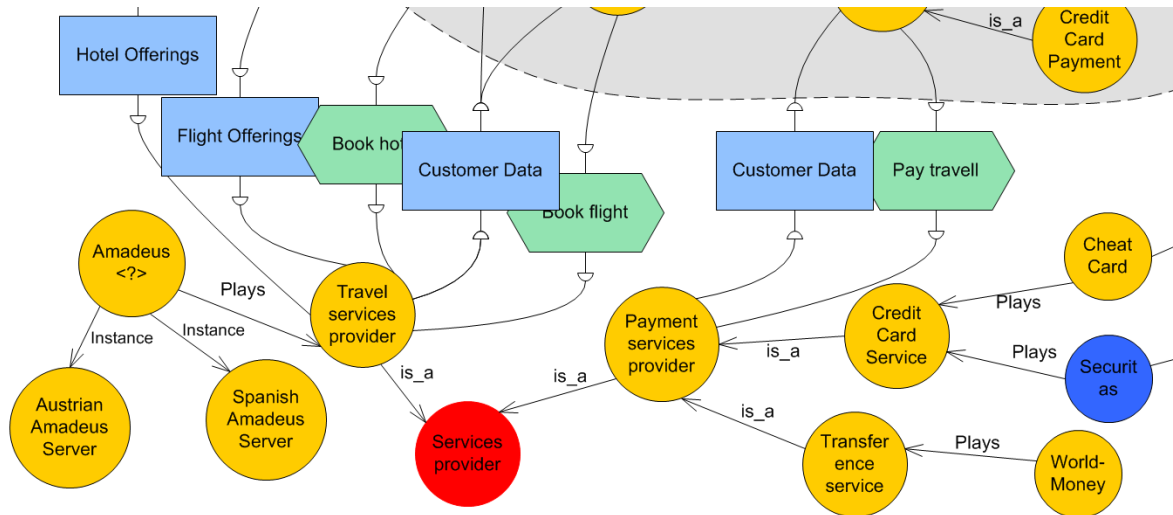
Figure 10.2: Example of variability in i* models.

We have identified a set of rules to generate the corresponding excerpt of a decision model from variation points found in the i* model. We define this correspondence in terms of the metamodel. The result is summarized in Table 10.1 and details can be found in [Clotet *et al.*, 2008].

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

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

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

A role inheritance variability rule (RI-VP) means that there are different kinds of agents, representing the same role. So one or more of the heirs will be chosen depending on the user characteristics.

Table 10.1: Rules for identifying variability in i* models, details in [Clotet *et al.*, 2008].

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

An intentional element inheritance variability rule (IEIVP) is applicable for actor classifications using inheritance if some inherited intentional elements are modified in their heirs. In the three cases of inheritance identified in [Clotet *et al.*, 2007a] (extension, refinement and redefinition), the intentional element placed in the parent has different ways to be achieved. In the case of extension, the new features are considered as alternatives to the parent. In the other cases, each intentional element declared as an heir is considered a way to achieve the intentional element in the parent.

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

10.1.2   Examples of Variability in Service-oriented Systems

Having identified the variability extraction rules for creating decision models out of i* models, we now describe typical situations, where such variability can occur (using an example of a travel service provider from [Stockhammer, 2008]). The examples also demonstrate how goal modeling plays an important role in this context.

Load-Balancing between services is important to achieve the customer goal: *high availability*. If one service is overloaded, a second service, which provides the same functionality, should overtake the incoming request, until the first service has served its stacked request queue. In order to achieve this, we need to measure the load carried by each service at runtime.

Automatic selection of the fastest service is required whenever there are different variants of the same service. The motivation for this scenario is that the client should be automatically selected to the fastest service. In order to achieve this, we need to measure the response time of the different variants of the services at runtime.

Ensuring privacy and security is important whenever third party services are used. There is variability in the level of security provided by service providers. However, this is something rather difficult to measure from outside. A common approach to guarantee a level of privacy and security is by validation through a trusted third party. This trusted party certifies the service and usually provides a certificate that expires after some time, which can be "measured" from the outside.

10.2 Monitoring Service-oriented Systems with $\mathcal{D}ecision\mathcal{K}ing$

10.2.1 Meta-model Adaptation

i* metamodel elements correspond to asset types in a variability metamodel. Figure 10.3 depicts the concrete asset meta-model which was created using $\mathcal{D}ecision\mathcal{K}ing$'s meta-model editor to parameterize the variability model editor. There are three different models involved in our modeling and monitoring process. Firstly the i* model which documents the high level goals of the stakeholders, secondly the variability model for the i* model containing automatically identified decision points. Thirdly, when the variability model is executed, we get an instance of the decision model (i.e., variability model, where the decisions have been taken, either by the user or automatically through monitor values). It is therefore logical to think of three different layers of assets, originating from the corresponding models. We identified the following asset types to be relevant and important for monitoring services at runtime.

Goal: Beginning at the highest level, this is where goals are defined in an i* model. The "Goal" block symbolizes the goal for the system, that the other components try to fulfill.

Service category: For the special case of web services, used in this thesis, the goal model has different service categories (e.g., Travel Service).

Service definition: The service categories are used to classify the service types into different categories based on their definitions.

Service types (e.g., Flight Service or Hotel Booking Service) represent an abstract service that contains one or more services.

Service: The actual implementation of a service type are the (web) services. From those services, one is used as the default service, the others are for replacement through adaptation. The services are deployed to application servers.

Metric: Different monitor categories for various basic metric types (e.g., request, response- time monitoring) exist on the right-hand side of the "Goal"-rectangle.

162
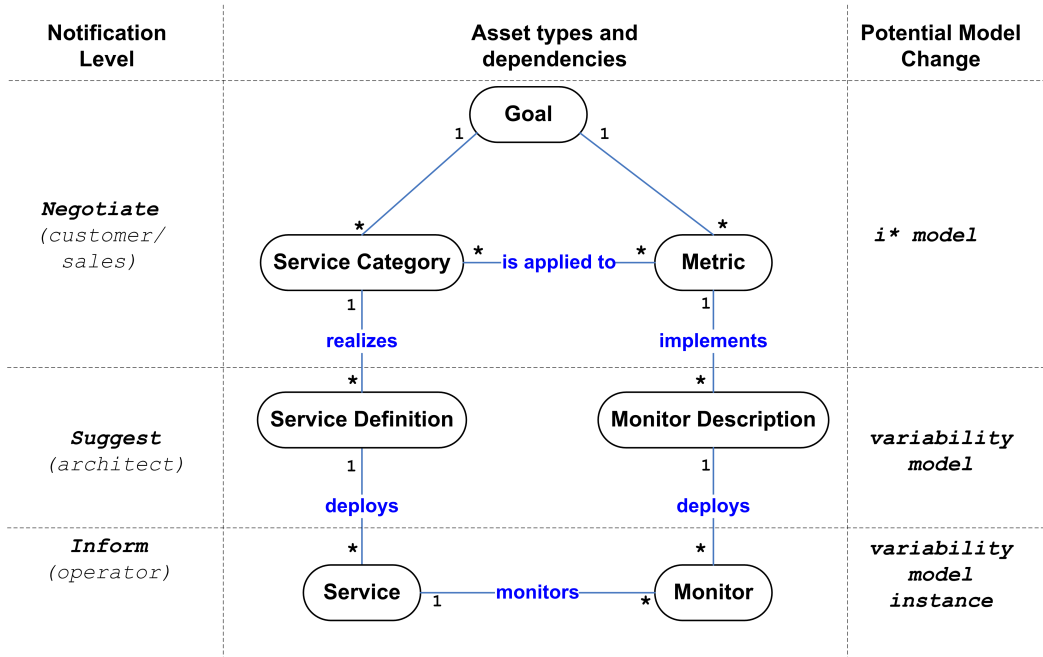
Figure 10.3: Asset meta-model used to configure $\mathcal{D}$ecision$\mathcal{K}$ing for monitoring service oriented systems at runtime.

Monitor description: For every metric, monitors can be defined using a the element monitor description. The monitor descriptions define a more specific monitoring metric as a type.

Monitor: For the monitor types, concrete implementations, the monitors, exist. A monitor is implemented to monitor one specific service. For every service to monitor, a monitor has to be implemented.

10.2.2 Tool Extensions

$\mathcal{D}$ecision$\mathcal{K}$ing was designed to be a generic variability management tool, which means it does not contain features for modeling and monitoring service oriented systems at runtime. However, due to the flexible design and extensible plugin architecture, it can be easily adapted to do so. Basically there are two aspects to be considered:

Modeling facilities include the identification of candidate variation points in i* models. As described in the previous section, we have largely automated the process of building decision models from i* models. In order to enable modeling of services, service instances and their dependencies, we also need to integrate the meta models of the i* models and $\mathcal{D}$ecision$\mathcal{K}$ing's asset meta models. *Runtime query facilities* include functionality to execute the variability models and querying the variability

models for allowed reconfigurations in case of changes. Changes can occur in two ways:

1. Bottom-up monitor-driven changes are detected automatically by the system based on runtime-parameters, e.g., response-time, service-availability etc.

2. Top-down user-driven changes are invoked explicitly by the user based on new requirements, technological changes, etc. For example, the customer wants a new payment service.

An overview of the tool architecture ($\mathcal{D}ecision\mathcal{K}ing$ + extensions, analysis tools, monitoring tools and user interaction tools) is presented in Figure 10.4. $\mathcal{D}ecision\mathcal{K}ing$ is used as the variability modeling and management engine. It is extended with special variability modeling facilities for special needs of the the domain.

i* meta-model integration is needed to facilitate the modeling of service oriented architectures (SOA). Due to the flexibility provided by SOA, systems can be easily changed at runtime– which means different parts of the architecture can change at runtime. In order to be able to track such changes at runtime and to analyze the consequences of a change, it is important to have modeling facilities to document changes according to monitored values.

We therefore integrated a JavaScript evaluation engine into $\mathcal{D}ecision\mathcal{K}ing$, which allowed us to formally describe and analyze different monitored values at runtime (e.g., response time, number of queries per second, etc.). Handling domain-specific metrics was then easy using the so called `Monitorscripts` which is enacted whenever a monitor is assigned a new value. The use of JavaScript also made the tool-suite very extensible as arbitrary algorithms can be used to analyze the consequences of a changes. `Monitorscripts` access the variability model at runtime, query for alternatives and even change the values of decisions automatically as needed.

The monitoring scripts provide the key functionality for the implementation of domain-specific metrics. Monitors are defined within the variability model and have a script either containing domain-specific computations or simply forwarding the value of the monitor to the decision model. Monitoring scripts are written in JavaScript and have access to the whole variability model. This is necessary to compute domain-specific values because the variability model in the running application is aware of the current context.

A $\mathcal{D}ecision\mathcal{K}ing$-plugin connects $\mathcal{D}ecision\mathcal{K}ing$ and a analyzer component and is called the DK2A adapter[Stockhammer, 2008]. It was required to enable querying the variability models at runtime. The DK2A component is connected to different analysis components, which fill the notification database with information analyzed and prepared for the user. The analyzer component itself is fed with data from a monitor database, which is filled by various domain-specific monitors collecting data from a running service oriented system.
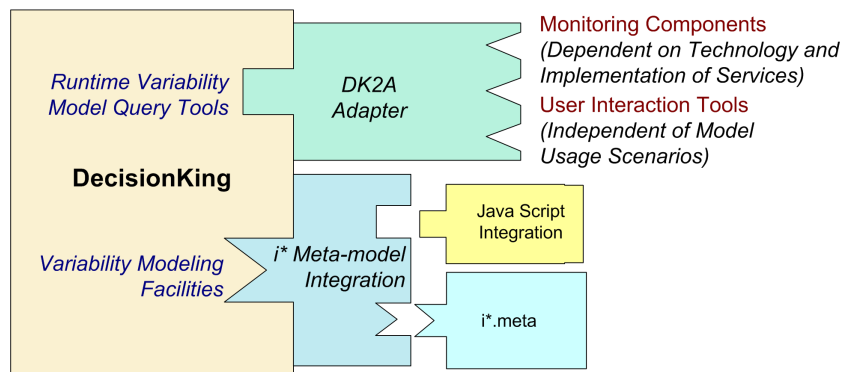
Figure 10.4: Tool architecture for monitoring services at runtime.
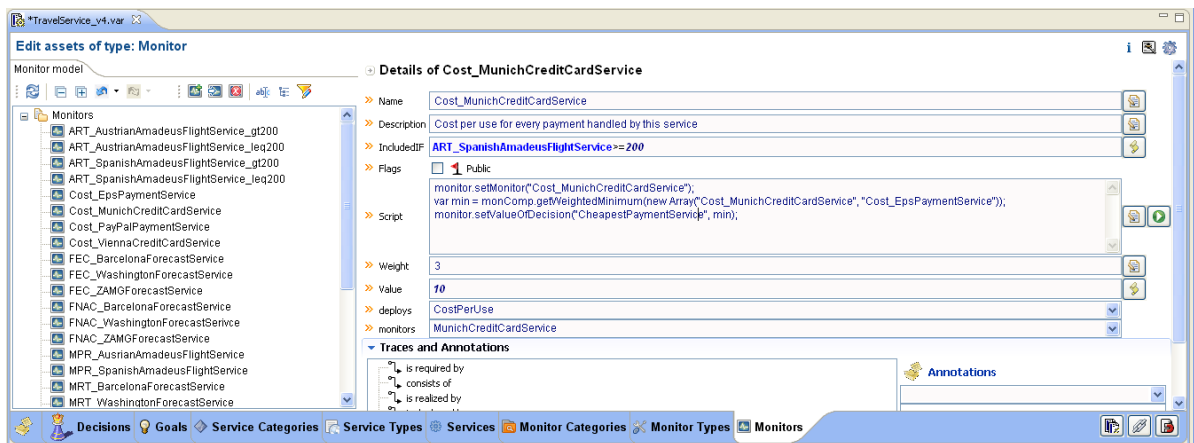
Figure 10.5: Screenshot of *DecisionKing* adapted to model the variability of service-oriented systems, depicting editor pane for monitors.

10.3 Summary

In this section, we presented our approach for automatically extracting variability models from i* goal models. The variability models were then used in the context of service-oriented systems for runtime monitoring and adaptation purposes. This case study, which was carried out together with Universitat Politècnica de Catalunya, Barcelona, was implemented by two students in their master thesis [Burgstaller, 2008, Stockhammer, 2008]. Details about the monitoring and adaptation approach can be found in the respective master theses, here we only exemplify the modeling capabilities and extensibility of our approach.

This case study provided further evidence that our modeling approach can support arbitrary asset

types and variability models can be used for different scenarios, other than in the context of product lines. We demonstrated the flexibility and adaptability of DecisionKing by describing how it was extended to match the requirements of this case study.

Part IV

Final Remarks

Chapter 11

Conclusions and Future Work

Modeling the variability of software systems involves modeling the problem space (i.e., stakeholder needs or desired features) and the solution space (i.e., the architecture and the components of the technical solution). Separation of concerns based on problem space and solution space was also dealt with by Metzger *et al.* [Metzger *et al.*, 2007] and Saleh *et al.* [Saleh & Gomaa, 2005]. Depending on the background of different researchers, the needs of different industrial contexts and the kinds of systems under investigation, a wide array of variability modeling tools and techniques are already available [Batory, 2005, Czarnecki *et al.*, 2004, Metzger *et al.*, 2007, Myllärniemi *et al.*, 2007, Schmid & John, 2004, Schobbens *et al.*, 2006].

The problem space, i.e., the decisions a customer has to take can be expressed in feature models [Kang *et al.*, 1990, Czarnecki *et al.*, 2004] or decision models [Schmid & John, 2004]. Similarly, methods, languages, and tools, have been proposed to model the solution space, i.e., the architecture and the components of the technical solution [Garlan *et al.*, 1994, van Ommering *et al.*, 2000, Medvidovic & Taylor, 2000]. Knowledge at the architectural level can be captured using existing architecture description languages (ADLs) [Medvidovic *et al.*, 1996]. Several extensions have been developed to ADLs to better support the modeling of product line aspects and architectural variability such as optional or variant components or connectors. Examples are Koala [van Ommering *et al.*, 2000] and the xADL[Dashofy *et al.*, 2002] product line extensions.

Several modeling approaches have been proposed to provide a remedy for challenges arising from product development processes practiced today. Some of these challenges are [Dhungana *et al.*, 2006]:

- Detailed knowledge about software architecture (implicit assumptions and rationale behind design decisions) is only in the minds of software engineers and not documented explicitly.

- There are only few people with an overview of the whole system and it is not always clear how the architectural knowledge is distributed.

- The needs of customers are best understood by sales people but not well shared with software architects. Customer requirements may be missed because of the gap between sales people and software architects.

A flexible and adaptable approach requires configurable and extensible tool support. It is important to enable *tailoring of the approach* as needed in different domains, which is also the basic idea behind Situational Method Engineering (SME). SME is the research area having the philosophy, that

system development projects should strive for controlled flexibility, being the balance between rigid general-purpose methods and adhoc, flexible development [Harmsen & Brinkkemper, 1995]. However, experience shows that many existing variability modeling tools are rather rigid and only allow minimal domain-specific adaptations. The variability implementation practices vary in diverse organizations and domains due to different concepts, technologies, and rules. A modeling tool needs to be customizable to support different types of core assets, architectural styles, or programming languages. Tools should be based on flexible meta-models that can be customized as needed.

Furthermore, large-scale software systems are often built using thousands of assets. It should be possible to create initial variability models by automatically extracting information from existing assets (*mining existing assets*). It is also important to automate the detection of changes in the asset base, to expedite the update of the models and to ensure their consistency with the asset base.

A variability modeling tool should support structuring of the modeling space as it is impossible for individual engineers or even a small team to create and maintain a variability model of the complete system in a large-scale system. Different teams are in charge of different parts of the product. Support for the distributed and coordinated creation of variability models created from different perspectives are thus essential.

The real value of a model is actually perceived, when the variability models can be utilized. Tool-support is therefore needed for different model utilization scenarios, e.g., for deriving a product from a product line, informing users of runtime reconfiguration possibilities or supporting product customization based on variability models. *Supporting model utilization* is crucial for the success of a modeling tool.

Many existing approaches assume a fairly stable software system, which enables the definition of domain and variability models. Through the collaboration with our industry partner Siemens VAI, we have learned that such stability cannot be taken for granted. Rather, we have observed continuous evolution in their product line. *Supporting model evolution* is therefore as important as the modeling part itself.

11.1   Modeling Approach

The widespread adoption of product line engineering is still hampered by the fact that existing approaches cannot be tailored to deal with organizational specifics such as architectural styles, languages, or modeling notations. Many of those approaches focus on process aspects and support only general-purpose modeling techniques such as feature modeling. Managing different kinds of assets in a PL relies on the precise definition of their specific characteristics in a domain-specific meta-model. Building such a model requires knowledge about the domain and the organization's settings and specifics. The meta-model defines the types of assets to be included in the product line (e.g., Components, Services, Documents, Properties, etc.) and the possible relationships between the different asset types. Based on an analysis of current practices and needs of our industrial partner, we have developed a model-based approach for defining, managing, and utilizing product lines [Dhungana *et al.*, 2007b,

Rabiser *et al.*, 2007].

An asset model is created on the basis of a domain-specific meta-model and describes the concrete assets in a product line and dependencies among them. If product line development does not start from scratch and core assets already exist, asset models can often be created semi-automatically. For example, call dependencies as defined in existing system configuration files can be utilized to automatically derive requires dependencies among assets that reflect the underlying technical restrictions. This dependency information is essential during later product derivation.

Variability stemming from technical or marketing considerations is expressed using decisions to be taken during product derivation. Decision models link external variability (decisions to be taken by customers or sales and marketing staff) with internal variability (customization and configuration decisions to be taken by engineers). Decision models reduce the complexity of modeling because variability is represented at a higher level of abstraction. This means that the variability mechanism in the asset base can be changed without having to change the variation points of the system.

In order to link assets and decisions, assets specify an inclusion condition which has to be satisfied for a particular asset to be included in the final product. Such a condition is a Boolean expression that can be composed of an arbitrary combination of decisions. The use of decisions and inclusion conditions allows establishing trace links between user demands and assets. We treat decisions as variables which can have special relationships to other variables. The relationships are expressed using a rule language. Decisions are presented to decision-takers in the form of questions. Validity conditions restrict the range of possible values. A decision model is a graph where the nodes represent decisions and the edges represent relationships between them. The core meta-model currently supports hierarchical dependencies specifying how the decisions are organized and logical dependencies specifying the known consequences of taking decisions.

11.2   Tool support

Tools for variability modeling need to fulfill a set of key requirements, which were considered in designing and implementing *DecisionKing*. It was clear early on the project that a rigid, one-customer tool can not deal with the diverse implementation practices in different domains. Therefore the tools supporting our approach had to be designed such that their functionality can be extended without the need to re-compile and re-distribute. It was important to enable addition of new functionality without requiring access to the original source code. Such flexibility requirements dealing with the flexibility and extensibility of the tool drove us to create an application framework for variability modeling. We also added meta-modeling capabilities to our tools to allow for easy adaptability.

A novelty of our approach is that the modeling tool *DecisionKing* has been designed for flexibility and extensibility early on. It contains a meta-model editor, which allows creating a domain-specific variability modeling tool for a particular organization. The editor treats variability as a prime modeling concept and supports the definition of an arbitrary number of domain-specific assets types and dependency types between model elements. It also provides a plug-in architecture that allows users to easily

extend it with company-specific plug-ins. We have been developing several plug-ins for Siemens VAI, demonstrating the feasibility of our meta-tool capabilities in several case studies, e.g., in the areas of industrial automation, ERP systems, and service-oriented systems. Another novelty lies in the fact that $\mathcal{DecisionKing}$ is not only a software engineering tool with a traditional user interface. Instead, the tool offers an API that makes it possible to include $\mathcal{DecisionKing}$ as a variability management component in different contexts. For instance, we have been using $\mathcal{DecisionKing}$ as a "variability management engine" in several case studies.

11.3   Ongoing and Future Work

We are currently working on more formal representations of decision-oriented variability models and their formal semantics. One longer term goal in this perspective is the formal definition and comparison to other available decision modeling approaches.

Apart from that, we are continuously extending the expression language used in our tool suite, which gives us the power to express variability constructs using functions at a higher level of abstraction. This includes implementation of different set operators, actions and other functions to model the dependencies among decisions/assets.

Ongoing work includes consistency checking and static analysis of decision-oriented models (e.g., by converting them into constraint satisfaction problems or petri nets) and further validation of the approach and tools in real world examples of our industry partner.

66 There will come a time when you believe everything is finished. That will be the beginning. 99 —*Louis L'Amour*

Bibliography

[Abi-Antoun *et al.*, 2006] Abi-Antoun, M., Aldrich, J., Nahas, N., Schmerl, B., & Garlan, D. 2006. Differencing and Merging of Architectural Views. *In: Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE'06)*. Tokyo,Japan: IEEE Computer Society.

[Allen & Garlan, 1997] Allen, R., & Garlan, D. 1997. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, **6**(3), 213–249.

[Anastasopoulos & Gacek, 2001] Anastasopoulos, M., & Gacek, C. 2001. Implementing product line variabilities. *Pages 109–117 of: 2001 Symposium on Software reusability: putting software reuse in context*. Toronto, Canada: ACM Press.

[Anastasopoulos & Muthig, 2004] Anastasopoulos, M., & Muthig, D. 2004. An Evaluation of Aspect-Oriented Programming as a Product Line Implementation Technology. *Pages 141–156 of:* Bosch, J., & Krueger, C. (eds), *8th International Conference on Software Reuse (ICSR 2004)*, vol. LNCS 3107. Madrid, Spain: Springer Berlin Heidelberg.

[Asikainen *et al.*, 2006] Asikainen, T., Männistö, T., & Soininen, T. 2006. A Unified Conceptual Foundation for Feature Modelling. *Pages 31–40 of: 10th International Software Product Line Conference (SPLC 2006)*. Baltimore, MD, USA: IEEE Computer Society.

[Atkinson *et al.*, 2000] Atkinson, C., Bayer, J., & Muthig, D. 2000. Component-based product line development: the KobrA Approach. *Pages 289–310 of: SPLC*.

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

[Basili, 1993] Basili, V. R. 1993. The experimental paradigm in software engineering. In Experimental Software Engineering Issues: Critical Assessment and Future Directives. *In: Dagstuhl-Workshop, H. Dieter Rombach, Victor R. Basili, and Richard Selby (eds)*. Lecture Notes in Computer Science: Springer-Verlag 1993.

[Batory, 2005] Batory, D. 2005. Feature Models, Grammars, and Propositional Formulas. *Pages 7–20 of: 9th International Software Product Line Conference (SPLC 2005)*, vol. LNCS 3714. Rennes, France: Springer Berlin Heidelberg.

[Batory *et al.*, 2006] Batory, D., Benavides, D., & Ruiz-Cortez, A. 2006. Automated analysis of feature models: challenges ahead. *Communications of the ACM*, **49**(12), 45–47.

[Bayer *et al.*, 1999] Bayer, J., Flege, O., Knauber, P., Laqua, R., Muthig, D., Schmid, K., Widen, T., & DeBaud, J. 1999. PuLSE: a methodology to develop software product lines. *Pages 122–131 of: SSR '99: Proceedings of the 1999 symposium on Software reusability*. New York, NY, USA: ACM.

[Benavides *et al.*, 2005] Benavides, D., Segura, S., Trinidad, P., & Ruiz-Cortes, A. 2005. Using Java CSP Solvers in the Automated Analyses of Feature Models. *In: Generative and Transformational Techniques in Software Engineering (GTTSE'05)*.

[Bentley, 1986] Bentley, J. 1986. Little Languages. *Communications of the ACM*, **29**(8), 711–721.

[Berg *et al.*, 2005] Berg, K., Bishop, J., & Muthig, D. 2005. Tracing software product line variability: from problem to solution space. *Pages 182–191 of: SAICSIT '05: Proceedings of the 2005 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries*. , Republic of South Africa: South African Institute for Computer Scientists and Information Technologists.

[Bézivin & Gerbé, 2001] Bézivin, J., & Gerbé, O. 2001. Towards a Precise Definition of the OMG/MDA Framework. *Pages 273–281 of: ASE '01: Proceedings of the 16th IEEE international conference on Automated software engineering*. Washington, DC, USA: IEEE Computer Society.

[Bosch, 2000] Bosch, J. 2000. *Design and use of software architectures: adopting and evolving a product-line approach*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co.

[Bosch *et al.*, 2002] Bosch, J., Florijn, G., Greefhorst, D., Kuusela, J., Obbink, J. H., & Pohl, K. 2002. Variability Issues in Software Product Lines. *Pages 13–21 of: PFE '01: Revised Papers from the 4th International Workshop on Software Product-Family Engineering*. London, UK: Springer-Verlag.

[Broek *et al.*, 2008] Broek, Pim van den, Galvao, Ismenia, & Noppen, Joost. 2008. Elimination of Constraints from Feature Trees. *In: First Workshop on Analyses of Software Product Lines in conjunction with Software Product Line Conference, SPLC 2008*.

[Burgstaller, 2008] Burgstaller, B. 2008. *Runtime Adaptation of Service-oriented Systems Based on Product Line Variability Models*. Institute for Systems Engineering and Automation, vol. Masters thesis. Linz: Johannes Kepler University.

[Campbell *et al.*, 1990] Campbell, G. H., Faulk, S. R., & Weiss, D. M. 1990. *Introduction To Synthesis*. Tech. rept. Software Productivity Consortium, Herndon, VA, USA.

[Cechticky *et al.*, 2004] Cechticky, V., Pasetti, A., Rohlik, O., & Schaufelberger, W. 2004. XML-Based Feature Modelling. *Pages 101–114 of:* Bosch, J., & Krueger, C. (eds), *8th International Conference on Software Reuse (ICSR 2004)*, vol. LNCS 3107. Madrid, Spain: Springer Berlin Heidelberg.

[Charles *et al.*, 2007] Charles, P., Fuhrer, R. M., & Sutton, S. M. 2007. IMP: a meta-tooling platform for creating language-specific IDEs in Eclipse. *Pages 485–488 of: ASE '07: Proceedings of the twenty-second IEEE/ACM Int'l Conf. on Automated software engineering*. New York, NY, USA: ACM.

[Chen, 1976] Chen, P. P. 1976. The entity-relationship model—toward a unified view of data. *ACM Transactions on Database Systems (TODS)*, **1**(1), 9–36.

[Clayberg & Rubel, 2006] Clayberg, E., & Rubel, D. 2006. *Eclipse: Building Commercial-Quality Plug-ins*. 2 edn. The Eclipse Series. Addison-Wesley Professional.

[Clements & Northrop, 2001] Clements, P., & Northrop, L. 2001. *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering, Addison-Wesley.

[Clotet *et al.*, 2007a] Clotet, R., Franch, X., López, L., Marco, J., Seyff, N., & Grünbacher, P. 2007a. The Meaning of Inheritance in i*. *In: 17th International Workshop on Agent-oriented Information Systems (AOIS-2007)*.

[Clotet *et al.*, 2007b] Clotet, R., Xavier, F., Grünbacher, P., López, L., Marco, J., Quintus, M., & Seyff, N. 2007b. Requirements Modelling for Multi-Stakeholder Distributed Systems: Challenges and Techniques. *In: RCIS'07: 1st IEEE Int. Conf. on Research Challenges in Information Science*.

[Clotet *et al.*, 2008] Clotet, R., Dhungana, D., Franch, X., Grünbacher, P, López, L., Marco, J., & Seyff, N. 2008. Dealing with Changes in Service-Oriented Computing Through Integrated Goal and Variability Modeling. *Pages 43–52, http://www.icb.uni–due.de/researchreports/ of: Second International Workshop on Variability Modelling of Software-intensive Systems (VAMOS 2008)*. Essen, Germany: ICB-Research Report No. 22, University of Duisburg Essen.

[Consortium, 1991] Consortium, Software Productivity. 1991. *Synthesis Guidebook*. Tech. rept. SPC-91122-MC. Herndon, Virginia: Software Productivity Consortium.

[Conway, 1968] Conway, M.E. 1968. How Do Committees invent? *Datamation*, **14**(4), 28–31.

[Coplien *et al.*, 1998] Coplien, J., Hoffman, D., & Weiss, D. 1998. Commonality and Variability in Software Engineering. *IEEE Software*, **15**(6), 37–45.

[Czarnecki & Antkiewicz, 2005] Czarnecki, K., & Antkiewicz, M. 2005. Mapping Features to Models: A Template Approach Based on Superimposed Variants. *Pages 422–437 of: Proceedings of the Fourth International Conference on Generative Programming and Component Engineering*. Tallinn, Estonia: Springer-Verlag, LNCS 3676.

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

[Czarnecki & Kim, 2005] Czarnecki, K., & Kim, C.H.P. 2005. Cardinality-Based Feature Modeling and Constraints: A Progress Report. *Pages 1–9 of: International Workshop on Software Factories at OOP-SLA'05*. San Diego, USA: ACM Press.

[Czarnecki & Pietroszek, 2006] Czarnecki, K., & Pietroszek, K. 2006. Verifying feature-based model templates against well-formedness OCL constraints. *Pages 211–220 of: GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*. New York, NY, USA: ACM.

[Czarnecki *et al.*, 2004] Czarnecki, K., Helson, S., & Eisenecker, U.W. 2004. Staged configuration using feature models. *Pages 266–283 of:* Nord, R. (ed), *Lecture Notes in Computer Science, Software Product Lines, Third International Conference (SPLC 2004)*, vol. LNCS 3154. Springer-Verlag.

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

[Czarnecki *et al.*, 2006] Czarnecki, K., Kim, C. H. P., & Kalleberg, K. T. 2006. Feature Models are Views on Ontologies. *Pages 41–51 of: SPLC '06: Proceedings of the 10th International on Software Product Line Conference*. Washington, DC, USA: IEEE Computer Society.

[Dashofy *et al.*, 2007] Dashofy, E., Asuncion, H., Hendrickson, S., Suryanarayana, G., Georgas, J., & Taylor, R. 2007. ArchStudio 4: An Architecture-Based Meta-Modeling Environment. *29th International Conference on Software Engineering. ICSE'07 Companion.*, May, 67–68.

[Dashofy *et al.*, 2001] Dashofy, E.M., van der Hoek, A., & Taylor, R.N. 2001. A Highly-Extensible, XML-Based Architecture Description Language. *Pages 103–112 of: Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*. Amsterdam, The Netherlands: IEEE Computer Society.

[Dashofy *et al.*, 2002] Dashofy, E.M., van der Hoek, A., & Taylor, R.N. 2002. An Infrastructure for the Rapid Development of XML-based Architecture Description Languages. *In: International Conference on Software Engineering (ICSE 2002)*.

[Dhungana *et al.*, 2006] Dhungana, D., Rabiser, R., Grünbacher, P., Prähofer, H., Federspiel, C., & Lehner, K. 2006. Architectural Knowledge in Product Line Engineering: An Industrial Case Study. *Pages 186–197 of: 32nd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. Cavtat/Dubrovnik, Croatia: IEEE CS.

[Dhungana *et al.*, 2007a] Dhungana, D., Rabiser, R., & Grünbacher, P. 2007a. Decision-Oriented Modeling of Product Line Architectures. *In: Sixth Working IEEE/IFIP Conference on Software Architecture*.

[Dhungana *et al.*, 2007b] Dhungana, D., Grünbacher, P., & Rabiser, R. 2007b. Domain-specific Adaptations of Product Line Variability Modeling. *In: IFIP WG 8.1 Working Conference on Situational Method Engineering: Fundamentals and Experiences*.

[Dhungana *et al.*, 2007c] Dhungana, D., Rabiser, R., Grünbacher, P., Lehner, K., & Federspiel, C. 2007c. DOPLER: An Adaptable Tool Suite for Product Line Engineering. *Pages 151–152 of: 11th International Software Product Line Conference (SPLC 2007), Tool Demonstration*, vol. Second Volume. Kyoto, Japan: Kindai Kagaku Sha Co. Ltd.

[Dhungana *et al.*, 2007d] Dhungana, D., Rabiser, R., Grünbacher, P., & Neumayer, T. 2007d. Integrated Tool Support for Software Product Line Engineering. *In: Tool Demonstration, 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*.

[Dhungana *et al.*, 2008] Dhungana, D., Neumayer, T., Grünbacher, P., & Rabiser, R. 2008. Supporting Evolution of Product Line Architectures With Variability Model Fragments. *In: Working IEEE/IFIP Conference on Software Architecture, WICSA 2008*.

[Dolan *et al.*, 1998] Dolan, T., Weterings, R., & Wortmann, J.C. 1998. Stakeholders in Software-System Family Architectures. *Pages 172–187 of:* van der Linden, F. (ed), *Second Int'l ESPRIT ARES Workshop on Development and Evolution of Software Architectures for Product Families (ARES'98)*, vol. LNCS 1429. Las Palmas de Gran Canaria, Spain: Springer Berlin Heidelberg.

[Eriksson *et al.*, 2005a] Eriksson, M., Börstler, J., & Borg., K. 2005a. The PLUSS Approach - Domain Modeling with Features, Use Cases and Use Case Realizations. *Pages 33–44. of: 9th International Software Product Line Conference*. Rennes, France: Springer Verlag.

[Eriksson *et al.*, 2005b] Eriksson, M., Morast, H., Börstler, J., & Borg, K. 2005b. The PLUSS toolkit: extending telelogic DOORS and IBM-rational rose to support product line use case modeling. *Pages 300–304 of: 20th IEEE/ACM international Conference on Automated Software Engineering (ASE'05)*. Long Beach, CA, USA: ACM.

[Estublier & Vega, 2005] Estublier, J., & Vega, G. 2005. Reuse and Variability in Large Software Applications. *Pages 316–325 of: 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*. Lisbon, Portugal: ACM Press.

[Fayad *et al.*, 1999] Fayad, E. M., Schmidt, C. D., & Johnson, R. E. 1999. *Building application frameworks: object-oriented foundations of framework design*. New York, NY, USA: John Wiley & Sons, Inc.

[Federspiel *et al.*, 2005] Federspiel, C., Bogner, J., Hübner, N., Leitner, R., Oberaigner, W., König, K., & Lindenberger, L. 2005. Next Generation Level2 Systems for Continuous Casting. *In: 5th European Continuous Casting Conference (ECCC)*. Nice, France: IOM Communications Ltd.

[Forgy & Shepard, 1987] Forgy, C. L., & Shepard, S. J. 1987. Rete: a fast match algorithm. *AI Expert*, **2**(1), 34–40.

[Forster *et al.*, 2008] Forster, T., Muthig, D., & Pech, D. 2008. Understanding Decision Models : Visualization and Complexity reduction of Software Variability. *Pages 111–119 of:* Heymans, P., Kang, K.C., Metzger, A., & Pohl, K. (eds), *Second International Workshop on Variability Modeling of Software-Intensive Systems*, vol. 22. Essen, Germany: ICB Research Report.

[Fritsch *et al.*, 2002] Fritsch, C., Lehn, A., & Strohm, T. 2002. Evaluating variability implementation mechanisms. *Pages 59–64 of: 2nd Int'l Workshop on Product Line Engineering*. Seattle, USA: Fraunhofer IESE (Technical Report No. 056.02/E).

[Froschauer, 2009] Froschauer, R. 2009. *Managing the Life-cycle of Industrial Automation Systems with Product Line Variability Models*. Ph.D. thesis, Johannes Kepler University.

[Froschauer *et al.*, 2006] Froschauer, R., Auinger, F., Grabmair, G., & Strasser, T. 2006. Automatic control application recovery in distributed IEC 61499 based automation and control systems. *IEEE Workshop on Distributed Intelligent Systems: Collective Intelligence and Its Applications*, June, 103–108.

[Froschauer *et al.*, 2008] Froschauer, R., Dhungana, D., & Grünbacher, P. 2008. Managing the Life-cycle of Industrial Automation Systems with Product Line Variability Models. *In: 34th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*.

[Gamma & Beck, 2003] Gamma, E., & Beck, K. 2003. *Contributing to Eclipse: Principles, Patterns, and Plug-Ins*. The Eclipse Series. Amsterdam: Addison-Wesley Longman.

[Garlan *et al.*, 1994] Garlan, D., Allen, R., & Ockerbloom, J. 1994. Exploiting Style in Architectural Design Environments. *Pages 175–188 of: Foundations of Software Engineering*.

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

[Grabmair *et al.*, 2006] Grabmair, G., Froschauer, R., Strasser, T., & Zoitl, A. 2006. Modelling Execution Order and Real-time Constraints in IEC 61499 Control Applications. *Distributed Intelligent Systems: Collective Intelligence and Its Applications, 2006. DIS 2006. IEEE Workshop on*, June, 115–120.

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

[Griss *et al.*, 1998] Griss, M. L., Favaro, J., & d' Alessandro, M. 1998. Integrating Feature Modeling with the RSEB. *Pages 76–86 of: ICSR '98: Proceedings of the 5th International Conference on Software Reuse*. Washington, DC, USA: IEEE Computer Society.

[Grubb & Takang, 2005] Grubb, P., & Takang, A. 2005. *Software maintenance, concepts and practice*. 2nd ed. edn. World scientific publishing.

[Gruler *et al.*, 2007] Gruler, A., Harhurin, A., & Hartmann, J. 2007. Development and Configuration of Service-based Product Lines. *Pages 107–116 of: 11th International Software Product Line Conference (SPLC 2007)*. Kyoto, Japan: IEEE CS.

[Grünbacher *et al.*, 2008] Grünbacher, P., Rabiser, R., & Dhungana, D. 2008. Product Line Tools Are Product Lines Too: Lessons Learned from Developing a Tool Suite. *In: 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)*.

[Grundy *et al.*, 2006] Grundy, J., Hosking, J., Zhu, N., & Liu, N. 2006. Generating Domain-Specific Visual Language Editors from High-level Tool Specifications. *Pages 25–36 of: 21st IEEE International Conference on Automated Software Engineering (ASE'06)*. Tokyo, Japan: IEEE.

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

[Halmans & Pohl, 2004] Halmans, G., & Pohl, K. 2004. Communicating the Variability of a Software-Product Family to Customers. *Informatik - Forschung und Entwicklung*, **18**(3-4), 113–131.

[Harel & Rumpe, 2000] Harel, D., & Rumpe, B. 2000. *Modeling Languages: Syntax, Semantics and All That Stuff, Part I: The Basic Stuff*. Tech. rept. Jerusalem, Israel, Israel.

[Harel & Rumpe, 2004] Harel, D., & Rumpe, B. 2004. Meaningful Modeling: What's the Semantics of "Semantics"? *Computer*, **37**(10), 64–72.

[Harmsen & Brinkkemper, 1995] Harmsen, F., & Brinkkemper, S. 1995. Design and Implementation of a Method Base Management System for a Situational CASE Environment. *Pages 430–438 of: APSEC*.

[Harry *et al.*, 2005] Harry, C. L., Krishnamurthi, S., & Fisler, K. 2005. Modular Verification of Open Features Using Three-Valued Model Checking. *Journal of Automated Software Engineering*, **12**(3), 349–382.

[Hazael-Massieux & Rosenthal, 2005] Hazael-Massieux, D., & Rosenthal, L. 2005. *Variability in Specifications*. Tech. rept. W3C Quality Assurance (QA) Group.

[Helferich *et al.*, 2006] Helferich, Andreas, Schmid, Klaus, & Herzwurm, Georg. 2006. Product management for software product lines: an unsolved problem? *Commun. ACM*, **49**(12), 66–67.

[Herbsleb & Grinter, 1999] Herbsleb, J. D., & Grinter, R. E. 1999. Architectures, Coordination, and Distance: Conway's Law and Beyond. *IEEE Software*, **16**(5), 63–70.

[Heymans *et al.*, 2007] Heymans, P., Schobbens, P., Trigaux, J., Matulevicius, R., Classen, A., & Bontemps, Y. 2007. Towards a comparative Evaluation of Feature Diagram Languages. *Pages 1–16 of: Software and Services Variability Management Workshop- Concepts, Models and Tools*. Helsinki, Finland.: HUT-SoberIT-A3.

[Hummer *et al.*, 2006] Hummer, O., Sunder, C., Zoitl, A., Strasser, T., Rooker, M.N., & Ebenhofer, G. 2006. Towards Zero-downtime Evolution of Distributed Control Applications via Evolution Control based on IEC 61499. *IEEE Conference on Emerging Technologies and Factory Automation, 2006. ETFA '06.*, Sept., 1285–1292.

[Jaaksi, 2002] Jaaksi, A. 2002. Developing mobile browsers in a product line. *IEEE Software*, **19**(4), 73–80.

[Jacobson, 1994a] Jacobson, I. 1994a. Business process reengineering with object technology. *Object Mag.*

[Jacobson, 1994b] Jacobson, I. 1994b. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc.

[Janota & Kiniry, 2007] Janota, M., & Kiniry, J. 2007. Reasoning about Feature Models in Higher-Order Logic. *Pages 13–22 of: SPLC '07: Proceedings of the 11th International Software Product Line Conference*. Washington, DC, USA: IEEE Computer Society.

[John, 2001] John, I. 2001. Integrating Legacy Documentation Assets into a Product Line. *Pages 113–124 of:* van der Linden, F. (ed), *Lecture Notes in Computer Science: Software Product Family Engineering: 4th Int'l Workshop, PFE 2001*, vol. LNCS 2290. Springer Berlin / Heidelberg.

[Kang *et al.*, 1990] Kang, K.C., Cohen, S., Hess, J., Nowak, W., & Peterson, S. 1990. *Feature-oriented domain analysis (FODA) feasibility study*. Tech. rept. Technical Report CMU/SEI-90TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA.

[Kang *et al.*, 2002] Kang, K.C., Donohoe, P., Koh, E., Lee, J., & Lee, K. 2002. Using a Marketing and Product Plan as a Key Driver for Product Line Asset Development. *Pages 366–382 of:* Chastek, G. (ed), *Lecture Notes in Computer Science: Second Software Product Line Conference - SPLC 2*, vol. LNCS 2379. Springer Berlin / Heidelberg.

[Kang *et al.*, 1998] Kang, Kyo C., Kim, S., Lee, J., Kim, K., Shin, E., & Huh, M. 1998. FORM: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, **5**, 143–168.

[Kleppe *et al.*, 2003] Kleppe, A. G., Warmer, J., & Bast, W. 2003. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

[Kraut & Streeter, 1995] Kraut, R. E., & Streeter, L. A. 1995. Coordination in software development. *Communications of ACM*, **38**(3), 69–81.

[Lewis, 2001] Lewis, R. W. 2001. *Modelling Control Systems Using IEC 61499: Applying Function Blocks to Distributed Systems*. IEEE Control Series. Institution of Engineering and Technology.

[Liaskos *et al.*, 2006] Liaskos, S., Lapouchnian, A., Yu, Y., Yu, E., & Mylopoulos, J. 2006. On Goal-based Variability Acquisition and Analysis. *Pages 76–85 of: RE '06: Proceedings of the 14th IEEE International Requirements Engineering Conference (RE'06)*. Washington, DC, USA: IEEE Computer Society.

[Magee & Kramer, 1995] Magee, J., Dulay N. Eisenbach S., & Kramer, J. 1995. Specifying Distributed Software Architectures. *Pages 137–153 of: In Proceedings of the 5th European Software Engineering Conference*. Lecture Notes In Computer Science, vol. 989. Springer-Verlag, London.

[Männistö *et al.*, 2001] Männistö, T., Soininen, T., & Sulonen, R. 2001. Modelling Configurable Products and Software Product Families. *Pages 64–70 of: Workshop on Configuration, collocated with the 18th International Joint Conference on Artificial Intelligence (IJCAI-01)*.

[Mansell & Sellier, 2004] Mansell, J.X., & Sellier, D. 2004. Decision Model and Flexible Component Definition Based on XML Technology. *Pages 466–472 of: Lecture Notes in Computer Science: Software Product-Family Engineering 5th International Workshop, PFE 2003*. Springer Berlin/Heidelberg.

[McAffer & Lemieux, 2005] McAffer, J., & Lemieux, J. 2005. *Eclipse Rich Client Platform: Designing, Coding, and Packaging Java(TM) Applications*. Addison-Wesley Professional.

[McGregor, June 2003] McGregor, John D. June 2003. *The Evolution of Product Line Assets*. Technical Report CMU/SEI-2003-TR-005 ESC-TR-2003-005. CMU/SEI.

[Medvidovic & Taylor, 2000] Medvidovic, N., & Taylor, R.N. 2000. A Classificiation and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, **26**(1), 70–93.

[Medvidovic *et al.*, 1996] Medvidovic, N., Oreizy, P., Robbins, J.E., & Taylor, R.N. 1996. Using Object-Oriented Typing to Support Architectural Design in the C2t'Style. *Pages 24–32 of: Fourth Symposium on Foundations of Software Engineering (FSE'96)*.

[Mellor *et al.*, 2004] Mellor, S. J., Kendall, S., Uhl, A., & Weise, D. 2004. *MDA Distilled*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc.

[Metzger *et al.*, 2007] Metzger, A., Heymans, P., Pohl, K., Schobbens, P.-Y., & Saval, G. 2007. Disambiguating the Documentation of Variability in Software Product Lines: A Separation of Concerns, Formalization and Automated Analysis. *Pages 243–253 of: 15th IEEE International Requirements Engineering Conference (RE'07)*.

[Mössenböck, 1991] Mössenböck, Hanspeter. 1991. A generator for production quality compilers. *Pages 42–55 of: CC '90: Proceedings of the third international workshop on Compiler compilers*. New York, NY, USA: Springer-Verlag New York, Inc.

[Myllärniemi *et al.*, 2007] Myllärniemi, V., Raatikainen, M., & Männistö, T. 2007. Kumbang Tools. *Pages 135–136 of: 11th International Software Product Line Conference (SPLC 2007), Tool Demonstration*, vol. Second Volume. Kyoto, Japan: Kindai Kagaku Sha Co. Ltd.

[O'Dell & Ostro, 1998] O'Dell, C., & Ostro, N. 1998. *If Only We Knew What We Know: The Transfer of Internal Knowledge and Best Practice*. Simon & Schuster. Preface By-C. Grayson.

[OSGi Alliance, 2003] OSGi Alliance. 2003. *OSGi Service Platform: The OSGi Alliance*. Tech. rept. IOS Press.

[Parnas, 1994] Parnas, D. L. 1994. Software aging. *Pages 279–287 of:* Press, IEEE Computer Society (ed), *International Conference on Software Engineering*. Sorrento Italy: IEEE.

[Parnas, 1972] Parnas, D.L. 1972. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, **15**(12), pp. 1053–1058.

[Penã *et al.*, 2006] Penã, J., Hinchey, M. G., & Ruiz-Cortés, A. 2006. Multi-agent system product lines: challenges and benefits. *Communications of ACM*, **49**(12), 82–84.

[Penserini *et al.*, 2006] Penserini, L., Perini, A., Susi, A., & Mylopoulos, J. 2006. From Stakeholder Needs to Service Requirements. *Service-Oriented Computing: Consequences for Engineering Requirements, 2006. SOCCER '06*, Sept., 8–18.

[Perry *et al.*, 1994] Perry, D.E., Staudenmayer, N.A., & Votta, L.G. 1994. People, organizations, and process improvement. *Software, IEEE*, **11**(4), 36–45.

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

[Prähofer *et al.*, 2008] Prähofer, H., Hurnaus, D., Schatz, R., Wirth, C., & Mössenböck, H. 2008. Monaco: A DSL Approach for Programming Automation Machines. *In: Software-Engineering-Konferenz*.

[Profactor, 2007] Profactor. 2007. *Framework for Distributed Industrial Automation and Control*.

[Rabiser, 2009] Rabiser, R. 2009. *A user-centered approach to product configuration in software product line engineering*. Ph.D. thesis, Johannes Kepler University.

[Rabiser *et al.*, 2007] Rabiser, R., Grünbacher, P., & Dhungana, D. 2007. Supporting Product Derivation by Adapting and Augmenting Variability Models. *In: 11th International Software Product Line Conference (SPLC 2007)*.

[Rabiser *et al.*, 2008] Rabiser, R., Dhungana, D, Grünbacher, P., & Burgstaller, B. 2008. Value-Based Elicitation of Product Line Variability: An Experience Report. *In: Second International Workshop on Variability Modelling of Software-intensive Systems (VAMOS 2008)*.

[Rabiser *et al.*, 2009] Rabiser, R., Wolfinger, R., & Grünbacher, P. 2009. Three-level Customization of Software Products Using a Product Line Approach. *In: 42nd Hawaii International Conference on System Sciences*.

[Redwine & Riddle, 1985] Redwine, S. T., & Riddle, W. E. 1985. Software technology maturation. *Pages 189–200 of: ICSE '85: Proceedings of the 8th international conference on Software engineering*. Los Alamitos, CA, USA: IEEE Computer Society Press.

[Reiser & Weber, 2006] Reiser, M.-O., & Weber, M. 2006. Managing Highly Complex Product Families with Multi-Level Feature Trees. *Pages 149–158 of: 14th IEEE Int'l Requirements Engineering Conference (RE'06)*. Minneapolis, MN, USA: IEEE CS.

[Riebisch *et al.*, 2002] Riebisch, M., Böllert, K., Streitferdt, D., & Philippow, I. 2002. Extending feature diagrams with UML multiplicities. *In: Integrated Design and Process Technology, IDPT-2002*.

[Saleh & Gomaa, 2005] Saleh, M., & Gomaa, H. 2005. Separation of concerns in software product line engineering. *Pages 1–5 of: MACS '05: Proceedings of the 2005 workshop on Modeling and analysis of concerns in software*. New York, NY, USA: ACM.

[Schmid, 1997] Schmid, H. A. 1997. Systematic framework design by generalization. *Communications of ACM*, **40**(10), 48–51.

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

[Schobbens *et al.*, 2006] Schobbens, P., Heymans, P., & Trigaux, J. 2006. Feature Diagrams: A Survey and a Formal Semantics. *re*, **0**, 139–148.

[Schobbens *et al.*, 2007] Schobbens, P., Heymans, P., Trigaux, J., & Bontemps, Y. 2007. Generic semantics of feature diagrams. *International Journal of Computer and Telecommunications Networking*, **51**(2), 456–479.

[Shaw, 2001] Shaw, M. 2001. The Coming-of-Age of Software Architecture Research. *icse*, **00**, 656.

[Shaw, 2003] Shaw, M. 2003. Writing good software engineering research papers: minitutorial. *Pages 726–736 of: ICSE '03: Proceedings of the 25th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society.

[Steger *et al.*, 2004] Steger, M., Tischer, C., Boss, B., Müller, A., Pertler, O., Stolz, W., & Ferber, S. 2004. Introducing PLA at Bosch Gasoline Systems: Experiences and Practices. *Pages 34–50 of:* Nord, R. (ed), *Third Software Product Line Conference (SPLC 2004)*. Boston, MA, USA: Springer Berlin Heidelberg.

[Stockhammer, 2008] Stockhammer, R. 2008. *Monitoring of Service-Oriented Systems Based on Variability Models*. Institute for Systems Engineering and Automation, vol. Masters thesis. Linz: Johannes Kepler University.

[Sünder *et al.*, 2006] Sünder, C., Zoitl, A., Favre-Bulle, B., Strasser, T., Steininger, H., & Thomas, S. 2006. Towards reconfiguration applications as basis for control system evolution in zero-downtime automation systems. *In: Intelligent Production Machines and Systems (IPROMS '06)*.

[Svahnberg & Bosch, 1999] Svahnberg, M., & Bosch, J. 1999. Evolution in software product lines: Two cases. *Journal of Software Maintenance: Research and Practice*, **11**(6), 391–422.

[Svahnberg *et al.*, 2005] Svahnberg, M., van Gurp, J., & Bosch, J. 2005. A Taxonomy of Variability Realization Techniques. *Software-Practice and Experience*, **35**(8), 705–754.

[Swanson, 1976] Swanson, E. B. 1976. The dimensions of maintenance. *Pages 492–497 of: ICSE '76: Proceedings of the 2nd international conference on Software engineering*. Los Alamitos, CA, USA: IEEE Computer Society Press.

[Tang *et al.*, 2007] Tang, M, Chou, S., & Dong, J. 2007. Conflicts classification and solving for collaborative feature modeling. *Advanced Engineering Informatics*, **21**(2), 211–219.

[Thiel & Hein, 2002] Thiel, S., & Hein, A. 2002. Modeling and Using Product Line Variability in Automotive Systems. *IEEE Software*, **19**(4), 66–72.

[Thurimella, 2008] Thurimella, A. K. 2008. *Issue-based Variability Modeling*. Ph.D. thesis, Technical University Munich.

[Tichy *et al.*, 1995] Tichy, W. F., Lukowicz, P., Prechelt, L., & Heinz, E. A. 1995. Experimental evaluation in computer science: a quantitative study. *Journal of System and Software*, **28**(1), 9–18.

[Tichy, 1998] Tichy, W.F. 1998. Should computer scientists experiment more? *Computer*, **31**(5), 32–40.

[Tolvanen & Rossi, 2003] Tolvanen, J. P., & Rossi, M. 2003. MetaEdit+: defining and using domain-specific modeling languages and code generators. *Pages 92–93 of: Conference on Object Oriented Programming Systems Languages and Applications (OOPLSA'03)*. Anaheim, CA, USA: ACM Press.

[van der Linden *et al.*, 2007] van der Linden, F., Schmid, K., & Rommes, E. 2007. *Software Product Lines in Action - The Best Industrial Practice in Product Line Engineering*. Springer Berlin Heidelberg.

[van Gurp *et al.*, 2001] van Gurp, J., Bosch, J., & Svahnberg, M. 2001. On the Notion of Variability in Software Product Lines. *Pages 45–54 of: Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*. Amsterdam, The Netherlands: IEEE Computer Society.

[van Ommering *et al.*, 2000] van Ommering, R., van der Linden, F., Kramer, J., & Magee, J. 2000. The Koala component model for consumer electronics software. *Computer*, **33**(3), 78–85.

[Verlage & Kiesgen, 2005] Verlage, M., & Kiesgen, T. 2005. Five years of product line engineering in a small company. *Pages 534–543 of: 27th International Conference on Software Engineering (ICSE 2005)*. St Louis, MO, USA: IEEE CS.

[Voelter & Groher, 2007] Voelter, M., & Groher, I. 2007. Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. *Pages 233–242 of: 11th International Software Product Line Conference (SPLC 2007)*. Kyoto, Japan: IEEE CS.

[Wallner, 2008] Wallner, Stefan. 2008. *Integration of a Rule Language in a Tool Suite for Software Product Line Engineering*. Linz: Master's Thesis, Johannes Kepler University.

[Walls & Breidenbach, 2007] Walls, C., & Breidenbach, R. 2007. *Spring in Action*. Manning Publications.

[Wang *et al.*, 2007] Wang, H. H., Li, Y. F., Jing, S., Zhang, H., & Pan, J. 2007. Verifying feature models using OWL. *Web Semantics: Science, Services and Agents on the World Wide Web*, **5**(2), 117–129.

[Wile & Ramming, 1999] Wile, D. S., & Ramming, J. C. 1999. Guest Editorial: Introduction to the Special Section 'Domain-Specific Languages (DSLs). *IEEE Transactions on Software Engineering*, **vol 25**(3).

[Wirth, 2008] Wirth, Christian. 2008. *Model-based Generation of End-User Programming Environments*. Linz: Master's Thesis, Johannes Kepler University.

[Wohlin *et al.*, 2000] Wohlin, C., Runeson, P., & Höst, M. 2000. *Experimentation in Software Engineering: An Introduction*. Kluwer International Series in Software Engineering.

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

[Yu, 1997] Yu, E. 1997. Towards Modeling and Reasoning Support for Early-Phase Requirements Engineering. *Page 226 of: RE '97: Proceedings of the 3rd IEEE International Symposium on Requirements Engineering (RE'97)*. Washington, DC, USA: IEEE Computer Society.

[Yu, 1996] Yu, E. S. 1996. *Modeling Strategic Relationships for Process Reengineering*. Ph.D. thesis, University of Toronto.

[Zhu *et al.*, 2007] Zhu, N., Grundy, J. C., Hosking, J. G., Liu, N., Cao, S., & Mehra, A. 2007. Pounamu: A meta-tool for exploratory domain-specific visual language tool development. *Journal of Systems and Software*, **80**(8), 1390–1407.

List of Abbreviations

DOPLER .... Decision-oriented Product Line Engineering for Effective Reuse
ADL .... Architecture Description Language
CBSE .... Component Based Software Engineering
CL2 .... Caster Level 2 Automation
CTL .... Computation Tree Logic
DoVML .... Decision-oriented Variability Modeling Language
DSL .... Domain Specific Language
EBNF .... Extended Backus-Naur Form
FODA .... Feature-oriented Domain Analysis
FORM .... Feature-oriented Reuse Method
GMF .... Eclipse Graphical Modeling Framework
GPL .... General Purpose Language
GUI .... Graphical User Interface
IoC .... Inversion of Control
KLoC .... 1000 Lines of Code
MDA .... Model-Driven Architecture
MDE .... Model Driven Engineering
MDSD .... Model-Driven Software Development
MOF .... Meta Object Facility
MSDS .... Multi-stakeholder Distributed Systems
OCL .... Object Constraint Language
OMG .... Object Management Group
OSGi .... Open Services Gateway initiative
OVM .... Orthogonal Variability Modeling
PIM .... Platform-Independent Model
PLE .... Product Line Engineering
PLUSS .... Product Line Use case modeling for System and Software engineering
PSM .... Platform-Specific Model
PuLSE .... Product Line Software Engineering
RSEB .... Reuse-Driven Software Engineering Business
SE .... Software Engineering
SME .... Situational Method Engineering
SPLE .... Software Product Line Engineering

UML	Unified Modeling Language
VAI	Voest-alpine Industrieanlagenbau
VML	Variability Modeling Language
xADL	XML based Architecture Description Language
XML	Extensible Markup Language

List of Figures

List of Tables

Listings

Textual Editor for Asset Meta-models

$\mathcal{D}ecision\mathcal{K}ing$ also supports creation of meta-model using a textual representation. Our experience showed that users (who are not familiar with textual notation) rather worked with the tabular meta-model editor. The following listing presents the EBNF for creating domain-specific refinements of the core meta-model.

Figure A.1: $\mathcal{D}ecision\mathcal{K}ing$ meta-model editor showing textual representation of Siemens VAI Meta-model.

```
VMLCompiler          =   "VML" ident "{"
                                AttributeTypeDecl
                                RelationshipTypeDecl
                                DecisionTypeDecl
                                ModelAttrDecl
                                DecisionAttrDecl
                                { DependencyTypeDecl }
                                { AssetTypeDecl }
                                RelationshipLinkDecl
                            "}".
AttributeTypeDecl    =   "ATTRIBUTE_TYPES" "[" ident { "," ident }"]" ";".
RelationshipTypeDecl =   "RELATIONSHIP_TYPES" "[" ident { "," ident } "]" ";".
DecisionTypeDecl     =   "DECISION_TYPES" "[" ident { "," ident } "]" ";".
ModelAttrDecl        =   "MODEL_ATTRIBUTES" "{" { AttrDecl } "}".
DecisionAttrDecl     =   "DECISION_ATTRIBUTES" "{" { AttrDecl } "}".
DependencyTypeDecl   = [ DependencyModifier ] "ASSET_DEPENDENCY" string
                            ":" ident "{" "inverse" string ";"
                            [ "description" string ";" ] "}".
AssetTypeDecl        =   [ "abstract" ] "ASSET_TYPE" ident
                            [ "extends" ident ]
                            "{" [ "plural" string ";" ]
                            [ "icon" string ";" ]
                            [ "description" string ";" ]
                            { AttrDecl } "}" .
RelationshipLinkDecl = "LINKS" "{" { LinkDecl } "}".
LinkDecl             = ident string ( ( "1" | "*" ) ) ident ";" .
AttrDecl             = [ AttrModifier ] ident ident   [
                            "[" string { "," string } "]" ] ";".
AttrModifier         = ( "mandatory" | "optional" ).
DependencyModifier   = ( "birectional" | "unidirectional" ).
/* end of VMLCompiler */
```

Listing A.1: EBNF used with CoCo/R to generate a scanner and a parser which were then used for syntax checking in the text based meta-model editor.

Appendix B

$\mathcal{D}ecision\mathcal{K}ing$'s Web-based Front-end

The model API provided by dk was also used to create web-based decision taking applications based on variability models. Figure B.1 depicts the start page of the application, where the user can either upload her own variability models or select an existing model to generate a questionnaire based on the decisions in the model. Figure B.2 depicts the questionnaire generated for one variability model of Siemens VAI's subsystem Caster.

Figure B.1: $\mathcal{D}ecision\mathcal{K}ing$'s web-based model execution platform.
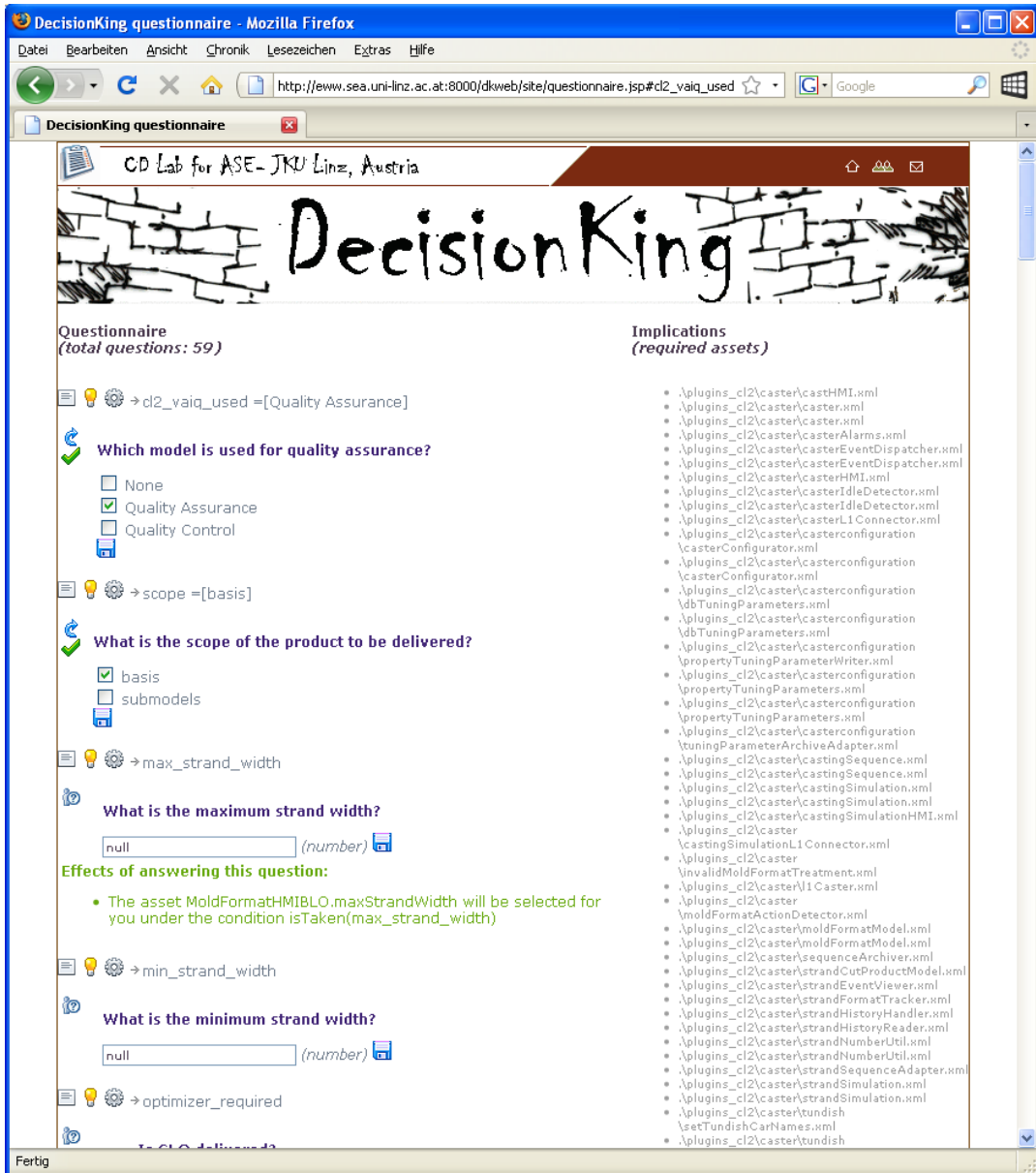
Figure B.2: Web application generated using the *DecisionKing*'s model API.