HELSINKI UNIVERSITY OF TECHNOLOGY

Department of Computer Science

Laboratory of Software Business and Engineering

Varvana Myllärniemi

Kumbang Configurator—A Tool for Configuring Software Product Families

Master's Thesis

Supervisor: Professor Tomi Männistö

Instructor: Professor Tomi Männistö

HELSINKI UNIVERSITY OF TECHNOLOGY

ABSTRACT OF THE MASTER'S THESIS

Author and name of the thesis:
Varvana Myllärniemi
Kumbang Configurator—A Tool for Configuring Software Product Families

Date: January 24th, 2005 **Number of pages:** 1+134

Department: Department of Computer Science, Laboratory of Software Business and Engineering

Professorship: T-76

Supervisor:
Professor Tomi Männistö

Instructor:
Professor Tomi Männistö

Software product families are an emerging trend that tries to cope with increasing variability and challenges of reuse. Software products are derived from the family in a prescribed way utilising common family assets. The derived product often includes some amount of integrating glue code or even product-specific tailoring.

In the domain of traditional, mechanical products, configurable products have been developed. Individual products are configured according to a predefined model, with no need for adaptive or innovative design.

When comparing these two approaches, remarkable similarities are found. In a configurable software product family, software products are configured based on a predefined configuration model. Building the product usually requires no product-specific programming.

The purpose of this thesis was to develop a tool for deriving product individuals from configurable software product families. This tool, called Kumbang Configurator, utilises existing modelling language Kumbang, which combines both feature-based and architecture-based modelling methods. Kumbang Configurator prevents configuration errors by checking whether the configuration is consistent and complete. This configuration reasoning employs inference engine *smodels*. The implementation was validated with two example cases.

Keywords: Software Product Family, Configuration, Tool Support

TEKNILLINEN KORKEAKOULU DIPLOMITYÖN TIIVISTELMÄ

Tekijä ja työn nimi:
Varvana Myllärniemi
Kumbang-konfiguraattori—Työkalu ohjelmistotuoteperheiden konfigurointiin
Date: 24. tammikuuta 2005 **Sivumäärä:** 1+134

Osasto:
Tietotekniikan osasto, Ohjelmistoliiketoiminnan ja tuotannon laboratorio

Professuuri:
T-76

Työn valvoja:
Professori Tomi Männistö

Työn ohjaaja:
Professori Tomi Männistö

Ohjelmistotuoteperheet ovat saavuttaneet suosiota tapana vastata kasvaneeseen varioituvuuteen ja uudelleenkäytön haasteisiin. Yksittäiset perheen ohjelmistotuotteet johdetaan etukäteen määritellyllä tavalla, käyttäen hyväksi yhteisiä varantoja. Usein derivoinnissa tarvitaan integroivaa koodia tai jopa tuotekohtaista räätälöintiä.

Tavallisista, mekaanisista tuotteista on kehitetty konfiguroitavia tuotteita. Yksittäiset tuotteet konfiguroidaan käyttäen hyväksi ennaltasuunniteltua mallia, lisäämättä jälkikäteen muokkaavaa tai innovatiivista suunnittelua.

Näiden lähestymistapojen välillä on huomattavia yhteyksiä. Konfiguroitavassa ohjelmistotuoteperheessä tuoteyksilöt konfiguroidaan ennaltasuunnitellun konfiguraatiomallin pohjalta. Varsinainen tuotteen rakentaminen ei vaadi ollenkaan tuotekohtaista ohjelmointia.

Tämän diplomityön tarkoituksena oli kehittää työkalu tuoteyksilöiden johtamiseen konfiguroitavasta ohjelmistuoteperheestä. Tämä työkalu, nimeltään Kumbang-konfiguraattori, perustuu Kumbang-kieleen. Kumbang-kielessä yhdistyvät sekä ominaisuuslähtöiset että arkkitehtuurilähtöiset mallitustavat. Kumbang-konfiguraattori tarkastaa konfiguraation oikeellisuutta ja valmiutta, ja täten ehkäisee konfiguraatiossa syntyviä virheitä. Konfiguraatiopäättely käyttää hyväkseen *smodels*-päättelykonetta. Toteutettu työkalu validoitiin kahdella esimerkkitapauksella.

Avainsanat: Ohjelmistotuoteperhe, Konfiguraatio, Työkalutuki

Acknowledgments

I would like to thank the following persons:

Timo Asikainen for his invaluable help and support - this thesis wouldn't have seen daylight without his knowledge of Kumbang and his willingness to answer all my questions.

Professor Tomi Männistö for his support, guidance and suggestions for improvement. He is clearly the best supervisor I've had so far. :)

Professor Timo Soininen for his comments and suggestions.

Andreas Andersson for his guidance on WeCoTin and suggestions for the implementation.

John MacGregor for allowing me to use a case from Robert Bosch GmbH.

Ville Partanen for providing the best home support one can ever have.

This research has been supported by National Technology Agency of Finland (Tekes) and Academy of Finland (project number 51394).

Espoo, December 2004

Varvana Myllärniemi

Contents

1 Introduction **7**

1.1 Background 7

1.2 Thesis Structure 11

2 Software Product Families **12**

2.1 Basics 12

2.1.1 Variability and Variation Points 14

2.1.2 Domain and Application Engineering 15

2.2 Domain Engineering in Software Product Families 17

2.2.1 Modelling Requirements with Feature Models 19

2.2.2 Modelling Product Family Architecture 21

2.3 Application Engineering in Software Product Families 24

3 Configurable Product Families **26**

3.1 Background 26

3.2 Modelling Configurable Product Families 28

3.2.1 Configuration Modelling Concepts 29

3.2.2 Example Configuration Model 31

3.3 Deriving Configurable Product Individuals 33

3.3.1 WeCoTin Configurator 34

4 Configurable Software Product Families **37**

4.1 Basics of Configurable Software Product Families 38

4.1.1 Configurable Product Base as the Highest Level of Reuse . . . 39

4.1.2 Applying Traditional Configuration Techniques to Software . 40

4.2 Koalish, Forfamel and Kumbang 42
4.2.1 Koala 42
4.2.2 Koalish 45
4.2.3 Forfamel 48
4.2.4 Kumbang 50

5 Research Aims **52**
5.1 Research Objectives 52
5.2 Research Questions 53
5.3 Research Method 54
5.4 Scope 55

6 System Implementation **57**
6.1 Requirements 57
6.1.1 How To Obtain Requirements 57
6.1.2 Overall Description of the System 59
6.1.3 Detailed Requirements 61
6.2 How the Tool Works 66
6.2.1 Implementation of Configuration Reasoning 70
6.2.2 Implemented Requirements 71
6.3 System Architecture 72
6.3.1 Language and Platform 72
6.3.2 System Context 74
6.3.3 Structural View 74
6.3.4 Layered View 78
6.3.5 Code Architecture View 80
6.4 Contribution from Other Developers 85

7 System Validation **86**
7.1 Case: Car Periphery System 86
7.1.1 Description of the Case 86
7.1.2 Constructing the Configuration Model 87
7.1.3 Configuration Task 92
7.2 Case: Weather Station Network 92

7.2.1 Description of the Case 92
7.2.2 Constructing the Configuration Model 93
7.2.3 Configuration Task 95
7.3 Validation Conclusions 96

8 Discussion **97**
8.1 Evaluation 97
8.1.1 Evaluation of Basic Requirements 97
8.1.2 Evaluation of Configuration Requirements 99
8.1.3 Evaluation of User Interface Requirements 100
8.1.4 Evaluation of Quality Attributes 102
8.1.5 Evaluation of Cases and Validation 105
8.1.6 Evaluation of Other Interesting Aspects 106
8.2 Related Work 110
8.2.1 Using Traditional Configurators for Configuring Software . . 111
8.2.2 Mae 116
8.2.3 Generative Programming and Domain-Specific Languages . . 119
8.2.4 Other Related Work 120
8.3 Comparison with Research Questions and Objectives 122

9 Conclusions and Future Work **124**

Bibliography **128**

Chapter 1

Introduction

1.1 Background

An emerging trend in software development is increasing variation. Instead of a homogeneous pool of customers, each customer or customer segment has a unique set of requirements. To satisfy these customers, one would have to include a potentially large set of different capabilities in one software product. This approach is not always feasible or even possible. For example, a hand-held device has a limited processing and memory capacity, and thus software in such a device must be kept as small and lightweight as possible. Further, software has to comply with varying hardware and varying regulations and laws. All these factors increase variability in software products.

However, it is inefficient to separately develop variant products that share some similar characteristics. In fact, if one reused existing pieces when developing new variant products, one would be able to cut development costs. But practice has shown that reuse, especially opportunistic and unplanned, is hard to apply effectively (Bosch, 2000).

A novel solution to this problem (managing variability with reuse) is applying software product lines, also known as *software product families* (SPF) (Clements and Northrop, 2001; Bosch, 2000). A software product family is a set of product individuals in which products are developed according to a common family architecture and are constructed from common reusable software components.

A software product family consists of *core assets*; examples of core assets include

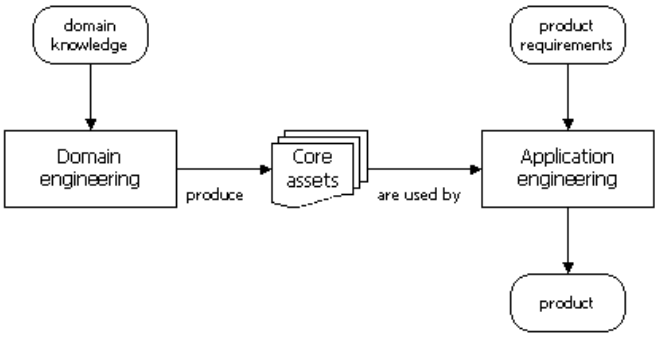

Figure 1.1: A simplified model of domain and application engineering. Domain engineering produces core assets, while application engineering utilises these assets when deriving individual products.

family architecture and reusable components (Clements and Northrop, 2001). A clear distinction is made between two processes: *domain engineering* and *application engineering* (see Figure 1.1). Domain engineering is interested in describing the domain and its common and variant properties, while application engineering is interested in deriving individual products from the product family. Domain engineering activities produce core assets, which are then used during application engineering.

The variability of the software product family manifests itself on many levels. Firstly, there is variability on the requirements level. Software product family research uses the term *feature* to represent an abstraction of requirements. There isn't one common definition of a feature, but it has been defined as "an end-user visible characteristic of a system" (Kang *et al.*, 1990) or as "a logical unit of behaviour that is specified by a set of functional and quality requirements" (Bosch, 2000). Secondly, the variability manifests itself in the family architecture. A software architecture is the organisation of a system that consists of components, their relationships, and the principles that guide system design (IEEE Std 1471, 2000). Thirdly, the variability manifests itself in the implementation units that are part of the software family.

If one applies software product family paradigm, satisfying varying customer requirements becomes a manageable task: one doesn't have to start from scratch when creating a new product individual. Still, the creation of product individuals requires development effort. In addition to selecting the desired features and components and integrating the components with glue code, the product derivation process might require

tailored, customer-specific components or modifications to the family architecture.

The idea of mass-customising products from a common product family is not new. In the domain of traditional, mechanical products this approach has been successfully applied for many years. To minimise customer-specific tailoring, one has developed this approach even further: a *configurable product family* is a product family where all product instances are derived in a routine manner from pre-existing components. The derivation of product individuals is called a *configuration task*. Because this configuration task can be a complex process, there are several tools to aid it; such a tool is called a *configurator*. One of these tools is WeCoTin configurator (Tiihonen *et al.*, 2003), which utilises an inference engine *smodels* (Simons *et al.*, 2002) for configuration reasoning.

Meanwhile, software engineering community has identified that reuse can happen at different levels (Bosch, 2002). When the domain is stable and well-known, it is possible that one can derive a product individual using only existing reusable assets. The derivation happens in a routine manner; thus one can say that products are not developed but configured from the family assets. This approach is often called a *configurable software product family* (CSPF) (Männistö *et al.*, 2000). Product individuals are composed of pre-existing assets in a routine manner, and actual implementation requires either no additional programming or just adding simple glue code. This approach is more or less analogous to traditional configurable product families.

However, one often needs tool support for configurable software product families, as has been the case with traditional configurable product families. There are several reasons for this. Firstly, tool support enables routine and automated derivation with only minimal effort. Secondly, it is typical that there are dependencies between different family elements; for example, one feature might exclude another. Especially in case of a large family, it might be extremely difficult or at least error-prone to resolve these dependencies manually.

In order to utilise tool support, the domain knowledge must be transferred to the tool. Some approaches encode this knowledge in the tool implementation itself. However, if one wants to provide a tool that is independent of the domain, one needs to separate the domain knowledge into a description that is given to the tool. In case of configurable software product families, this description is called a *configuration model*. A configuration model is an unambiguous and machine-readable description

that portraits the commonality and variability in the family.

But since similar problems have been tackled with traditional product configuration earlier, one should investigate on applying these ideas to software. Can existing modelling mechanisms for traditional products be used for modelling software product lines? What are the differences between software and traditional products, and how do these differences affect the situation? Moreover, can one use existing product configurators for configuring software?

Since applying techniques from traditional product configuration to software product families is a novel research area , there aren't any certain answers to these questions. There exists several examples on how one could configure software using traditional configurators, but it is unclear whether these tools can be used in all situations.

There exist three modelling languages, Koalish, Forfamel, and their combination Kumbang, that have been developed as a part of the research conducted in this area. These languages take their conceptual and formal basis from the traditional product domain, which enables the use of existing configuration techniques and tools. Yet they exhibit concepts that are specific to the software domain. In essence, these languages provide means for describing the product family in a configuration model. This model can then be used in the product derivation process.

Koalish (Asikainen, 2004) is a modelling language that is derived from software architecture description language Koala by adding mechanisms for variability. Thus it offers the possibility to manage variability at the architecture level. In contrast, *Forfamel* (Asikainen, 2004) adapts and synthesises existing feature modelling languages that have been proposed for software product families. Thus it is meant for managing variability at the requirements level. *Kumbang* combines the ideas of Koalish and Forfamel into one language. All these languages are based on a configuration ontology for traditional product families. Further, they are provided a formal basis by mapping them to a general-purpose weight constraint rule language (WCRL). This mapping enables the use of *smodels* inference engine (Simons *et al.*, 2002) for configuration reasoning.

A following question arises: how can one implement a software configurator tool that utilises Koalish, Forfamel and Kumbang languages and applies existing tool support from traditional product configuration domain? This is the question this thesis tries to answer.

1.2 Thesis Structure

The rest of the thesis is organised as follows. Chapter 2 introduces software product families and discusses domain and application engineering activities. Chapter 3 discusses the concepts of traditional configurable product families and the methods used in the configuration. Chapter 4 discusses how these two approaches can be combined, and introduces Koalish, Forfamel and Kumbang. Together these chapters constitute the literature survey, which gives background and motivation for the the actual research conducted in this thesis.

Chapter 5 presents aims of this research, such as research questions and goals, research methods used and scope of the research. Chapter 6 describes the system that was developed to meet the research aims. Chapter 7 validates the system by presenting two example cases. Chapter 8 discusses and evaluates the results and does comparison with other related work. And finally, Chapter 9 draws conclusions and suggestions for future work.

Chapter 2

Software Product Families

2.1 Basics

The idea of reuse is not new in software engineering. Software systems are complex and their development is time-consuming and error-prone. Thus reuse has been seen as a mean to alleviate these problems: instead of developing a software product from scratch, one would be able to compose a product out of existing pieces. But it seems that software reuse has failed to deliver on its promises. Since the first attempts at software reuse, one has learned two important lessons. First, opportunistic reuse is not effective in practice; reuse must be a planned and proactive effort. Second, bottom-up reuse does not function in practice; successful reuse programmes are required to employ a top-down approach and develop components that fit the higher-level structure defined by software architecture. (Bosch, 2000, Chap. 1)

Software product families (also known as software product lines) are a rapidly emerging paradigm that provides means to incorporate reuse as a part of software development (Clements and Northrop, 2001; Bosch, 2000). The idea of software product families is simple: instead of developing variant products independently, one reuses existing components that are designed to be reused.

The terms product line and product family seem to have slightly different meanings, although they are often used as synonyms. A product line is a set of systems scoped to satisfy a given market need, while a product family is a set of systems sharing enough common properties to be built from a common set of assets (Czarnecki and Eisenecker, 2000). Thus a product line isn't necessarily a product family and vice

versa. But the distinction between these two is not always so clear—some use these terms interchangeably. For example, Software Engineering Institute (SEI) defines software product line as "a set of software-intensive systems that share a common, managed feature set satisfying a particular market segment's specific needs or mission and that are developed from a common set of core assets in a prescribed way" (Northrop, 2002). This definition clearly includes both aspects: satisfying market needs *and* being developed from common set of assets. Thus it can be argued that these terms are more or less equal. For consistency, the term "product family" is used throughout this text.

The definition of SEI uses the term *core asset*. Core assets are the building blocks of software product families; these include the architecture, reusable common components, documentation, requirements, test cases and so on. The architecture is the key among these assets, since it describes the overall structure of the family and tells how product individuals are composed from reusable components "in a prescribed way".

As said earlier, software reuse in general hasn't been able to deliver its promises. So why does reuse in software product families work? There are several reasons for this.

Firstly, past reuse agendas were mainly concentrating on fortuitous small-grained reuse. Software developers wrote small-sized components for one application, and added these components to a reuse library. Further, there was rarely support for locating, configuring and integrating components in the reuse library. Consequently, it was often more effective to write a new component from scratch than to use an existing one. In a software product line approach, the reuse is planned, enabled and enforced. Reusable assets include more than just components. Further, all assets are designed to be reused, designed so that they fit the overall product family. (Clements and Northrop, 2001)

Secondly, reuse was often seen as a technical activity only. However, software product families are as much about business practices as they are about technical practices (Northrop, 2002). Applying software product families is a strategic decision which requires management commitment (Northrop, 2002) and even organisational changes (Bosch, 2000).

And finally, software product families are not a solution to every situation. It certainly costs more to develop reusable common assets than to develop components for one application only. In order to gain benefits, one has to derive several products from

the product family that use these assets. Bosch (2000) identifies three primary issues that determine the applicability of the product family approach: amount of commonality between single products, ownership of software and negotiability of requirements. If these three aspects are fulfilled to a sufficient extent, the software product family approach can be applied successfully (Bosch, 2000).

2.1.1 Variability and Variation Points

There are are two concepts that often come up when discussing software product families: commonality and variability.

Commonality is the functionality or quality that all the products in the product family share. Since commonalities possess a high reuse potential in the family scope, they should be incorporated as part of the reusable common components (Geyer and Becker, 2002). In contrast, *variability* describes how products in the family differ from each other; it is the ability to change or customise a system. Variability might be expressed as a set of optional or alternative elements, as a numerical range, or it might simply be an open issue (van Gurp *et al.*, 2001).

A concept closely related to variability is variation point. A *variation point* is characterised as a location in a software asset where variation will take place (Geyer and Becker, 2002). Thus variation points reflect the variability found in the assets. There is a n to m relation between variation points and variabilities: selection of one variant might affect several variation points, and vice versa (Geyer and Becker, 2002).

Associated with each variation point is its *binding time*, which is the moment at which the decision is made and the variability is resolved (Geyer and Becker, 2002). For example, variation point might be resolved at the design stage, during the compilation, or even at runtime (compare to Figure 2.1).

The trend has been towards later binding of variation points: one wants to delay decision making to the latest possible moment. The situation can be depicted as two life-cycle funnels in Figure 2.1. The space between the arrows represents the amount of variability in the system. At each step of development, one makes design decisions that limit the number of possible systems. In a software product family, the delayed decision making corresponds to late binding of variation points: variation is not resolved until later stages of product development. (van Gurp *et al.*, 2001)

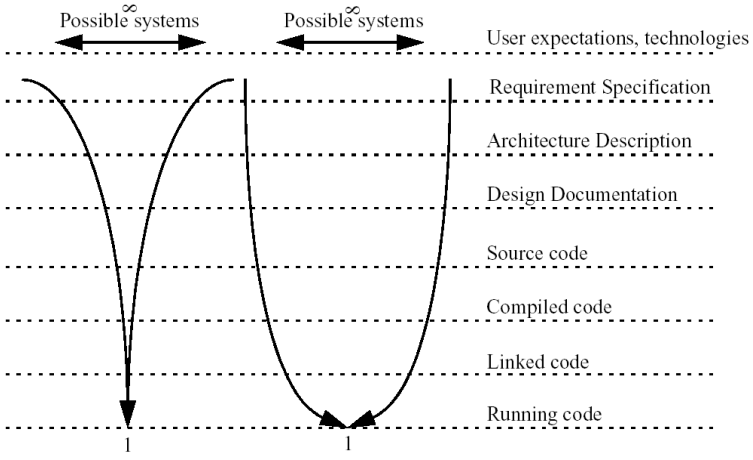

Figure 2.1: Two systems with early and delayed variability (van Gurp *et al.*, 2001)

However, later binding doesn't come without a cost. For example, if binding is delayed until runtime, the running application must include implementation for all possible variations. This potentially increases memory consumption. In some cases the cost of delayed binding might simply be too much. An example of this case is Linux Familiar: one cannot install all possible Linux Familiar packages into a hand-held device, since that would consume too much memory (Ylinen *et al.*, 2002).

2.1.2 Domain and Application Engineering

An important characteristic of a software product family is the separation of two equally important processes: domain engineering and application engineering.

Domain engineering is the activity that designs and produces core assets of the family, while *application engineering* activities develop individual products based on these core assets. Weiss and Lai (1999) define domain engineering as "a process of creating the production facilities for a family", while application engineering is defined as "a process for rapidly creating members of a family (applications) using the production facilities for the family".

In other contexts, these two separate processes are called with different names. Some call these processes development and deployment (Bosch, 2000), or software product family development and product derivation (Männistö *et al.*, 2000), or even core asset development and product development (Clements and Northrop, 2001).

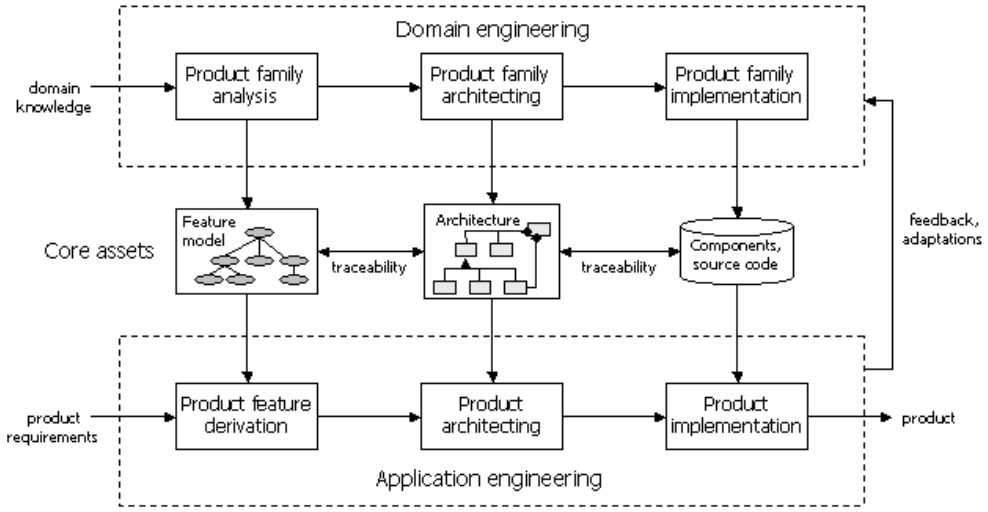

Figure 2.2: Example activities in both domain engineering and application engineering. The core assets serve as links between these activities. (Figure modified and compiled from Thiel and Hein, 2002a; MacGregor, 2002; SEI Software Technology Roadmap, 1997, p.160)

In practice, it has been reported that a successful software product family program must consider this separation of domain and application engineering, even at organisational level. When pressure for the next product release is high, it is easier to add product-specific features than to develop core assets. Consequently, effort is geared from domain engineering towards application engineering, which diminishes reuse potential. One solution is to dedicate parts of the organisation for domain engineering only. This ensures that releases of the family are not degenerated into separate products. (Bosch, 2000)

The separation of domain and application engineering is also highlighted by Software Engineering Institute (SEI). SEI defines three essential activities that constitute product family development. These activities are called *core asset development*, *product development* and *management* (Clements and Northrop, 2001). Besides domain engineering and application engineering, this definition separates management as an important activity. This partly reflects the idea that applying software product families is a planned and managed effort.

Figure 2.2 identifies some example activities in both domain engineering and appli-

cation engineering. According to Figure 2.2, domain engineering contains three basic activities: product family analysis, product family architecting and product family implementation. These activities produce a feature model that describes the features of the family, a product family architecture and reusable components and implementation units. In contrast, application engineering activities derive the product features from the family feature model, create the product architecture, and finally implement the product utilising reusable components.

Although the model shown in Figure 2.2 corresponds to the approach taken in this thesis, it worth noting that the activities and core assets listed in Figure 2.2 are just examples. There are some approaches that do not utilise feature models at all, and some approaches that generate needed components during application engineering.

The following sections discuss domain engineering and application engineering in more detail. Section 2.2 discusses some relevant aspects of domain engineering, especially how product family features and architecture can be modelled. Section 2.3 shows how one can derive product individuals based on these models.

2.2 Domain Engineering in Software Product Families

Domain engineering (also known as core asset development, or software product family development) is the activity that produces and maintains core assets in a software product family. These core asset include not just software components, but models that describe the product family. As shown in Figure 2.2, these models might include feature models and product family architecture. Figure 2.3 shows how Thiel and Hein (2002a) see the typical activities in domain engineering (which they call core asset development), and how these activities produce some of the core assets.

Product line analysis (see Figure 2.3) is interested in describing the domain and its requirements. This analysis partly resembles traditional requirements engineering processes, but there is one special issue that should be concerned: variability. Product family requirements must capture both commonality and variability found in the products. There are several approaches for describing product family requirements, many of which are feature-oriented. Section 2.2.1 discusses these feature modelling approaches more thoroughly.

After product line analysis is done and family requirements are known, product

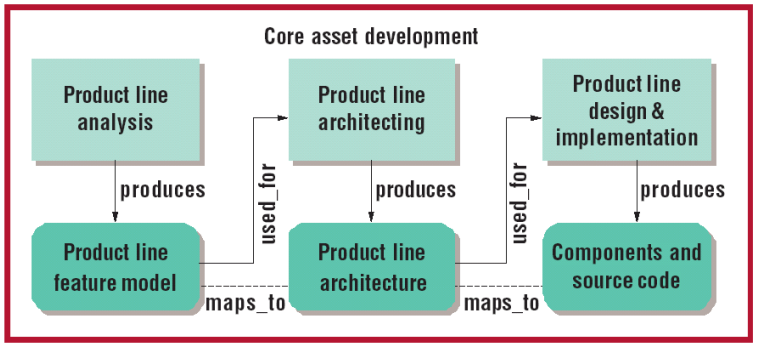

Figure 2.3: Typical activities in core asset development, and how they relate to main core assets (Thiel and Hein, 2002a)

line architecting (see Figure 2.3) produces the product family architecture. This architecting phase resembles traditional architecture design, but again variability brings its twist. For example, Bosch (2000) discusses how functionality-based architecture design could be applied to designing a product family architecture. Again, the family architecture should be able to depict the variability in the family. A couple of methods for modelling software product family architectures are discussed in Section 2.2.2.

Finally, product line design and implementation (see Figure 2.3) produces the actual reusable components and other implementation units. This activity includes specifying requirements for individual components, designing and implementing them (Bosch, 2000).

As was seen in the activities discussed above, handling variability is as essential part of domain engineering. Variability affects all product family artefacts, from requirements to code. Despite this, variability is often underlooked (Thiel and Hein, 2002a): designers might give variability incidental treatment. What is needed is a systematic and explicit way of expressing and modelling variability in software product families (Thiel and Hein, 2002a).

As is illustrated in Figure 2.3, family requirements, architecture and reusable components are all related to each other. The variability in the family can be traced from requirements to architecture and from architecture to components. As Thiel and Hein (2002a) point out, product family requirements and architecture depict the system from different perspectives—variation points in the requirements manifest themselves as variation points in the architecture. Thiel and Hein (2002a) raise two observations

about this correspondence. Firstly, variation points in the architecture do not introduce new variability, they are just manifestations of variation points in the requirements. Secondly, variation points in the requirements can be mapped to several variation points in the architecture, and vice versa.

The following sections discuss different approaches for modelling variability in both requirements and architecture. Section 2.2.1 presents feature modelling concepts for requirements, while Section 2.2.2 presents approaches for architecture modelling.

2.2.1 Modelling Requirements with Feature Models

In many approaches, a feature model is an essential result of product family requirements analysis. Feature models capture product family members' functional and non-functional features, as well as their commonalities and variabilities. (Thiel and Hein, 2002a)

There are several definitions of features, but none of them seem to unambiguously capture the concept. A feature can be seen as a characteristic of a system that is visible to the end-user; for example, when a user buys an automobile, a decision must be made about which transmission feature (automatic or manual) the car will have (Kang *et al.*, 1990). But this requirement of direct effect on end-user might sometimes be too restrictive. Therefore, a broader definition is made: a feature is a distinguishable characteristic of a concept that is relevant to some stakeholder of the concept (Czarnecki and Eisenecker, 2000). However, this definition is quite broad, since it could cover almost anything in a software system.

One of the most widely-known feature modelling approach and the first one to introduce feature models is Feature-Oriented Domain Analysis (FODA) (Kang *et al.*, 1990). FODA is a method discovering and representing commonalities among related software systems; its primary focus is the identification of features of the software system. The FODA process consists of three phases: context analysis, domain modelling and architectural analysis. Domain modelling is the key among the FODA process, since it produces the feature model. A FODA feature model consists of the following four elements:

Feature diagram a hierarchical composition of features

Feature definitions description of features, including classification of binding time

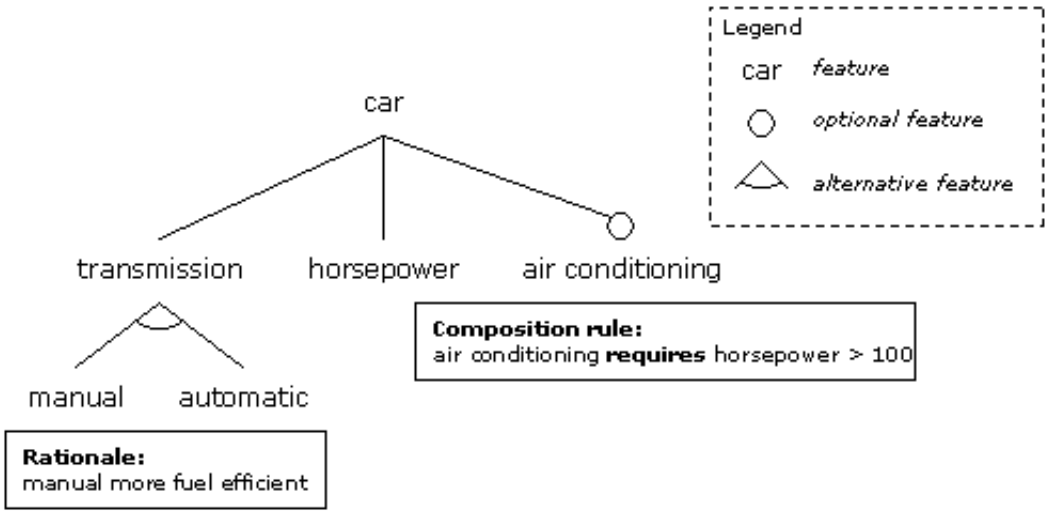

Figure 2.4: An sample feature diagram describing feature hierarchy in a car (Kang *et al.*, 1990). Every car must have transmission and horsepower, but air conditioning is optional. Transmission can be either automatic or manual. In addition, the figure shows one composition rule and one rationale for this diagram.

Composition rules tells which feature combinations are legal

Rationale reasons for selecting or not selecting features

Figure 2.4 shows an example of a feature diagram, which also includes a composition rule and a rationale. This diagram illustrates some of the basic ideas of feature models. Features are organised in a tree, and the structure of the tree reflects the structure of the features: a particular feature cannot be selected into a system if its parent is not selected. A mandatory feature (feature without decorations) must be selected into a system when its parent is selected. An optional feature (empty circle) may or may not be selected. Arcs represent alternatives between different features - they form a set from which exactly one feature can be chosen. Similar or roughly similar notations are used in many other feature modelling methods.

Besides FODA, there exist numerous other approaches for modelling features. Feature-Oriented Reuse Method (FORM) (Kang *et al.*, 2002) extends FODA to incorporate reuse-oriented development methods. Czarnecki and Eisenecker (2000) extend FODA diagrams with several new concepts, such as or-features, alternative or-features and so on. Later on, they also propose adding cardinalities and attributes to feature

models (Czarnecki *et al.*, 2002, 2004). Furthermore, one approach proposes external features, which are features that are external to the system, but that are still used by the system - for example, features offered by a target platform (van Gurp *et al.*, 2001). Finally, this work is based on a synthesised feature modelling method called Forfamel (see Section 4.2.3).

However, despite the ample variety of different feature modelling methods, they often lack rigorous semantics. The example FODA diagram in Figure 2.4 shows some of the problems with vague semantics. The diagram implies that feature "horse power" has something to do with numbers, since the composition rule says that horse power must be more than 100. However, the notation of feature "horse power" is exactly similar to notation of feature "transmission", although transmission definitely has nothing to do with numbers. These and other similar issues limit the use of feature models to human reading only. Without rigorous semantics, they cannot be used in computer-aided deduction. Only lately has one tried to define the semantic basis for feature models (see e.g. Asikainen, 2004; Czarnecki *et al.*, 2004).

2.2.2 Modelling Product Family Architecture

Although it is widely known that architecture is a central product family asset, and that architecture for a software product family must portrait the variability in the family, there are only few approaches for actually depicting variability in architecture. This is in clear contrast with the number of methods proposed for describing requirements of a product family.

Architecture Description Languages (ADL) provide dedicated languages and notations for describing software architectures. However, none of the ADLs offer explicit mechanisms for describing the variability and optionality in architectures (van der Hoek *et al.*, 1999). It is certainly possible to capture some aspects in some languages, but this often quickly leads to a complicated and chaotic architectural description (van der Hoek *et al.*, 1999).

Although van der Hoek *et al.* (1999) claim that ADLs aren't really suitable for describing variability, Koala (van Ommering, 2004) has been used for building software product populations at Philips. This is possible, since product populations are built by plugging reusable components to into product-specific framework. Thus Koala does

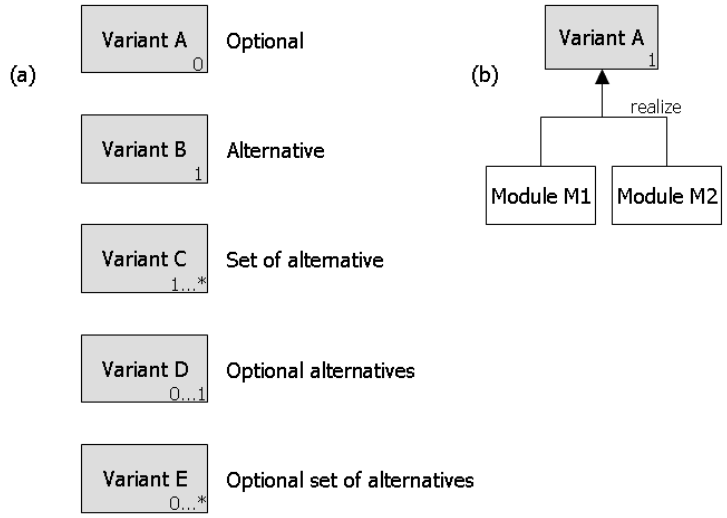

Figure 2.5: Figure (a) shows how different types of variants could be described with subtyped numbers. Figure (b) shows how a variant could be decomposed into modules. (Bachmann and Bass, 2001)

not need to provide explicit mechanisms for variability. For further discussion about Koala and its variability mechanisms, please refer to Section 4.2.1.

Bachmann and Bass (2001) propose a notation for describing variability of software modules. In particular, they distinguish different types of variants: optional, alternative or a set of several alternatives or options. To support these and all possible combinations, they offer notation shown in Figure 2.5. However, the numbering scheme they use differs from the typical way cardinalities are expressed. For example, they represent an optional element with number zero, while optionality is often described with cardinality `[0...1]`. Further, Bachmann and Bass discuss variation points as a mechanism to build a connection from features to places in the architecture that are designed to support those variations. However, their ideas of variation points seem to be contradictory with the notion that variation points manifest themselves in both feature models and architecture models (see e.g. Thiel and Hein, 2002a).

Another approach is presented by Thiel and Hein (2002b). They propose an extension to IEEE 1471 recommended practice for architectural description (IEEE Std 1471, 2000) to support product-family specific issues. In particular, they propose four basic extensions: product line extension, feature variability extension, architecture variabil-

ity extension, and design element extension. Out of these extensions, architecture variability extension covers the explicit representation of variability in the architectures. It contains an architectural variability model, which is described by architectural variation points, which in turn address architectural view models of the IEEE 1471 recommendation.

Extensions proposed by Thiel and Hein (2002b) do not commit to any particular way of describing architectural variation points or architectural variability models. However, they present an example how their extensions could be applied in practice. In effect, their extensions separate the actual architectural model (that describes the structure of the system) from the variability found in the architecture. This separation has its pros and cons: it makes the derivation of a particular product architecture easier, but it may hinder the understandability of the whole product family architecture.

Yet another method for describing variability in the architectures is xADL2.0 (van der Hoek, 2004) and its supporting environment Mae (Roshandel *et al.*, 2004). xADL2.0 is an architectural description language that is build as a set of extensible XML (eXtensible Markup Language) schemas. It supports both space variability (traditional variability) and time variability (evolution). For space variability, it recognises optional elements, variant elements and optional variant elements. Variabilities are managed and resolved with Boolean guards. Mae provides support for the specification of these variabilities, for resolving these variabilities, and for consistency checks. In general terms, this approach aims to represent the same concepts as in many ADLs, namely components, interfaces and connectors. However, this solution is language-independent, since the elements used in the descriptions can be tailored. Mae is further discussed in Section 8.2.2.

And finally, the architectural modelling language that is the basis of this work is Koalish (Asikainen, 2004), which provides several mechanisms for explicitly describing variability. Koalish is presented in Section 4.2.2.

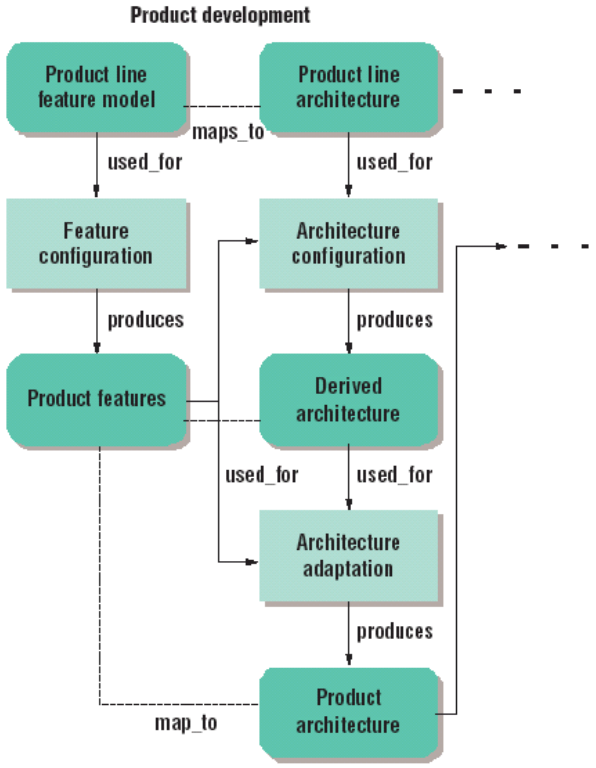

Figure 2.6: How individual products are derived using the product family feature model and product family architecture (Thiel and Hein, 2002a)

2.3 Application Engineering in Software Product Families

Application engineering (also known as product development, or product derivation) is the process of deriving product individuals from a software product family. In essence, the whole idea of setting up a software product family is to make product derivation as easy as possible. This is achieved by providing reusable assets that can be used in the deployment of product individuals.

Figure 2.6 (Thiel and Hein, 2002a) illustrates how product development can utilise feature model and architectural model. The feature model of the product family acts as a starting point for deriving the product. The feature configuration process selects the features for the individual product, and the result is a description of the product features. After that, architecture configuration process determines the product archi-

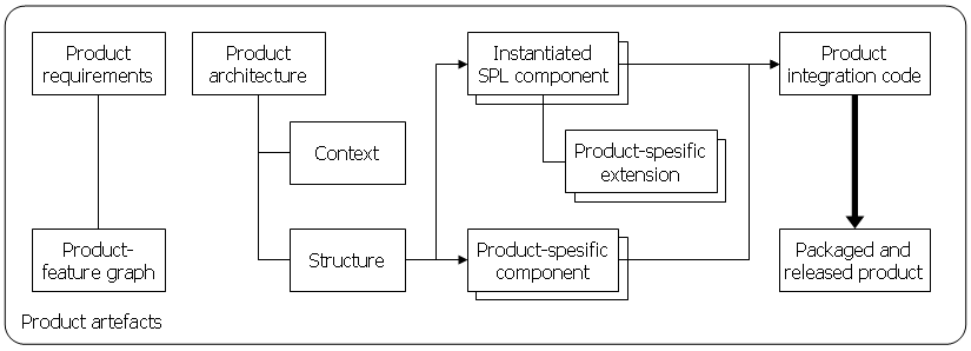

Figure 2.7: Artefacts that are produced during the product instantiation process (Bosch, 2000, Chap. 12)

tecture based on the product family architecture and product-specific features. The result is a derived architecture that conforms to product features. It also serves as a basis for potential customisations in the architecture-adaptation process, which yields the actual product architecture. (Thiel and Hein, 2002a).

Another approach for product instantiation activities is given by Bosch (2000, Chap. 12). The artefacts that are produced in this process are depicted in Figure 2.7. Firstly, one needs to specify the product-specific requirements and produce the corresponding feature model (called feature graph in Figure 2.7). Secondly, the product architecture is derived from the product family architecture. This derivation might include activities such as pruning unneeded parts of the architecture, extending the architecture with product-specific features, resolving conflicts, and assessing the architecture. After that, one needs to develop components that match the derived architecture. This can include both selection and instantiation of existing product family components, and development of product-specific components. Naturally the intention is to use existing product family components to the widest possible extent. After the components have been instantiated, they are integrated into one product and the system and its components are validated. Finally the product is packaged and released.

These two approaches are roughly similar but address the issue from different perspectives. Thiel and Hein (2002a) cover the earlier activities (selecting features, deriving and adapting architecture), while Bosch (2000) shows more details concerning the later stages (developing and building the implementation).

Chapter 3

Configurable Product Families

Chapter 2 discussed how software development has been organised into software product families. Meanwhile, product families have been previously utilised in the domain of traditional, mechanical products. A *configurable product family* is such that individual products are configured from pre-designed assets in a routine manner, with no need for innovative design. Thus this activity assumes that the product is assembled from a fixed set of well-defined components, and that these components interact with each other in a predefined way (Sabin and Weigel, 1998).

This chapter presents the basic concepts of configurable product families. In particular, this chapter discusses two topics that are relevant to this thesis: how configurable product families are modelled, and how configurator tools are used for deriving individual products. Section 3.1 gives a brief overview of the basic concepts of configurable product families. Section 3.2 describes how the domain knowledge of the product families can be modelled, while Section 3.3 discusses the tools and techniques that are used for deriving the product individuals from the product family.

3.1 Background

The industrial revolution, that is, the transition from cut-to-fit craftsmanship to the automated mass-production of goods from interchangeable parts, took about 200 years (Czarnecki and Eisenecker, 2000). Although products are still being mass-produced, customers increasingly demand adaptation to their own requirements. However, producing a specific design for each customer is not economical. Instead, producers use

standardised sets of parts that can be *configured* into products satisfying a wide range of requirements (Faltings and Freuder, 1998). This kind of hybrid between mass-production and customisation is often called *mass-customisation*.

Mass-customisation affects both product-realisation and order-realisation processes of the organisation. At the product-realisation level, the goal shifts from designing single products towards designing families of products. At the order-realisation level, one needs to understand the requirements of the customer and to configure a product individual description to match those needs. (Sabin and Weigel, 1998)

According to Tiihonen *et al.* (1998), a configurable product has the following properties:

1. Each delivered product individual is tailored to the individual needs of an individual customer.
2. The product has been pre-designed to meet a given range of different customer requirements.
3. Each product individual is specified as a combination of pre-designed components or modules. Thus, there is no need to design new components as a part of the sales-delivery process.
4. The product has a pre-designed general structure.
5. The sales-delivery process requires only systematic variant design, not adaptive or original design.

Thus a configurable product is configured from a pre-designed *configurable product family* to meet the requirements of a given customer. This derivation activity is called a *configuration task*, and it produces a *configuration*, which is a description of the product individual to be delivered. As stated in the above definition, configuration task is performed in a routine manner, with no need for creative design or product-specific implementation.

Tiihonen *et al.* (1998) make a separation between the product development process, which produces a model of the family, and the configuration process, which produces a specific configuration. This distinction between two phases is also mentioned by Sabin and Weigel (1998), although they call these phases describing the domain knowledge and specifying the desired product. In any case, the main target of the first activity

is to produce a description of domain, its elements and how these elements can be combined. This description of the family is often called a *configuration model*, and it contains all the information on the possibilities of adapting the product to customer needs (Tiihonen *et al.*, 2003).

Often the required flexibility of a product family means hundreds or even thousands of configurable parts, which in turn increases the possibility of errors during configuration. These errors can create majors slips in the schedule and lead to costly iterations (Sabin and Weigel, 1998). Since computing correct and optimal configurations quickly is so critical, the situation strongly favours automating the configuration process (Faltings and Freuder, 1998).

Thus the configuration task must often be supported by dedicated tools that prevent configuration errors. A *product configurator* (or a *configurator* for short) is a tool that enables the creation and management of configuration models and supports the configuration task that produces a description of the product specification (Tiihonen *et al.*, 2003).

There are several areas applicable for configurable product families. Examples include computer industry (PC configuration), telecommunication industry (configuration of switching systems) and automotive industry (car sales configuration) (Felfernig *et al.*, 2001).

3.2 Modelling Configurable Product Families

Instead of explicitly defining a set of product variants in a product family, a configurable product is associated with a *configuration model* that contains all the information of the possible products. This model defines a set of pre-designed components, rules on how these can be combined into valid products and rules how one can achieve the desired functionality for the customer (Tiihonen *et al.*, 1998).

As Sabin and Weigel (1998) point out, most of the complexity of solving the configuration problem lies in representing domain knowledge. This is because the configuration model must explicitly state all the relations and constraints among different components. Modelling the domain knowledge is thus a critical part of the configuration problem. (Sabin and Weigel, 1998)

There are several approaches for capturing the domain knowledge. For example,

configuration knowledge can be captured with rules (rule-based approach), hierarchies of concepts, such as taxonomical hierarchies or structure hierarchies (structure-based approach), with constraints between elements (constraint-based approach), using resource exchange (resource-based approach) or using case-based technologies. Together these approaches are often called knowledge-based configuration methods. (Günter and Kühn, 1999)

3.2.1 Configuration Modelling Concepts

In order to distill domain knowledge into a configuration model, one needs a modelling language for this purpose. There exist several languages for representing configuration knowledge, most of which are designed for certain tools. But many of these languages share similar properties. These similar properties can be captured in a *configuration ontology*, which tries to define a set of concepts that can be used for representing the configuration knowledge. Soininen *et al.* (1998) present a synthesised configuration ontology, while Felfernig *et al.* (2001) present a configuration ontology that is based on Unified Modelling Language (UML). These two ontologies bear many similararities with each other. But since this thesis is based on the configuration ontology by Soininen *et al.* (1998), called configuration ontology in the following, this section covers that particular ontology in more detail.

The configuration ontology presented by Soininen *et al.* (1998) consists of a set of concepts for representing the configuration knowledge and the restrictions on possible configurations. The ontology synthesises existing modelling approaches, such as resource-based, rule-based and constraint-based methods.

The configuration ontology distinguishes three classes of configuration knowledge: configuration model knowledge, configuration solution knowledge and requirements knowledge. Configuration model knowledge specifies the elements that may appear in the configuration and how they can be combined. In other words, this knowledge specifies all correct configurations that may be derived. In contrast, configuration solution knowledge specifies a configuration, while requirements knowledge specifies the requirements of a particular configuration. The configuration ontology provides concepts only for the first two knowledge classes, since requirements knowledge can be specified with the concepts provided by other two classes. (Soininen *et al.*, 1998)

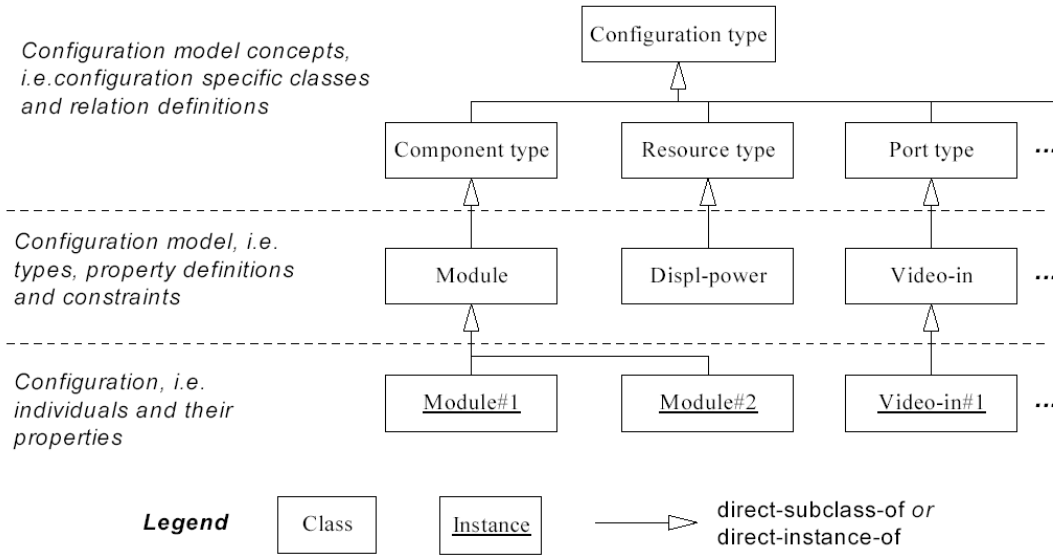

Figure 3.1: Basic structure of the configuration ontology presented by Soininen *et al.* (1998)

The basic structure of the configuration concepts by Soininen *et al.* (1998) is depicted in Figure 3.1. It distinguishes three levels of abstraction. On the top-level are configuration model concepts, which form a taxonomy of configuration-specific concepts and relation definitions. On the second level are product-specific types in one particular configuration model. These types are direct subclasses of the configuration model concepts. On the lowest level are individuals that exist in a particular configuration. These individuals are instantiated from the types occurring in the configuration model. (Soininen *et al.*, 1998)

The distinction between types and individuals is important. However, it is common that configuration knowledge is discussed using terms that do not distinguish between types and instances. For example, a sentence “car has an engine as a part” can have two meanings. If it is part of configuration model knowledge, it states that every car individual must have an engine individual as a part. If it is part of configuration solution knowledge, it states that a configuration includes a car individual that has an engine individual as a part. (Soininen *et al.*, 1998)

In the following, one presents briefly the basic configuration model concepts in the ontology, which correspond to the highest hierarchy level in Figure 3.1. For further

details of the concepts, please refer to (Soininen *et al.*, 1998).

A *configuration type* is either a component type, port type, resource type or function type. A *component type* represents a distinguishable entity in a product that is meaningful for product configuration. A configuration is composed of component individuals that are instantiated from component types in a configuration model. A component type can specify its part roles as a set of *part definitions*. A part definition specifies a part name, possible part types and a cardinality. Further, ports are used to connect components through compatible interfaces. A *port type* is a definition of a connection interface, whereas port individual represents a "location" where another port individual in other component individual may be connected. In addition to component type and port type, a *resource type* captures the production and use of abstract entities, such as power or space in rack. Finally, functions bring a non-technical aspect to the configuration model. A *function type* represents a functional specification, which in turn is mapped to technical elements through implementation constraints.

In addition to configuration types, the ontology offers *attributes*, which represent the characteristics of a type. Finally, one can use *constraints* to restrict other concepts in the configuration model.

3.2.2 Example Configuration Model

This section tries to clarify the concepts presented in Section 3.2.1 by presenting an example configuration model. Figure 3.2 shows a part of a configuration model for drilling machines (Tiihonen *et al.*, 1998). The example model is written with a language that conforms to the configuration ontology concepts presented in Section 3.2.1, and it illustrates several examples of those concepts.

Figure 3.2 contains several components, which are either abstract or concrete. The top-level component in the part hierarchy is *Drilling boom assembly*, which contains exactly one component *Boom & Feeder*, one component *Rockdrill*, and an optional component *Suction head*. Further, a *Rockdrill* could be either *HL500*, *HL600* or *HL700*, which consume different amounts of power (resource *Power*).

The configuration model in Figure 3.2 specifies several ports. Ports *BA* and *DA* are compatible, and they both specify two concrete ports (*HBA* and *LBA* for *BA*, *HDA* and *LDA* for *DA*). Component *DA* (drilling attachment) contains port *DA*, while component

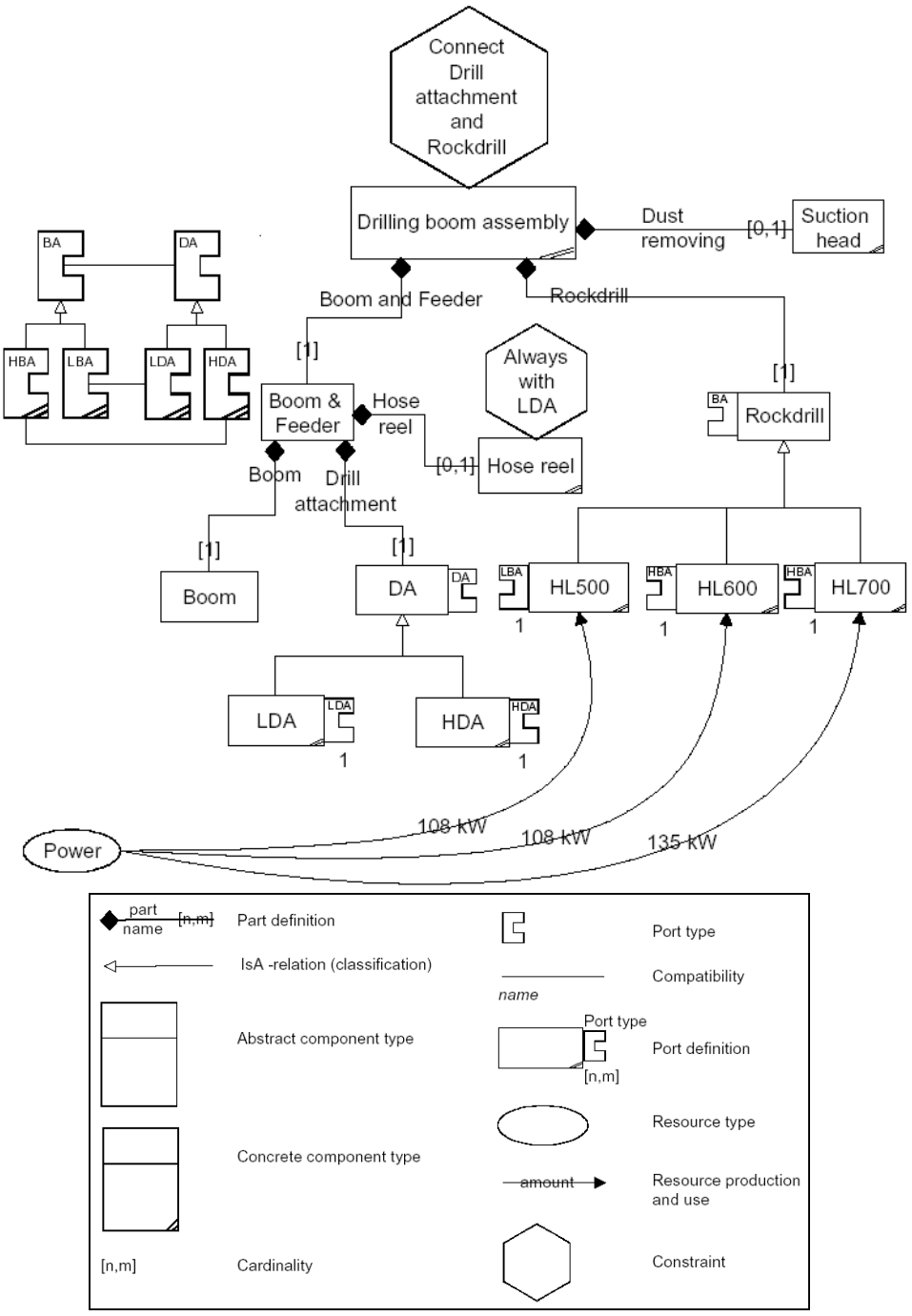

Figure 3.2: An example configuration model (or a part of it) and the corresponding legend (Tiihonen *et al.*, 1998)

Rockdrill contains port *BA*. Further, different concrete drilling attachments and rock drills contain different concrete ports (for example, component *HL500* contains port *LBA*).

The hexagons in Figure 3.2 represent constraints. For example, if component *Hose reel* is present in the configuration, drilling attachment must be of type *LDA*.

Finally, there are some concepts in the ontology, such as attributes and functions, that do not have examples in Figure 3.2.

3.3 Deriving Configurable Product Individuals

This section discusses how tools can be used for deriving product individuals.

To recapitulate Section 3.1, the derivation activity is called a configuration task, and it produces a configuration, which is a description of the product individual. Further, the configuration task is performed in a routine manner, without creative design or separate implementation.

Because of the complexity of the configuration task, it must often be supported by dedicated software tools called *configurators* (Soininen *et al.*, 2002). A configurator enables the user to specify the configuration; the tool generates a description of a product individual that meets the customer requirements and complies with the configuration model (Tiihonen *et al.*, 2003).

Fundamentally, a configurator must check *completeness* and *consistency* of the configuration. Completeness means that all the necessary selections are made, while consistency means that no rules of the model are violated. It should be impossible to order an inconsistent or incomplete configuration. (Tiihonen *et al.*, 2003)

Configuration techniques have been studied in the field of artificial intelligence, and there exists a wealth of techniques for this purpose. Examples include constraint satisfaction problems, description logics and different specialised problem solving methods. (Soininen *et al.*, 2002)

Although several techniques exist, many of them have been shown to be potentially computationally very expensive. In the worst case configuration task requires at least an exponential amount of time in the size of the problem. However, conventional wisdom states that practical problems do not exhibit this kind of behaviour. Although this statement hasn't been proven yet, there are cases where configuration has been

efficient. (Tiihonen *et al.*, 2002)

Besides configuration techniques, there exist implemented tools for the same purpose (e.g. Günter and Hotz, 1999; Hollmann *et al.*, 2000; Tiihonen *et al.*, 2003). One of them is an academic configurator prototype WeCoTin (Tiihonen *et al.*, 2003). Since the work done in this thesis bears many similarities with WeCoTin, the following section discusses WeCoTin more thoroughly.

3.3.1 WeCoTin Configurator

WeCoTin (Tiihonen *et al.*, 2003) is an academic configurator prototype that can be used for configuring product individuals from configurable product families. It provides both a web-based configuration tool and a modelling tool. The configuration tool supports the configuration task by visualising and checking the configuration and provides a remote repository of configuration models that the user can choose to configure. The modelling tool provides a graphical user interface for creating and editing configuration models. The user interface of WeCoTin is presented in Figure 3.3.

WeCoTin has been used for modelling and configuring real products from different domains. For these products, it has been shown that the system is efficient enough for practical use. (Tiihonen *et al.*, 2003)

The configuration modelling language that WeCoTin employs is called PCML (Product Configuration Modelling Language). PCML is based on the configuration ontology presented in Section 3.2.1. The main concepts of PCML are similar to the on-

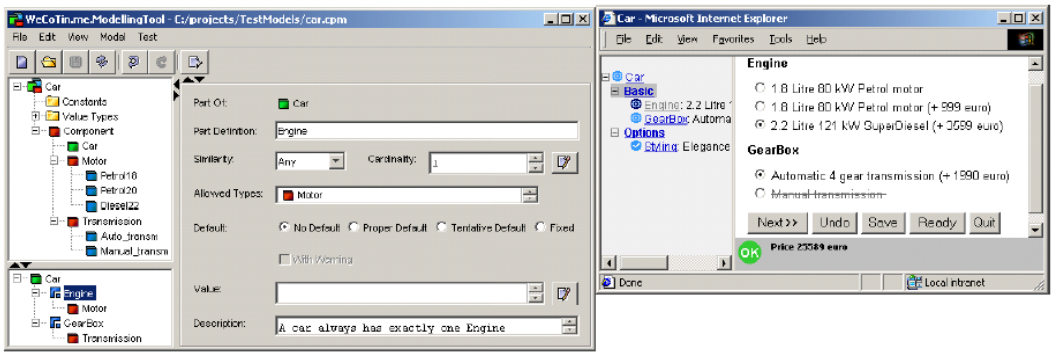

Figure 3.3: Graphical user interface of WeCoTin environment. Modelling tool is on the left and configuration tool is on the right. (Tiihonen *et al.*, 2003)

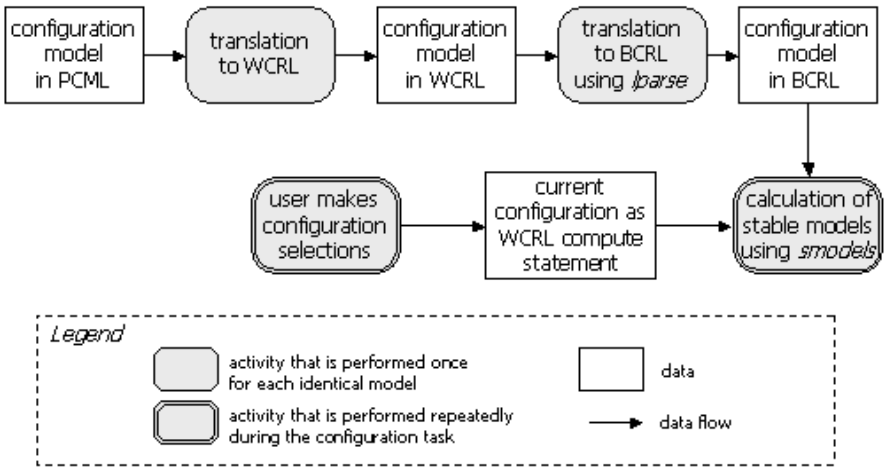

Figure 3.4: A data flow model that shows how WeCoTin uses *smodels* for configuration reasoning

tology: component types, compositional structure with cardinalities and variant types, attributes of components, contraints to components, and so forth. However, there are some aspects of the configuration ontology that are missing from PCML, such as ports and resources. (Tiihonen *et al.*, 2003)

PCML has been given semantics by mapping it to WCRL. WCRL (Weight Constraint Rule Language) is a general-purpose knowledge representation language, which is a form of logic programs (Simons *et al.*, 2002). The translation from PCML to WCRL is provided by Soininen *et al.* (2001). The basic idea is to treat the sentences of the model as a set of rules in WCRL. A configuration is a logical model (so called *stable model*) of the set of rules representing the configuration model.

A closely related concept is BCRL (Basic Constraint Rule Language), which serves as a "normal form" for general weight constraint rule language. Basic constraint rules play a major role in implementing general weight constraint rules. (Simons *et al.*, 2002)

By providing a mapping from PCML to WCRL one enables the use of *smodels* (Simons *et al.*, 2002), which is an inference engine operating on WCRL and BCRL programs. The *smodels* system has a two-level architecture: a front-end *lparse* and actual *smodels* kernel. To compute the stable models of a WCRL program, *lparse* first translates the program to WCRL. This compilation is called *grounding*, and it

is potentially very costly, since it removes variables in the WCRL program. After grounding, *smodels* kernel computes the desired stable models. This computation is implemented with an effective search algorithm. (Simons *et al.*, 2002)

Figure 3.4 shows how WeCoTin utilises *smodels* for calculating the stable models.

Before the configuration task starts, the system translates the configuration model written in PCML first to WCRL and then to BCRL (upper part of Figure 3.4). Since some of these activities (e.g. translation to BCRL, also called as grounding) take a lot of time, it is beneficial to perform them in advance.

During the configuration task, the configurator tool (see Figure 3.3) provides a web-based user interface. This interface contains a configuration tree, which gives an overview of the configuration, its compositional structure and properties. The user makes configuration selections by setting the properties of the elements (see Figure 3.3). Every time the user makes configuration selections, the system checks the configuration state (lower part of Figure 3.4). The current configuration is given as a *compute statement*, which represents the selections made so far. This compute statement is then used for calculating desired stable models.

Chapter 4

Configurable Software Product Families

Chapter 2 discussed software product families and their properties, while Chapter 3 presented configurable product families for traditional products. When comparing these two approaches, it seems that they have quite a lot in common. Both share the notion of two separate processes: one that designs the family structure and the other that derives product individuals based on that design. The main idea in both approaches is to ease the derivation process and enable quick derivation of new variant products. In fact, configurable product families for traditional products take this idea even further: the derivation process does not include innovative design or implementation at all. Instead, the derivation is performed in a highly routine manner, with the support of dedicated configurator tools.

The similarity between these areas raises questions. Could the idea of configurability be applied to software product families? Could one automate the product derivation process so that software product individuals are not developed but configured out of existing core assets? This would mean that a software product family would become a *configurable software product family*.

This chapter presents some basic ideas of configurable software product families and presents the conceptual basis of this thesis. Section 4.1 discusses some basic aspects of configurable software product families. Section 4.2 presents three modelling languages, named Koalish, Forfamel and Kumbang, that synthesise ideas from software product families and from traditional product configuration domain.

4.1 Basics of Configurable Software Product Families

A *configurable software product family* is such that individual software products are created in a highly routine manner from a predefined set of variants. Such a family may potentially contain millions of variants. This means that one must be able to manage a large set of variants and to select and produce a correct individual for a particular need from the set of all variants. This approach becomes important, for example, in case of embedded software: if the memory is limited, the loaded software cannot include all possible implementations for variability. (Männistö *et al.*, 2000, 2001a,b)

In case of large number of variants, it is clear that tool support is needed, both for managing the variety and for producing correct product individuals. However, there are different approaches for managing and expressing the variability. Männistö *et al.* (2001b) propose a model-based approach: the variability of the family is captured in a *configuration model* that describes the functional capabilities as well as software architecture of the product family. But there are also industrial cases in which variability is encoded in the tool implementation itself. One can argue that utilising a model-based approach improves portability and maintainability: if the domain knowledge is in the model, it is easier to utilise the same implementation for many domains.

Since configurable software product family is a novel research area, there exists relatively little research on the topic. However, there are some recorded examples of industrial configurable software product families. Raatikainen *et al.* (2003) present a case study of two configurable software product families. Both companies had developed proprietary tools for configuring individual products, and both domains were relatively stable. Further, Bosch (2000, 2002) presents one configurable product family in which the company involved produced fire alarm systems. Each customer received the same code base, which was configured and installed using a configuration tool developed for this purpose.

This section tries to highlight some of the research aspects of configurable software product families. Section 4.1.1 discusses configurability as the highest level of reuse, while Section 4.1.2 discusses how techniques and ideas from traditional configurable products could be employed.

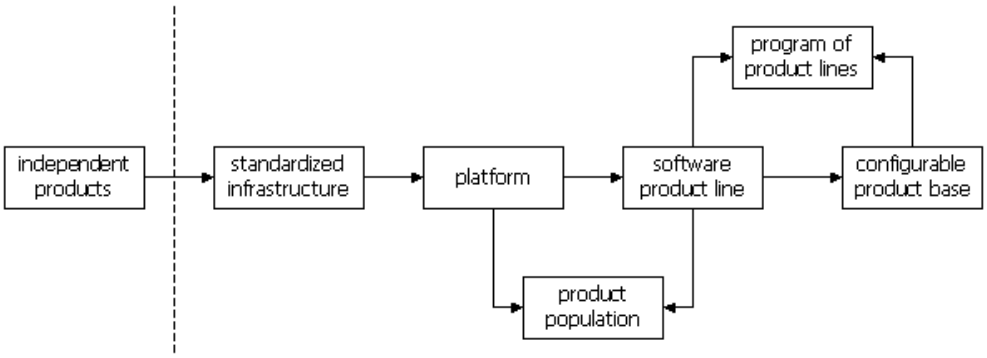

Figure 4.1: Maturity levels for software product families and how an organisation typically evolves along these levels (Bosch, 2002)

4.1.1 Configurable Product Base as the Highest Level of Reuse

The most obvious distinction between configurable product families and software product families is the amount of innovative design needed in the product derivation process. The usual case for a software product family is that one has to implement product-specific design during the derivation process (see Section 2.3). The amount of product-specific implementation is often inversely proportional to the amount of reuse. The more one can reuse existing assets when deriving the product individual, the less one needs to add product-specific implementation.

This amount of reuse is one of the affecting factors in the classification presented by Bosch (2002). Bosch proposes a hierarchy of levels, which he calls maturity levels, that correspond to different approaches for software product family development (see Figure 4.1). A higher maturity level corresponds to a situation where effort has moved from product development (application engineering) to family development (domain engineering). This in turn reflects the amount of intra-organisational reuse.

Bosch (2002) points out that although literature often presents only one particular technique to adopting a product family approach, there are several levels at which reuse can take place in the organisation. As the organisation changes and evolves, the approach taken may also change. Starting from a situation in which each product is developed independently, the main maturity development path consists of a *standardised infrastructure*, a *platform*, a *software product line*, and finally, a *configurable prod-*

uct base. Two additional developments can be identified: a *product population* and a *program of product lines*. (See Figure 4.1.)

A configurable product base corresponds to the highest maturity level. Rather than developing a number of different products, the organisation moves towards developing only one configurable product base that is configured into the desired product. This configuration is typically supported by automated tools or techniques. Once this approach is fully adopted, all development effort has moved from application engineering to domain engineering. All variation points must have an explicit representation in the tool that is used for resolving the variability. (Bosch, 2002)

If we look at this characterisation of a configurable product base, it very much resembles a traditional configurable product, with the exception that it is performed for software. However, Bosch (2002) seems to imply that the variability should be encoded in the tool itself, not in a separate configuration model. Further, it seems that he does not make a distinction between configuration task and building, similar to what is done in traditional product configuration.

Also, it is worth noting that "mature" is a bit confusing term, since it easily implies that all organisations should aim at maturity. Instead, one should choose the most appropriate level for a particular situation. The right level depends on several factors: maturity of the organisation and maturity of the domain itself (Bosch, 2002). Thus a configurable product base as an approach is suitable for situations where a large number of products is developed in a highly stable domain and in which the organisation is stable enough in terms of domain understanding, project organisation and management. The requirement for domain stability is quite essential: since effort has moved from application engineering to domain engineering, the initial cost of starting a configurable product base is substantial. If requirements and the domain itself are constantly changing, there is not enough time for the investments to be pay off.

4.1.2 Applying Traditional Configuration Techniques to Software

As stated, a key question is: Could the practices used with traditional products be transferred to the software product family domain? Are there some differences between the domains that need to be considered?

According to Brooks (1987), software has the following inherent properties: *com-*

plexity, *conformity*, *changeability* and *invisibility*. Since software is so complex, it forces the developers to design conform interfaces. And although it is complex, it is seemingly easy to change software. And finally, due to its invisibility, it is hard to concretise software. Hotz and Krebs (2003a) suggest that at least these factors might affect how software can be configured, but they also state that this is a subject worth studying.

Männistö *et al.* (2001a) present a comparison between concepts in both domains. In particular, they take the configuration ontology presented by Soininen *et al.* (1998) as a representative for traditional product configuration. They find remarkable similarities, but also areas where some mismatch exist. Based on these findings, they present two potential areas where knowledge could be transferred between these domains. First, the appropriate level of abstraction for software product families should be adjusted, especially for dynamic behaviour. Second, it is not clear whether connections between components should be represented as first-class entities like in many architecture description languages. In fact, modelling connectors and evolution are two areas where knowledge might be transferred from software domain to traditional product domain. (Männistö *et al.*, 2001a)

Geyer and Becker (2002) claim that techniques known from the traditional knowledge-based community are insufficient for software configuration, since they concentrate too much on inheritance structure and neglect aggregation. Unfortunately Geyer and Becker (2002) do not give any examples of traditional techniques that have this property. In fact, if we look at the configuration concepts and techniques presented in Chapter 3, aggregation of components is a central concept in them.

Indeed, there are some approaches that have applied traditional, knowledge-based configuration techniques to software. Some attempts have been made to use existing configurators as such for configuring software product families (Asikainen *et al.*, 2004; Ylinen *et al.*, 2002). Other approaches take existing knowledge-based configuration techniques and build software-specific tools on top of them (see e.g. Hein and MacGregor, 2003; Hotz and Krebs, 2003b; Hotz *et al.*, 2004). For further discussion about other approaches that use either directly or indirectly techniques from the traditional product domain, please refer to Section 8.2.

The dilemma is as follows: how to avoid re-inventing the wheel but take the characteristics of software into account? It seems that one answer is to build software-specific

configurators on top of traditional techniques. Indeed this is exactly the approach taken in this thesis: to build a software configurator that reuses techniques and tools from traditional product configurators.

4.2 Koalish, Forfamel and Kumbang

This section presents three modelling languages, Koalish and Forfamel, and their combination, Kumbang.

These languages are developed to be used for modelling configurable software product families, but they can also be used for other modelling purposes also—for example, for modelling ordinary software product families.

Koalish, Forfamel and Kumbang are influenced by concepts from the traditional product configuration domain. In particular, they are developed to conform to the configuration ontology presented in Section 3.2.1. However, these languages are designed for software domain; they take the characteristics of software into account. Further, these languages are designed so that they can be used with *smodels* inference engine (Simons *et al.*, 2002). This way, one can build configuration support that employs *smodels* for configuration reasoning, similar to what is done in WeCoTin product configurator (see Section 3.3.1).

The rest of the section is organised as follows. Section 4.2.1 presents Koala, which is an architecture description language that is the basis of Koalish language. Section 4.2.2 covers Koalish, which is a Koala-based modelling language specifically designed for configurable software. Section 4.2.3 presents Forfamel, which is a feature modelling language that synthesises other existing feature modelling methods. Finally, Section 4.2.4 discusses briefly how Koalish and Forfamel could be combined into one modelling language, Kumbang.

4.2.1 Koala

Koala (van Ommering *et al.*, 2000; van Ommering, 2002, 2004) is a modelling language that can be used for describing component-based architectures for embedded software. There are many architectural description languages available, but Koala is one of the very few that has practical value. Koala was developed at Philips Consumer

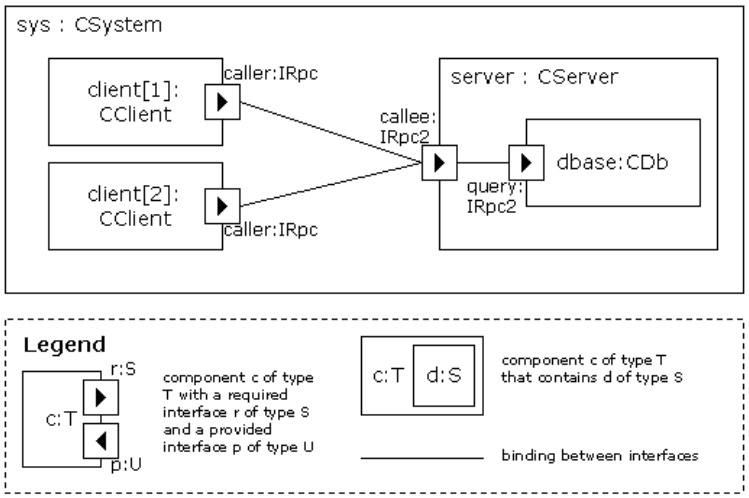

Figure 4.2: An example of an architecture described using Koala (Asikainen *et al.*, 2003b)

Electronics to aid in developing software for embedded television sets (van Ommering *et al.*, 2000). There are a number of products developed using Koala in the market, and a couple of hundred developers are using the tools based on Koala (van Ommering, 2002). Thus its industrial usage gives Koala practical relevance.

The main concepts of Koala are *components* and *interfaces*. A component in Koala is "an encapsulated piece of software with an explicit interface to its environment". Components can be compound components, meaning that one component can have several components as parts. Together they form a hierarchy of components, which constitutes the system: a *configuration* is a component that has no interfaces and that is not contained in any other component. (van Ommering *et al.*, 2000)

As the definition implies, a component communicates with its environment through interfaces, which are small sets of related functions. Each interface is either a *requires* or *provides* interface—components access all external functionality through required interfaces, while provided interfaces provide this external functionality to other components. Since interfaces are basically sets of functions, one can compare these sets. Thus one can say that an interface type is a subtype of another type if it contains at least all the functions that are contained in its supertype interface.

In addition, components can be connected with each other. Since all communica-

tion happens via interfaces, only interfaces can be connected with each other. These *bindings* between interfaces correspond to function calls between components. The general rule is that one can connect a calling interface (a "tip") to a called interface (a "base"). If components are at the same level, one can connect a required interface to a provided interface only. In addition to topological rules, one must ensure that connected interfaces are type-compatible. This means the called interface must provide at least all functions found in the calling interface.

Figure 4.2 shows an example of a system that is described using Koala. The system contains a three-level aggregation hierarchy of components, and it has three bindings that connect interfaces.

Koala makes a distinction between types and instances, both for components and interfaces. Thus each component and interface instance is of one certain component or interface type. The elements in Figure 4.2 are component and interface instances. Types and names of these instances are written with notation *name:Type* in the figure.

Koala has been used for managing product populations at Philips. Product populations are software product families that incorporate reuse between product families. However, explicit variability mechanisms of Koala are rather weak, especially when comparing to the expressiveness of many traditional configuration techniques (see for example configuration ontology in Section 3.2.1). Koala provides a couple of explicit mechanisms for variability. Examples of these constructs include switches that are used for binding varying components, diversity spreadsheets that are used for parametrising components, and interfaces that can be optional (van Ommering, 2004).

But if Koala has been used for building product populations, how is it possible that Koala doesn't provide strong mechanisms for expressing variability? The answer lies in the difference between philosophies. Typical product family consists of a common family architecture, into which one can include common, varying and product-specific components. This means that one has to be able to explicitly specify how components can be combined in the family architecture. In contrast, Koala assumes that products are built by plugging reusable, parametrised components into a product-specific framework (van Ommering, 2004). Thus Koala does not have to provide explicit mechanisms for variability, since there is no need to specify explicit variation of components.

Fundamentally, the difference between configurable software product families and

Koala lies in the assumption about how products are built. Koala assumes that developers who build products know what they are doing and know how components can be combined. In contrast, the person who performs the configuration task for a configurable software product family does not ideally need to know anything about the technical issues of the family. Further, if one aims at providing computer-aided configuration reasoning, one needs to provide explicit mechanisms for specifying the rules of the family. One definitely cannot assume that a computer knows without explicit instructions how components can be combined into products.

In (Asikainen, 2002), Koala is compared with the configuration ontology discussed in Section 3.2.1. The thesis concludes that there are some concepts that are missing from Koala, but that Koala could rather easily be modified to meet the configuration ontology concepts. This would enable it to be used with traditional configuration techniques. Next section presents Koalish, a modification of Koala, which tries to incorporate ideas from the configuration ontology into Koala.

4.2.2 Koalish

Koalish is a language for modelling the architecture of configurable software product families (Asikainen *et al.*, 2003a,b; Asikainen, 2004). The language is obtained by combining basic modelling elements of Koala with a number of variability mechanisms adopted from the product configuration domain. In particular, Koalish uses the concepts from the product configuration ontology (see Section 3.2.1). This enables the use of existing methods of traditional knowledge-based configuration. However, Koalish is designed for software architectures. Thus it can be used for modelling architectures of software product families (see Section 2.2.2).

The basic building blocks of Koalish are components, interfaces, attributes and constraints. Koalish components and interfaces are similar to Koala components and interfaces in many respects. Koalish makes a distinction between types and instances. This distinction is also included in Koala and the configuration ontology (see Section 3.2.1). A *Koalish model* is a description of a configurable software product family, and it may contain component types, interface types, attribute value types and constraints. A *Koalish configuration* is derived from a certain model, and it may contain component instances with attributes, interface instances and bindings between

interface instances. Thus the distinction between types and instances is shown in the relationship between models and configurations. Models contain types, while configurations derived from those models contain instances derived from the corresponding types.

Koalish introduces two variability mechanisms that are lacking from Koala, but are included in the configuration ontology. Namely, it offers the possibility to explicitly select the number and type of parts in the composition hierarchy. In addition, constraints can be used for restricting these variability mechanisms.

The Koalish composition hierarchy is quite similar to the Koala composition hierarchy: components can contain other components as parts. In addition, the Koalish composition hierarchy includes the variability of the parts, namely the possibility to select the number and type of parts. Thus component type specifies its parts by stating a part name, a set of possible part types and a cardinality definition. The cardinality is an integer range that specifies the number of component instances that must exist as parts under the given component.

Figure 4.3 contains an example Koalish model that resembles the Koala system described in Figure 4.2. For example, component *CSystem* can have one, two or three clients, which may each be of type *CBasicClient* or *CExtendedClient*.

Similar to Koala, a Koalish interface type is a set of functions. These interface types can be inherited from each other. For example, interface type *IRpc2* with functions *f* and *g* is a subtype of interface *IRpc* with function *f*.

There are some concepts that exist in Koala but have been omitted from Koalish. For example, there is no counterpart for module, switch or diversity spreadsheet. There concepts were left out in order to keep the model as simple as possible while still having the necessary concepts for modelling variability. In addition, there are some concepts that exist in the configuration ontology (see Section 3.2.1), but are missing from Koalish. For example, one cannot inherit Koalish components from each other. This means that component taxonomy is always only one level deep.

Koalish has been provided semantics by mapping it to Weight Constraint Rule Language (WCRL) (Simons *et al.*, 2002). Besides semantics, this mapping provides a way to utilise *smodels* inference engine (Simons *et al.*, 2002), which operates on WCRL programs. Thus *smodels* can be used for providing configuration reasoning support for Koalish.

```
Koalish model ClientServer
  root component CSystem

interface IRpc {
  f;
}

interface IRpc2 {
  f; g;
}

component CBasicClient {
  requires IRpc caller;
}

component CExtendedClient {
  requires IRpc2 caller;
}

component CDb {
  provides IRpc2 query { grounded };
}

component CServer {
  provides IRpc callee;
  contains CDb dbase;
  connects callee = dbase.query;
}

component CServer2 {
  provides IRpc2 callee;
  contains CDb dbase;
  connects callee = dbase.query;
}

component CSystem {
  contains (CExtendedClient, CBasicClient) client[1-3];
  contains (CServer, CServer2) server;
  contains CServer2 extraServer[0-1];
  attributes PerformanceLevel performance;
  constraints present(extraServer) => value(performance) = high;
}

attribute type PerformanceLevel = { low, medium, high };
```

Figure 4.3: An example of a Koalish configuration model (Asikainen, 2004)

4.2.3 Forfamel

There exists a large number of different feature modelling languages for software product families (see Section 2.2.1). These languages describe the product family through its variant and common features. However, many of these languages seem to be unsuitable for modelling configurable software product families. Since configuration task must be at least partially automated, automation sets several requirements for the modelling language. Firstly, the language must be expressive enough to capture all relevant information in the configuration model itself. Secondly, the language must have rigorous and unambiguous semantics so that the configuration engine can reason about the model.

Forfamel (Asikainen, 2004) is a feature modelling language that has been developed for configurable software product families. Forfamel synthesises many existing feature modelling languages, namely FODA, FORM and that of Czarnecki (see Section 2.2.1). This means that Forfamel is compatible with the ideas present in the software product family domain. In addition, Forfamel applies techniques and concepts utilised in traditional knowledge-based configuration (see Section 3.2.1). This in turn enables the use of existing configuration techniques for traditional product families.

The main modelling elements of Forfamel are features. Forfamel makes a distinction between feature types and feature instances, just like the configuration ontology does (see Section 3.2.1). A *Forfamel model* defines feature types and their relations, while a configuration contains feature instances derived from the corresponding types. A feature type can be either abstract or concrete. If a feature type is abstract, it cannot be instantiated directly.

Features can be organised into two kinds of hierarchies: taxonomies and compositional structures. Firstly, feature types can inherit other feature types. This inheritance is a IsA-relation: a feature type inherits all properties of its supertype.

Secondly, features can contain other features as parts. This relationship corresponds to the hierarchy relations in many feature modelling approaches: a feature instance cannot exist in the configuration without its parent feature. The compositional structure offers a variability mechanism. Unlike many other feature modelling approaches, Forfamel does not include separate concepts for variant or optional features. Instead, Forfamel delivers expressiveness for variability and optionality by providing a

```
Forfamel model TE
  root component TextEditor

feature type TextEditor {
  subfeatures
    Language uiLanguage;
    EquationEditor primaryEquationEditor;
    EquationEditor secondaryEquationEditor[0-1];
    SpellChecker spellChecking[0-1];
    Clipboard copyPaste;
    (OCI, JDBC) sqlImport[0-1];
  constraints
    instance_of(uiLanguage, English) => present(spellChecking);
    not present(sqlImport) or not present(secondaryEquationEditor);
}

abstract feature type Language {}
feature type English extends Language {}
feature type Finnish extends Language {}
feature type Swedish extends Language {}

abstract feature type EquationEditor {}
feature type EqEdit extends EquationEditor {}
feature type MathPal extends EquationEditor {}

feature type Clipboard {}
feature type MultiItemClipboard extends Clipboard {
  attributes
    Cap capacity;
}

feature type OCI {}
feature type JDBC {}

feature type SpellChecker {
  Language language;
}

attribute type Cap = { 3, 5 ,9 }
```

Figure 4.4: An example of a Forfamel model (Asikainen, 2004)

possibility to select the number and type of subfeatures. This is achieved through subfeature definitions. This definition specifies a set of possible feature types from which one can choose, and a cardinality that indicates the minimum and maximum number of subfeatures that can be instantiated.

In addition to being part of type or compotitional hierarchies, features in Forfamel models can specify attributes. These attribute definitions are inherited along all other properties in the type taxonomies.

Finally, Forfamel offers the possibility to specify constraints in the model.

Figure 4.4 depicts an example configuration model written in Forfamel. The system contains a root feature type *TextEditor*, which contains several subfeatures as parts. For example, the system can have optional support for SQL-imports (subfeature *sqlImport* with cardinality from zero to one), and the SQL-import can either be of type *OCI* or *JDBC*. In addition, the system must specify exactly one language for user interface (subfeature *uiLanguage*), which can be either English, Finnish or Swedish (inherited from an abstract feature type *Language*). Further, a feature type *MultiItemClipboard* defines an attribute *capacity*, which has either value 3, 5 or 9. Finally, the system contains two constraints that specify how certain elements can exist in the configuration.

Forfamel has been provided with semantics by mapping it to Weight Constraint Rule Language (WCRL) (Simons *et al.*, 2002). Besides semantics, this provides a way to utilise *smodels* inference engine (Simons *et al.*, 2002), which operates on WCRL programs. Thus *smodels* can be used for providing configuration reasoning support for Forfamel.

4.2.4 Kumbang

Two previous sections presented two modelling languages that are designed for modelling configurable software product families: Koalish can be used for modelling the family architecture, while Forfamel describes family features. As was discussed in Section 2.2, feature models and architecture models represent the same software product family from different viewpoints. If one provides a mapping between these models, one can use features to specify product-specific architecture.

Thus Koalish and Forfamel have been combined into one modelling language,

```
Kumbang model KumbangExample
  root feature FSystem;
  root component CSystem

feature FSystem {
  subfeature
    (FeatureA,FeatureB) f;
  implementation
    instance_of(f, FeatureA) <=> value($, attr) = a;
    instance_of(f, FeatureB) <=> value($, attr) = b;
}

feature FeatureA {}
feature FeatureB {}

component CSystem {
  attributes
    ABValue attr;
}

attribute type ABValue = { a, b }
```

Figure 4.5: An example of a Kumbang model

Kumbang. Kumbang has been developed by our research group, but it's full specification has not yet been published. In essence, Kumbang is a combination of Koalish and Forfamel elements. A Kumbang model may contain all the elements that can exist in Koalish and Forfamel models. In addition, one can map elements in the feature model to elements in the architecture model through implementation constraints. These implementation constraints describe the relations between features and components in the product family. An example Kumbang model that illustrates implementation constraints is shown in Figure 4.5.

Chapter 5

Research Aims

5.1 Research Objectives

The main objective of this work was to develop a configutor tool called Kumbang Configurator. Kumbang Configurator can be used in the configuration task for configurable software product families. The tool employs three existing modelling languages, Koalish, Forfamel and Kumbang (see Chapter 4). These languages provide concepts, such as features and architectural elements, that are designed for software domain.

However, Kumbang Configurator utilises existing techniques that have been applied in the domain of traditional configurable products. The tool uses state-of-the-art inference engine *smodels* (Simons *et al.*, 2002) for resolving dependencies between configuration decisions.

Kumbang Configurator combines feature-based and architecture-based configuration techniques. This means that the tool can be used for deriving feature configurations, architecture configurations and configurations that contain both features and architectural elements.

Since configuration task is often error-prone, Kumbang Configurator has mechanisms for preventing configuration errors. The tool checks the configuration for consistency and completeness. A *consistent* configuration does not contain any conflicts against the rules of the configuration model. A *complete* configuration is such that all necessary configuration decisions are made.

Finally, Kumbang Configurator and its implementation must be validated. This means that the tool has to be tested using some kind of example cases.

Research questions	Method or means of development
	Method for analysis
	Design, evaluation or analysis of a particular instance
	Generalisation or characterisation
	Feasibility
Research results	Procedure or technique
	Qualitative or descriptive model
	Empirical model
	Analytic model
	Notation or tool
	Specific solution
	Answer or judgement
	Report
Validation techniques	Analysis
	Experience
	Example
	Evaluation
	Persuasion
	Blatant assertion

Table 5.1: Different types of research questions, results and validation methods (Shaw, 2002)

5.2 Research Questions

Shaw (2002) discusses the characteristics of good research in software engineering and classifies different types of research questions, results and validation methods (see Table 5.1). A particular research strategy is constructed by selecting a suitable combination of these types. Naturally not all combinations make sense; Shaw (2002) gives examples of combinations that have existed on submitted conference papers.

Given the general objectives presented in Section 5.1, one can start to construct research questions for this work. Firstly, this work can be seen as a feasibility study. Examples of research questions for feasibility include questions like *Is it possible to accomplish X at all?* and *Does X even exist, and if so what is it like?* (Shaw, 2002).

One can indeed treat this work as a feasibility study for tool-supported configuration based on Kumbang language and on methods that employs techniques from traditional product configuration. Secondly, this work can also be seen as a method or means of development. An example of this type of research question is *How can we do or automate doing X?* (Shaw, 2002). It makes sense to treat this work as providing means for configuration task for configurable software product families. However, it must be noted that the goal of this work is to develop a prototype, not a tool ready for production. This also means that the feasibility purpose of this work is much more predominant.

Thus, the research questions of this paper can be summarised as follows:

1. Is it possible to build a configurator tool that can be used for configuring product individuals from a configurable software product family described with Koalish, Forfamel or Kumbang language, so that the implementation utilises existing inference engine *smodels*?
2. If it is possible to build the aforementioned configurator tool, does the tool support the user in the configuration task and provides service that outweigh manual configuration?
3. Given the developed tool presented above, can the feasibility of the approach be validated with example cases?

5.3 Research Method

Given the research questions and objectives presented above, what is the research method that is used for answering those questions? Table 5.1 identifies different types of research questions, results and validation methods. As was discussed, this work tries to answer both feasibility questions and to provide means of development.

Firstly, to give background for the research, this thesis contains a literature survey. The first part of the study analyses the domains of software product families and traditional product families, and then discusses the techniques for combining these two. This material is achieved through literature, and it is meant to give proper background information to the rest of the work. The literature study spans Chapter 2, Chapter 3 and Chapter 4.

Secondly, this work tries to answer the research questions using constructive methods. This means that this work provides an implemented tool that embodies a certain technique (category *notation or tool* of research results in Table 5.1). This tool tries partly to show the feasibility of the approach and partly to provide means for product derivation in configurable software product families. The implementation is discussed in Chapter 6.

Finally, the research result is validated through example cases (category *example* of validation techniques in Table 5.1). Shaw (2002) gives two categories for examples: a toy example, perhaps motivated by reality, and a system that is a slice of life. The validation technique of this work contains both types of examples: one invented example and one real-life case. The research result (a tool) is thus validated by showing how it works on two kinds of example cases. The validation is discussed in Chapter 7.

In summary, the research strategy of this thesis combines the following categories from Table 5.1. The research questions combine both feasibility questions and means of development, while the result is a tool that is validated with two examples. The research strategy of development of method/means as a question, notation/tool as a result and example as a validation technique was also found in submitted conference papers (Shaw, 2002). In addition, this work adds the feasibility aspect to that particular strategy.

5.4 Scope

This section covers the scope of this thesis. Firstly, the goals of this thesis affect the scope. For example, this thesis takes Koalish, Forfamel and Kumbang languages for granted, since the goal of this thesis is to develop a configurator tool based on these languages.

The scope of this thesis can be easily summarised in the following way. Figure 5.1 describes the some of the activities in both domain engineering and application engineering. (For further discussion about this issue, see Section 2.1.2.) The scope of this thesis is the highlighted area in Figure 5.1. Taken the scope of this thesis, feature models are described with Forfamel elements, architecture models with Koalish, and Kumbang provides traceability between these elements. Together these form a configuration model, which is used for deriving product individual during the application

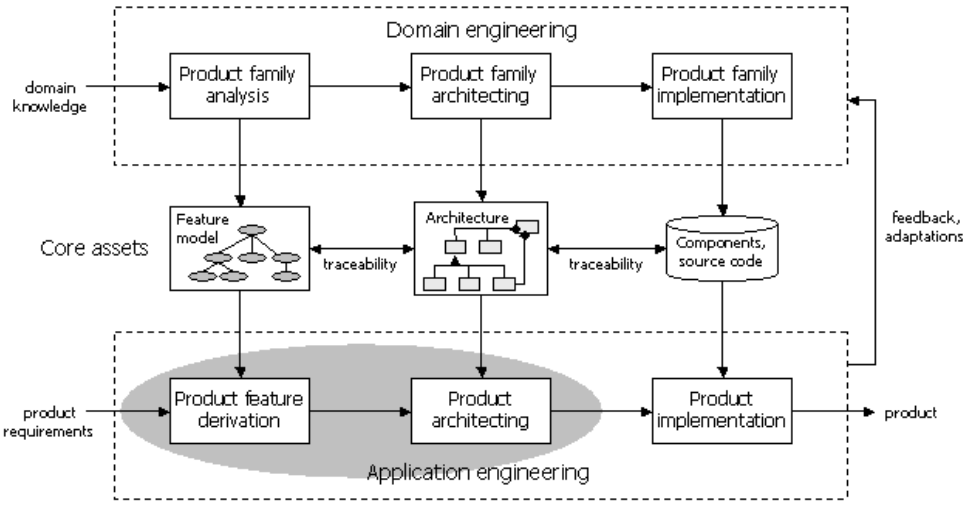

Figure 5.1: Examples of core assets and activities in both domain engineering and application engineering. The scope of this thesis is the highlighted area in the figure.

engineering.

As is illustrated by Figure 5.1, this thesis concentrates on application engineering activities only. This means that core assets are taken for granted. For example, this thesis assumes existing configuration models written in Koalish, Forfamel or Kumbang, and that reusable components have been implemented. However, one could support product family analysis and product family architecting by providing a graphical modelling tool that can be used for creating Koalish, Forfamel and Kumbang configuration models.

In addition, this thesis leaves out implementation issues (product implementation in Figure 5.1). The tool produces only a description of the complete and consistent product individual. How the actual product in built is out of the scope. This task could also be supported by providing tools for building the actual product.

Finally, how the product requirements are achieved is out of the scope of this thesis. The product requirements are taken for granted, and it is assumed that the user of the configurator tool knows how these requirements affect configuration decisions (selecting features and components).

Chapter 6

System Implementation

This chapter describes Kumbang Configurator, which that was implemented to meet the research goals and answer the research questions presented Chapter 5.

The rest of the chapter is organised as follows. Section 6.1 describes the requirements of the system. Section 6.2 gives a brief overview of how the tool works from the user point of the view, while Section 6.3 describes the system internals and its architecture. Finally, Section 6.4 identifies contribution from other developers that took part in the implementation.

The aim is that this chapter does not include evaluation of the implemented system. However, this chapter tries to give rationale for the implementation decisions made. For the complete evaluation of the system, please refer to Section 8.1.

6.1 Requirements

6.1.1 How To Obtain Requirements

In general terms, requirements for a software set out what the system should do and define constraints on its operation (Sommerville, 2004). Requirements act as a bridge between system goals and the its implementation. If requirements are not explicitly specified or are specified incorrectly, it is really difficult to implement a system that actually meets the given goals and satisfies the user needs. Thus this section specifies the requirements that were set for the system. But the question is: how does one obtain these requirements?

The process of finding out and analysing the requirements of a system is often called *requirements elicitation and analysis*. This activity includes discovering the requirements, organising them, giving them priorities and documenting them. Out of these activities, requirements discovery is the probably the most challenging one. Possible sources of information during requirements discovery include documentation, system stakeholders and specifications of similar systems. In particular, gathering the requirements from the system stakeholders directly often yields best requirements. (Sommerville, 2004)

Considering the requirements of Kumbang Configurator, there are several sources for requirements:

Literature review Requirements are discovered from the scientific literature. This is one major source of requirements for this system, but consideration must be applied. Firstly, all ideas might not be applicable to this case. Secondly, literature gives requirements at a very superficial level, yielding only major requirements of the tool.

Traditional configurators Requirements are discovered by comparing to traditional product configurators. Many of the requirements have been gathered by comparing to WeCoTin configurator (see Section 3.3.1). However, requirements cannot be taken directly, since there are differences between the domains that might affect the requirements.

Software configurators Requirements are discovered by comparing to other existing software configurators. Unfortunately, there are only few of them, and they have very little industrial relevance.

Industrial experience Requirements are discovered from industrial stakeholders through interviews or similar methods. This would have been the best solution, but there were several difficulties. Firstly, the process of identifying and interviewing system stakeholders from the industry takes a lot of time, and given the scope and resources of this thesis, it is too much. Secondly, even if one was able to ask the stakeholders what they want, the stakeholders might not know **what** they want. This is due to the novelty of the research area.

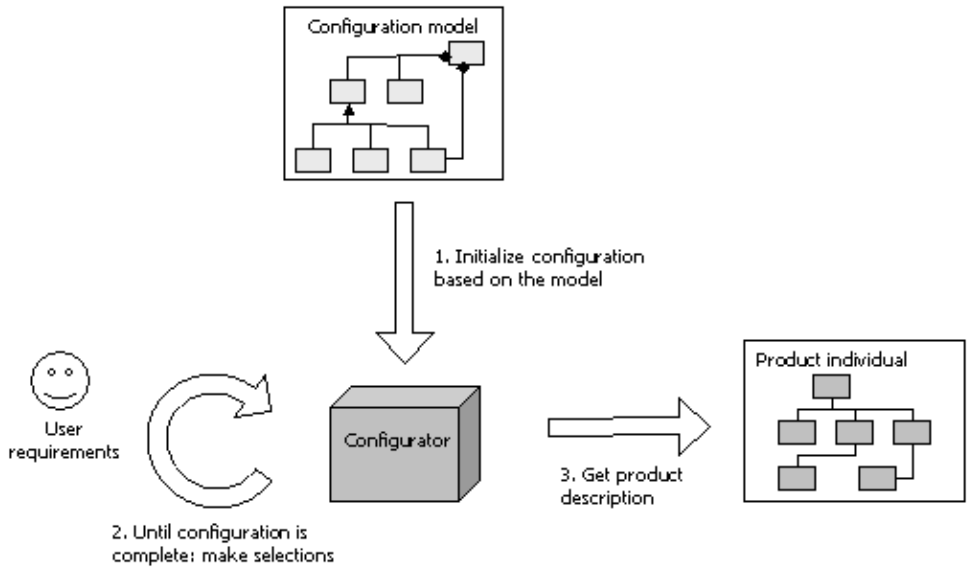

Figure 6.1: A high-level black-box view that shows how the configurator tool developed in this thesis works.

6.1.2 Overall Description of the System

This section gives a brief overview of the requirements of Kumbang Configurator.

Figure 6.1 describes the high-level black-box view of the tool. It shows how the tool supports the configuration task for deriving product individuals. The tool takes a configuration model as input. This configuration model describes the product family, and it is written in Koalish, Forfamel or Kumbang. Based on this model, the tool offers the user the possibility to make configuration decisions. An example decision could be "I want feature X in this product" or "I want component Y with attribute Z" or "I want to connect interface I1 to interface I2". Based on these configuration decisions, the tool builds the configuration.

As the user makes the configuration decisions, the tool checks the configuration for consistency and completeness. In addition, the tool is able to deduce the consequences of the configuration decisions made so far. This means that the tool automatically fills those instances that are implied by previous decisions and identifies those instances that are conflicting with previous decisions.

The completeness and consistency checking and further deducing the configura-

tion is a complex and possibly time-consuming task. This is where the tool employs existing techniques from traditional product configuration. This is done by using a state-of-the-art inference tool called *smodels* (Simons *et al.*, 2002), which can resolve dependencies between configuration decisions.

When the configuration is complete, meaning that all the necessary decisions are made, the tool produces a description of the product individual. This is a feature and architecture description that can be used for building the product from existing assets. Thus the tool does not generate code nor provide a mapping from software components to actual implementation.

In order to ease the configuration task, the tool offers a graphical user interface. This means that the tool visualises the current configuration. Since the configuration consists of instances, which are in turn composed into instance hierarchies, the tool visualises this composition hierarchy. In addition, the tool shows the configuration decisions that the user can make. In general, the idea is that the user doesn't have to know a thing about the configuration model when using the tool. This is partly justified by the fact that often the user of the configurator tool is not the same person who designs the configuration models.

The visualisation of the tool resembles existing approaches where possible. For example, the component configuration can be visualised with Koala-like notation.

The tool conforms with the given languages, Koalish, Forfamel and Kumbang, as tightly as possibly. The configuration can consist of a feature hierarchy, or of a component hierarchy, or it can be a combination of both. The feature configuration consists of feature instances, their part relations and attributes in features. The component configuration consists of component instances and their part relations, attributes, interfaces and bindings. The user can add and remove these instances freely, as long as these modifications conform to the configuration model at hand.

Since this tool is merely a prototype that is used for demonstrating the feasibility of this approach, requirements concerning quality attributes are not so strict. However, maintainability, usability and performance are at least considered in the implementation. Maintainability eases the further development of the tool, while usability and performance make demonstration easier.

Priority	Description
1	Requirement is essential.
2	Requirement is conditional.
3	Requirement is optional.

Table 6.1: Priority descriptions of the requirements.

6.1.3 Detailed Requirements

Based on the above sources and overall description of the system requirements, a detailed list of requirements was constructed. Further, requirements were organised into three groups, and each requirement was given a priority from one to three, which correspond to essential (priority one), conditional (priority two) and optional (priority three) requirements. (See Table 6.1.) An essential requirement is part of the core functionality, conditional requirement eases the use of the tool noticeably, and optional requirement may be implemented if there is enough time.

The requirements presented in the following are deliberately kept on user requirements level. *User requirements* are statements, often presented in natural language, of what the system is expected to provide from the user point of the view; they do not depict detailed specifications of the system (Sommerville, 2004). Detailed *system requirements* cannot be included in this document, partly because they occupy too much space, partly because the amount of detailed information would not give any real value to the reader of this thesis.

Table 6.2 contains requirements that cover the basic functionality of the system. For example, Table 6.2 contains requirements for modifying the configuration, which means adding, removing or editing instances in the configuration.

Table 6.2 also contains requirements that concern configuration models. These models are used as a basis for the configuration task. Thus the configuration task is started by initialising the configuration based on a pre-designed configuration model. The system must also manage configuration models in a centralised repository. This way the user of the tool does not need to possess the configuration model, or even know anything about the model. This enables separation of domain and application engineering organisations.

ID	*Requirement*	*Priority*
B1	The system conforms to Koalish, Forfamel and Kumbang languages and offers all the functionality they offer.	1
B2	The user can open a pre-designed configuration model, which is written either with Kumbang, Forfamel or Koalish. The tool should initialise the configuration based on the model.	1
B3	The user can save an existing (partial) configuration.	2
B4	The user can open a saved (partial) configuration.	2
B5	The system can produce a description of the configuration.	1
B6	The user can add new instances (features, components, interfaces, bindings) to the configuration, according to the rules of Kumbang and the configuration model.	1
B7	The user can remove instances (features, components, interfaces, bindings) from the configuration, according to the rules of Kumbang and the configuration model.	2
B8	The user can set attribute values for components and features, according to the rules of Kumbang and the configuration model.	1
B9	The user can do function binding, that is, to bind interfaces using only a subset of available functions, according to the rules of Kumbang and the configuration model.	3
B10	The user can use the system without knowing anything about the rules of the configuration model.	2
B11	The system must enable centralised management of configuration models.	2
B12	The user can load configuration models to a centralised repository.	2
B13	The user can store additional information along with the loaded configuration models, such as name, version and description.	3
B14	The user can initialise a configuration by opening a centrally-managed configuration model.	2

Table 6.2: Basic requirements

ID	*Requirement*	*Priority*
C1	Configuration reasoning employs *smodels* and *lparse* modules.	1
C2	Every time the user makes a configuration selection, the system should check the configuration for consistency and completeness.	1
C3	The system shows constantly whether the current configuration is consistent or inconsistent.	1
C4	The system shows constantly whether the configuration is complete or incomplete.	1
C5	If the user makes a selection that causes the configuration to be inconsistent, the user can cancel that selection.	1
C6	If the user makes a selection that causes the configuration to be inconsistent, the user can try to manually correct the situation.	2
C7	If the user makes a selection that breaks a constraint, the system must tell which constraint is broken.	3
C8	If the user makes a selection that causes the configuration to be inconsistent, the system must tell a reason for inconsistency.	3
C9	The system should be able to deduce positive consequences (selections that must be present in the configuration) and select them in the configuration, either manually or automatically.	2
C10	The system should be able to deduce negative consequences (selections that must not be present in the configuration) and inform these selections to the user.	2
C11	If there exists at least one complete and consistent configuration, the system should be able to find it and fill in missing selections.	2

Table 6.3: Requirements that cover configuration reasoning

ID	*Requirement*	*Priority*
U1	The system shows the compositional hierarchy of the current configuration in a tree, both for features and for architectural elements.	1
U2	The system provides a taxonomy of the types available in the configuration model in a tree.	3
U3	The user can browse the architecture configuration in a diagram, one level at a time.	1
U4	The user can browse the feature configuration in a diagram, one level at a time.	2
U5	The system shows those locations in the diagrams into which new instances can be added.	2
U6	The system must initialise layout of elements in the diagrams.	1
U7	The user can move the elements in the diagrams.	2
U8	The system should use graphical notation for the configuration that resembles existing notations, such as Koala and feature trees.	1
U9	The user can show or hide elements in the diagram.	3
U10	The user can select among different graphical representations for the elements (such as UML).	3

Table 6.4: Requirements that cover the graphical user interface

Table 6.3 contains requirements that cover the aspects of the configuration reasoning. When the user modifies the configuration, the system must constantly check the consistency and completeness of the configuration. When the user makes a selection that causes the configuration to be inconsistent, the user can either cancel that selection or try to manually correct the situation. In addition, configuration reasoning includes deduction of consequences, which means adding those instances that must be present in the configuration and preventing those that cannot be present. Further, the system could tell why a selection makes the configuration inconsistent. However, being able to tell reasons for inconsistency is probably very hard, and in general it requires separate explanation mechanisms built on top of the configuration mechanism.

Table 6.4 contains requirements that cover the graphical user interface of the system. Basically the graphical user interface must show the current configuration and enable its modification. This calls for two separate representation mechanisms that show different views to the configuration. Firstly, one needs trees that show the whole

ID	*Goal*	*Priority*
Q1	The usability of the system should be good enough for demonstration purposes.	1
Q2	The system should be as extensible and modifiable as possible in order to support further development of the system.	1
Q3	The system performance should be good enough for demonstration purposes.	2

Table 6.5: System goals that cover quality attributes of the system

configuration in one compositional hierarchy. (For example, WeCoTin configurator employs trees for this purpose, see Section 3.3.1.) Unfortunately, it is quite difficult to represent bindings in a tree, since bindings connect two separate interfaces in the configuration. One solution could be adding two nodes to the list to represents bindings, but this isn't as informative as the graphical presentation. In addition, adding bindings is definitely easier when one can just draw a line from one interface to another. Further, one might want to have more detailed information in a structure that shows how elements are contained in other elements. This requires a separate diagram representation, which shows only one composition level of the configuration. (An example level could look something like in Figure 4.2, if one removed component *dbase:CDb* from the figure.) These levels can then be browsed up and down along the composition hierarchy (for example, by double-clicking component *server:CServer* in Figure 4.2 in order to reveal component *dbase:CDb*).

Finally, Table 6.5 contains some system goals that are related to quality attributes. They are called goals, not requirements, since they are on a very abstract level. A good requirement should be measurable, so that one can verify whether the implemented system fulfils that requirement. The purpose of this system is to demonstrate the feasibility of the approach and provide a first prototype around the concepts. Thus it was seen to be unnecessary to specify strict limits for quality attributes. However, one can give guidelines for quality attributes that best support the purpose of this system. Especially system usability and modifiability are important goals. Without usability, it is hard to demonstrate the system so that others understand what is going on. Usability is especially important for Kumbang Configurator, since underlying configuration concepts are rather difficult.

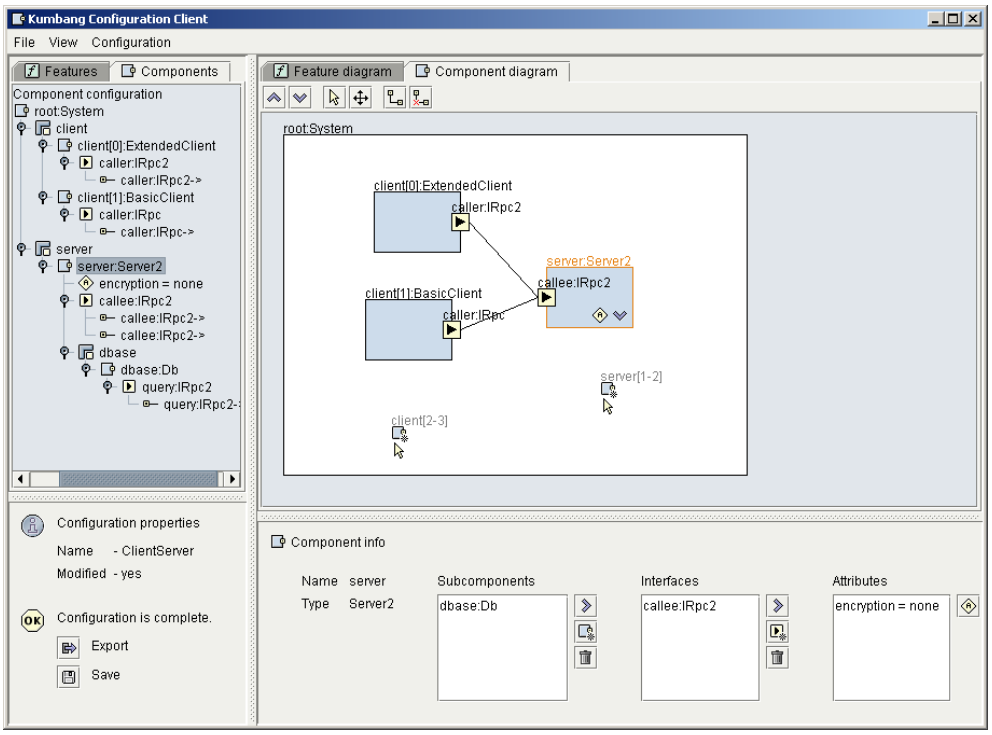

Figure 6.2: A screen shot from the system. This is the view that shows the architecture configuration. The configuration tree is on the left, and it shows the whole compositional hierarchy. Diagram representation is on the right, and it shows the compositional hierarchy one level at a time.

6.2 How the Tool Works

This section presents an overview of the implemented tool and shows briefly how the tool works. Figure 6.2 shows a screen shot from the tool. This screen shot illustrates some of the basic principles of the tool. Firstly, the graphical user interface is divided into two separate areas, one for feature configuration (see Figure 6.3) and one for architecture configuration (see Figure 6.2). In a usual scenario, user modifies the configuration from the feature area, and then these selections are reflected automatically to the architecture configuration. However, user can also modify the architecture configuration directly.

The graphical user interface is divided into four separate areas. The tree structure on the left contains a hierarchical list of the current component configuration (Fig-

ure 6.2) or feature configuration (Figure 6.3). Each node in the list represents one instance or attribute in the configuration. The diagram area on the right shows the same configuration, but one level at a time (Figure 6.2 or Figure 6.3). The user can browse these levels up and down. The area on the bottom left corner shows the status of the configuration: it is either incomplete (Figure 6.3), complete (Figure 6.2) or inconsistent. The area on the bottom right corner shows detailed information of one particular instance. When user selects an instance from the tree or from the diagram, this area shows detailed information and offers buttons for several actions on that particular instance.

As can be seen from Figure 6.2 and Figure 6.3, trees and diagrams are tabbed. Thus user can freely select which trees or diagrams to show at a time. Trees and diagrams are constructed so that basically user can modify the configuration from either one of them. In a typical scenario, user selects feature tab from the tree and component tab from the diagram. The modifications made to the feature tree are then reflected to the component diagram.

There are several ways to modify the configuration. Firstly, one can modify the configuration through trees, since nodes in the tree provide pop-up menus that contain needed functionality. Secondly, the configuration can be modified through diagrams. Diagrams provide locations into which new instances can be added (an example location can be found in Figure 6.4). Further, elements in the diagram provide pop-up menus for other purposes. Finally, information area on the bottom-right (see Figure 6.2 and Figure 6.3) provides buttons for modifying the configuration.

As was presented in Figure 6.1, a typical usage scenario consists of three steps. Firstly, the user opens a configuration model and the tool initialises the configuration based on that model. Figure 6.5(a) shows a dialog that lets the user to select a pre-loaded configuration model from the repository. Secondly, the user modifies the configuration by adding, editing or removing instances. For example, when the user wants to add a new component, the system opens a dialog in Figure 6.5(b). The user can select the type of the component, set attribute values and select interfaces. Finally, after the configuration is complete, the user can export a description of the configuration (see Figure 6.5(c)). The description can either be written in XML (eXtensible Markup Language), or it can be a text file that resembles Kumbang model files.

The graphical user interface mimics graphical UML (Unified Modelling Language)

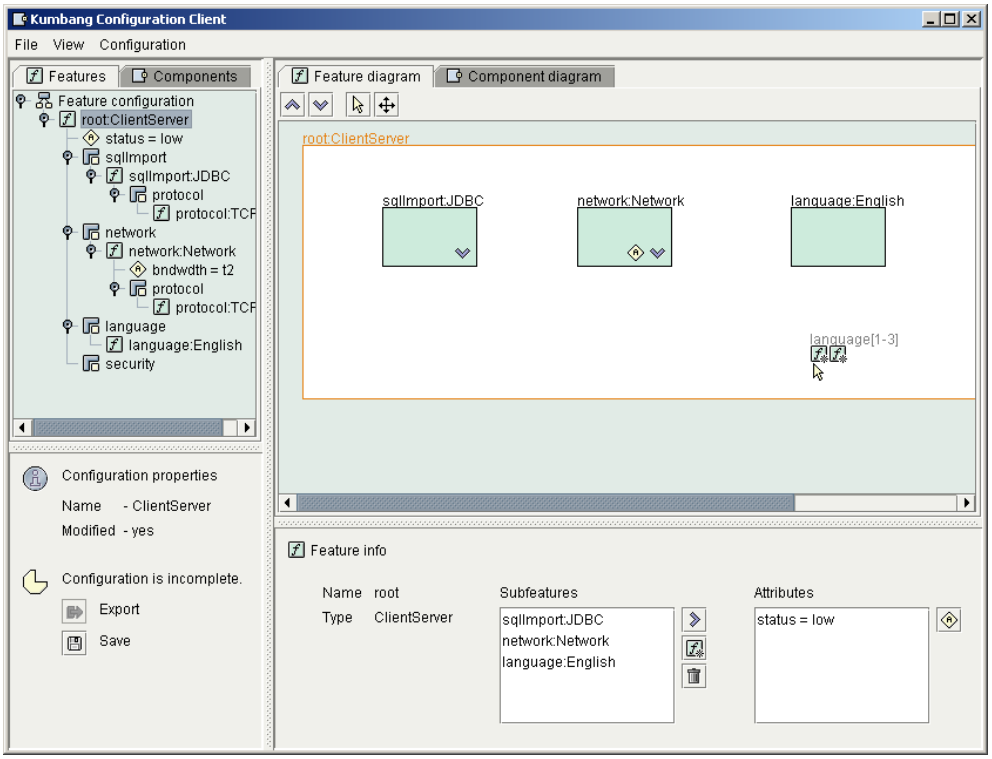

Figure 6.3: A screen shot from the system. This is the view that shows the feature configuration. The configuration tree is on the left, and it shows the whole compositional hierarchy. Diagram representation is on the right, and it shows the compositional hierarchy one level at a time.

Figure 6.4: An example of the locations in the UI where the user can add new instances. It shows the name and cardinality of the part. The number of icons corresponds to the number of instances that can still be added (in this example, one can add one component). There is one location for each part in the diagram, and the user can move or hide them if desired.

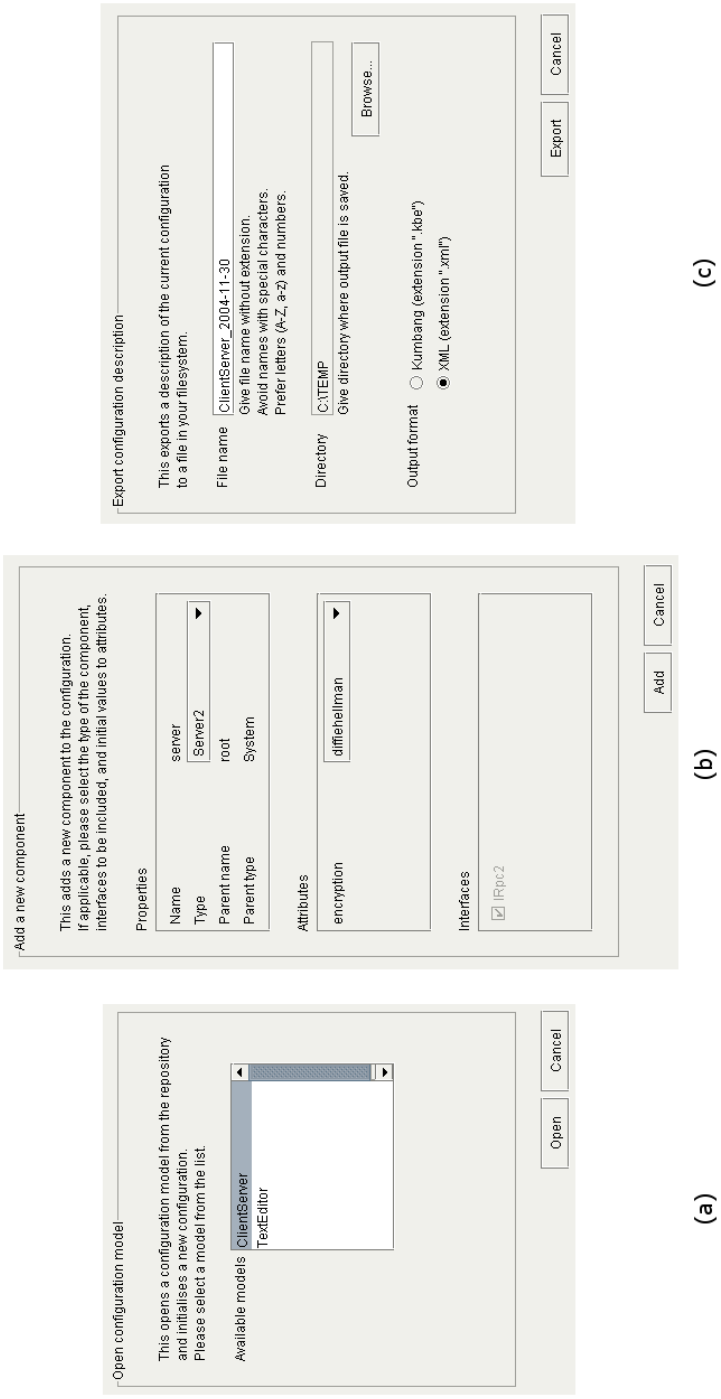

Figure 6.5: These screen shots from the dialogs of the system exemplify three basic steps: opening a configuration model (a), adding new instances to the configuration (b) and exporting a description of the configuration (c).

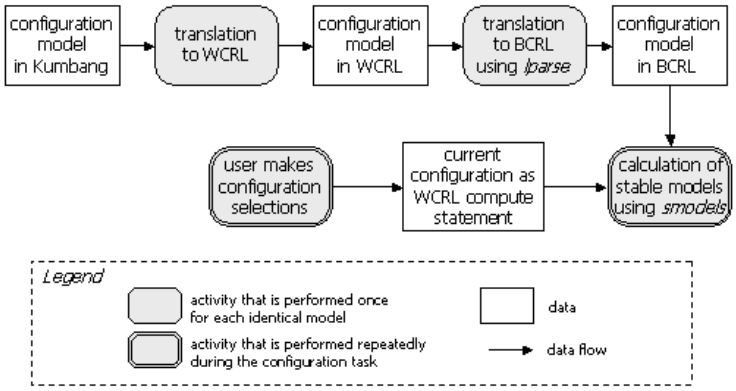

Figure 6.6: This figure depicts how the system uses *smodels* for configuration reasoning

editors in many respects. For example, Poseidon UML editor (Poseidon for UML, 2004) has the same basic structure for creating UML diagrams. It has a list of elements, a graphical diagram and a panel for detailed information. But there are obvious differences also. For example, the hierarchical structure is much more predominant in Kumbang Configurator. But it can be stated that it is often beneficial for a tool to resemble other typical tools. If user is familiar with the layout, it is easier to use the tool.

6.2.1 Implementation of Configuration Reasoning

The configuration reasoning implemented in the system resembles the configuration reasoning of WeCoTin (see Section 3.3.1) in many respects.

First and foremost, the system uses *smodels* and *lparse* (Simons *et al.*, 2002) for configuration reasoning. Figure 6.6 shows how the system achieves this.

Before one can utilise *smodels*, the configuration model must be translated into a form understood by *smodels*. These activities constitute the upper part of Figure 6.6. They are performed before the configuration task starts, and only once for each configuration model. Firstly, the configuration model is translated into WCRL. WCRL (Weight Constraint Rule Language) is a general-purpose knowledge representation language, which is a form of logic programs (Simons *et al.*, 2002). After that, the model is translated into BCRL (Basic Constraint Rule Language), which serves as a

ID	*Priority*	*Status*	*ID*	*Priority*	*Status*	*ID*	*Priority*	*Status*
B1	1	x	C1	1	x	U1	1	x
B2	1	x	C2	1	x	U2	3	-
B3	2	x	C3	1	x	U3	1	x
B4	2	x	C4	1	x	U4	2	x
B5	1	x	C5	1	x	U5	2	x
B6	1	x	C6	2	-	U6	1	x
B7	2	x	C7	3	-	U7	2	x
B8	1	x	C8	3	-	U8	1	x
B9	3	-	C9	2	x	U9	3	x
B10	2	x	C10	2	x	U10	3	-
B11	2	x	C11	2	x			
B12	2	x						
B13	3	-						
B14	2	x						

Table 6.6: A requirement matrix that shows how each requirement has been fulfilled. Column *Status* shows whether the requirement has been implemented (x) or not (-).

"normal form" for general weight constraint rule language (Simons *et al.*, 2002). This compilation is called *grounding*, and it is potentially very costly, since it removes the variables in a WCRL program.

During the configuration task, the system repeatedly performs the activities in the lower part of Figure 6.6. When the user modifies the configuration, the system constructs a *compute statement*, which specifies the current configuration. Using this statement, one can call *smodels* to produce desired *stable models* that identify the status of the configuration. Further, configuration consequences can be identified from a partial stable model. A partial stable model tells which statements concerning the configuration must be true, which must be false and what is still unknown.

6.2.2 Implemented Requirements

Table 6.6 contains a matrix that shows whether each requirement has been implemented. The requirements are labeled with same ID numbers as in Tables 6.2, 6.3, 6.4.

Quality attribute goals (Table 6.5) have been omitted, since it is very hard to say exactly whether one has met such unmeasurable goals or not. Out of 35 total requirements, the system implements 28. Out of 28 total implemented requirements, 14 requirements are essential, 13 are conditional and one is optional. Out of 7 unimplemented requirement, 6 are optional and one is conditional. Thus one can roughly summarise that the system implements almost all essential and conditional requirements and almost none of the optional requirements.

Implemented and unimplemented requirements are discussed more throroughly in Section 8.1.

6.3 System Architecture

This section discusses the architecture of Kumbang Configurator. It presents the scope and context of the system and describes the architecture through several architectural views.

According to IEEE standard (IEEE Std 1471, 2000), every system has an architecture, whether it is recorded or not. Architectures are recorded by an architectural description, which is organised into one or more architectural *views*. Each view addresses one or more concerns of the system stakeholders. A certain view of the system is characterised by a *viewpoint*. Thus a viewpoint defines the language and methods for composing a certain view for the system. (IEEE Std 1471, 2000)

A view can adopt one or more styles. An architectural *style* is a common form of design. It determines the vocabulary of how component and connectors can be arranged. This can include topological constraints (for example, no cycles). Styles can be applied system-wide, or they can be applied to a very specific area of the system. Examples of well-known styles include pipes and filters, layers, client-server and blackboard. (Shaw and Garlan, 1996)

6.3.1 Language and Platform

Kumbang Configurator is implemented using Java programming language. There are several reasons for this. Firstly, Java is virtually platform independent. Thus the configurator tool can easily be ported to many platforms, including Windows, Linux and

Unix variants. Secondly, Java offers many built-in mechanisms that ease the implementation. These mechanisms include network protocols like Remote Method Invocation (RMI), graphical user interface (GUI) libraries like Swing, logging services, and many others. Finally, the author of this thesis has good knowledge in Java, thus enabling faster development cycles. Naturally, one could argue that there are some drawbacks in Java. Considering the performance, C++ might be a better solution. Fortunately, the computationally heaviest part, checking the consistency of the configuration, is performed by external modules *smodels* and *lparse*.

The implementation uses Java version 1.4.2. There are some parts of the system, like logging, that use capabilities that are available only in versions 1.4 or higher. Thus the code cannot be ported to older versions unless these parts of the system are rewritten.

At the moment, the implementation works only on Windows platform. This is because the tool requires binaries of *smodels* and *lparse*, which are platform-dependant. In order to use the tool on other platforms, one needs to compile *smodels* and *lparse* binaries to these platforms also.

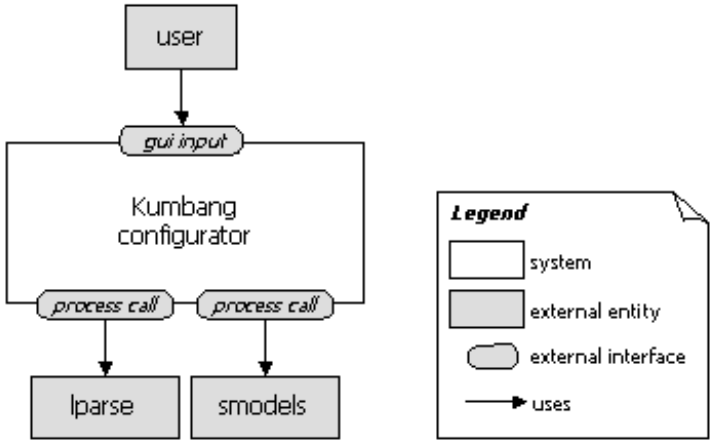

Figure 6.7: The context of the system and system interactions with the external world.

6.3.2 System Context

Figure 6.7 describes the system context. Almost all software system have interfaces to one or more external entities. A system context model specifies externally visible properties of the system through these interfaces. The system may have interfaces to several external entities at different levels—an external system can be either a higher-level, lower-level or peer-level entity. (Bosch, 2000)

Kumbang Configurator is a relatively autonomous system. Basically it takes user input through a graphical user interface. At the other end, the tool sends commands to *smodels* and *lparse* modules in order to check the configuration for consistency and completeness. This communication happens through process calls.

6.3.3 Structural View

The structural viewpoint defines a view that shows the system divided into its parts. These parts, or components, hide some chosen aspects of the overall system. Clearly separated responsibilities and interfaces are essential to this view. According to IEEE standard (IEEE Std 1471, 2000), structural viewpoint tries to answer the following concerns: what the computational elements of a system are, how those elements are organised, what components and interfaces constitute the system and how they are connected.

Figure 6.8 shows a simple model that presents the system from the structural viewpoint. Further, Table 6.7 lists the elements found in the view and describes their responsibilities briefly.

One can identify several architectural decisions that concern the structural view of the system. In the following, one discusses decisions and their rationale.

Decision: The system is distributed into one server and several clients. The structural view employs an architectural style that is called *client-server*, which is one form of distributed styles. In such a system, server provides services to clients, but the server does not know the identities of the clients in advance. On the other hand, clients know the identity of the server and access its services through remote procedure calls. (Shaw and Garlan, 1996)

There were several reasons for applying the client-server style. In any case, the

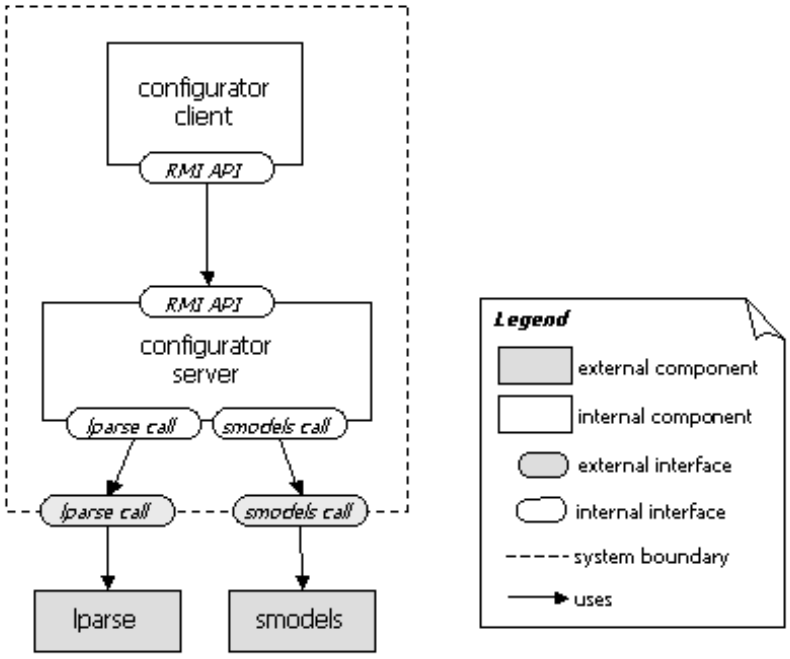

Figure 6.8: The structural view of the system. The system is divided into two components, client and server. In addition, server operates two external components, *lparse* and *smodels*.

system has to provide some kind of distribution. The need for distribution comes from the requirement that configuration models have to be managed centrally. With a central model repository, user can operate the tool without having a configuration model at hand. Thus centrally managed models enable the separation of domain engineering and application engineering organisations. But having distribution is also beneficial to configuration reasoning. Configuration reasoning is the computationally heavy part, and it operates external modules *smodels* and *lparse*. Thus the configuration reasoning can be assigned to a dedicated, robust high-end computer, while the client can run on a light-weight hardware. Finally, distribution brings flexibility to the system. In fact, if the above-mentioned issues are not applicable, one can run the client and the server in the same machine.

Among different distributed styles, the client-server style was chosen partly because it is relatively simple and easy to implement, partly because it fits intuitionally with the responsibilities of the components.

Element	Description
configurator client	Configurator client takes care of direct user interaction. It provides a graphical user interface that shows the configuration and allows its modification. Client also manages the current configuration and stores data structures that represent the configuration.
configurator server	The main responsibility of configurator server is to provide services for configuration reasoning. Server also manages model repository that contains loaded configuration models. In addition, server manages sessions with several clients.
RMI API	Server provides a RMI interface for clients. This interface contains services for configuration reasoning as well as loading and fetching models and starting and ending sessions with clients.
lparse call	Server calls external *lparse* component by creating a separate process and giving the input as a WCRL file that contains the configuration model. The result of this call is the same program translated into BCRL.
smodels call	Server calls external *smodels* component by creating a separate process and giving the input as a BCRL program.

Table 6.7: Element catalogue for structural view of the system.

Decision: Client has two main responsibilities: providing a graphical user interface and managing the configuration. The main responsibility of the client is to offer a graphical user interface, which visualises the configuration and configuration task activities. Another responsibility is to manage the current configuration; this means that the client must manage data structures that represent the configuration.

It is quite natural to assign the graphical user interface to the client. Trickier part is to decide whether the configuration data structures should reside on the client or on the server. This system implements *thick client* scheme for several reasons. Firstly, the graphical user interface is quite tightly connected to the actual structures that represent the configuration. If these structures resided on the server, the client would be constantly asking information about the current configuration. This would increase remote traffic. Furthermore, especially large configurations can take quite a lot memory. It is thus beneficial to divide this memory consumption between all clients, not to

dump all structures from all clients to the server.

Decision: Server has three main responsibilities: providing configuration reasoning, managing model repository and managing sessions with clients. The main responsibility of the server is to provide services for configuration reasoning. Configuration reasoning employs two external components, *lparse* and *smodels*. This communication happens through process calls. Unfortunately, external *smodels* and *lparse* modules accept input only through files. When the system needs to check the the configuration, the system constructs a file that contains the current configuration along the rules of the configuration model in WCRL. After that, the system creates a process and calls *lparse* in order to compile this file into BCRL. Finally the system creates a process and calls *smodels* with the BCRL file as input. Fortunately, parts of these operations can be done in advance, so that the actual configuration reasoning query can be conducted faster. Especially the potentially costly operation of translating the WCRL file into BCRL file can be done in advance. Thus only the current configuration must be updated for each configuration reasoning query.

Another responsibility is to manage configuration models. This enables centralised model repository that every client can use. Clients can load pre-designed configuration models to the server, and then initialise configurations by selecting a configuration model from a list. The server stores each loaded model in four forms. One is a textual Kumbang model, another is a serialised binary file that contains the model parsed into Java objects. Third and fourth are WCRL and BCRL files that contain the model rules. Since these files are constructed and stored in advance, one can speed up the initialisation of the configuration and configuration reasoning.

Thirdly, since the server must manage several clients with different configurations, the server must keep track of sessions with each client. For each session, the server stores identity of the client and configuration information.

Decision: Clients use server through RMI (Remote Method Invocation). The communication technique between clients and a server is Remote Method Invocation (RMI). Another option would have been using raw network sockets and sending commands with a proprietary protocol. However, there would have been several drawbacks with this approach. Firstly, one should have invented the protocol for communication.

Although the protocol would have been quite simple, this task would have taken some time. Secondly, RMI is quite easy to implement.

Using RMI, the communication between clients and a server is quite straightforward. The server offers one remote interface for clients, which use this interface for calling the server. In addition, there is a set of serialisable objects that are passed as parameters in the communication. The RMI interface offers services for configuration reasoning, loading and fetching models and starting and ending sessions.

Decision: Server uses *lparse* and *smodels* through process creation. One of the requirements was that the system must employ *lparse* and *smodels* modules for configuration reasoning. These modules offer two ways to operate. Firstly, one can call these modules from the command line, which basically creates a new process for each command line call. Secondly, one can directly operate *smodels* through C++ application programming interface (API). Unfortunately, given the scarce resources of the implementation, the task of bridging a C++ interface to the system through Java Native Interface (JNI) would be too great. Thus this system employs direct process calls, which are less effective performance-wise.

6.3.4 Layered View

According to Bachmann *et al.* (2000), the layered view of the architecture is one of the most commonly used view in software architecture. Layering reflects a division of the software into units, which are called layers. A layer is a collection of software units that provide together provide a cohesive set of services. The relation among layers is "allowed to use". If two layers are related by this relation, any unit of software in the first layer is allowed to use any unit of software in the second. The layers in a system are often ordered, and the usage in them flows downwards. This means that layers are dependent on layers below them and independent on layers above them. (Bachmann *et al.*, 2000)

This system uses a modified version of the layer style. Figure 6.9 depicts the layered view of the system. There are five layers that are ordered on top of each other, and the relations between these layers are typical to the layering scheme: a layer may use the services of the lower layer. In addition, there is one layer that provides services

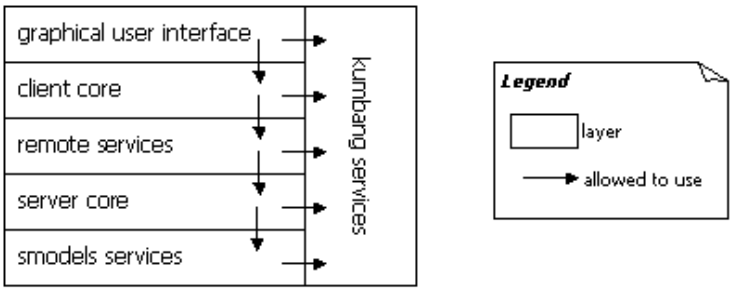

Figure 6.9: The layered view of the system. The system is divided into six separate layers. One of the layers provides services for all the other layers, while other layers use only services provided by the adjacent lower layer.

Element	**Description**	**Mapping to elements in structural view**
graphical user interface	Provides a graphical user interface.	Maps to *configurator client*
client core	Manages current configuration and connects to the server.	Maps to *configurator client*, including required interface *RMI API*
server services	Provides remote services to the client, manages sessions.	Maps to *configurator server*, including provided interface *RMI API*
server core	Manages model repository, handles and relays configuration reasoning requests.	Maps to *configurator server*
smodels services	Provides configuration reasoning services.	Maps to *configurator server*, including required interface *lparse call* and *smodels call*
kumbang services	Provides services to the whole system.	Maps to all elements

Table 6.8: Element catalogue for layered view of the system, and mapping to elements in the structural view.

for all the other layers. This particular layer is drawn vertically in Figure 6.9.

Table 6.8 lists the elements found in the layered view. In addition, it provides a mapping to the elements found in the structural view (see Section 6.3.3). The mapping is quite straightforward. Client maps to two layers that separate two main responsibilities of the client: providing a graphical user interface and managing configuration data structures. Server maps to three layers that roughly correspond to the three main responsibilities: session management, model management and configuration reasoning. But there are some services and operations that are clearly needed in many layers. These services include basic operations and data structures that represent configuration models and configurations. Thus one of the layers is called *kumbang services*, and it provides services that stem from Kumbang language. These services are needed throughout the system.

There are a couple of architectural decisions that are relevant to this view.

Decision: Each major responsibility in the structural view roughly corresponds to one layer in the layered view. This decision is quite easy to justify, since assigning responsibilities to layers provides the cohesion that is required from architectural layers in general.

Decision: All those services and structures that relate to Kumbang language and are needed in other parts of the system are separated into one layer. This decision separates all functionality that stems from Kumbang language into a separate layer. This decision tries to enhance the extendibility of the system. Kumbang language is independent of the configurator implementation, which means that those services that provide Kumbang functionality are independent from the rest of the system. This means that Kumbang layer can be separated from the implementation of this system, and be used in the implementation of another system. For example, Kumbang layer could potentially be used as such for implementing Kumbang modelling tool.

6.3.5 Code Architecture View

Code architecture view describes the view divided into its implementation units, which means that it relates directly to the system implementation. An example implementation unit is a package, which can be further aggregated into a package hierarchy. A

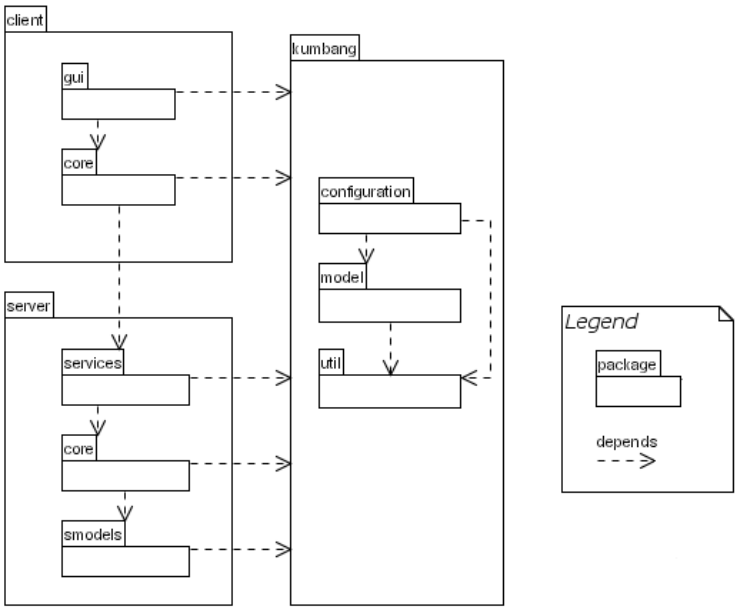

Figure 6.10: The code architecture view of the system. This model shows the code packages that constitute the system and how they depend on each other. Packages below these packages are omitted in order to keep the model easy to read.

package is said to be a leaf package if it does not contain any other packages, otherwise it is an aggregate package. Relations between packages are dependency relations: one package depends on another package if it uses the services provided by that package.

Figure 6.10 depicts a model from the code architecture view of the system, while Table 6.9 lists the elements found in that model. The elements presented in Figure 6.10 and Table 6.9 are Java packages in the implementation. Figure 6.10 and Table 6.9 omit some of the leaf packages that aren't architecturally interesting. However, these packages are mentioned in Table 6.9 as child elements for other packages.

There are a couple of major architectural decisions that relate to this view. There are also other decisions, such as division of packages *client.gui*, *kumbang.configuration* and *kumbang.model* into smaller packages, but these decisions are not architecturally interesting enough to be discussed here.

Decision: Each layer will get its own package, and package dependencies follow layer relations. It is relatively easy to justify that each layer has been assigned its

Element	**Parent element**	**Child elements**	**Implements**
client	-	*gui, core*	Client-related functionality
server	-	*services, core, smodels*	Server-related functionality
kumbang	-	*configuration, model, util*	Functionality covering Kumbang language
gui	*client*	*core, diagram, tree, dialog, template*	Graphical user interface
core	*client*	*task*	Management of the current configuration and connection to the server
services	*server*	-	Functionality that provides services to the client and session management
core	*server*	-	Model management and handling and relaying configuration reasoning requests
smodels	*server*	-	Configuration reasoning and connecting to *lparse* and *smodels*
configuration	*kumbang*	*instance, core, task*	Structures that represent instances in a configuration and some basic operations how those can be modified
model	*kumbang*	*type, parser, visitor, syntaxtree*	Structures that represent types in a configuration model and services that parse these models
util	*kumbang*	-	Kumbang-related utilities, such as logging operations

Table 6.9: Element catalogue for code architecture view of the system. NOTE: packages not shown in Figure 6.10 do not have a description in this catalogue.

own package. Layers in the layered view form coherent units, which can easily be associated into packages. The dependencies between packages (see Figure 6.10) follow exactly the relations of "allowed to use" in Figure 6.9.

Decision: Package *kumbang* is further divided into packages *configuration*, *model* and *util*. The rationale behind splitting *kumbang* package is to separate model-related and configuration-related issues from each other. Entities in the configuration model (such as types) are independent of entities in the actual configuration (such as instances). A configuration model can exist without a configuration, but a configuration cannot exist without a model. If one separates these two issues into packages, one sees that package *configuration* depends on package *model*, since model is used as a basis for the configuration. These dependencies are also shown in Figure 6.10. Further, this separation also acts as a separation of responsibilities, since package *model* was not implemented by the author of this work (see Section 6.4).

In addition to packages *configuration* and *model*, there is a separate package *util* for functionality that is not directly related to model or configuration. This funtionality includes for example logging.

Decision: There is a separate package that represents types in the configuration model and there is a separate package that represents instances in the configuration. The key assets in packages *configuration* and *model* are the data structures that represent instances in the configuration and types in the model. Because they are so central to the system, they are both separated into packages that contain nothing but the data structures. Package *kumbang.model.type* contains classes that represent types in the configuration model, while package *kumbang.configuration.instance* contains classes the represent instances in the configuration. Since the latter package is part of the core functionality implemented by the author, the following paragraphs briefly present the contents of that package.

Figure 6.11 shows the inner structure of package *kumbang.configuration.instance*. The taxonomy of the instances in Figure 6.11 is quite deep, but is still reasonable. The basic building blocks of configurations are components and features (classes *ComponentInstance* and *FeatureInstance*). Since these both can be combined into compositional hierarchies, it is reasonable to derive them from one superclass. Components

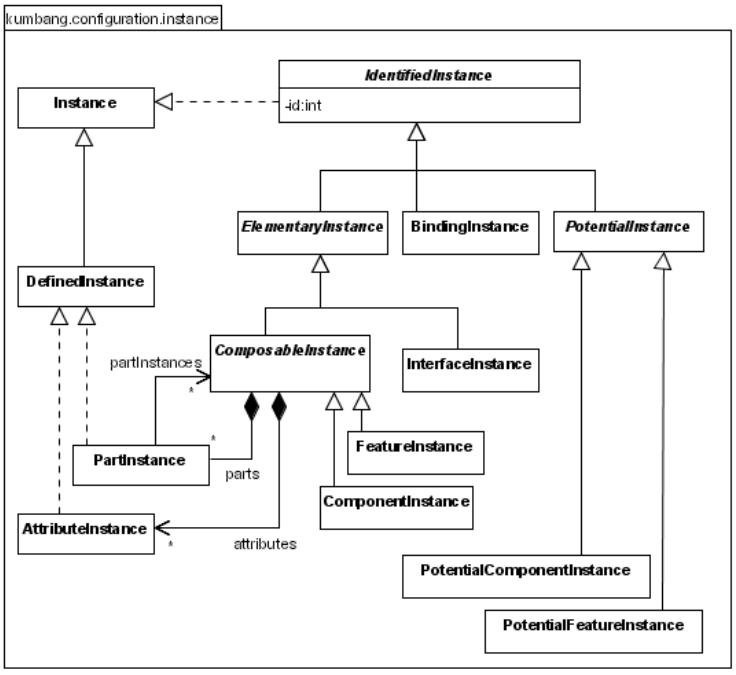

Figure 6.11: Package *kumbang.configuration.instance* provides data structures that represent instances in a configuration. This UML diagram shows some of these classes and their relationships.

and features can have attributes, and they are composed into hierarchies through part instances (class *PartInstance*).

Interfaces, components and features all have a name and a type, thus they extend superclass *ElementaryInstance*. In contrast, bindings do not have a type nor name. But all these instances are extended from *IdentifiedInstance*. Thus all instances have a unique ID that is used for many purposes, such as for naming them to *smodels*.

Class *PotentialInstance* is kind of a convenience class in the taxonomy (see Figure 6.11). This instance represents those locations configuration where new instances can be added. There exists one potential instance for each part in the configuration, and it keeps track of those instances that have been instantiated under that part. Thus it knows how many instances can still be added and how many instances can be removed.

6.4 Contribution from Other Developers

It is important to emphasise that this work was conducted as a part of research in a research group. Thus there are other developers who have contributed to the system implementation. Before the actual implementation of the system started, Timo Asikainen[1] had implemented a preliminary system. This implementation included data structures for configuration models, a parser for parsing a model file into Java objects, and a preliminary implementation for translating the configuration model to WCRL. During the implementation, he continued to make improvements to these parts of the system.

It can be roughly said Timo Asikainen implemented package *kumbang.model* (see Section 6.3.5), although the author of this thesis later re-organised this implementation into the current package hierarchy. Further, the author of this thesis integrated this implementation into the whole system.

However, the line between the responsibilities of each developer is not so clear-cut. Timo Asikainen has made modifications to other parts of the system, while the author of this thesis has made modifications to package *kumbang.model*.

[1] Helsinki University of Technology, Laboratory of Software Business and Engineering

Chapter 7

System Validation

This chapter presents the validation of the system that was implemented to meet the research goals. Kumbang Configurator is depicted in more detail in Chapter 6.

As was discussed in Section 5.3, the system is validated through two example cases: one that is a real-life case from automotive industry (Section 7.1) and one toy example that shows chosen aspects of the system (Section 7.2). These two example types as validation technique are also mentioned by Shaw (2002).

These two example cases are presented in Section 7.1 and in Section 7.2. Both sections first describe the case and its background, describe how it was modelled into a Kumbang model and describe the configuration task. Finally, Section 7.3 draws some conclusions based on these two cases.

The validation techniques, cases and methods are evaluated in Section 8.1.5.

7.1 Case: Car Periphery System

7.1.1 Description of the Case

The case presented in this section is originates from Robert Bosch GmbH (MacGregor, 2004). Robert Bosch GmbH is a company that developes various embedded systems for cars. A distinguishing characteristic of automotive industry is the large number of variants. This is partly because of the diversity of the hardware underneath.

The case depicts a part of a car periphery system (CPS). The CPS product can contain several applications, such as parking assistance, pre-crash detection, blind spot

detection, parking spot detection and so forth. The product that is configured in this case contains only two example applications, parking assistance and pre-crash detection.

Parking assistance application monitors the distance to objects while the vehicle is parking. This includes displaying the distance to the object and sounding an alarm when boundaries are crossed. There are four zones (near, very near, imminent, in proximity) that represent different distances to objects nearby. When an object enters one of these zones, the system uses the corresponding sound and colour to inform the user.

Pre-crash applications try to detect an imminent crash. This includes sensitising the airbag sensor (application PreSet) and firing a seatbelt tensioner (application PreFire). Both applications have corresponding activation zones and trigger zones that either activate the capability or trigger it.

The case material has been fully obtained from a presentation by MacGregor (2004). Figures 7.1-7.4 show the original model that represents the case. (Some parts of the configuration model have been omitted. The original material (MacGregor, 2004) contains the full configuration model.)

The original presentation (MacGregor, 2004) included a demonstration that showed how this particular CPS system can be configured. The configurator tool used in the demonstration is presented by (Hotz *et al.*, 2004). Similar to Kumbang Configurator, it is based on techniques and concepts derived from traditional product configuration. (For further discussion about the tool presented by (Hotz *et al.*, 2004), please refer to Section 8.2.1.)

7.1.2 Constructing the Configuration Model

In many respects, the case is ideal for constructing a configuration model. Firstly, this particular case already represents a configurable product. Secondly, the concepts used in the original case are approximately similar to the concepts utilised in Kumbang language.

However, it was relatively difficult to construct a working configuration model based on the case. The main reason is that the diagrams shown in Figures 7.1-7.4 are meant for communication. The primary driver for the diagrams has been clarity,

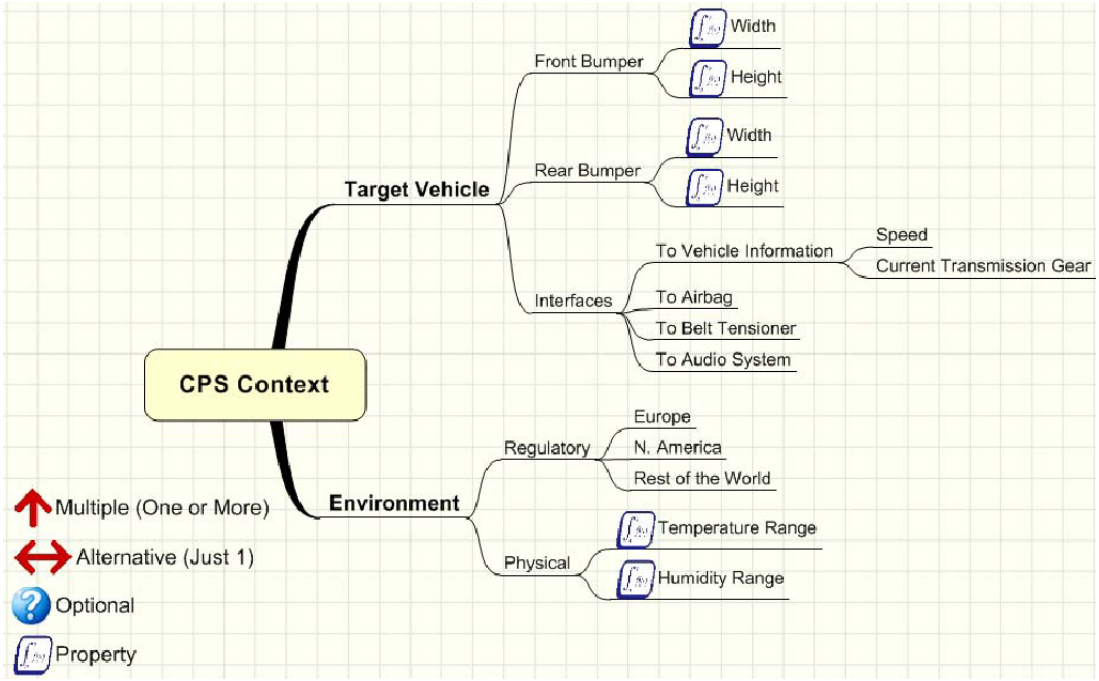

Figure 7.1: The context hierarchy presented in the original source (MacGregor, 2004)

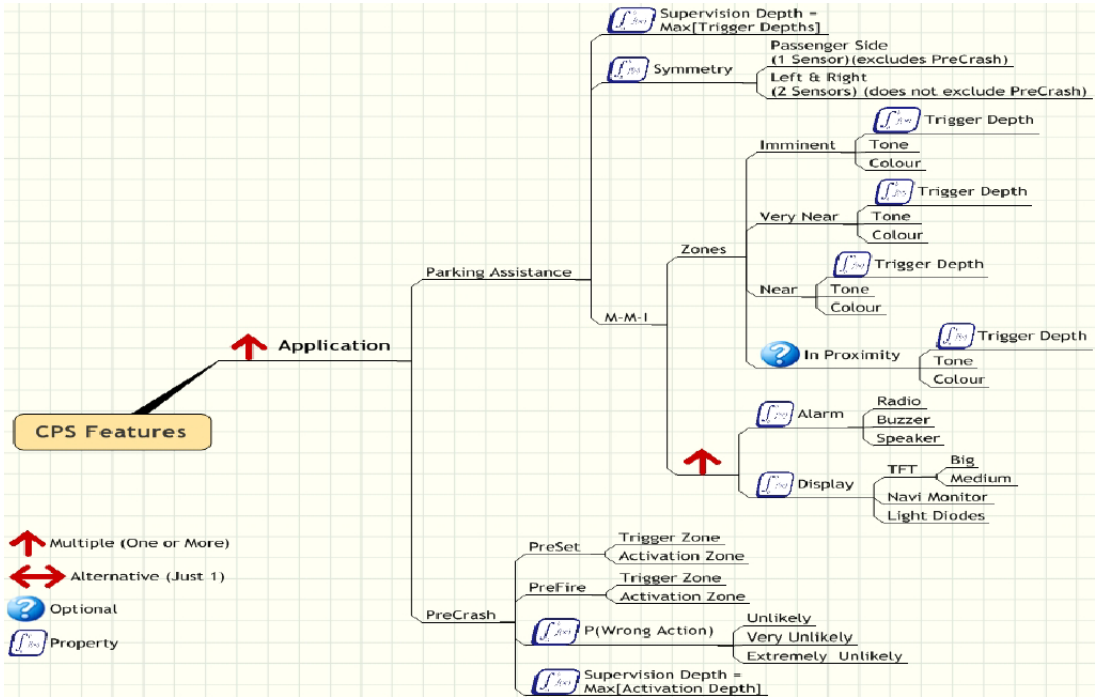

Figure 7.2: The feature hierarchy presented in the original source (MacGregor, 2004)

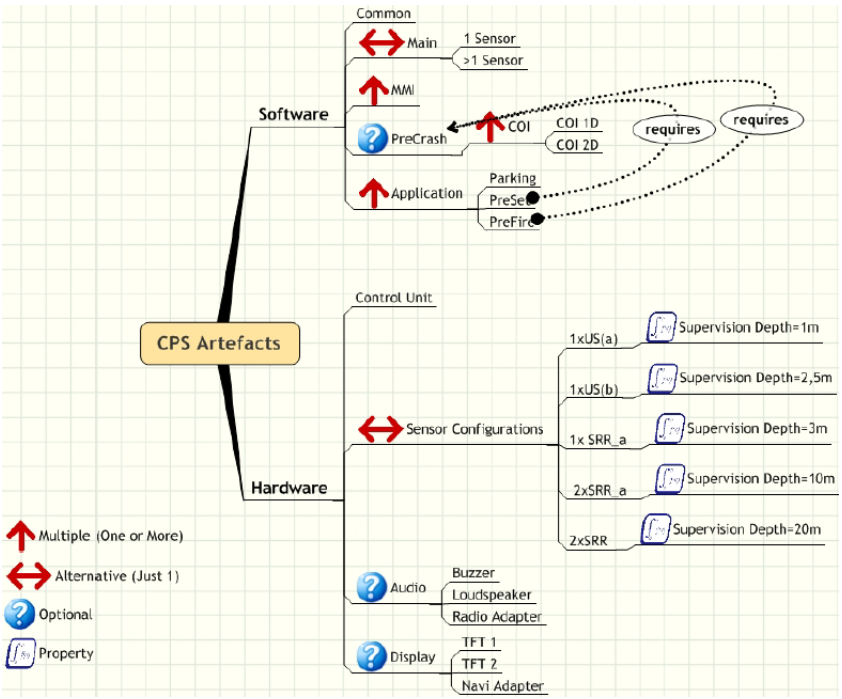

Figure 7.3: The artefact hierarchy presented in the original source (MacGregor, 2004)

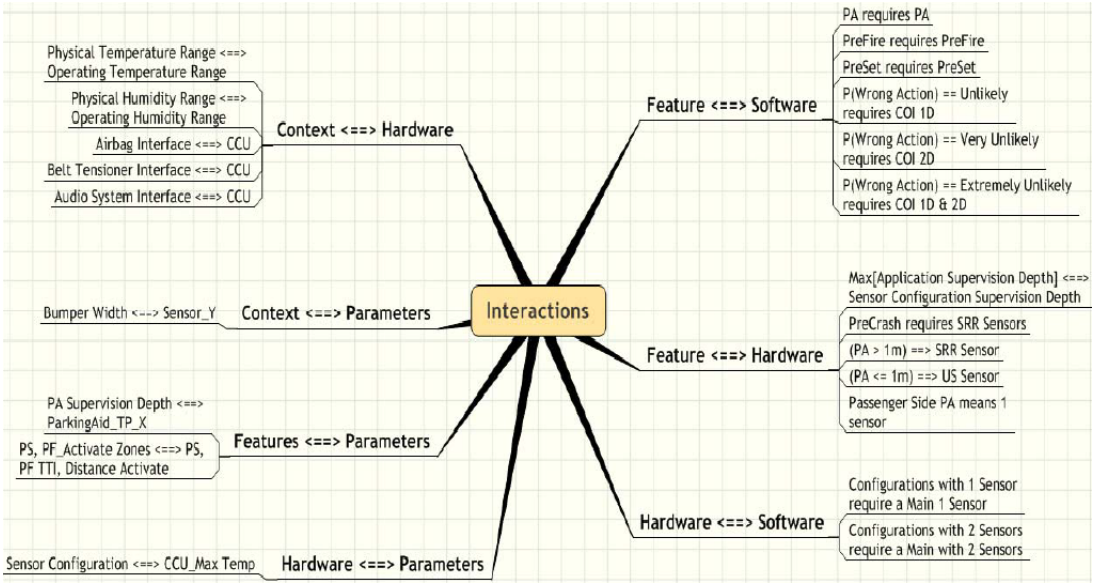

Figure 7.4: The interactions presented in the original source (MacGregor, 2004)

not accuracy. There are clearly some details and underlying relations that have been omitted from the diagrams to make them more readable. In contrast, a configuration model should be fully accurate and unambiguous. Another reason for inaccuracy might be the background of the person who produced the diagrams. A person who knows the domain really well may think that some aspects of the model are so "trivial" that one doesn't even have to explicate them. A similar situation has been identified in requirements engineering research (Sommerville, 2004).

The rest of this section presents how the case described in Figures 7.1-7.4 was modelled into a working Kumbang model.

The basic concepts utilised in the original case are features (Figure 7.2), software and hardware artefacts (Figure 7.3), context (Figure 7.1), parameters, properties and interactions (Figure 7.4). Features in the original case are directly mapped to Forfamel features. Since Kumbang does not provide a concept for application context, context elements are mapped to a separate branch in the feature hierarchy. Software and hardware artefacts are mapped to Koalish components, each into a separate branch in the component hierarchy. Properties in all elements are modelled as Kumbang attributes. Since this tool does not support property ranges, ranges are modelled as two separate attributes. Interactions in the original case are modelled as constraints in the Kumbang model. Finally, parameters are mostly omitted, but some of them are modelled as attributes.

But direct mapping doesn't yield a complete configuration model. For example, the original model contains some properties that are clearly not meant to be attributes set by the user. For example, each sensor configuration (Figure 7.3) is associated with a fixed property value that describes the maximum operating range of the sensor. Further, the feature hierarchy contains two supervision depth properties that are calculated from zone depths. The idea is that the zone depths should not exceed the operating range of the sensor configuration. To reflect this idea, these properties are not modelled as attributes, but are encoded in the configuration model as constraints.

Another issue that is not covered in the original material is attribute values. That is, what are the possible values for each property? Without proper domain knowledge, one has to guess applicable values for each attribute in the configuration model. Further, there are some features that seem to be empty (for example, *Tone* and *Colour* in Figure 7.2). The situation is corrected by modelling them as attributes that have fixed

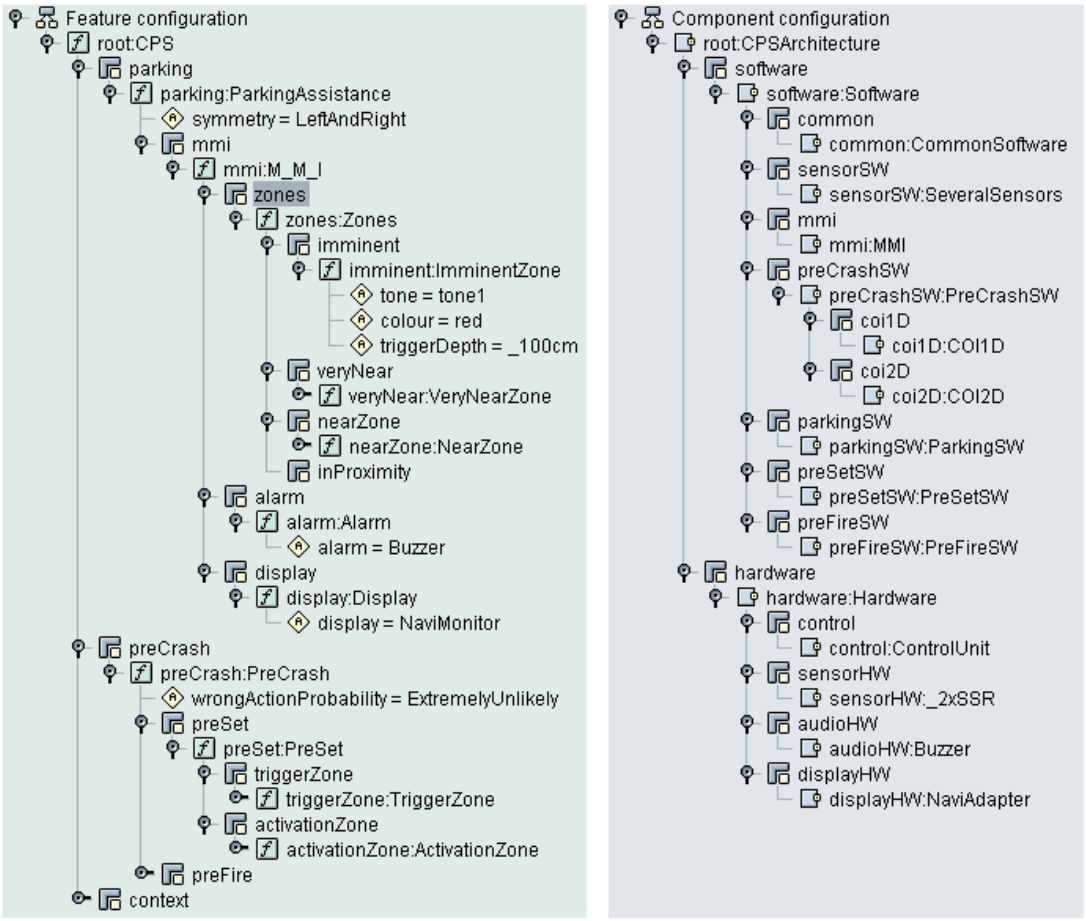

Figure 7.5: A screenshot that shows an axample configuration. Feature configuration is on the left, while component configuration is on the right. (Note that some instances in the feature configuration have been collapsed to keep the figure size reasonable.)

sets of colours and tones available. Finally, the system lacks some interactions (Figure 7.4) between elements. For example, the Kumbang model contains constraints that map display features and alarm features to corresponding hardware.

In addition, there are some issues that are completely omitted from the Kumbang model. For example, it seems that interfaces in the context hierarchy (Figure 7.1) don't have much relevance during the configuration task. Thus they are omitted altogether from the Kumbang model. Further, the original case specifies a large number of parameters. Some of these parameters are modelled as attributes, but most of them have been left out. It is still a bit unclear what the role of these parameters is in the configuration task.

7.1.3 Configuration Task

Figure 7.5 contains an example configuration that has been derived from the Kumbang model. At the moment, the feature configuration does not fully determine the component configuration. In some cases, depending on the values given to zone depths, the choice of the sensor configuration is still open. It is probable that this is due to some missing relation that one is not able to deduce from the original description of the case.

During the configuration task, it became evident that one could indeed need some support for the configuration order. When the configuration is large, it is hard to remember which selections have already been made and which are still open. If the tool provided an order according to which selections were made, the configuration task would become easier.

For approximate performance measurements of the case, see Table 7.1 on page 96.

7.2 Case: Weather Station Network

7.2.1 Description of the Case

The problem with the case presented in Section 7.1 is that it doesn't portrait all capabilities of the system. Namely, it doesn't utilise interfaces or connectors. This is partly because the tool that was originally used for configuring the case does not support connectors. Since interfaces and connectors are one of the key contributions in this work, this section presents a case that utilises those concepts.

Unfortunately, this is an invented toy example, not a real-life case as the one presented in Section 7.1. However, the case is motivated by a real-life system. Further, this case was developed to demonstrate Kumbang Configurator and its capabilities, which was one of the underlying goals of this research. The demonstration was held in a seminar targeted for industrial partners, and the audience had mainly technical background.

The case depicts a distributed weather station network that is used for measuring weather at separate physical locations (see Figure 7.6). The system consists of one central server and one to three measurement nodes. Each measurement node contains one collecting main module and several sensors. There are three different sensors, one for measuring temperature, one for measuring pressure and one for measuring wind.

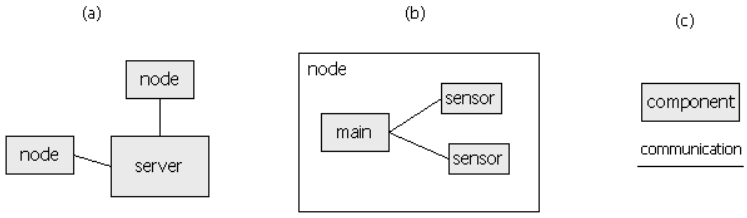

Figure 7.6: A sample view of one product instance in the case family. Figure (a) shows how nodes are connected to the server, while (b) shows internals of one particular node. Figure (c) contains a legend.

The main module collects the data from the sensors and sends it to the central server. The central server stores this data in a database.

7.2.2 Constructing the Configuration Model

The system is modelled to contain three features with several attributes in each. It would have been possible to use features instead of attributes, but attributes provided a simple and conventient way of modelling and configuring the case. At the moment, features are used for grouping together similar attributes; they do not provide any variability as such.

Feature *Measurement* specifies the number of measurement nodes (attribute *numberOfNodes* with value range from one to three) and types of weather measures collected (attributes *windMeasurement*, *temperatureMeasurement* and *pressureMeasurement* with Boolean values). Feature *Network* specifies two attributes that determine whether *bandwidth* is high or low, and whether communication is *wireless* or not. Finally feature *Storage* specifies attribute *dbTechnology* (either mySQL or Oracle).

The model is designed so that any complete feature configuration fully determines the corresponding component configuration. Firstly, the number of *Node* components is specified by one attribute in feature *Measurement*. Measurement attributes in feature *Measurement* specify which sensor components are present in each node - possible types are *WindSensor*, *TemperatureSensor* and *PressureSensor*. Secondly, attribute *dbTechnology* specifies whether component *Server* contains component *DBmySQL* or *DBOracle*. Finally, feature *Network* determines the protocol used in the *Server* component.

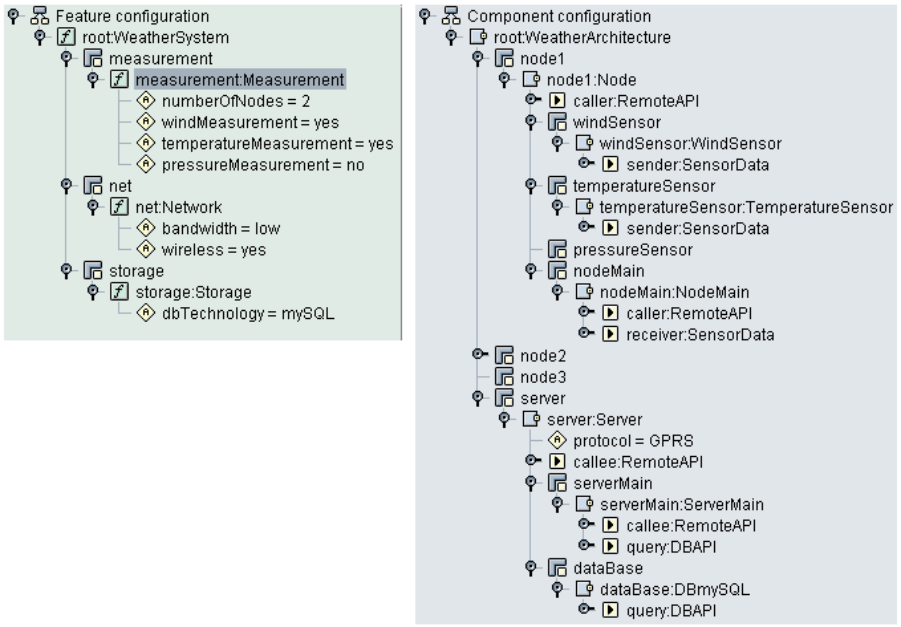

Figure 7.7: An example configuration derived from the case family. Feature configuration is on the left, while component configuration is on the right. (Note that some instances in the component configuration have been collapsed. Thus this figure does not show bindings or internal structure of the second node.)

Unfortunately, there was one thing that couldn't be modelled as desired. It would have been appropriate to model the nodes as one part that has cardinality from one to three (see Figure 7.8). But in order to write implementation constraints for nodes and node sensors, one would have needed logical operators *all* ($\forall$) and *exists* ($\exists$). And since the system doesn't support such operators, one was forced to model the nodes differently. At the moment, the model specifies three separate parts for nodes, where each part represents one node that can be either present or not present (see Figure 7.8).

```
component WeatherArchitecture {
  contains
    Server server;
    Node node[1-3];
  ...
}
```

```
component WeatherArchitecture {
  contains
    Server server;
    Node node1[0-1];
    Node node2[0-1];
    Node node3[0-1];
  ...
}
```

Figure 7.8: It would have been appropriate to use cardinalities from one to three (left side). Unfortunately, the model had to be written the way shown on the right side.

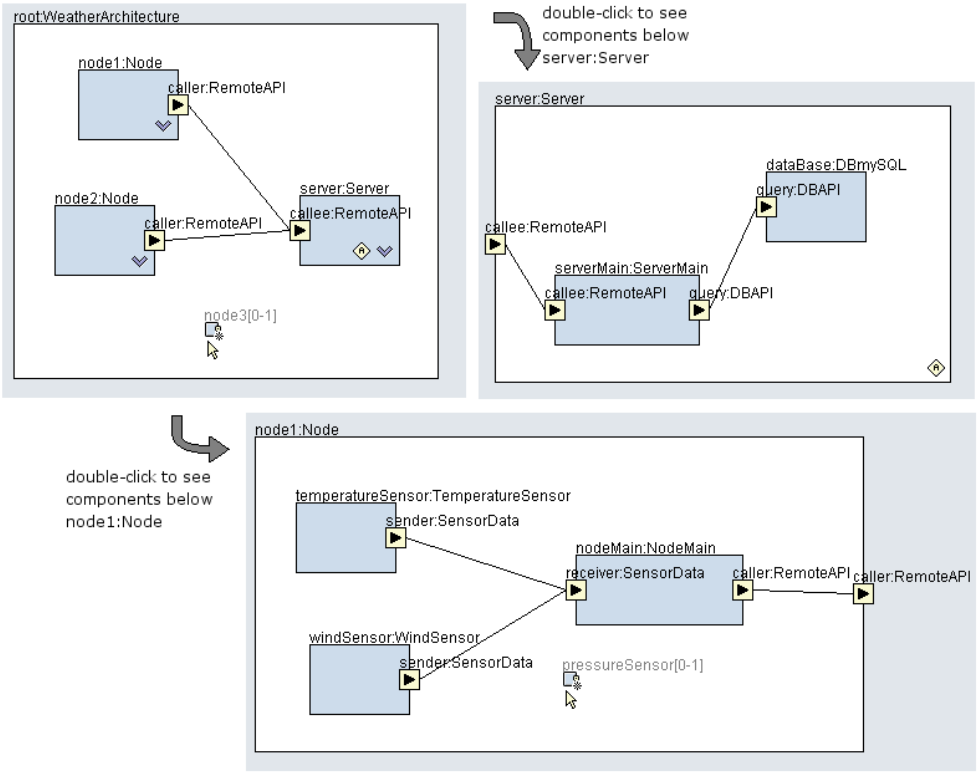

Figure 7.9: Component diagrams that represent an example configuration derived from the case family.

7.2.3 Configuration Task

The configuration task of the case went quite smoothly. Figure 7.7 and Figure 7.9 show screen shots from an example configuration. This configuration was constructed so that one first specified all attribute values in the features, and the tool completed the component configuration from those selections.

Unfortunately, the tool failed to automatically add one binding in the component configuration. At the moment, the configuration rules written in WCRL are constructed so that *smodels* cannot deduce the existence of a binding between *DBAPI* interfaces (see lower right corner in Figure 7.9). This is because the component type that contains the provided *DBAPI* can be either *DBmySQL* or *DBOracle*. It thus seems that one should update the translation rules to WCRL to cover this kind of situation.

For approximate performance measurements of the case, see Table 7.1.

Response time	Grounding the model	Invoking configuration engine
Case 1	4,0s	0,5s
Case 2	0,3s	0,2s

Table 7.1: Approximate performance of the cases with a Windows 2000 PC, 800MHz Pentium processor, 512MB RAM

7.3 Validation Conclusions

The feasibility of the tool was clearly demonstrated by these two examples. However, the validation raised several issues that could be improved.

It seems that a couple of new capabilities to the conceptual basis would ease the modelling task.

Firstly, in many cases one could utilise hidden attributes and default attribute values. Hidden attributes could act as 'value locations' and provide a way to simplify complex constraints. Secondly, the rules that are used for mapping a Kumbang model to WCRL should be enhanced. This way, *smodels* would be able to deduce the missing binding that was discussed in Section 7.2.3. Finally, the constraint language should be extended. For example, although equivalence relation is useful for describing the mapping between features and components, the system does not support writing equivalence constraints. One can replace an equivalence with two implications, but it is tedious to write them explicitly in the model.

Also configuration tasks revealed some areas that could be improved.

Firstly, the system should provide support for selection order during the configuration task. This need was more evident in the first case, since the size of the second configuration is rather small. Secondly, the configuration trees quickly become very deep. This problem could be alleviated by hiding part nodes from the tree. Especially when there is only one instance in a part, an instance could be located directly under its parent. Finally, the performance of the system could be further investigated and improved. At the moment, small configuration models are configured rather quickly, but as the configuration grows, the performance may become an issue.

Chapter 8

Discussion

This chapter discusses the work that was done in this research. Section 8.1 evaluates the work from different perspectives, while Section 8.2 presents related work and compares it to the work done in this thesis. Finally Section 8.3 compares the overall work with the research objectives set in Chapter 5.

8.1 Evaluation

This section evaluates the research conducted in this thesis. Namely, it evaluates the system implementation (Chapter 6) and validation of the system (Chapter 7). Since these subjects have been covered elsewhere in the thesis, it is assumed that the reader is familiar with the system implementation and validation covered in this thesis.

In the following, one first evaluates the system against the requirements set in Section 6.1. This is done by evaluating basic requirements (Section 8.1.1), requirements covering configuration reasoning (Section 8.1.2), user interface requirements (Section 8.1.3) and quality attribute goals (Section 8.1.4). After that, Section 8.1.5 evaluates the validation of the system through example cases. Finally, Section 8.1.6 covers some other aspects that are worth discussing because of their future relevance.

8.1.1 Evaluation of Basic Requirements

Basic requirements of the system (see Table 6.2) cover such issues as opening, saving and modifying the configuration and model management. These requirements were

fulfilled quite well.

The first basic requirement stated that the system must fully conform to Koalish, Forfamel and Kumbang languages. This requirement was easily achieved. This might be due to the fact that one of the developers had originally designed these languages. In fact, there were some minor modifications to the Koalish, Forfamel and Kumbang that were found necessary or helpful during the implementation. For example, now each constraint ends with a semicolon (;). In addition, the translation of the languages to WCRL was also modified a bit (original Koalish translation can be found in (Asikainen *et al.*, 2003b)).

All requirements that covered opening, saving and modifying the configuration were implemented, and they were implemented pretty well. At the moment, one can think of one enhancement only: the system should be able to save layout information along with the partial configuration.

The model management now resides on a dedicated configuration server. At the moment, the user can specify a name for the model, and this name is used as a file name for saving appropriate model files. A better solution could utilise separate property file for each model. This property file could contain a description of the model, a version number, a password and so forth. Even a better solution could utilise a database for storing model information and model files. Further, the model uploading and fetching now happens through configuration client. A better solution could include a separate password-protected client for managing configuration models. This would enable separation of tasks for domain engineering people, who are responsible of model management, and application engineering people, who are responsible of configuration.

There are two basic requirements that aren't implemented, but both of them are optional. First unimplemented requirement covers function binding. Basically the system has been implemented so that this is easy to add. However, this would also require extensions to the Koalish language. Second unimplemented requirement covers saving additional data with the models. Although this requirement is rather easy to implement, there wasn't enough time to start implementing it.

8.1.2 Evaluation of Configuration Requirements

Configuration requirements of the system (see Table 6.3) cover issues that are related to configuration reasoning. These requirements were probably the hardest to implement. Unfortunately, this difficulty was reflected in how they were implemented. The system currently implements eight out of eleven configuration requirements (five essential and three conditional requirements).

The first configuration requirement states that configuration reasoning must employ *smodels* and *lparse* modules. This requirement was absolutely mandatory, since it wouldn't have been feasible to implement the configuration engine from scratch. But using *smodels* and *lparse* also brought some difficulties. Firstly, one had to build the configuration knowledge so that one could utilise *smodels* and *lparse* functionality. Secondly, integrating two external modules to the system required quite a lot of work and increased testing effort of other parts. Thirdly, *smodels* gives very little feedback about inconsistent configurations, especially when it is invoked through a process call.

The integration to *smodels* and *lparse* modules is one of the weak points of the system. At the moment, the system invokes these modules through process creation. Further, *lparse* takes its input as filesystem files only. It is obvious that this is an extremely heavy way of calling an external module, especially in operating systems where process creation takes a lot of resources. In addition, writing input to files takes some time. Fortunately, the system utilises pre-grounding: WCRL and BCRL files for each configuration model are constructed in advance. For one particular call on *smodels*, one needs to write only a brief list of instances to an input file.

But there exists a better solution to invoking `smodels`. This solution is based on using *smodels* API (Application Programming Interface) directly. Unfortunately, *smodels* API is written in C++, whereas Kumbang Configurator is written in Java. One solution to bridge the gap is to use JNI (Java Native Interface). JNI enables calling native C++ programs from Java programs. With JNI, one could have created a `Smodels` instance in C++ and called its methods directly. This would clearly have been a better solution, but it would have taken too much resources. It would have required setting up a C++ development environment with editors and compilers, getting to know JNI techniques, writing some C++ code on top of *smodels* API and integrating this all with the system using JNI. Thus it was better to implement a *brute force* solution that used

process creation. This solution required only minimal effort.

There are three unimplemented configuration requirements, out of which one is conditional and two are optional requirements. First unimplemented requirement states that user should be able to manually correct an inconsistent situation. Although this requirement is moderately important (conditional requirement), it was added to the requirement list only at the end of the development. Thus there wasn't enough time to implement it. Second unimplemented requirement states that the system should identify inconsistent constraints. Unfortunately, this requirement cannot be implemented using *smodels* as an external module, since it doesn't give much feedback about inconsistent situations. Third unimplemented requirement states that the system must tell a reason for inconsistent situations in general. This was known to be very hard to implement in the first place, so it was not a surprise that it was not implemented.

8.1.3 Evaluation of User Interface Requirements

User interface requirements of the system (see Table 6.3) cover issues that are related to the graphical user interface. These were pretty well implemented: only two requirements out of ten were not implemented, and both unimplemented requirements were optional.

User interface requirements stated that the system must show the compositional structure of the configuration in two separate representations: a tree and a diagram. This is in a line with Geyer and Becker (2002): they emphasize that the compositional hierarchy must be central in the presentation. They claim that this eases navigation and enhances understanding of the model.

At the moment, the graphical user interface makes a separation between features and components. The tree area has separate tabs for features and components, and the diagram area does the same. Another possibility would have been to put features and components in one tree or one diagram. But if configurations are large, it soon becomes difficult to use and understand such combined structures. Besides, another advantage of the separation is that the user can select which trees and which diagrams to show independently of each other. (In a typical scenario, the user might want to see feature tree and component diagram side by side.) If one took the current solution one step further, the best solution would provide full configurability of the user

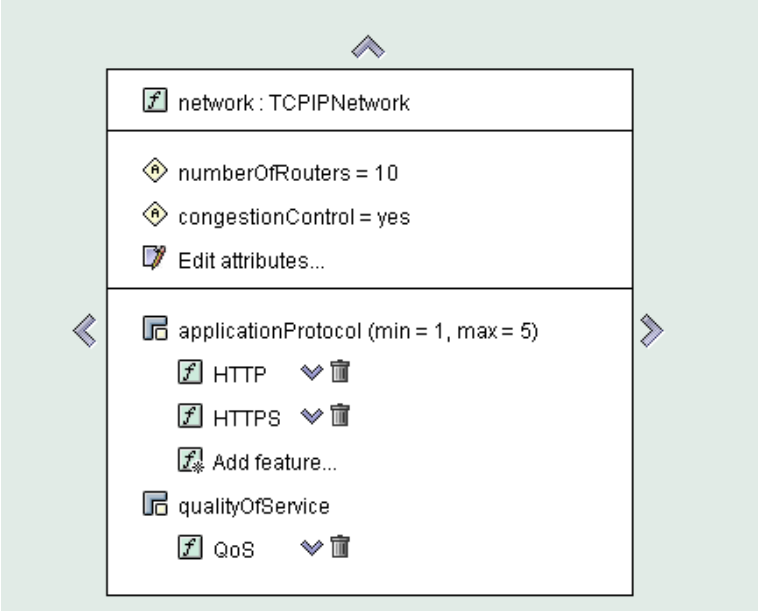

Figure 8.1: A draft that shows how feature diagrams could be replaced in the future.

interface, similar to some IDE tools (Integrated Development Environment). In such a system, the user could choose desired UI elements, and the tool would hide the rest. A configurable user interface could be implemented using internal frames, for example.

One of the requirements states that the graphical notation of the tool should resemble existing notations. The component configuration mimics Koala as much as possible, but it was hard to come up with a proper graphical notation for the feature configuration. The feature tree is quite intuitive for those who know feature models, but the feature diagram is certainly a bit strange.

In fact, it can be argued that the whole feature diagram is unnecessary as such. Instead, one could utilise a concept similar to WeCoTin (see Section 3.3.1). Instead of a feature diagram, one could show detailed information about one particular feature. An example draft of such detailed feature diagram is in Figure 8.1. This detailed information could show attributes and subfeatures, and offer a possibility to edit both. In addition, this diagram could offer navigation into three directions: to parent feature, to child features and to sibling features. This detailed feature diagram could quite easily be integrated with configuration process support - navigation happens along process steps, not along feature hierarchy (see Section 8.1.6).

One implementation issue was how much the tool should concentrate on attributes.

For example, the user interface of WeCoTin (see Section 3.3.1) emphasises attributes in the components. In contrast, attributes do not play a major role in the user interface of this system. Components and features are mainly described using their types and compositional relations. There exists industrial experience to support this choice (Hotz *et al.*, 2004). At their industrial partners, it was noticed that only a small number of parameters are used for modelling features and software components. This is a difference between software and hardware domains. However, it can be argued that the user interface should provide better support for the visualisation of attributes. The proposal in Figure 8.1 could partially improve the situation.

As discussed, there are two unimplemented requirements, and both are optional requirements. The first unimplemented requirement states that the system should show a type taxonomy. But since the user of the tool should be able to use the tool without knowing anything about the configuration model, it can be argued that it is not necessary to provide a type taxonomy.

The second unimplemented requirement states that there should be separate representations for graphical elements. The actual implementation is probably rather easy: each instance in the configuration has a separate graphical element, which draws itself independently on the screen. Thus it is easy to add new representations for graphical elements. The biggest problem is to find a proper notation that is compatible with the underlying concepts. A potential candidate is UML 2.0 (Unified Modelling Language version 2.0). Since UML is a well-known notation for software architectures, it would be beneficial if the tool had UML-compliant notations. It seems that many concepts in Koala have similar counterparts in UML 2.0. For example, UML 2.0 shows ports for components with joint lollipop connections, while Koala uses triangles.

8.1.4 Evaluation of Quality Attributes

This system had three primary goals for quality attributes (see Table 6.5). Although it is hard to measure exactly whether these goals have been met or not, it can discussed how the system reflects these goals. In the following, usability, modifiability and performance of the system are discussed.

Usability

One of the quality attribute goals (Table 6.5) states that the usability of the system should be good enough for demonstration purposes. There are several aspects of the system that work towards this goal.

Firstly, there are usually multiple ways of doing things. For example, user can add new components using either component tree, component diagram, information panel or menu bar. This way one can choose the usage mode that fits one's preferences.

Secondly, the graphical user interface tries to guide the user with textual information. For example, many items have informative tool tip texts that are shown when the user points the item with a mouse. Further, dialogs usually contain a couple of informative sentences that describe the purpose of the dialog.

Thirdly, besides textual information, the graphical user interface uses icons and logos in a consistent manner. For example, each instance has its own icon, and they are used throughout the user interface - in menus, buttons, trees and so forth.

Of course, there are a lot of areas that could be enhanced. Most notably, support for configuration process (see Section 8.1.6) would improve usability. Further, the tool does not provide any kind of help. It would be beneficial to offer at least context sensitive help and help buttons in dialogs. Further, the tool does not provide key shortcuts. Many users prefer keyboard to mouse, which means that each operation and menu item should have its own key shortcut. Finally, the configurability of the user interface (see Section 8.1.3) would definitely enhance usability of the tool.

Modifiability

One of the quality attribute goals (Table 6.5) states that the system should be as extensible and modifiable as possible in order to support further development of the system.

In general, the system supports modifiability by utilising modularisation. The aim was to keep modules coherent and to minimise coupling between modules. To support this aim, the system has been structured into hierarchical modules, and dependencies between these modules have been kept at minimum. For example, all classes that are related to Kumbang language are separated into one module, which is further divided into two submodules for configuration-related and model-related concepts. Further, the system utilises layering style, which is said to support modifiability (Bachmann

et al., 2000). For further discussion about system modules and their dependencies, see Section 6.3.

In addition to modularisation, one has tried to utilise generic structures and supporting elements. For example, the client provides a hierarchy of tasks that are used for modifying the configuration. Thus all information related to a modification can be bundled into one task, which can be passed as a parameter to different handlers of the system.

One requirement stated that the system should provide different representations for the configuration. Although this requirement was not fulfilled, it has been taken into account in the implementation. The system makes a strict separation between data elements (instances and types) and graphical elements. All graphical elements are built as separate class hierarchies, and data elements are independent of the graphical elements. Further, graphical elements draw themselves independently on the screen. In order to change the layout for one element, one only needs to modify its drawing methods.

Performance

One of the quality attribute goals (Table 6.5) states that system performance should be good enough for demonstration purposes.

There are some implementation decisions that enhanced the performance of the tool. One decision was to use *smodels* inference engine instead of implementing a configuration engine from scratch. Another decision concerned translation from WCRL to BCRL (or grounding), which is potentially a costly operation. Grounding is done beforehand, and the grounded BCRL program is stored for future use. When *smodels* is invoked, the only thing that needs to be added to the grounded BCRL program is the current configuration.

Further, the configuration reasoning hides all unnecessary statements from WCRL programs and from output. Thus only those statements that are relevant are parsed from *smodels* output.

But there are some decisions that also degrade the performance of the system. One of such decisions was using *smodels* and *lparse* as external modules, which brought the need for process creation and reading and writing big files. These operations are both quite ineffective compared to direct procedure calls. Given the current situation,

it seems that this is one of the weak points in the system performance. If one integrated *smodels* directly through application programming interface, one would probably enhance the system performance a lot.

Another decision that degrades the performance of the system is utilising total configurations. At the moment, the tool calculates the total configuration (all possible instances that can exist in one configuration) and keeps it in the main memory during the configuration task. Total configuration is essential for finding topologically correct bindings, and it eases the configuration task remarkably in other aspects also. But total configurations grow exponentially in size, which potentially means huge memory consumption for large configurations. (For example, the total configuration of the model in Figure 4.3 takes a little less than 10 kB as a serialised object. This is quite a lot when one considers the size of the model and the compressing effect of serialisation.) In order to enhance the situation, one could store the total configuration on a disk or on a database, and fetch instances only when needed. Further, the system could utilise some kind of cache for fetched instances.

There exists one technique that could improve the system performance: symmetry breaking (Tiihonen *et al.*, 2002). Symmetry breaking means that one removes redundant configurations by telling explicitly in which order instances can reside in the configuration. At the moment, the system does not implement symmetry breaking, but this issue is worth investigating.

8.1.5 Evaluation of Cases and Validation

As stated in the research method, the system developed was also validated through example cases. Chapter 7 presented two example cases and showed how they were modelled and configured with the system. Chapter 7 also discussed briefly the issues that were brought up during the configuration task.

If one compares the two cases with each other, it is clear that the first case (Section 7.1) is more representative. Firstly, it is a real-world case of a real configurable product. Secondly, it is considerably larger, which highlights the need for automated configuration reasoning. The weakness of the first case is that it does not include connections. Thus it is yet to be seen how real-world cases use connections and whether it corresponds to the approach taken in this system. Further, it is still unclear how

connections affect the performance of large-scale models.

However, it is clear that these two example cases are not nearly enough for empirical validation, especially if one wants to validate whether the system really provides value to application engineering activities. In order to provide better empirical validation, one should gather several real-world cases with different properties. These cases should be modelled and configured in cooperation with industrial partners in order to gather feedback and comments.

Further, the performance of the system should be tested more thoroughly. For example, the randomised performance tests conducted by Tiihonen *et al.* (2002) could provide one way of measuring the performance.

8.1.6 Evaluation of Other Interesting Aspects

This section covers some aspects that are not addressed by the requirements but have future relevance or are interesting as such. In general, the implementation of the functionality discussed below would require modifying and extending the underlying modelling concepts.

Inheritance of Components

As was mentioned in Section 4.2.2, Koalish does not support inheritance of components. (In contrast, Forfamel allows features to extend other features.) The reasoning behind this decision is conformity with Koala: since Koala does not offer inheritance of components, Koalish does neither. But it can be argued that inheritance can neatly group together similar concepts. Figure 8.2 shows two examples that could utilise inheritance of components.

The problem that must be solved when applying inheritance is how to treat contained elements inside an inherited component. That is, should a subcomponent inherit all interfaces, attributes, part components and constraints from the parent component? Since inheritance should be an IsA-relation, the answer must be yes: all characteristics of the parent component must be present in the subcomponent. The first example in Figure 8.2 shows how component *C2* inherits interface *interf1* from parent component *C1* and specifies additional interface *interf2*.

But inheritance of interfaces should be taken into account. The second example

```
Example 1: without inheritance          Example 1: with inheritance

component C1 {                          component C1 {
  provides I1 interf1;                    provides I1 interf1;
}                                       }
component C2 {                          component C2 extends C1 {
  provides I1 interf1;                    provides I1 interf1;
  provides I2 interf2;                  }
}
```

```
Example 2: without inheritance          Example 2: with inheritance

component CServer {                     component CServer {
  provides IRpc callee;                   provides IRpc callee;
  contains CDb dbase;                     contains CDb dbase;
  connects callee = dbase.query;          connects callee = dbase.query;
}                                       }
component CServer2 {                    component CServer2 extends CServer {
  provides IRpc2 callee;                  provides IRpc2 callee;
  contains CDb dbase;                   }
  connects callee = dbase.query;
}
```

Figure 8.2: Two examples that show how Koalish could utilise inheritance of components.

in Figure 8.2 is an excerpt from Figure 4.3, and it shows two components, *CServer* and *CServer2*. These components are otherwise exactly similar, except that *CServer* contains interface *callee:IRpc* and *CServer2* contains interface *callee:IRpc2*. Since interface *IRpc2* contains all functions in interface *IRpc* ($IRpc \subset IRpc2$), these two components are in an IsA-relation. But how could component *CServer2* be extended from component *CServer*? There are two choices. First alternative is to add a new interface type $IRpc'$ so that $IRpc2 = IRpc \cup IRpc'$ and $IRpc \cap IRpc' = \emptyset$, and then follow the scenario presented in the first example in Figure 8.2. But it is much easier to allow direct definition shown in Figure 8.2: component *CServer2* defines an interface with the same name but with extended interface type.

Sharable Components

Hotz *et al.* (2004) present experience that was gained when applying software configurators in an industrial context. One issue that was brought up was the need for sharable parts. Sharable parts, such as libraries, are used in many software applications. (Hotz *et al.*, 2004)

At the moment, the tool supports sharing only at the same level of compositional

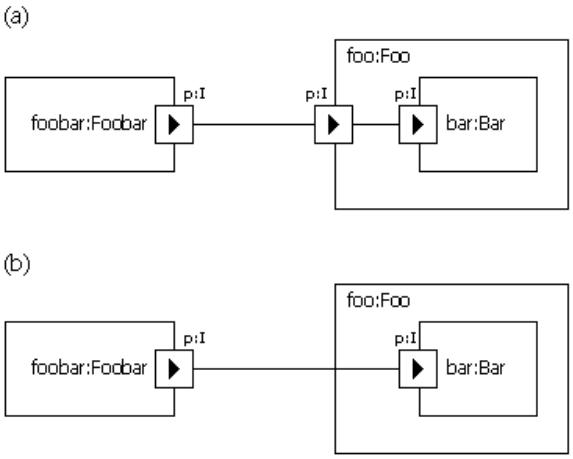

Figure 8.3: An example that shows how one could bind interfaces across component levels. Figure (a) shows the current situation, figure (b) shows how one could relax the rules of binding.

hierarchy. If there is a need to globally share a component, it can happen through type-instance mechanism. This means that one can use a certain type in several places in the configuration. But it can be argued that this isn't real sharing, since instances are in any case separate and thus can have different states (for example, different attribute values). In some special cases one can represent sharable libraries with separate instances of the same type, but in general this solution is not enough.

In order to support sharing in general, one should be able to choose already existing instances as parts, not just to create new instances. In addition, one could extend Koalish to include declaration of static components. A static component type couldn't be instantiated, but other component instances could access static component type directly.

Binding Interfaces across Component Levels

In some cases, one might need to relax the rules of connecting interfaces with each other. At the moment, interfaces can only be connected with interfaces on the same compositional level, or between interfaces whose components are parts of each other (see Figure 8.3a). This requires that the family architecture ensures strict encapsulation—components cannot see or access inner structure of other components. But in many real

world situations, one might need to relax this encapsulation a bit. This would require that one could connect interfaces to interfaces that are contained in other components (see Figure 8.3b).

It can be argued that this extension is a bit questionable, since it violates the rules of encapsulation. Further, the situation can be circumvented by defining a separate interface at the border that acts as a bridging interface (see Figure 8.3a). This corresponds to a situation in which new public interfaces are added to a component.

Evolution

It is said that changeability is one of the characteristics of software. It is no wonder that the need and challenge of evolution in configurable software product families has been addressed in many contexts (see e.g. Männistö *et al.*, 2001a; Krebs *et al.*, 2003; Geyer and Becker, 2002). In order to support evolution, configuration techniques and tools should explicitly take evolution into account.

Unfortunately, neither this tool nor the conceptual basis beneath it support evolution. At the moment, each version of the configuration model is treated as a separate model. An enhancement to this approach could add a version number to the configuration model itself. But this doesn't solve the problem, since components can be evolved more or less independently of each other. Thus one should add a separate version number to elements in the model. Further, one should be able to refer to these revisions in the configuration model, and even write constraints that refer to version numbers.

But which elements in the model should be attached with version numbers? Since software components evolve when their implementation is modified, it is natural to assign version numbers to components. Further, requirement specifications may also change, which means that features should also have version numbers. But it can be argued that other elements might also need version numbers. For example, one can modify the function specifications in an interface type, or modify the value set of one particular attribute type. To conclude, when specifying the versioning system that supports evolution, one should carefully decide which elements in the configuration model should be taken into account.

Support for Configuration Process

According to Geyer and Becker (2002), a configurator tool should support the user during the configuration process, so that features can be selected in an optimal order. This is also reflected by Hotz *et al.* (2004), who use procedural knowledge for describing the order in which configuration decisions are processed. Further, the order of the configuration is also supported in WeCoTin configurator (see Section 3.3.1).

At the moment, Kumbang Configurator does not provide any kind of support for configuration process. Instead, the user can modify the configuration in any order he or she chooses, as long as the configuration is instantiated in a top-down order. This means that upper parts must be instantiated before one can start to instantiate parts below them. But once the upper parts are instantiated, one can modify them in any order one chooses.

There are several advantages in providing support for configuration process. Firstly, some dependencies might be easier to handle when the configuration order is specified. For example, it might be easier to avoid dead-ends and conflicting situations, if one makes some selections first and lets the tool deduce consequences of those selections. Secondly, configuration process support acts as a "wizard" for deriving a configuration, which helps the users to understand the configuration better. Finally, one can attach detailed information and explanations with the procedural knowledge.

But it can be argued that some experienced users don't want to follow a fixed order of configuration decisions, just like some users don't want to use wizards in other tools. This means that one should be able to use the tool both ways: with and without configuration process support.

8.2 Related Work

This section discusses and compares related work to the work done in this thesis. Thus the purpose of this section is to find differences as well as similarities with approaches that try to solve the same problems.

But what should be taken as related work? The scope of this thesis does not enable comparing this work to all approaches and issues presented in the literature survey. The idea is to find work that is similar enough so that comparison is meaningful. Thus

this section assumes that related work covers product derivation as part of application engineering. And not just any kind of product derivation, but product derivation that has been automatised or is aided by dedicated tools and is performed in a routine manner. In addition, one limits the scope of this section to software product families. Comparing to traditional configurable products or services would naturally be fruitful, but it is unfortunately too vast a subject for this section. In summary, the scope of this section is *tool-aided routine product derivation for software product families*.

The rest of the section is organised as follows. Section 8.2.1 discusses how traditional product configurator tools have been used for configuring software. These cases include situations where configurator tool was used directly, as well as situations where new software has been built on top of existing configurator tools. Section 8.2.2 presents an approach that combines software configuration management (SCM) ideas with software architectures. Section 8.2.3 discusses generative programming, and finally Section 8.2.4 presents a couple of other approaches.

8.2.1 Using Traditional Configurators for Configuring Software

As was discussed in Section 4.1.2, one has tried to apply traditional product configuration tools to configure software product families. These can be categorised by the effort required to model software so that it can be given to the configurator tool. In some cases, the model can be given almost directly to the tool, while in some cases one needs to translate the original description of the configurable software product into some other format. In some cases, the configurators cannot directly support all characteristics of the software. Instead, one has built new software on top of existing configurators to enable mapping the software description to a format understood by the tool.

Configuring Linux Familiar

The question of whether software products can be modelled and configured with traditional configurators is investigated by configuring Linux Familiar with WeCoTin configurator (Ylinen *et al.*, 2002).

Linux Familiar is a distribution of the Linux operating system developed for Compaq iPAQ hand-held computers. Linux Familiar consists of a large number of software

components, called packages. Each package can have multiple versions, which in this case are revisions in time. Packages can have various relations between them: a package can depend on, conflict with, replace, or provide another package. When installing the software, these relations must be taken into account. Further, an interesting characteristic of software running in a hand-held device is the limited amount of resources available. One cannot install all possible packages into the device; instead, one must prioritise them. (Ylinen *et al.*, 2002)

In order to solve the problem of configuring a correct set of packages, Ylinen *et al.* (2002) provide a mapping to the modelling concepts used in WeCoTin configurator tool (see Section 3.3.1). In particular, they provide an automatic mapping from Linux Familiar package description to PCML (Product Configuration Modelling Language). As a result of this mapping, they conclude that PCML is largely suitable for modelling this kind of configurable product.

In order to support evolution and reconfiguration of Linux Familiar, Kojo *et al.* (2003) propose a conceptualisation for modelling evolution and variability of configurable software product families. In addition, they present a prototype built on top of WeCoTin configurator that supports the conceptualisation presented. Finally, they use this prototype for modelling and configuring Linux Familiar over multiple releases.

When comparing these two cases of configuring Linux Familiar, one can note several similarities with Kumbang Configurator. It seems that one could indeed model and configure Linux Familiar using the tool developed in this work. This is due to the fact that this tool resembles WeCoTin configurator in many respects. But there are several differences also.

Firstly, there are some concepts that are provided by Kumbang Configurator but are not actually needed in case of Linux Familiar. (Fortunately these issues do not prevent modelling and configuring Linux Familiar with the tool developed in this work. One should just leave out those concepts that aren't needed.) Ylinen *et al.* (2002) question the usefulness of cardinality in software configuration. In case of Linux Familiar, there were only optional and required packages, which means that the notion of cardinality could be replaced with the notion of optionality. Another obvious difference is the lack of interfaces and connectors. It is the responsibility of the packages themselves to know how they use each other, and thus package connections cannot be configured separately.

Secondly, there are some concepts that are needed in case of Linux Familiar but are not provided by Kumbang Configurator. One obvious difference is the support for evolution. Kojo *et al.* (2003) provide support for evolution of packages, whereas this tool lacks that support completely. Another difference is the need for resources. Ylinen *et al.* (2002) note that resource-based configuration would be beneficial for balancing the memory consumption of the configuration. But resources are not provided by WeCoTin either, so they are not essential for the configuration task.

Finally, there is even one issue that is provided by this tool but is lacking from WeCoTin: separation of features and components. Ylinen *et al.* (2002) point out that virtual Linux packages would more naturally be modelled as functions or features, not as ordinary packages. With Kumbang Configurator, virtual packages could be modelled as features, while other packages could be modelled as components.

In summary, it seems that it would be beneficial to try to do similar mapping from Linux Familiar to this tool. This would also give valuable empirical data on how this tool handles very large configurations.

Configuring Feature Models

As was discussed in Section 2.2.1, feature models have become a prominent method for describing software product families. However, no generally accepted method or tool for deriving feature configurations exists. Asikainen *et al.* (2004) investigate whether traditional configurators could be used for deriving valid feature configurations from feature models. In particular, they take WeCoTin configurator (see Section 3.3.1) and use it for modelling a sample feature model and for deriving feature combinations from the model.

The central observation of the demonstration is that it is indeed feasible to use WeCoTin for modelling and deployment of feature models. The translation to PCML (Product Configuration Modelling Language) used by WeCoTin was quite straightforward, and the configuration task went smoothly. However, WeCoTin is not perfectly suitable for this purpose. WeCoTin makes a distinction between feature types and their usage in the hierarchy, whereas most feature modelling methods make no such distinction. One manifestation of this mismatch is that compositional hierarchy in WeCoTin configurator has separate part names for features as parts, whereas in feature models there is no conceptual difference between a role of subfeature and a subfeature filling

that role. But it is noted that this difference can also be an advantage. In some cases, it might be beneficial to be able to distinguish a feature type or a role of subfeature as separate concepts. (Asikainen *et al.*, 2004)

If one compares the demonstration conducted by Asikainen *et al.* (2004) to this work, one sees that the feature modelling concepts of Kumbang are very similar to those found in WeCoTin. That is, Kumbang Configurator makes a separation between types and instances as well as roles and subfeatures. But it is true that in some cases this might be unnecessary. Especially when there is only one subfeature in a role, one should be able to define the subfeature directly under that role, without separate type definitions and role names. (This is somewhat similar to the concept of anonymous inner class found in Java programming language.) This way, one could avoid useless repetition in both configuration models and in the configurator tool.

Configuring Embedded Automotive Systems

There exists a pool of research that tries to integrate knowlegde-based configuration techniques with software product families of embedded automotive systems (see e.g. Hein *et al.*, 2001; Hein and MacGregor, 2003). In the context of embedded automotive systems, hundreds or event thousands variants are produced every year. This makes every effort in automating product derivation worthwhile. (Hein *et al.*, 2001)

As a result of the research, one has developed tool support to ease the configuration task in such systems (see e.g. Hotz *et al.*, 2004; Hotz and Krebs, 2003b). (However, the developed tool support is independent of the domain.) This tool support utilises several existing structure-based configurator tools, such as KONWERK (Günter and Hotz, 1999) and EngCon (Hollmann *et al.*, 2000). (Figure 8.4 contains a screen shot from a case that utilises KONWERK configurator tool.) The main characteristic of this tool support is the separation between features and artifacts. Further, ther are both hardware or software artifacts. The concepts of the software domain are then mapped into structural language understood by the configurator tool.

The case that is presented in Section 7.1 originates from the research conducted for embedded automotive systems, and it is meant to be used with the tool support (see a description of the case in Hotz *et al.*, 2004). One can almost directly configure the case presented in Section 7.1 with both tools. This is a strong indicator of the similarity between Kumbang Configurator and the tool discussed by Hotz *et al.* (2004).

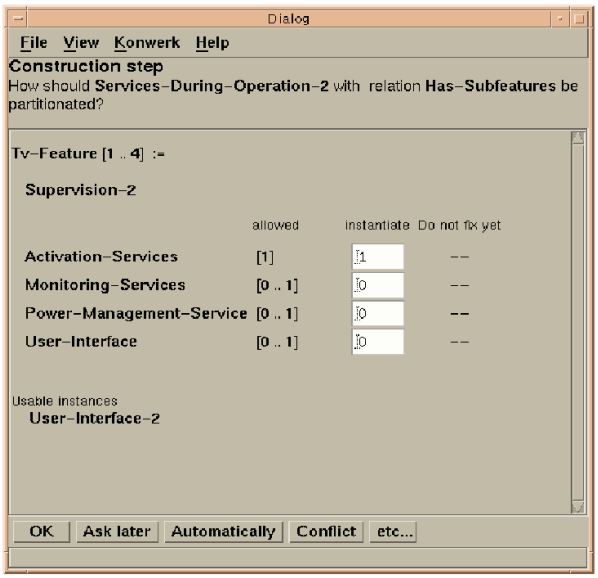

Figure 8.4: A screen shot from the configurator KONWERK (Hotz *et al.*, 2004)

For example, both tools support compositional hierarchies of elements, cardinalities, attributes, separate features and components (artifacts), constraints between elements, and so forth.

In addition to many similarities, there are also differences between these tools. Firstly, the tool discussed by Hotz *et al.* (2004) supports the configuration task by offering a mechanism for guiding the user through configuration selections. In contrast, Kumbang Configurator does not provide any kind of guidance, but the user can freely make selections in any order. It can be argued that support in the configuration task is beneficial for the user, especially in large configurations. But it should also be possible to perform the steps in any order.

Secondly, the tool discussed by Hotz *et al.* (2004) does not support interfaces or connectors, whereas this tool offers both. This might partly be because the underlying configurator tools do not support connecting elements.

Thirdly, the tool discussed by Hotz *et al.* (2004) allows configuration only through features, and artifacts are automatically deduced based on feature selections. In contrast, the tool developed in this work allows the user to make configuration selections on components also, and reflect those selections to the feature configuration. It is probable that the user usually wants to select features first and get component configuration automatically. However, in some cases it might be beneficial to offer the possibility to

do vice versa.

Finally, it is a bit unclear whether the tool discussed by Hotz *et al.* (2004) supports evolution. At least a conceptualisation of the evolution has been studied (Krebs *et al.*, 2003), but to one's knowledge, these ideas have not yet been implemented in the tool support.

Concluding Remarks

In all the cases presented in this section, there is one clear difference to the approach taken in this thesis. None of the demonstrations or tools support connecting components through interfaces. Since none of the underlying configurator tools support connectors, it seems that it is quite difficult to build support for connectors on top of these tools. Connecting components implies topological relations that are quite difficult to model using other relations, such as composition or inheritance, available in the tools.

8.2.2 Mae

Mae (Roshandel *et al.*, 2004; van der Hoek, 2004) is a supporting environment for managing architectural variability and evolution. The background of Mae lies in the principles of software configuration management (SCM); such systems have long been used to provide support with managing configurations. But since configuration management systems tend to concentrate on source code and file management, they aren't really suitable for managing architectures. On the other hand, architectural description languages (ADLs) lack explicit mechanisms for optionality and variability. The goal of Mae is to solve these issues by providing an SCM system for managing configurable architectures written in ADLs that explicitly model variability and optionality. In addition, Mae enables evolution of architectures by providing a versioning system for the elements in the architecture. (Roshandel *et al.*, 2004; van der Hoek, 2004)

Mae provides a system model that tries to combine software architecture and configuration management concepts into a single representation. The system model is built around a notional architecture model, which captures elements commonly found in most existing ADLs. Thus Mae can be used in conjunction with many ADLs. The system model is implemented as an extension to xADL 2.0, and it is built a a set of

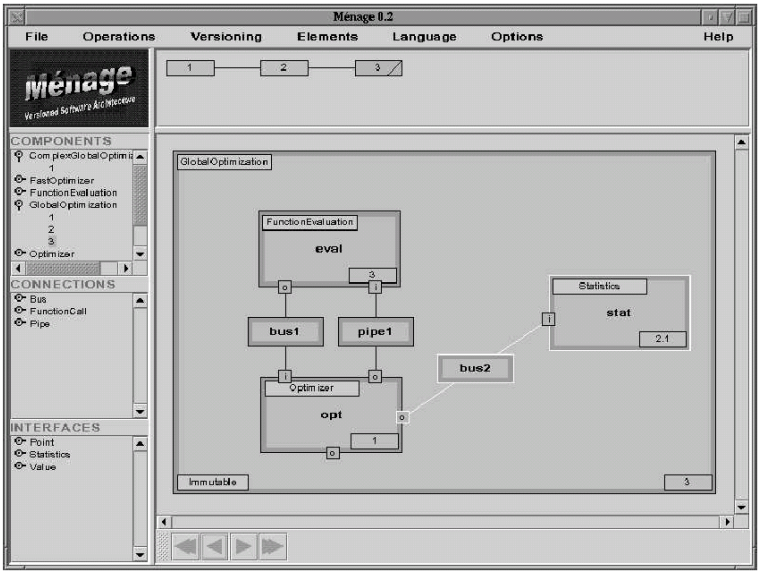

Figure 8.5: A screen shot from the Ménage modelling tool (van der Hoek *et al.*, 1999)

XML (eXtensible Markup Language) schemas. (Roshandel *et al.*, 2004; van der Hoek, 2004)

Mae contains three subsystems that perform separate tasks in the management of evolvable and configurable arhictectures. Firstly, a *design* subsystem called Ménage (van der Hoek *et al.*, 1999) provides a graphical user interface for designing the architectures. Secondly, a *selector* subsystem is used for specifying a certain architecture configuration out of the available version space. Finally, an *analysis* subsystem performs consistency checks, either on a selected configuration or on an architecture designed with Ménage. (Roshandel *et al.*, 2004)

Although Mae originates from a completely different background (software configuration management compared to traditional product configuration), it bears many similarities with Kumbang Configurator. Firstly, it provides a tool for modelling variability in architectures, deriving specific architectures from those models and checking the consistency of the configurations. Secondly, the architectural concepts used in Mae are quite similar to the ones used in this work. This is partly because both Kumbang Configurator and Mae use concepts that are commonly found in many ADLs, such as components, interfaces, connectors and constraints. And like many ADLs, both systems make a separation between types and instances.

But there are also differences between these tools. Firstly, Mae provides a lot of

functionality that is lacking from this tool: check-in/check-out mechanism, evolution and versioning for architecture, possibility to use different languages for describing architectures and so forth. Further, Ménage can be used to model architectures, while this system is currently lacking modelling support.

Secondly, there are some issues that have been implemented somewhat differently. Perhaps the biggest difference is the definition of variability and optionality. Variability of components is achieved in Mae by using *variant components*. Variant components are similar to traditional components, except that they offer a set of other components out of which one component can be chosen. Further, variability mechanism is complemented with optional elements. In contrast, this system uses cardinalities and sets of possible types together in part definitions. It can be argued that variability mechanisms used in this tool are more powerful in the sense that they can express more in simple definitions. For example, one needs several variant components combined with optional components to express the part definition *(BasicClient,ExtendedClient) client[2-3]* in Figure 4.3.

Another big difference is how selections are made. Mae attaches each optional and variant element with a property guard - a property must have a certain value in order to include a certain optional element, for example. When deriving a configuration from the architecture, the user sets these property values globally, and they are propagated down the architecture to make corresponding selections. In contrast, the selection mechanism in this system is explicit: one makes the selections by choosing instances directly, not through property values.

Yet another difference is the mechanism for consistency checking. In Mae, the consistency of a configuration is checked after the configuration has been derived - this is called *after-the-fact* checking. In contrast, this system checks the consistency of the configuration after every selection and offers the possibility to cancel inconsistent selections. It can be argued that this *after-the-fact* checking might sometimes be annoying to the user, especially if one has to iterate over and over again to find a correct configuration.

Finally, there are a couple of things that this tool supports but that are lacking from Mae. First and foremost, the tool developed for this system makes a separation between features and components, while Mae addresses only architectural models and configurations. This might partly be because this work has its background in software

product family research, in which feature models are seen as an integral part of the software product family development. Secondly, it seems that Mae does not support attributes of any kind. Mae offers properties for elements, but they are used for binding variability, not for modelling the domain.

8.2.3 Generative Programming and Domain-Specific Languages

Generative Programming is defined as "a software engineering paradigm based on modelling software system families such that, given a particular requirements specification, a highly customised and optimised intermediate or end-product can be automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge" (Czarnecki and Eisenecker, 2000).

The idea of generative programming is to mass-customise products from a software product family. Generative programming is a direct consequence of the "automation assumption": if one can compose components manually, one can also automate that process. Generative programming aims at building generative models for software product families; these models are then used to generate concrete software products from these models. This derivation (or generation) reuses existing implementation components and configuration knowledge. (Czarnecki and Eisenecker, 2000)

Generative programming makes a distinction between domain engineering and application engineering activities. Domain engineering includes modelling the family with feature models and generative domain models and building generators and reusable implementation components. Application engineering uses these assets for generating desired product instances from the family. (Czarnecki and Eisenecker, 2000)

In many respects, generative programming resembles the approach taken in this study. For example, generative programming utilises feature models for describing the family in a similar fashion. In fact, the feature modelling concepts that are presented by Czarnecki and Eisenecker (2000); Czarnecki *et al.* (2002, 2004) are quite similar to the concepts utilised in this work. Examples of similar concepts are attributes, cardinalities, constraints and so forth. But there are also differences: Czarnecki and Eisenecker (2000) utilise many FODA-like concepts, such as characterising features as either optional or mandatory instead of representing the same information with cardi-

nalities. Further, Czarnecki and Eisenecker (2000) do not make a distinction between types and instances.

Perhaps the biggest difference between generative programming and this work lies in the application engineering. Firstly, generative programming generates components instead of selecting them to the product instance. Although generation might utilise existing components, they are typically much smaller than those components that are selected in product configuration.

Secondly, generative programming does not make a clear distinction between selecting the configuration and building the product. Instead, these tasks are often intertwined into one generative model. In contrast, this tool makes a separation between these activities; in fact this tool does not build the product at all. Although the user often wants to perform configuration task and build task together, it is better to keep them separate conceptually and implementation-wise. One reason for this is maintainability: it is much easier to modify either one of these tasks, when they are cleanly separated.

Thirdly, this tool is domain-independent, since the domain knowledge is encoded in the configuration model, not in the tool implementation. This means one can utilise the same tool in many domains. In contrast, generative programming often utilises domain-specific languages (DSL), which are used for generating the system implementation. The problem with domain-specific implementation and generators is portability: it is hard to reuse same generators in many domains, and it is harder to make big changes to the domain. There exist some tools that utilise domain-specific languages for generating software systems. An example system, MetaEdit, is presented by Arion and Tolvanen (2004).

8.2.4 Other Related Work

CONSUL and pure::variants

CONSUL (Beuche *et al.*, 2004) and its commercial successor pure::variants (Beuche, 2004) are tools that support the product family development chain from features to final binary product. These tools use feature models as their main model for describing variability. In addition, the tools provide mappings from features to components and from components to actual implementation. The chain of deriving an executable prod-

uct is highly customisable, and the system can be used with different implementation languages. (Beuche *et al.*, 2004)

The concepts related to feature models are quite similar to the ones used in for example FODA (see Section 2.2.1. That is, the system does not make a difference between types and instances, and variability concepts (alternatives, optionals etc.) are incorporated in the features themselves. In these respects, the system developed in this work differs from CONSUL system. In addition to basic concepts, CONSUL provides feature values and restrictions, which pretty much correspond to attributes and constraints provided by this system.

The component side of CONSUL is quite close to implementation. That is, components in CONSUL are defined as a set of parts, which are mapped to one or more sources. Sources in turn are physical representations to logical elements, such as files or flags. In contrast, this system treats components as high-level architectural abstractions that can contain other components, attributes, interfaces and bindings between interfaces. In CONSUL, it seems that components are merely a group of sources with some additional restrictions.

CONSUL supports consistency checks, which are implemented using Prolog logical programming language. Based on the papers published, it is though a bit unclear how and when these checks are performed.

RequiLine

RequiLine (von der Maßen and Lichter, 2003) is a requirements engineering tool for software product families that utilises feature models for describing requirements. The user can specify and manage features and requirements, and perform queries on the specified model (for example, by filtering certain features from the model). In addition, the system performs consistency checks on the specified model. (von der Maßen and Lichter, 2003)

At the moment, RequiLine does not provide support for instantiating products through configuring, but this functionality is under construction. The aim is that RequiLine could provide assistance in resolving the variabilities of the model and building consistent configurations. (von der Maßen and Lichter, 2003)

The concepts utilised in RequiLine are quite close to the concepts used in this system, but there are differences also. Just like this system, RequiLine makes a sep-

aration between features and their optinality or variability, but the mechanism is a bit different. For example, RequiLine does not provide cardinalities, but specifies separate relationships for mandatory, optional, alternative and or features. Further, features cannot contain attributes. Finally, it seems that RequiLine does not make a separation between types and instances, like this system does.

8.3 Comparison with Research Questions and Objectives

This section discusses the research aims presented in Chapter 5 and compares them to the research results presented in Chapter 6 and Chapter 7.

To iterate, the research questions set for this work were:

1. Is it possible to build a configurator tool that can be used for configuring product individuals from a configurable software product family described with Koalish, Forfamel or Kumbang language, so that the implementation utilises existing inference engine *smodels*?
2. If it is possible to build the aforementioned configurator tool, does the tool support the user in the configuration task and provides service that outweigh manual configuration?
3. Given the developed tool presented above, can the feasibility of the approach be validated with example cases?

The answer to the first research question was found. The system can be used for configuring product individuals based on configuration models written in Kumbang, Koalish and Forfamel, and the implementation utilises *smodels* inference engine for configuration reasoning. Further, the system fulfils all research objectives set for the implementation in Section 5.1. Thus one can conclude that the answer to the first research question is yes: it is indeed possible to build such tool.

The answer to the second research question is not so straightforward. The idea was to develop such solutions that ease the configuration task and provide real value to the user of the tool. There are many aspects of the tool that clearly target for this aim. For example, the graphical user interface visualises the configuration and provides support

for making configuration selections. Further, configuration engine prevents configuration errors and ensures that the configuration is consistent and complete. However, it is clear that this system lacks many capabilities that could support application engineering activities. Many of these capabilities are discussed in Section 8.1; examples include support for guided configuration process, support for evolution, explanation mechanisms, better performance and so forth. Thus one can say that the research done in this thesis only scratches the surface of the second research question. There is still a lot of work to be done in this area.

The third research question sets the guidelines for validating the research. This thesis presents two sample cases that were modelled and configured with the system. Thus one can say that the feasibility of the approach (the first research question) has been validated. However, although the success with the cases give some indication of the usefulness of the system, these cases are not enough to validate the second research question (providing real value and support). In order to validate the second research question, one would need further empirical research. Firstly, one would need to expand the number and quality of example cases to ensure that these two cases were not accidental. Secondly, one needs to verify the system by applying it to industrial usage. This way one can collect valuable data on whether the system really provides value to the user.

In summary, the third research question has been achieved for feasibility, but real value of this system is yet to be proven.

Chapter 9

Conclusions and Future Work

Variability in software products has been an increasing trend. To cope with increasing variability and short time-to-market, a company can organise its product development into a software product family. Individual products in software product families are developed according to a common family architecture and are constructed from common reusable assets. In general, software product families make a distinction between two separate processes: domain engineering, which designs and produces core assets, and application engineering, which uses core assets for deriving individual products.

Since variability in a software product family manifests itself in requirements, architecture and implementation, variability must be managed at all these levels. Feature models are often used for modelling varying requirements; there exists a wealth of different feature modelling approaches. On the other hand, there are only a few architecture modelling methods that can explicitly model variability.

The idea of easing the derivation of individual product by designing the family beforehand has been applied also in the domain of traditional, mechanical products. A configurable product family is such that all product instances are derived in a routine manner, possibly with the aid of a specialised tool called configurator. There exists several configurators; this work adopts many techniques that are similar to WeCoTin configurator.

If one applies the idea of configurable product families to software, one gets a configurable software product family. In such a family, product derivation is extremely effortless; often product derivation is supported with dedicated configurator tools.

This purpose of this work was to build a working prototype tool for configuring

product individuals from configurable software product families. The demonstration tool, called Kumbang Configurator, is based on modelling languages Koalish, Forfamel and Kumbang. These languages utilise concepts from software domain and from traditional product configuration. Further, Kumbang Configurator uses *smodels* inference engine for configuration reasoning.

The requirements of the tool were mainly gathered from literature and comparison with similar tools. After that, the system was implemented to correspond to these requirements.

The implemented tool was validated with two example cases: one real-life case of a configurable product, and one toy example. The validation revealed some minor issues that could be improved, but overall it showed that the tool can indeed be used for deriving such products.

However, as was discussed in Section 8.1, there is still plenty of work to be done. The following paragraphs discuss possible future work and present how the tool could be improved.

Extending the capabilities of the tool in the current scope: There are several issues that could be improved as such, without extending the scope or conceptual basis of the tool. Firstly, one should improve quality attributes of the system. For example, performance could be improved by calling *smodels* API directly, by removing total configurations from main memory, or by breaking symmetries that may exist in configurations (Tiihonen *et al.*, 2002). Secondly, configuration reasoning should be extended, at least by implementing those requirements that are not fulfilled in the current system. Thirdly, it is clear that one should build support for configuration process. This could be implemented as a separate declarative model built on top of Kumbang model.

Extending the conceptual basis of the tool: But there are also many enhancements that require direct modifications of Koalish, Forfamel and Kumbang languages. Perhaps the most critical enhancement is support for evolution. One should design and implement an extension that takes the evolution and versioning of family elements into account. In addition, the constraint language of Kumbang should be extended considerably. For example, an equivalence relation would ease writing constraints between

features and components. Further, it should be investigated how one could support different binding times both in the model and in the tool implementation. Besides these three issues, there are also other, not so critical enhancements. These include support for component inheritance and for resource modelling.

Extending the scope along application engineering activities: The scope of this thesis was intentionally limited to actual configuration task. In future, this scope could be extended from both ends. Firstly, one should provide tool support for constructing the actual product. There are two choices: generation and building. To generate the implementation, one needs generators and generative models for architectural components. If components have already been implemented, one can provide a mapping from components to implementation units and then build the product with dedicated build tools (such as *make* or *ant*).

Secondly, one can also provide support for requirements. Instead of plain feature models, the tool could offer detailed descriptions of features. Further, features could be mapped to actual requirements, which usually reside on separate System Requirement Specification (SRS) documents.

Extending the scope towards domain engineering activities: In addition to extending the scope along application engineering, one could also provide support for domain engineering activities. This essentially means providing a separate modelling tool that can be used for creating valid Kumbang models. At the moment, Kumbang models have to be written by hand; this approach is too difficult and error-prone to be used in practice.

Integration with other tools: It is worth remembering that no tool is an island. In order to provide real value to the daily development, one should integrate this system with other development tools. For example, it should be investigated whether this tool or corresponding modelling tool could be integrated with Eclipse (Eclipse IDE, 2004). Further, software configuration management tools (see e.g. CVS, 2004) can be used for managing implementation units and other core assets. Finally, if one provides support for mapping between features and requirements, one must investigate whether this system could be integrated with requirements engineering tools.

Gaining further empirical knowledge: Previous paragraphs mainly concentrated on enhancing the capabilities of the system. However, if one hasn't empirically verified the system, it is quite useless to put effort in providing lots of new capabilities. Thus it is very important to empirically test the system.

Naturally, the validation approach taken in this thesis should be continued and extended. One should model and configure several new products with different properties and varying sizes. But this validation technique is not enough, since results are not gathered directly from real world situations. Thus the tool should be subjected to industrial scrutiny. One could provide demonstrations and gather comments and feedback from industrial partners, or one could even pioneer tool usage in a small-scale project.

Finally, quality attributes of the system should be empirically tested. For example, one could evaluate performance with similar randomised inputs as was done by Tiihonen *et al.* (2002). In particular, it would be interesting to see how connections affect the performance of the system. It is possible that connections increase the complexity of the configuration task in general. However, this is yet to be proven.

Bibliography

Arion Slava, and Tolvanen Juha-Pekka. 2004. Metaedit+: Domain-Specific Modeling and Code Generator Environment. *In: Proceedings of the Workshop on Software Variability Management for Product Derivation, at Software Product Line Conference (SPLC3)*.

Asikainen T., Soininen T., and Männistö T. 2003a. A Koala-Based Approach for Modelling and Deploying Configurable Software Product Families. *In: Proceedings of the Fifth International Workshop on Product Family Engineering (PFE-5)*.

Asikainen T., Soininen T., and Männistö T. 2003b. A Koala-Based Ontology for Configurable Software Product Families. *In: IJCAI 2003 Configuration workshop*.

Asikainen Timo. 2002. *Representing Software Product Line Architectures Using a Configuration Ontology*. M.Sc.Tech. thesis, Helsinki University of Technology, Department of Industrial Engineering and Management.

Asikainen Timo. 2004. *Modelling Methods for Managing Variability of Configurable Software Product Families*. Licentiate Thesis, Helsinki University of Technology.

Asikainen Timo, Männistö Tomi, and Soininen Timo. 2004. Using a Configurator for Modelling and Configuring Software Product Lines Based on Feature Models. *In: Proceedings of the Workshop on Software Variability Management for Product Derivation, at Software Product Line Conference (SPLC3)*.

Bachmann Felix, and Bass Len. 2001. Managing Variability in Software Architectures. *In: Proceedings of the Symposium on Software Reusability*.

Bachmann Felix, Bass Len, Carriere Jeromy, Clements Paul, Garlan David, Ivers James, Nord Robert, and Little Reed. 2000. *Software Architecture Documentation in*

Practice: Documenting Architectural Layers. Tech. rept. CMU/SEI-2000-SR-004. Software Engineering Institute.

Beuche Daniel. 2004. Demonstration: Variants and Variability Management with pure::variants. *In: Proceedings of the Workshop on Software Variability Management for Product Derivation, at Software Product Line Conference (SPLC3).*

Beuche Daniel, Papajewski Holger, and Schröder-Preikschat Wolfgang. 2004. Variability Management with Feature Models. *Science of Computer Programming*, 333–352.

Bosch Jan. 2000. *Design and Use of Software Architectures: Adapting and Evolving a Product-Line Approach*. Addison-Wesley, Boston.

Bosch Jan. 2002. Maturity and Evolution in Software Product Lines: Approaches, Artefacts and Organization. *Pages 257–271 of:* Chastek Gary J. (ed), *Proceedings of the Second Software Product Line Conference (SPLC2).*

Brooks Frederick P. 1987. No Silver Bullet: Essence and Accidents of Software Engineering. *Computer*, **20**(4), 10–19.

Clements Paul, and Northrop Linda. 2001. *Software Product Lines—Practices and Patterns*. Addison-Wesley, Boston.

CVS. 2004. *Concurrent Versions System*. http://www.cvshome.org/. Visited December 2004.

Czarnecki K., and Eisenecker U.W. 2000. *Generative Programming*. Addison-Wesley, Boston.

Czarnecki K., Besnasch T., Unger P., and Eisenecker U.W. 2002. Generative Programming for Embedded Software: An Industrial Experience Report. *Pages 156–172 of: ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*.

Czarnecki Krzysztof, Helsen Simon, and Eisenecker Ulrich. 2004. Staged Configuration Using Feature Models. *Pages 266–283 of: Proceedings of the 3rd Software Product Line Conference (SPLC3).*

Eclipse IDE. 2004. *Eclipse Integrated Development Environment*. http://www.eclipse.org/. Visited December 2004.

Faltings Boi, and Freuder Eugene C. 1998. Configuration (Guest Editor's Introduction). *IEEE Ingelligent Systems*, **13**(4), 32–33.

Felfernig A., Friedrich G., and Jannach D. 2001. Conceptual Modeling for Configuration of Mass-Customizable Products. *Artificial Intelligence in Engineering*, **15**(2), 165–176.

Geyer Lars, and Becker Martin. 2002. On the Influence of Variabilities on the Application-Engineering Process of a Product Family. *Pages 1–14 of:* Chastek Gary J. (ed), *Proceedings of the Second Software Product Line Conference (SPLC2)*.

Günter Andreas, and Hotz Lothar. 1999. KONWERK—A Domain Independent Configuration Tool. *In: Proceedings of the AAAI 1999 Workshop on Configuration*.

Günter Andreas, and Kühn Christian. 1999. Knowledge-Based Systems—Survey and Future Directions. *Pages 47–66 of: Proceedings of the 5th Biannual German Conference on Knowledge-Based Systems*.

Hein Andreas, and MacGregor John. 2003. Managing Variability With Configuration Techniques. *Pages 19–23 of: Proceedings of Software Variability Management ICSE 2003 Workshop*.

Hein Andreas, MacGregor John, and Thiel Steffen. 2001. Configuring Software Product Line Features. *In: Proceedings of the Workshop on Feature Interaction in Composed Systems in ECOOP 2001*.

Hollmann Oliver, Wagner Thomas, and Günter Andreas. 2000. EngCon—A Flexible Domain-Independent Configuration Engine. *In: Proceedings of the Configuration Workshop in conjunction with the 14th European Conference on Artificial Intelligence ECAI 2000.*

Hotz Lothar, and Krebs Thorsten. 2003a. Configuration—State of the Art and New Challenges. *Pages 145–157 of: Proceedings of the 17th Workshop, Planen, Scheduling and Configuration, PuK2003*.

Hotz Lothar, and Krebs Thorsten. 2003b. Supporting the Product Derivation Process with a Knowledge-based Approach. *Pages 24–29 of: Proceedings of Software Variability Management ICSE 2003 Workshop*.

Hotz Lothar, Krebs Thorsten, and Wolter Katharina. 2004. Combining Software Product Lines and Structure-based Configuration—Methods and Experiences. *In: Proceedings of the Workshop on Software Variability Management for Product Derivation, at Software Product Line Conference (SPLC3).*

IEEE Std 1471. 2000. *IEEE Standard 1471-2000: IEEE Recommended Practice for Architectural Description of Software-Intensive Systems*.

Kang K.C., Cohen S.G., Hess J.A., Novak W.E., and Peterson A.S. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Tech. rept. CMU/SEI-90-TR-21, ADA 235785. Software Engineering Institute.

Kang K.C., Lee Jaejoon, and Donohoe P. 2002. Feature-Oriented Product Line Engineering. *IEEE Software*, **19**(4), 58–65.

Kojo Tero, Männistö Tomi, and Soininen Timo. 2003. Towards Intelligent Support for Managing Evolution of Configurable Software Product Families. *Pages 86–101 of: Software Configuration Management (ICSE Workshops SCM 2001 and SCM 2003 Selected Papers).*

Krebs Thorsten, Hotz Lothar, Ranze Christoph, and Vehring Guido. 2003. Towards Evolving Configuration Models. *In: Proceedings of the 17th Workshop, Planen Scheduling und Konfigurieren (PuK2003) – KI 2003 Workshop*.

MacGregor John. 2002. Requirements Engineering in Industrial Product Lines. *In: Proceedings of REPL02, International Workshop on Requirements Engineering for Software Product Lines*.

MacGregor John. 2004. *CONIFP—Configuration in Industrial Product Families*. Presentation in Workshop on Software Variability Management for Product Derivation, at Software Product Line Conference (SPLC3).

Männistö Tomi, Soininen Timo, and Sulonen Reijo. 2000. Configurable Software Product Families. *In: ECAI 2000 Configuration Workshop, Berlin*.

Männistö Tomi, Soininen Timo, and Sulonen Reijo. 2001a. Modelling Configurable Products and Software Product Families. *In: IJCAI 2001 Configuration workshop*.

Männistö Tomi, Soininen Timo, and Sulonen Reijo. 2001b. Product Configuration View to Software Product Families. *In: International Workshop on Software Configuration Management (SCM-10) at ICSE 2001*.

Northrop Linda M. 2002. SEI's Software Product Line Tenets. *IEEE Software*, **19**(4), 32–40.

Poseidon for UML. 2004. *Poseidon for UML by Gentleware*. http://www.gentleware.com/. Visited November 2004.

Raatikainen Mikko, Timo Soininen, Männistö Tomi, and Antti Mattila. 2003. A Case Study of Two Configurable Software Product Families. *In: Proceedings of the Fifth International Workshop on Product Family Engineering (PFE-5)*.

Roshandel Roshanak, van der Hoek Andre, Mikic-Rakic Marija, and Medvidovic Nenad. 2004. Mae—A System Model and Environment for Managing Architectural Evolution. *ACM Transactions on Software Engineering and Methodology*, **18**(2), 240–276.

Sabin Daniel, and Weigel Rainer. 1998. Product Configuration Frameworks—A Survey. *IEEE Ingelligent Systems*, **13**(4), 42–49.

SEI Software Technology Roadmap. 1997. *SEI Software Technology Roadmap (28th September 1997)*. http://www.sei.cmu.edu/str/. Visited November 2004.

Shaw M., and Garlan D. 1996. *Software Architecture - Perspectives on an Emerging Discipline*. Prentice Hall. Chap. 2.

Shaw Mary. 2002. What Makes Good Research in Software Engineering? *International Journal of Software Tools for Technology Transfer*, **4**(1), 1–7.

Simons Patrik, Niemelä Ilkka, and Soininen Timo. 2002. Extending and Implementing the Stable Model Semantics. *Artificial Intelligence*, **138**, 181–234.

Soininen T., Tiihonen J., Männistö T., and Sulonen R. 1998. Towards a General Ontology of Configuration. *AI EDAM (Artificial Intelligence for Engineering Design, Analysis and Manufacturing)*, **12**(4), 357–372.

Soininen T., Tiihonen J., Männistö T., and Sulonen R. 2002. Special Issue: Configuration (Guest Editorial Introduction. *AI EDAM (Artificial Intelligence for Engineering Design, Analysis and Manufacturing)*, **17**(1–2).

Soininen Timo, Niemelä Ilkka, Tiihonen Juha, and Sulonen Reijo. 2001. Representing Configuration Knowledge with Weigth Constraint Rules. *In: AAAI Spring 2001 Symposium on Answer Set Programming*.

Sommerville Ian. 2004. *Software Engineering*. 7th edn. Addison-Wesley.

Thiel Steffen, and Hein Andreas. 2002a. Modelling and Using Product Line Variability in Automotive Systems. *IEEE Software*, **19**(4), 66–72.

Thiel Steffen, and Hein Andreas. 2002b. Systematic Integration of Variability into Product Line Architecture Design. *In: Proceedings of the 2nd Software Product Line Conference (SPLC2)*.

Tiihonen J., Lehtonen T., Soininen T., Pulkkinen A., Sulonen R., and Riitahuhta A. 1998. Modeling Configurable Product Families. *In: Proceedings of 4th WDK Workshop on Product Structuring*.

Tiihonen Juha, Soininen Timo, Niemelä Ilkka, and Sulonen Reijo. 2002. Empirical Testing of a Weight Constraint Rule Based Configurator. *In: ECAI 2002 Configuration Workshop, July 22-23, Lyon, France*.

Tiihonen Juha, Soininen Timo, Niemelä Ilkka, and Sulonen Reijo. 2003. A Practical Tool for Mass-Customising Configurable Products. *In: Proceedings of the 14th International Conference on Engineering Design (ICED'03)*. Accepted for publication.

van der Hoek A., Heimbigner D., and Wolf A. 1999. *Capturing Architectural Configurability: Variants, Options and Evolution*. Tech. rept. CU-CS-895-99. Department of Computer Science, University of Colorado, Boulder, Colorado.

van der Hoek Andre. 2004. Design-Time Product Line Architectures for Any-Time Variability. *Science of Computer Programming*, **53**(3).

van Gurp J., Bosch J., and Svahnberg M. 2001. On the Notion of Variability in Software Product Lines. *Pages 45–54 of: Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA2001)*.

van Ommering R., van der Linden F., Kramer J., and Magee J. 2000. The Koala Component Model for Consumer Electronics Software. *IEEE Computer*, **33**(3), 78–85.

van Ommering Rob. 2002. Building Product Populations with Software Components. *Pages 255–265 of: Proceedings of the 24th International Conference on Software Engineering (ICSE2002)*.

van Ommering Rob. 2004. *Building Product Populations with Software Components*. Doctor's Thesis, University of Groningen.

von der Maßen Thomas, and Lichter Horst. 2003. RequiLine: A Requirements Engineering Tool for Software Product Lines. *In: Proceedings of the 5th International Workshop on Product Family Engineering (PFE-5)*.

Weiss David, and Lai Chi Tau Robert. 1999. *Software Product-Line Engineering—A Family-based Software Development Process*. Addison-Wesley, Boston.

Ylinen Katariina, Männistö Tomi, and Soininen Timo. 2002. Configuring Software with Traditional Methods—Case Linux Familiar. *In: ECAI 2002 Configuration Workshop*.