Foreword

This report summarises the result from a research project called *Sarcous* carried out by Software Business and Engineering Institute (SoberIT) of Helsinki University of Technology. The project was funded by Tekes (National Technology Agency of Finland) and participating industrial companies.

The Sarcous project was set up to study and formulate methods for managing software product families and re-usable software assets on the basis of modelling them as configurable software product families.

The project met its objectives well. A large amount of knowledge about the state-of-the-art of software product families was collected. Further, modelling concepts and concrete languages for modelling configurable software product families were developed. A large number of demonstrations and tools were built, both based on existing tools and techniques and from scratch. These results were reported in a large number of high-quality publications, published at the most important forums of the field.

During the course of the project, several individuals have contributed to the results. They include: *Timo Asikainen*, *Tero Kojo*, *Varvana Myllärniemi*, *Antti Mattila*, *Mikko Multimäki*, *Tomi Männistö*, *Mikko Raatikainen*, *Timo Soininen*, and *Katariina Vuorio (née Ylinen)*.

We thank professor *Reijo "Shosta" Sulonen*, who was the main originator of the project ideas. Further, we thank all the people who have contributed to build the discipline of product configuration and product data management, on which the project was built. We also thank all our colleagues for their support and discussions. Finally, we thank the companies that participated in the project for their co-operation.

Espoo, Finland, October 2004

Timo Asikainen and Tomi Männistö

Contents

Chapter 1

Introduction to Sarcous

Tomi Männistö, Timo Asikainen, and Timo Soininen

Sarcous is a research project that has been carried out by Software Business and Engineering Institute (SoberIT) of Helsinki University of Technology. The project was funded by Tekes (National Technology Agency of Finland) and participating industrial companies. The project was started in 2000 and ended in 2004, during which time three projects periods were conducted, each lasting slightly over one year.

This chapter introduces the Sarcous project, its background, goals, and summarises the results of the project.

1.1 Background

Software product families (SPF) (or *lines*, as they are also known) are a means for increasing the efficiency of software development and to control the complexity and variability of products. A SPF can be defined to consist of a *common architecture*, a *set of reusable assets* used in systematically producing, i.e., deploying, products, and the *set of products* thus produced.

The concept of product family has existed for decades (Parnas, 1976), but only recently have research results shown that a software product family may provide industrially relevant benefits, such as decreased development effort and time-to-market (Knauber *et al.*, 2002; Cohen, 2002), and that the family itself and some of the issues it addresses are important and predictive for success of reuse (Tracz, 1988; Frakes & Fox, 1995; Morisio *et al.*, 2002). The constituting assets of a software product family are typically assumed to be specifically developed for the family, and then shared and reused in the development of individual products. Figure 1.1 illustrates the main concepts of software product families.

One of the most important benefits of the software product family approach is the increase in reuse. The reuse needs not to be limited to a single kind of asset, such as program code, but may cover multiple asset kinds, such as architecture (Clements & Northrop, 2001) or requirements. Further, in the context of software product families reuse tends to be systematic (Bosch, 2000). Finally, reuse in the context of software product families is not seen as a merely technical problem, but to concern other issues as well, such as business, process,

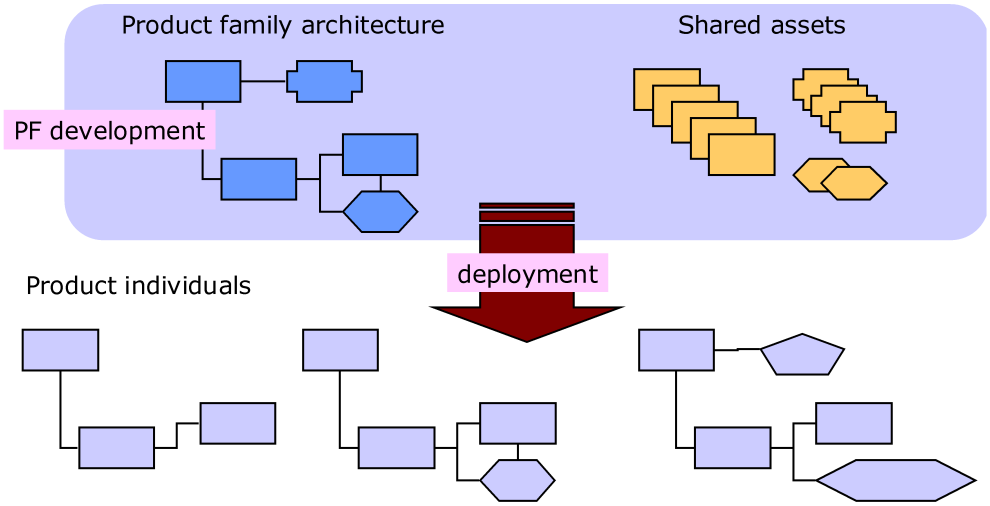

Figure 1.1: Main software product line artefacts and processes

organizational, and architecture (van der Linden, 2002).

Sarcous project was set up to study a new approach to the development of software product families under the following assumptions:

- The software development consists of systematically developing, modelling, managing and applying re-usable components using supporting intelligent tools. The components represent information on functional subsystems rather than single files or atomic classes.
- The software is *configurable*, i.e., it has a common architecture into which (mostly) pre-defined components can be combined according to the needs of the user, while respecting the known design constraints.

Hence, the project studied a special class of software product families that we call *configurable software product families (CSPF)*. A configurable software product family allows derivation of product individuals without customer-specific design or programming effort; in this sense, configurable software product families represent an extreme in the ease of producing new variants. This can also been graphically in Figure 1.2, which illustrates different approaches to managing the variability of software product families.

It should be noted that although a configurable software product family provides significant benefits, it is not the optimal solution for all companies: building a CSPF requires that many issues related to the product family must be established in the organisation.

1.2 Goals

The objectives of Sarcous project was to *study and formulate methods for managing software families and re-usable software assets on the basis of modelling them as configurable software product families*. The re-usable assets may be requirements, features, designs or pieces of implementation.

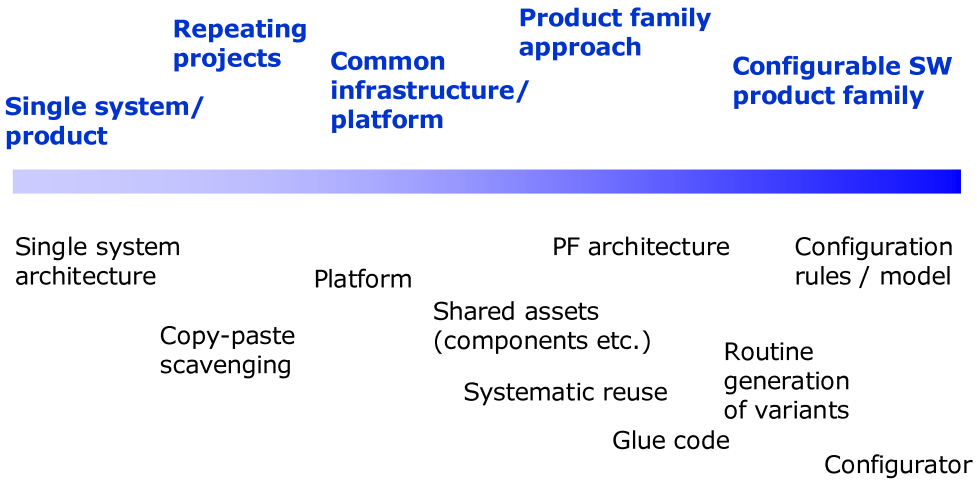

Figure 1.2: Approaches to variability management

The overall goal of the project was stated as: *To develop methods for managing software product families on the basis of modelling them as configurable software product families*. This goal was refined into two research questions:

1. How to model and manage SPFs, especially their variability and evolution?
2. What kind of intelligent support for reuse and configuration of SPFs can be offered?

1.3 Tasks

The principal research philosophy applied in the project was that of SoberIT in more general: finding industrially relevant research problems, solving them, and turning the results achieved into industrial practices. This philosophy is illustrated in Figure 1.3.

To address the research questions, the work in the project was divided into major tasks as follows:

1. Gaining an understanding of the existing work in related scientific fields.
2. Gaining an understanding of the real needs companies by investigation of the software product family problem modelling concepts, especially *features*, *components*, *interfaces* and *versioning* in case companies
3. Development of a modelling language and method to support SPFs
4. Validation of developed methods in industrial cases and by demonstrating the configuration concept by building prototype tools

The very first task of the project was to gain an understanding of the related work that had been conducted in related research fields. The next task was to survey the state-of-the-practice of software product families in the Finnish industry. The goal of the survey was to find out

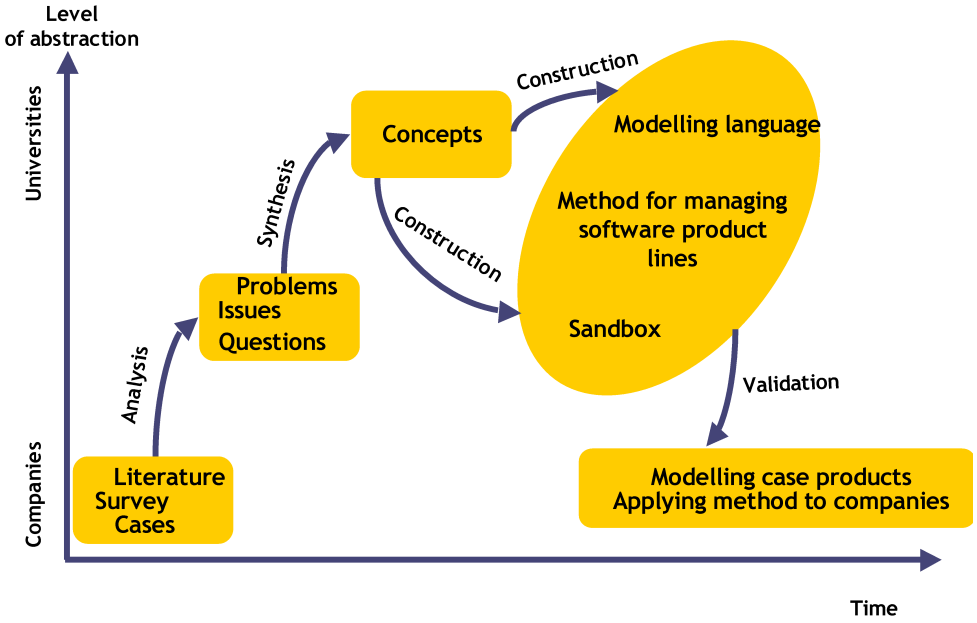

Figure 1.3: Sarcous research philosophy

where industrial companies stand with respect to software product families and what are the challenges they are facing. Experiences from the survey were fed as a feedback to the project for guiding the remaining tasks towards highly relevant research results.

The main constructional research task was to develop a conceptual foundation for modelling configurable software product families, to devise a modelling language with formal semantics based on the conceptual foundation, and to demonstrate the language.

1.4 Results

The results of the Sarcous project are divided in the four categories introduced in the following subsections. Only a brief summary of the main results is given here; more details can be found from the corresponding chapters of this report.

1.4.1 Survey on software product families

Six companies were surveyed and analysed. The results thus obtained were reported to the companies and in anonymised form to the scientific community. The reporting work is still partly in progress and will yield some result after the project has ended. This topic is covered in Chapter 2.

1.4.2 Modelling concepts and languages

The work done on concepts concepts can be divided in three main categories:

1. *Architectural description* of the product in terms of *components* that may contain other components as their *parts*; *interfaces* of components, and *connections* between them

2. *Feature modelling* concepts, which include *features* organised in a hierarchy by the *subfeature* relation, concepts for variability, such as *optional* and *alternative features*, and *constraints* that may be used to express relationships involving features, such as *requires* or *incompatibility*

3. *Evolution* of components

A configuration language called *Kumbang* was developed on the basis of Koala software architecture description language of Philips (van Ommering *et al.*, 2000), a number of *feature modelling* methods, and lessons leearned from configuration of non-software products.

The modelling methods were validated by applying them in collaboration with industrial partners to real cases.

This topic is covered in Chapter 3.

1.4.3 Tools and demonstrations

The tool support research in Sarcous had two main lines of activities:

- Applying existing configuration technology designed for non-software products to configurable software product families

- Development of prototype tools that are built on software specific variability modelling and management concepts

The first category includes the following demonstrations based on *WeCoTin*, a product configurator developed at SoberIT:

- Modelling and configuring features

- Configuring Familiar, a Linux distribution for Compaq iPAQs

- (Re)configuration with multiple Linux over multiple releases

In addition, the following prototype tools were developed from scratch:

- A tool for configuring and installing Linux Familiar on Compaq iPAQ

- A prototype modelling tool for creating models of configurable software product families combining component- and feature-based aspects

- A prototype configuration tool called *Kumbang* for software product families combining component- and feature-based modelling approaches.

This topic is covered in Chapter 4.

1.4.4 Dissemination and international collaboration

Active participation of the project in the international community of software product families is evident from the activities listed in Chapter 5, and from the project publications listed in Chapter 6.

1.5 Conclusions and future work

A central issue in variability management is understanding of variability in a manner that allows explicit description of it. The Sarcous project took concrete steps towards making this happen in realistic and feasible manner.

However, we are not there yet, as many issues are to be resolved in a more systematic manner. These include but are not limited to the following two main areas:

1. Tool support covering different life cycles, such as requirements, architecture, feature models, components and product derivation
2. Best practices in SPF adoption, variability management, variability documentation, traceability

The underlying goal is to provide a company working with or considering moving towards software product families a *readily-applicable, validated set of methods*. To achieve this, the SoberIT research philosophy needs to be applied, see Figure 1.3: gathering the state-of-the-art practices from the industry, generalising and tailoring these to match the specific needs of other companies, and bringing the results back into the industry.

1.6 Outline

The remainder of this report is structured as follows. Next, in Chapter 2 we will discuss a survey carried out within the project. Thereafter, in Chapter 3, we will present the modelling concepts and languages developed. A discussion of the demonstrations and tools built follows in Chapter 4. The dissemination of the project results is discussed in Chapter 5. Finally, Chapter 6 contains a list of the publications of the project.

Chapter 2

Software Product Family Survey

Mikko Raatikainen, Timo Soininen, and Tomi Männistö

In this chapter, we discuss the product family survey carried out in Finnish software companies as a part of the Sarcous project. In more detail, we will first discuss the framework used for the survey. Thereafter, we will provide some details of the results of the survey. Finally, we will characterise configurable software product families.

2.1 Framework for studying software product families

The quality of the data gathered in any survey or other empirical study heavily depends on the methods applied in gathering the data. Therefore, we did not want to use *ad hoc* methods in our survey. Unfortunately, there seemed not to exist readily-available research instruments for studying software product families in the industry. Consequently, our first step was to develop a research instrument for this purpose.

The framework developed follows the qualitative case study strategy described in (Yin, 1994). In more detail, the framework consists of five steps: *designing case study*, *preparing data collection*, *collecting evidence*, *analysing the evidence*, and *composing a case study report*. In the following, we will discuss each of these steps in some detail.

In designing the case study, we formulated the *research problem* in the form of a question as follows: What kind of a software product family a company develops? Based on our preliminary understanding of software product families, a number of *study propositions* was made. As an example, it was postulated that the products in the family share a managed set of features. The cases were selected using *theoretical sampling*: the selection was not random, but was based on specific criteria. Finally, the *unit of analysis* was defined to be either a single software product family, or many of these, in the case that a company developed many.

In preparing the data collection, the most important issues were to decide the *field procedures* to be applied, and to prepare the *case study questions* to be used. Concerning the former, it was decided that at least two investigators participate in each interview, and the investigators should familiarise themselves with the questions before the interview. The latter issue, namely case study questions, were prepared based on the BAPO (Business, Architec-

ture, Process, Organisation) framework (van der Linden, 2002).

The collection of evidence was carried out as interviews in the participating companies. Each interview was tape recorded; in addition, notes were taken. In order to ensure that different viewpoints were taken into account, the interview was carried out in three distinct sessions, each concentrating on slightly different issues.

The analysis methods employed were not fixed in the framework. Similarly, no procedure was predetermined for composing the case report.

Further details of the framework can be found in (Raatikainen, 2003; Raatikainen *et al.*, 2004b).

2.2 State of the practice

The software product families of the six companies are summarised in Table 2.1. In the following, we will discuss a number of topics arising from the table, or otherwise worth discussion.

A number of reasons for applying the product family approach were mentioned. In most of the companies, the following issues were brought up in the interview: managing variability, shortening time-to-market, enhancing quality, increasing the number fo products, smoothening flow of projects, and more efficient use of resources. In addition, issues such as enhancing co-operation within the company, price categorisation were mentioned in some of the companies.

The size of the staff varied across the companies: the software engineering staff was in the magnitude of scores in most companies, but one company employed 200 software engineers. Also, the type of software varied, being embedded software in half of the companies, and software running on a PC in the other half.

The application domain, price of a product, the number of variants, and the number of deliveries varied greatly across the companies. This suggests that these attributes do not restrict the applicability of the software product family approach.

Table 2.1: Software product families in the companies studied

Factor	**Company A**	**Company B**	**Company C**	**Company D**	**Company E**	**Company F**
Application domain	Consumer electronics	Automation systems	Medical devices	Factory automation	Medical information management	CAD/CAE
Staff	200	5000	Hundreds	200	130	500
SPF engineers	20	25	200	25	35	35
Type of software	Embedded	Embedded	Embedded	PC software	PC software	PC software
Type of product	Embedded software in an electronic gadget	Mechanics controlled by embedded software	Embedded software in an electronic gadget	Mechanics controlled by software on a PC	Software product	Software product
Price	100-300€	100000€ +	1000-50000€	50000€	20000€	20000€
SPF coverage	50%	70-95%	75%+	100%	100%	100%
Variants/year	-[a]	10	-[a]	50	35	Thousands
Variants	20	-[a]	10	-[a]	-[a]	-[a]
Deliveries	Thousands	1	Hundreds	1	1	1

[a] The factor is not applicable
[b] Asset developers, no data about product developers
[c] The software engineers develop other product as well
[d] The number includes employees who develop, for example, complex algorithms that product includes

2.3 Characterizing configurable software product families

In (Raatikainen *et al.*, 2004a), we discuss two of the six companies that participated in the survey in more detail. These two companies have developed configurable software product families, from which we can conclude that configurable software product family is applied in the industry and hence is not a mere theoretical peculiarity. Furthermore, the paper shows that a configurable software product family is a feasible and efficient way to systematically develop a family of products and manage its variability: the two companies achieved competitive advantage through their configurable software product families, compared to companies with no such a family.

For both companies, the configurable software product family approach was an efficient way to systemize the software development and enabled an efficient control over variants, and even reconfiguration of individual products. They even went as far as to state that the approach is the only reasonable way to do business, regardless of the significant initial investment.

The study also exemplifies the feasibility of developing and using configurators, i.e., information systems supporting the derivation process: both companies had developed a configurator of their own. Using the configurator, the derivation process was made such that no software engineering skills were required.

The possible drawbacks of the approach seemed to pertain mainly to the evolution of the product family: the topic was seen as especially unpromising as far as the ability to meet customer requirements by configuring was concerned.

We also discovered a number of factors that affect the feasibility of the configurable software product family approach:

- Number of deliveries

  Supporting a configurable software product family requires a relatively large numbers of deliveries. However, it seems that a large number of deliveries is not enough to guarantee the feasibility of the approach. Further, what is a large enough number depends on the company: what is large enough for a company may not be enough to another.

- Application domain understanding

  A thorough understanding of the targeted application domain has a positive effect on the feasibility. Further, stable application domains are more likely to support a successful configurable software product family than emerging or otherwise unstable ones.

- Clear organisational separation of development and deployment

  Even for small organisation, it seems to be beneficial, from the configurable software product family point of view, to separate the development and deployment activities. The separation should be done at least at the role level, i.e., developing assets and deploying individual products should be considered as different activities.

We also found a number of factors that are seemingly irrelevant for the success of configurable software product families.

- Special software engineering skills

- Advanced process models, methods, modelling and implementation tools
- Application domain
- Company maturity

In (Raatikainen *et al.*, 2005), we give detailed accounts of the derivation process in the two configurable software product families; these are the same families that were studied already in (Raatikainen *et al.*, 2004a). The findings shed new light on the previous characteristics of configurable software product families, and gave grounds for new ones (Eisenhardt, 1989; Strauss & Corbin, 1998). These characteristics are the following:

1. Configurable product base

   A configurable software product family includes a configurable product base that constitutes at least a significant part of any product individual

2. High- and low-level variability

   Variability takes place at a *high level*, which pertains to important characteristics, and at a textitlow level, pertaining to detailed characteristics. Both forms of variability have constraints but only the high-level variability is visible to a customer.

3. Light-weight derivation

   The derivation process requires only minor effort. Further, derivation activities are highly flexible both spatially and temporally, and their order may be permuted. Derivation requires mainly application domain knowledge, understanding of the products, and system administrator skills; in other words, no significant software engineering skills are required.

4. Limited tailoring during product derivation

   Only certain aspects of a product may be tailored. Further, requirements for tailoring are carefully analyzed. Finally, tailoring is performed in separation of other activities.

5. Late and flexible binding of variability

   Variability is bound in a late point of time, and bindings can be changed.

6. Derivation organisationally independent

   There is an organisational unit concentrating on deriving product individuals, and the unit has no other, significant responsibilities.

Chapter 3

Modelling Concepts and Languages

Timo Asikainen, Tomi Männistö, Tero Kojo, and Timo Soininen

In this chapter, we will discuss the modelling concepts and languages developed during the Sarcous project. First, we will briefly discuss how research on *configurable products* has been used as the basis of our research efforts. Thereafter, we will go into the specific modelling concepts and languages developed for configurable software product families. These include *Koalish*, a language for modelling configurable software product families from an architectural point of view; *Forfamel*, a language for modelling them based on the common and variable features provided by the individual product in the product family; and *Kumbang*, a language that combines the modelling concepts of Koalish and Forfamel into a single language.

3.1 Product configuration and configurable products

The work done on modelling concepts and languages is based on earlier work done on *configurable* (non-software) *products*: they have been researched extensively in the *product configuration* domain, a subfield of artificial intelligence. Our guideline has been to reuse work done in the product configuration domain at three levels of abstraction. These levels include:

1. Tool level

   Tool level is the lowest level of abstractions. Reuse at this level pertains to applying supporting tools developed for configurable (non-software) products to software product families. Reuse of existing tools is discussed in detail in Section 4.1.

2. Language level

   Language level is the second lowest level of abstraction on which reuse may occur. Reuse at the language level pertains to applying existing modelling languages to configurable software product families. The demonstrations discussed in Section 4.1 are also implicit examples of reuse at the language level.

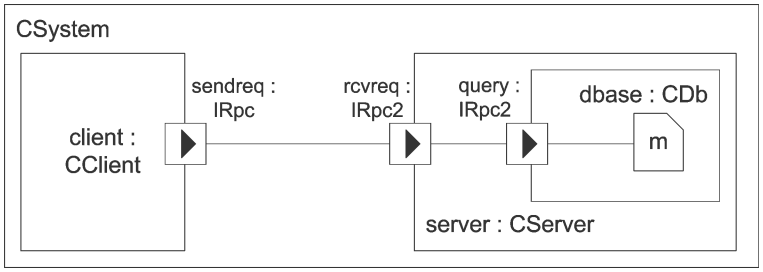

Figure 3.1: An architectural model of a simple client-server system represented in Koala

3. Conceptual level
   Reuse at the conceptual level pertains to modelling configurable software product families using existing modelling concepts directly, or translating software-specific concepts to these concepts. Reuse at the conceptual level may be used to provide models of software product families formal semantics, given that the reused concepts have been defined one. The modelling methods discussed later in this chapter reuse many concepts originally defined in the product configuration domain for configurable (non-software) products.

The similarities between configurable product and configurable software product families have been studied in (Männistö *et al.*, 2000; Männistö *et al.*, 2001a; Männistö *et al.*, 2001b).

3.2 Koalish—Software architecture

The level of design concerning the overall structure of software systems is commonly referred to as the *software architecture* level of design. This level includes structural issues such as the organisation of a system as a composition of components, the protocols for communication, the assignment of functionality to design elements, the composition of design elements etc. (Garlan, 2001).

Informally, software architecture is used to refer to the structure of a software system on a high level of abstraction. Explicitly, software architecture does not concern the fine-grained structure or the properties of a software system, or the process used to develop it (Medvidovic & Taylor, 2000).

We considered software architecture to be an important aspect of configurable software product families. Therefore, it was decided that a method for describing the architectures of configurable software product lines should be developed. Towards this goal, a number of *architecture description languages* (ADLs) were studied.

Loosely defined, ADLs are formal notations with well-defined semantics for describing software architectures. A large number of ADLs have been proposed. The greatest common denominator for the class of ADLs is the concept of computational elements, usually termed *components*, present in all of them; in other respects, they differ from each other radically. (Medvidovic & Taylor, 2000)

Figure 3.1 contains an example of an architectural description of a simple client-server system, the ADL used is Koala (van Ommering *et al.*, 2000). This is an example of graphical

architectural description. However, architectural descriptions may also be represented in text: for instance, Koala includes a textual syntax as well.

The first step in the study of ADLs was a paper (Asikainen *et al.*, 2002). In this paper, three ADLs, namely Acme (Garlan *et al.*, 1997), Wright (Allen & Garlan, 1997), and Koala (van Ommering *et al.*, 2000) were analysed and compared with the configuration ontology of (Soininen *et al.*, 1998). The same line of work was carried further in (Asikainen, 2002), in which a more detailed analysis, along with a mapping between the concepts of the ADLs and the configuration modelling concepts was presented.

The overall outcome from this work was that the configuration modelling concepts are remarkably similar to those found in ADLs, but nevertheless, the concepts would have to be modified in order to fully support the modelling primitives of ADLs, especially the interface mechanism of Koala.

Consequently, concepts for modelling the architecture of configurable software product families were developed (Asikainen *et al.*, 2003b) . The conceptualisation is termed *Koalish*. The same name is used for the language built on the concepts (Asikainen *et al.*, 2003a). The uses the same basic concepts as Koala, namely *component types*, their *compositional structure* in terms of other components contained in them, *interface types* and the *interfaces* of components, and *bindings*, or *connections*, as they are also called, between interfaces. In addition, Koalish includes constructs for expressing *optionality* and *alternatives*, and *constraints* concerning different aspects of the above-mentioned concepts.

3.3 Forfamel—Feature modelling

Feature modelling has become has become a popular method for modelling software product families. *feature* lacks an agreed-upon definition. Popular definitions include:

1. An end-user visible characteristic of a system

2. A distinguishable characteristic of a concept (e.g., system, component, and so on) that is relevant to some stakeholder of the concept

3. ”We define feature as a logical unit of behaviour that is specified by a set of functional and quality requirements. A feature generally captures a considerable set of requirements and is, as such, used to group requirements, which simplifies requirements handling. In addition, a feature represents a logical unit of behaviour from the perspective of one or several stakeholders of the product. For instance, the user of a product generally considers the product to consist of a number of functional units that are identified as different. Each such functional unit, we refer to as a feature.” (Bosch, 2000)

We agree with the first definition in that end-user visible characteristics of systems can be termed features. Further, in the spirit of the second definition, be believe that it may be fruitful to consider also things that are visible to end-users as features. Further, we agree with the third definition in that it may be useful to consider features as abstractions from requirements. However, unlike suggested by the second definition, we believe that it is necessary to make a

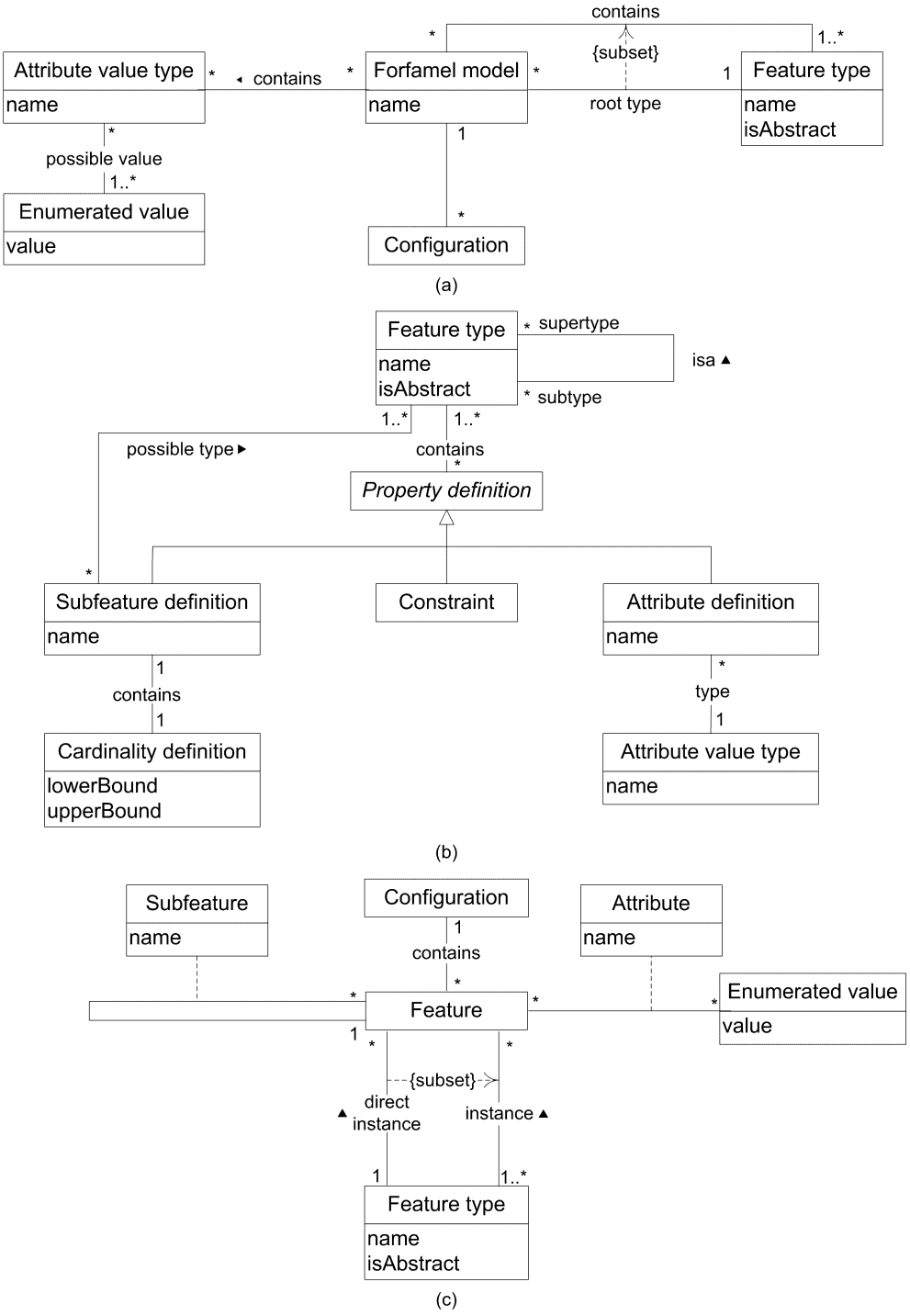

Figure 3.2: Forfamel concepts. (a) Forfamel model and its configurations (b) The properties of feature types (c) Configurations.

distinction between architectural concepts and features. Consequently, we believe that entities such as components, interfaces, subsystems etc. should *not* be considered features.

Similarly as in the case of ADLs, we conducted a comparative study between a number of feature modelling methods, and the configuration modelling concepts used in the product configuration domain. This study has been reported in (Asikainen *et al.*, 2004b). The most important finding from the study was that the configuration modelling concepts can be used to capture feature modelling concepts. However, a fundamental difference between feature modelling methods configuration modelling concepts is that unlike the latter, the former includes distinct notions for types and instances. This implies that features models represented using the configuration modelling concepts differ somewhat from the original ones.

We have developed a feature modelling method called *Forfamel*. The method has been applied to a software product family of one of our industrial partners. The special requirements posed by configurable software product families have been taken into account when developing the method: a number of modelling concepts not present in existing methods have been integrated with the feature modelling concepts. In more detail, the feature models are decorated with *binding information*. In addition, the method makes a distinction between feature types and instances. However, this distinction is made primarily at the conceptual level; in terms of syntax, the feature models created based on our method may be highly similar to those creating using existing methods.

The concepts of Forfamel are illustrated in Figure 3.2 as a UML class diagram. Figure 3.3 contains a sample Forfamel model that represents the features of a software product line of advanced text editors.

Further details about Forfamel can be found in (Asikainen *et al.*, 2004a). However, some aspects of Forfamel are still work-in-progress.

3.4 Kumbang—Features and architecture combined

Given the two above-described methods for modelling configurable software product family, it is natural to ask: If both kinds of models are created for a single configurable software product family, how are they related? Obviously, it needs to be somehow ascertained that the two models are mutually consistent.

We have chosen the approach that the architectural model captured by the Koalish model, and the feature model represented using Forfamel is *orthogonal* in the sense that they describe the product family from two, different points of view. We will refer to these views as the *architectural* and *feature views*.

Above, when discussing features, we imposed a modelling guideline that entities related to the architecture of the product family should not be modelled as features. At this point, the utility of this guideline should be clear: by making the feature model free of architectural issues we will make the two views as independent as possible from each other.

Of course, it is not reasonable for the two views to be completely independent from each other. The mechanism for relating the two views are *implementation constraints*. The locus of these constraints is features, or more exactly, feature types. Their semantics is that in order for an individual product to provide a feature of the type, the architecture of the product must satisfy the conditions stated in the constraint. Intuitively, the dependencies between the

```
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 3.3: A sample Forfamel model representing the features of a software product line of advanced text editors.

architectural and feature views are directed: features include descriptions what they require from the architecture, but architectural entities may not pose similar requirements for the features.

The combination of the concepts of Koalish and Forfamel is called *Kumbang*. The conceptual basis of Kumbang has not yet been published. However, a configuration tool operating on Kumbang is described in (Myllärniemi *et al.*, 2004).

3.5 Modelling evolution

Building a configurable software product family is very likely to require significant initial investment. Consequently, in order to get paid back for the investment, the family must be long-lived. On the other hand, the requirements for the products are likely to change, as well as the platform on which it is running etc. Due to this combination of a long life span and changing requirements, many configurable software product families are bound to *evolve*.

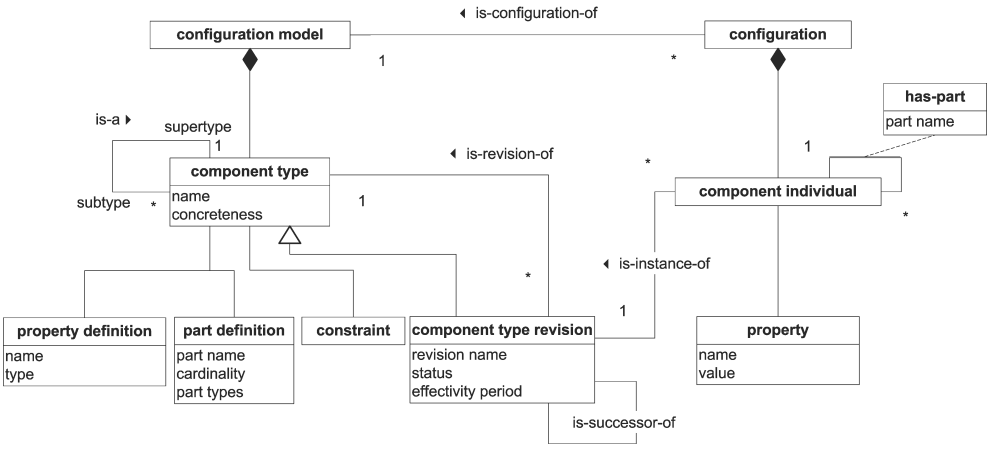

Figure 3.4: Modelling concepts for evolution

During the Sarcous project, concepts for modelling the evolution of configurable software product families were developed. In this section, an overview of these results is presented: a conceptual foundation for modelling evolution of configurable SPFs with the main concern being the deployment phase and generation of valid configurations.

The conceptualisation of evolution uses concepts, such as *components*, their *properties*, *compositional structure* and *constraints*; these concepts are shared with Koalish, see Section 3.2. However the conceptualisation adds a set of concepts that are useful in describing the evolution of a software product family.

These concepts are described briefly below and illustrated in Figure 3.4.

A *component type* has a set of *revisions*, called *component type revisions*, which are related to the component type by *is-revision-of* relation, and ordered by *is-successor-of* relation. Revisions capture the evolution of a component type in time.

Status describes the life-cycle status of a component type revision. It is a measure of the maturity of a component type revision, and may have values such as *unstable*, *stable*, and *end of life*, and can be used to convey additional information in configuration task. Status is a useful concept, e.g., for expressing user requirements, but has no relevance in determining the correctness or other properties of a configuration.

Effectivity period is a time interval stating when an component type revision may legally appear in a configuration. Effectivity period is thus an additional concept needed in determining the correctness of a configuration.

In the metamodel, component type revision is defined as a subtype of component type to indicate that component type revisions have the same properties as component types plus the additional concepts for representing evolution.

Each component individual is directly an instance of a component type revision, represented by the *is-instance-of* relation. This basically means that component individuals are component type instances with additional revision information.

The model is presented in more detail in (Kojo *et al.*, 2003).

The task of configuring Debian Familiar Linux packages over many releases and package versions was used as an example of the evolution of a software product family. This example

is discussed in Section 4.1.3.

Chapter 4

Tools and Demonstrations

Empirical studies are required to evaluate the feasibility of ideas and validate developed concepts, such as those discussed in the previous chapter. For this reason, we produced several demonstrations and prototype tools during the project.

The tools developed serve two purposes. First, they provide evidence that the concepts developed are sensible in the sense that they can be supported by tools. Second, tool support is essential in showing that the ideas and concepts can be applied in practice: industrial software product lines are often too complex to be managed simply by using a textual language and text editors; instead, supporting tools are needed.

As discussed in the introduction to the previous chapter, our guideline has been to reuse existing tools, languages, and concepts to the maximum possible extent. In more detail, we have applied an existing product configurator called *WeCoTin*, and the modelling language (*Product Configuration Modelling Language, PCML* underlying it (Tiihonen *et al.*, 2003). Section 4.1 reports demonstrations implemented using WeCoTin.

However, it turned out that WeCoTin or other existing tools developed for non-software products do not cover all relevant aspects of software product lines. Consequently, we set out to implement new tools. These are described in Section 4.2.

4.1 Demonstrations based on WeCoTin

In this section, we study a number of demonstrations that are based on WeCoTin. These demonstrations prove that tools developed for non-software products are indeed applicable to configurable software product lines. Further, the fact that none of the demonstrations to be discussed required significant amounts of coding or other development effort suggests that the tools provide an efficient way to provide tool support for software product lines.

In the first subsection, we will discuss how WeCoTin can be used to create and manage feature models, and to configure these models to find individual products. Thereafter, in the second subsection we will show how WeCoTin can be used to configure Linux Familiar. The third subsection extends this discussion to cover configuring multiple releases simultaneously.

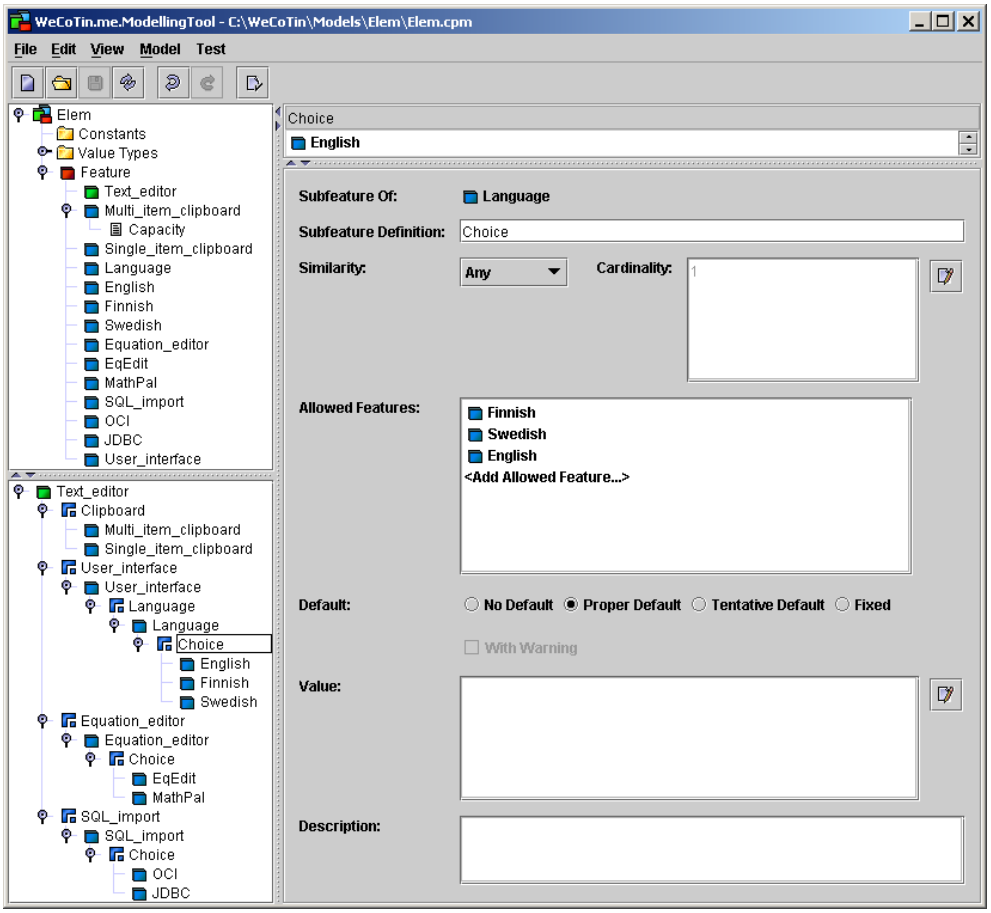

Figure 4.1: WeCoTin Modelling applied to modelling features

4.1.1 Modelling and configuring features

Timo Asikainen, Tomi Männistö, and Timo Soininen

The idea underlying this demonstration is the assumption that feature models, if provided with rigorous semantics, can be used as the basis for configuring. Further, the hypothesis was that the tool support required for efficient configuration could be provided by WeCoTin.

It turned out that the modelling capabilities of WeCoTin were more than sufficient to match those found in most feature modelling methods: in addition to the standard modelling primitives found in feature modelling methods, WeCoTin includes a number of constructs especially suited for modelling configurable software product families. These constructs include the possibility to define a cardinalities for parts, and a constraint language for imposing additional rules that must be obeyed by valid individuals. In consequence, it was found that WeCoTin can be used as a feature modelling tool that enables the configuration of the models, that is, finding descriptions of individual products in the product family.

However, WeCoTin seems not to be an optimal tool for feature modelling: the distinction between types and instances made in WeCoTin seems unnecessary in many cases in the

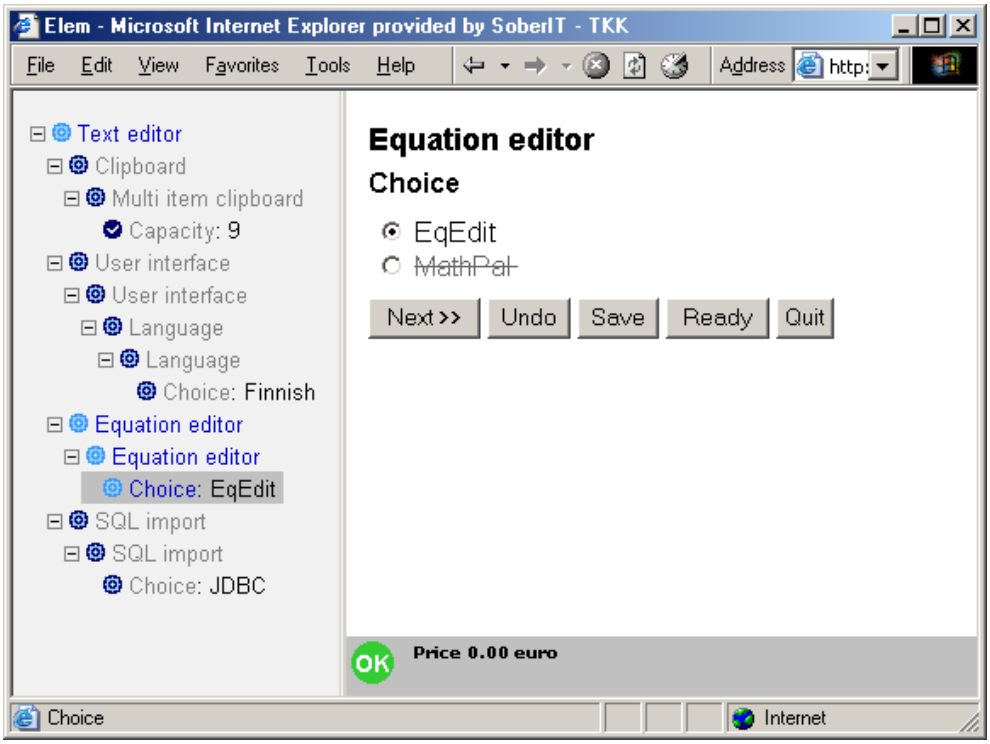

Figure 4.2: Configuration features using the WeCoTin configuration interface

context of feature modelling. Instead, the distinction appears to cause overhead in both the modelling and configuration tasks. A better solution might be to provide the user the option of defining types and thus reusing feature information.

Figure 4.1 illustrates the user interface of the WeCoTin modelling tool applied to feature modelling; using this interface, it is possible to create feature models. Further, the configuration interface is illustrated in Figure 4.2.

This demonstration has been reported in more detail in (Asikainen *et al.*, 2004c).

4.1.2 Configuring Linux Familiar

Katariina Vuorio, Tomi Männistö, and Timo Soininen

Familiar is a distribution of the Linux operating system developed for a line of handheld computers, namely Compaq iPAQ. Software running in handheld devices such as iPAQ is interesting from the configuration point of view for two reasons: first, resources, such as memory, available in such devices are very limited, which places stringent requirements on the software running on them. Second, Linux in general represents a challenge from the configuration point of view: the software is composed of a hundreds of *packages*, of which there may be multiple *versions* and between which there may be different kinds of *dependencies*.

In the demonstration, the package descriptions of Linux Familiar were automatically translated into Product Configuration Modelling Language (PCML), the modelling language un-

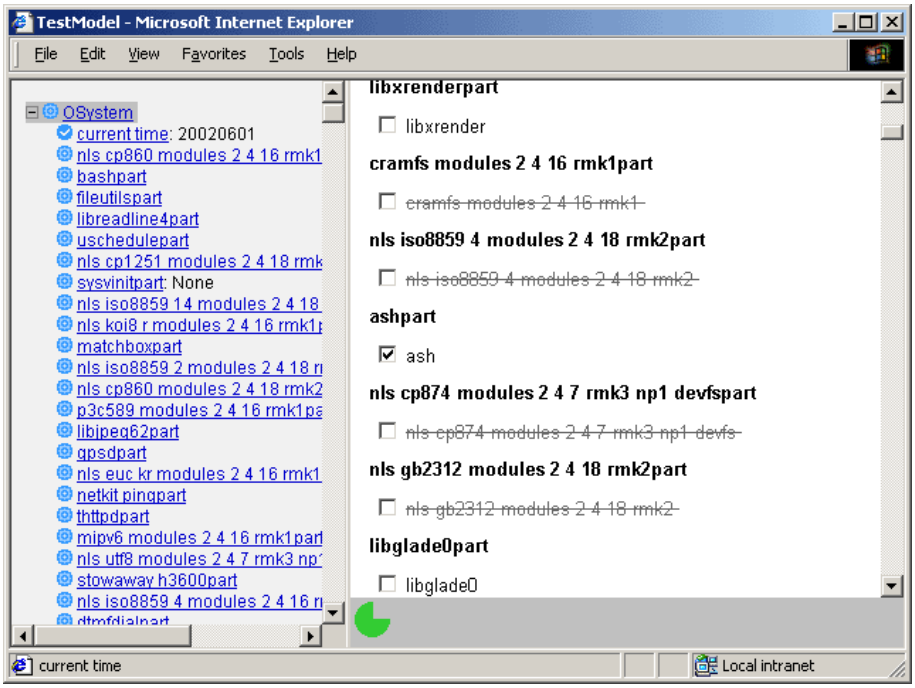

Figure 4.3: Configuring software over time using WeCoTin

derlying WeCoTin. Using the translated model, it is possible find valid configurations using the WeCoTin configurator.

The demonstration has been reported in more detail in (Ylinen *et al.*, 2002).

4.1.3 Configuring Linux Familiar over multiple releases

Tero Kojo, Tomi Männistö, and Timo Soininen

This is a very large scale configuration demonstration that takes the first steps towards configuring software over time. The approach followed relies on the modelling concepts presented in Section 3.5. However, WeCoTin provides no intrinsic support for versioning. Consequentely, the version knowledge had to be translated to other concepts available in PCML, the modelling language underlying WeCoTin. This was seen as a limitation, as versioning is an essential part of software development.

WeCoTin enables structuring the Familiar packages into categories and provides an intuitive web interface. The interface provides a user with intelligent support, such as greying out packages that may not be selected. These features are illustrated in Figure 4.3

The work was a complete success: it proved that even considerable large software product families can be efficiently configured over time. It also showed that product configuration is intimately related to the areas of software deployment and design (van der Hoek *et al.*, 1997), through the discipline of *software configuration management* (Conradi & Westfechtel, 1996).

The details of this work are further elaborated in (Kojo *et al.*, 2003).

4.2 Tools for Configurable Software Product Families

In the previous section, it was shown that tools developed for non-software products provide a feasible way to model and configure configurable software products. However, this approach is not feasible in all situations, due to several conceptual differences between software and traditional products (Männistö *et al.*, 2000; Männistö *et al.*, 2001a; Männistö *et al.*, 2001b). For example, traditional product configuration techniques lack means for describing software architectures: as an example, WeCoTin includes no notion of connection points in components comparable to interfaces in Koala and Koalish, see Section 3.2 or (Asikainen *et al.*, 2003a). As interfaces and bindings between them are a fundamental ingredient of Koalish, WeCoTin is arguably inapplicable to Koalish. Further, WeCoTin and, to the best of our knowledge, every other product configurator, lacks intrinsic support for versions of, e.g., components.

Consequently, we set out to develop tool support dedicated to configurable software product families. This section presents three prototypes. First, we will discuss a configuration tool developed for configuring Linux Familiar for a line of handheld devices. Thereafter, we will discuss a modelling tool that enables the creation of models of configurable software product families. Finally, we will discuss a configurator tool, *Kumbang Configurator* that can be used to search for configurators of software product family modelled using Kumbang.

4.2.1 Confuse – Configuration of Compaq iPAQ

Katariina Vuorio and Tero Kojo

The Confuse Configurator was implemented to demonstrate the feasibility of configuring the operating system of a handheld device. In more detail, Confuse showed that Linux Familiar, the operating system of Compaq iPAQ handheld devices, can be automatically configured, including reconfiguration, and installed.

The configuration environment consists of a configurator running on a desktop PC, and of the iPAQ, the software of which is to be configured. The PC and the iPAQ are connected via a TCP/IP network. The configurator provides an easy-to-used, web based user interface. Both the PC and the iPAQ run Linux operating systems.

The configuration process follows the following pattern. First, the current configuration of the iPAQ can be loaded in the configuration. Thereafter, the user can repeatedly modify the configuration by adding or removing packages. The configurator can be at any point used to check the current configuration for consistency, i.e., is it valid in the sense that it works properly. Nothing is installed to the iPAQ until the user explicitly decides to do; when she decides to install the configuration, only one click is needed. Next, the configurator compares the current configuration to the new one, and installs and removes the necessary packages to make the iPAQ's configuration match the new configuration. A configurator screenshot is presented in Figure 4.4.

The reason for running the configurator on the PC instead of the iPAQ is that the handheld device provides insufficient resources for performing the reasoning tasks necessary to check that the configuration is consistent; such checks are needed in order to avoid installing an invalid configuration. Also, a desktop PC enables creating a more easily accessible user interface.

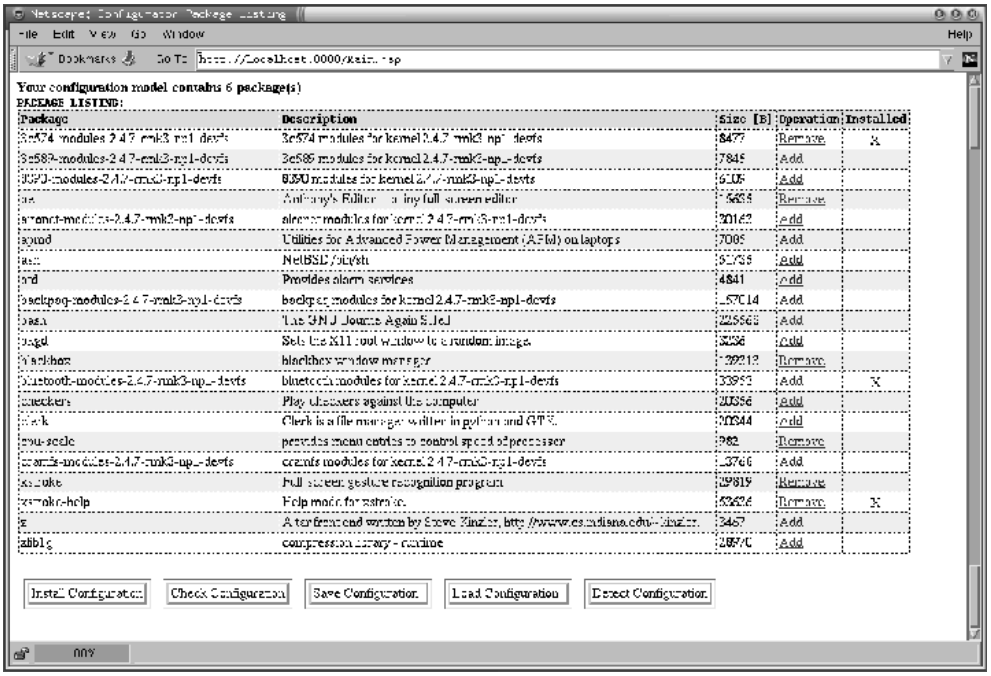

Figure 4.4: The user interface of Confuse Configurator

4.2.2 Comet GCMT modelling tool

Tero Kojo and Katariina Vuorio

Another important form of tool support for configurable software product families is support the *modelling task*. The purpose of a modelling tool is to provide an easy way of creating *configuration models*. A configuration model is a description of the whole product family, and it describes all alternative product individuals. Such a model can be used for designing and documenting the family, and as the basis for configuring individual products with a configurator tool, see next subsection. Without a modelling tool, all configuration models would have to be written manually.

The Comet GCMT (Graphical Configuration Modeling Tool) is a tool that provides a graphical user interface which can be used to create new configuration models, and to edit existing configuration ones. Comet GCMT was the first prototype of a modelling tool for configurable software product families.

The Comet GCMT consists of two main parts; a GCMT GUI (Graphical User Interface) and the GCMT engine. The GCMT is designed so that the GCMT GUI can be easily replaced, to enable creating different representations of configuration models. To put it in another way, a graphical user interface plugged in the GCMT engine, only creates a visual representation of the data stored and manipulated by the engine.

Figure 4.5 illustrates the first GCMT GUI developed.

However, as the ideas related to modelling concepts became more clear, a need to build another user interface to the configuration modelling tool emerged. Thus a new GUI was created. This new GUI resembles many UML (Unified Modeling Language) editors, since the

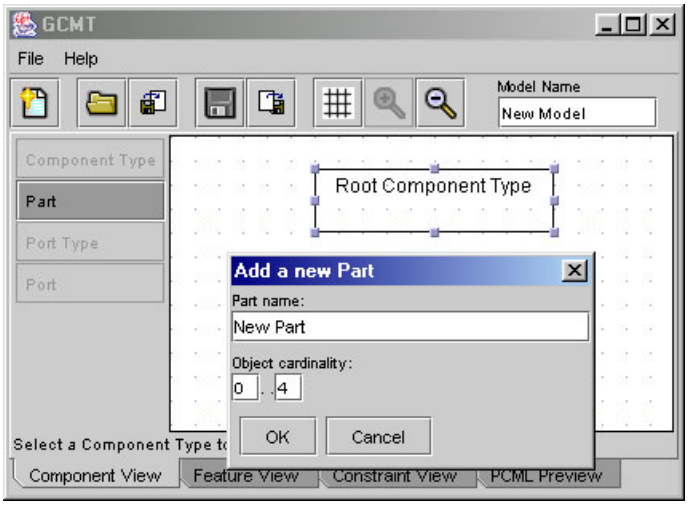

Figure 4.5: The initial user interface for Comet GCMT

user can draw UML-like diagrams of product families and assign dependencies like *requires* and *conflicts* in the diagram. The new GUI is illustrated in Figure 4.6.

Comet CGMT showed that it is possible to implemente tool support for creating models of configurable software product lines.

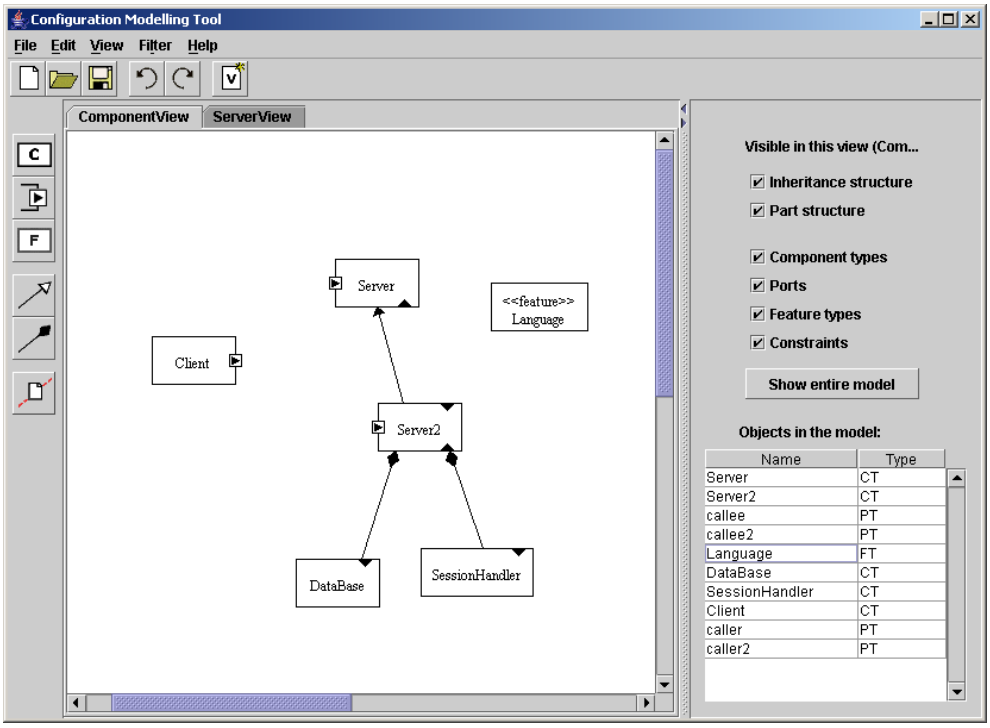

Figure 4.6: An improved user interface of Comet GCMT

4.2.3 Kumbang Configurator

Varvana Myllärniemi, Timo Asikainen, Tomi Männistö, and Timo Soininen

Kumbang Configurator is a tool intended for deriving product individuals from configurable software product families. The tool is based on Kumbang (Section 3.4), and concequently includes both architecture- and feature-based modelling facilities. It is designed specially for software domain, but it employs techniques established in the product configuration domain. Figure 4.7 illustrates the user interface of the tool.

The purpose of Kumbang configurator is to support the user in the configuration task, i.e., finding a valid product individual matching her specific needs. Towards this goal, Kumbang Configurator provides a graphical user interface. The need for a configurator tool is motivated by the fact that the configuration task, especially when configurations are large, is a complex and error-prone activity. There typically exists complex dependencies between elements that would be extremely hard to resolve manually.

The tool takes as an input a configuration model, and is able to illustrate it in its graphical user interface. Next, the user has the possibility to modify the configuration and to select alternatives that best suite her requirements. For example, the user might reason "I want component *client* connected to component *server* through interface *caller*" or "I want attribute *bandwidth* in feature *connection* to have value *high*", and consequently enter these choices

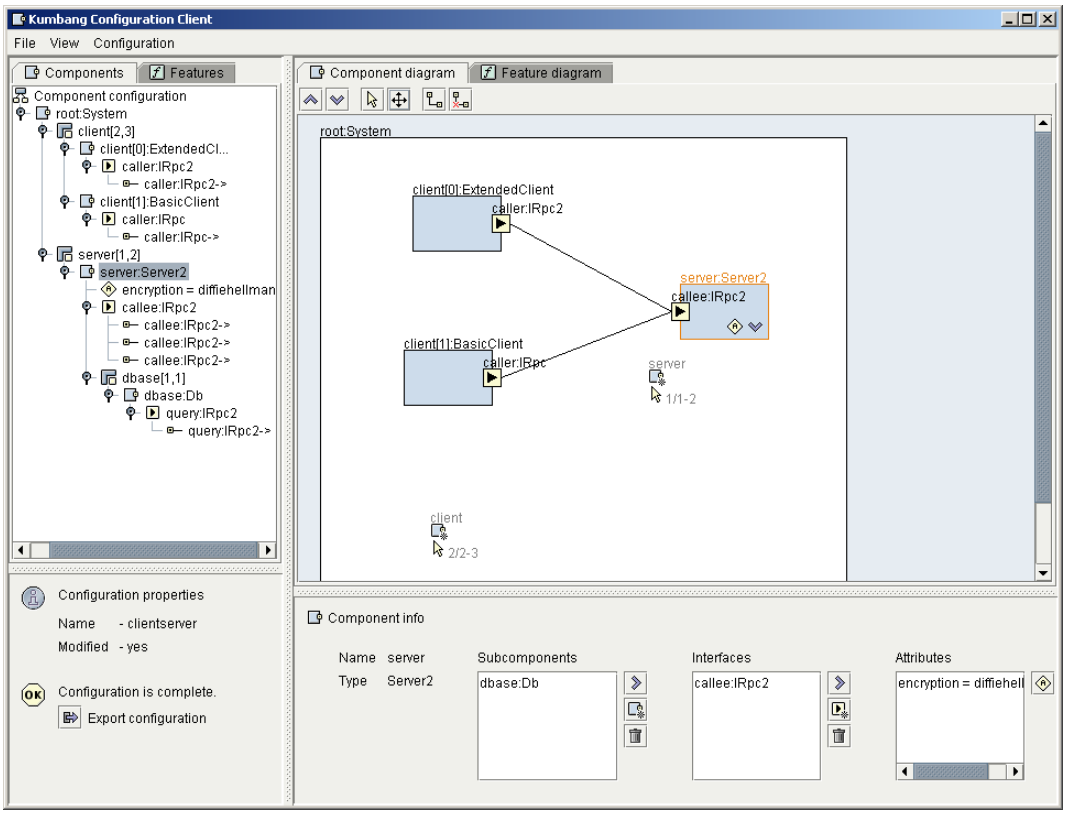

Figure 4.7: A screenshot from the Kumbang configurator tool

using the graphical user interface. After each choice entered, the tool checks the *validity* of the configuration. The validity of the configuration includes *completeness* (all necessary elements are in the configuration) and *consistency* (no rules of the model have been violated). In addition, the tool can also deduct some of the consequences of the decisions made so far, and make changes based on these deductions. The checking and deductions are implemented by providing the models formal semantics by mapping them to WCRL (Weight Constraint Rule Language) (Simons *et al.*, 2002), and utilising an existing inference tool operating on WCRL, *smodels* (Simons *et al.*, 2002).

After the configuration is completed, the tool produces a description of the corresponding product individual. This description can then be used for assembling the resulting product from the existing assets of the family.

The tool follows a distributed client-server architecture: a client includes a graphical user interface. A server, in turn may be used to store multiple configuration models, and serve multiple clients. This enables both distributed configuration, and centralized management of the configuration models; the configuration models need not be distributed to the clients, and can hence be easily managed.

More details of the Kumbang Configurator can be found in (Myllärniemi *et al.*, 2004).

Chapter 5

Dissemination and International Collaboration

In this chapter, we discuss how the results of the project have been disseminated.

5.1 Conferences

During the project, the members of the project group have both participated in organising conferences, and actively participated in other conferences.

5.1.1 Organising conferences

During the project, members of the project group have been engaged in organising the following conferences and events.

Software variability management

Tomi Männistö was the chair of a workshop titled *Workshop on Software Variability Management for Product Derivation—Towards Tool Support*, together with Jan Bosch (Männistö & Bosch, 2004). He is also a member of the program comittee of the *2nd Groningen Workshop on Software Variability Management*. As their names imply, both workshops are devoted to issues related to software variability, the former especially from the tool support point of view.

Configuration workshop

A workshop devoted issues related to configurable products, both software and non-software, has been held annually since 2001 in conjunction with the ECAI (European Conference on Artificial Intelligence, even years) and IJCAI (International Joint Conference on Artificial Intelligence, odd years) conferences. Timo Soininen was the chair of 2001 workshop. In addition, he has been a member of the organising and program committees of all four workshops. Further, Tomi Männistö was a member of the program committee of the 2003 workshop.

5.1.2 Participating in conferences

- 23rd International Conference on Software Engineering (ICSE), May 12–19, 2001, Toronto, Ontario, Canada
  Presentation: (Männistö *et al.*, 2001b) in *Tenth International Workshop on Software Configuration Management (SCM-10)* held in conjunction with the conference

- 17th International Joint Conference on Artificial Intelligence (IJCAI), August 4–10, 2001, Seattle, Washington, USA
  Presentation: (Männistö *et al.*, 2001a) in the *Configuration workshop* of the conference

- 24th International Conference on Software Engineering (ICSE), May 19–25, 2002, Orlando, Florida, USA

- 15th European Conference on Artificial Intelligence (ECAI), July 22–26, 2002, Lyon, France
  Presentations: (Asikainen *et al.*, 2002) and (Ylinen *et al.*, 2002) in the *Configuration workshop* of the conference

- 21st International Conference on Conceptual Modelling (ER), October 7–11, 2002, Tampere, Finland

- Software Variability Management Workshop, February 13–14, 2003, Groningen, The Netherlands
  Presentation: (Asikainen *et al.*, 2003c)

- 25th International Conference on Software Engineering (ICSE), May 3–10, 2003, Portland, Oregon, USA
  Presentation: (Kojo *et al.*, 2003) in *Eleventh International Workshop on Software Configuration Management (SCM-11)* held in conjunction with the conference

- 18th International Joint Conference on Artificial Intelligence (IJCAI), August 9–15, 2003, Acapulco, Mexico
  Presentation: (Asikainen *et al.*, 2003b) in *Configuration workshop* of the conference

- Fifth International Workshop on Product Family Engineering (PFE-5), November 4–6, 2003, Siena, Italy
  Presentations: (Asikainen *et al.*, 2003a) and (Raatikainen *et al.*, 2004a)

- International Conference on Economic, Technical and Organisational Aspects of Product Configuration Systems (PETO), June 28–29, 2004, Copenhagen, Denmark

- 16th European Conference on Artificial Intelligence (ECAI), August 22–27, 2004, Valencia, Spain
  Presentation: (Asikainen *et al.*, 2004b) in *Configuration workshop* of the conference

- Software Product Line Conference (SPLC), August 30–September 2, 2004, Boston, Massachusetts, USA
  Presentations: (Asikainen *et al.*, 2004c) and (Myllärniemi *et al.*, 2004) in *Workshop*

on Software Variability Management for Product Derivation—Towards Tool Support, a workshop held in conjunction with the conference

5.2 Talks

The members of the project group have given a number of talks related to the topic of the project.

- Talk on Configurable software product families, Nokia Research Center (NRC), 2001, Boston, Massachusetts, USA
  Given by Tomi Männistö.

- Full-day seminar on software architecture and software product families for continuing education course, 2002, Espoo, Finland
  Lectured by Tomi Männistö

- Full-day seminar on software product families for industrial audience, 2002, Espoo, Finland
  Lectured by Tomi Männistö

- Half-day seminar on software product families for Nordea (a bank), 2002, Finland
  Lectured by Tomi Männistö

- Lectures on special issues in software product families for industrial audience
  Lectured by Tomi Männistö

- Seminar on latest trends in product configuration, May 22, 2003, Espoo, Finland
  In this seminar held in conjunction with WeCoTin project, Tomi Männistö gave a lecture on variability management in software product families, Mikko Raatikainen on the product family survey, and Tero Kojo a presentation and demonstration on configuring Linux Familiar over multiple releases.

- SoftaProfessional Summit, 2003, Espoo, Finland
  Tomi Männistö gave a lecture on software product families and architectures.

- Seminar on software product families and reuse for industrial audience, 2004, Espoo, Finland
  Lectured by Tomi Männistö.

5.3 Teaching

During the project, members in the project group have been engaged in a number of teaching activities at the Helsinki University of Technology.

- T-76.150 Software architecture (3 cr)
  Full-term undergraduate course concentrating on various aspects related to software architecture and software product families. Lectured by Tomi Männistö during fall terms 2001-2004. Timo Asikainen and Mikko Raatikainen have given guest lectures at the course. Katariina Vuorio and Varvana Myllärniemi have been assistants for the course.

- T-76.650 Special seminar on designing software architectures /
  T-76.270 Special seminar on mastering quality attributes in software architectures
  Full-term undergraduate seminar that purports to provide the participants with a clear and concrete understanding of the design process for software architectures. The seminar was held during fall 2002 and spring 2004. Tomi Männistö has been the responsible teacher of the seminar at both times, and Varvana Myllärniemi was the assistant during spring 2004.

- T-76.611 Software design and specification methods (2 cr)
  Full-term undergraduate course that covers methods related to specifying the requirements for software systems, and designing them; mainly concentrated on the Unified Modelling Language (UML). Timo Asikainen has been the assistant for the course during spring terms 2002-2004. In addition, during the 2004 course he gave a guest lecture on version 2.0 of the UML.

- T-76.614 Software configuration management (2cr)
  Full-term undergraduate course focusing on software configuration management (SCM). Tero Kojo lectured the course during spring terms 2003 and 2004, and was assistant for the course during spring 2002. The responsible teacher of the course is Tomi Männistö.

- T-86.150 Special Assignment in Information Technology (2-6 cr)
  During the Sarcous project, Timo Soininen has supervised a number of individual assignments written related to software product families.

- T-86.165 Seminar in Product Data Technology (2-6 cr)
  Full-term undergraduate seminar concentrating on various aspects of product data management. The topic of the seminar varies. Spring 2003 the topic of the seminar was *product families from information technology point of view*, and spring 2004 *software product families and tool support for product families*. The responsible teacher of the seminar are Timo Soininen and Mikko Raatikainen.

5.4 Reviewing

The members of the project group have acted as reviewers in a number of international journals and conferences related to the topic of project.

Tomi Männistö has reviewed articles for the following journals: *Journal of Computing and Information Science in Engineering*, *Science of Computer Programming*, and *Software Practice and Experience*; and for the *4th Product Family Engineering Workshop (PFE-4)*.

Timo Soininen has acted as a referee for the following journals: *Science of Computer Programming* and *Concurrent Engineering: Research and Applications Journal*; and for the workshop on *Software Variability Management* held in Groningen, The Netherlands, 2003.

Timo Asikainen has acted as a reviewer for the journal *Software Practice and Experience*, and as a peer reviewer for the workshop on *Software Variability Management* held in Groningen, The Netherlands, 2003.

Mikko Raatikainen was a peer reviewer for the special issue on *Software Variability: Process and Management* of *International Journal of Software Process: Improvement and Practice (SPIP)*.

5.5 Other

Tomi Männistö is a member of the IFIP (International Federation for Information Processing) WG-2.10 Software Architecture working group, and has actively participated in its meetings.

Chapter 6

Publications

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, Soininen Timo, & Männistö Tomi. 2002. Representing Software Product Family Architecture Using a Configuration Ontology. *In: Configuration workshop of the 15th European Conference on Artificial Intelligence (ECAI 2002)*.

Asikainen Timo, Soininen Timo, & Männistö Tomi. 2003a. A Koala-Based Approach for Modelling and Deploying Configurable Software Product Families. *In: Proceedings of the 5th International Workshop on Product Family Engineering (PFE-5). Lecture Notes in Computer Science 3014.*

Asikainen Timo, Soininen Timo, & Männistö Tomi. 2003b. A Koala-Based Ontology fro Configurable Software Product Families. *In: Configuration Workshop of 18th International Conference on Artificial Intelligence (IJCAI-03)*.

Asikainen Timo, Soininen Timo, & Männistö Tomi. 2003c. Towards Managing Variability Using Software Product Family Architecture Models and Product Configurators. *In: Proceedings of Software Variability Management Workshop. IWI preprint 2003-7-01*.

Asikainen Timo, Männistö Tomi, & Soininen Timo. 2004a. Forfamel: Feature Modelling for Configurable Software Product Families. *In: 2nd Groningen Workshop on Software Variability Management (submitted)*.

Asikainen Timo, Männistö Tomi, & Soininen Timo. 2004b. Representing Feature Models of Software Product Families Using a Configuration Ontology. *In: Configuration workshop of the 16th European Conference on Artificial Intelligence (ECAI 2004)*.

Asikainen Timo, Männistö Tomi, & Soininen Timo. 2004c. Using a Configurator for Modelling and Configuring Software Product Lines Based on Feature Models. *In: Proceedings of Software Variability Management for Product Derivation – Towards Tool Support, a workshop in SPLC 2004.*

Kojo Tero, Soininen Timo, & Männistö Tomi. 2003. Towards Intelligent Support for Managing Evolution of Configurable Software Product Families. *Pages 86–101 of: Proceedings of the 11th International Workshop on Software Configuration Management (SCM-11). Lecture Notes in Computer Science 2469*.

Männistö Tomi, Soininen Timo, & Sulonen Reijo. 2000. Configurable Software Product Families. *In: Configuration workshop of the 14th European Conference on Artificial Intelligence (ECAI 2000)*.

Männistö Tomi, Soininen Timo, & Sulonen Reijo. 2001a. Modelling Configurable Products and Software Product Families. *In: Configuration Workshop of 17th International Joint Conference on Artificial Intelligence (IJCAI-01)*.

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

Männistö Tomi, & Bosch Jan (eds). 2004. *Software Variability Management for Product Derivation—Towards Tool Support*. SoberIT Technical Reports.

Myllärniemi Varvana, Asikainen Timo, Männistö Tomi, & Soininen Timo. 2004. Tool for Configuring Product Individuals from Configurable Software Product Families. *In: Proceedings of Software Variability Management for Product Derivation - Towards Tool Support, a workshop in SPLC 2004*.

Raatikainen Mikko. 2003. *A Research Instrument for an Empirical Study of Software Product Families*. M.Sc.Tech. thesis, Helsinki University of Technology.

Raatikainen Mikko, Soininen Timo, Männistö Tomi, & Mattila Antti. 2004a. A Case Study of Two Configurable Software Product Families. *In: Proceedings of the 5th International Workshop on Product Family Engineering (PFE-5). Lecture Notes in Computer Science 3014.*

Raatikainen Mikko, Männistö Tomi, & Soininen Timo. 2004b. Towards a Scientific Approach to Study Software Product Families in Industry. *In: 2nd Groningen Workshop on Software Variability Management (submitted)*.

Raatikainen Mikko, Soininen Timo, & Männistö Tomi. 2005. Characterizing Product Derivation in the Configurable Software Product Family. *Software Process: Improvement and Practices*, **to appear**.

Simons Patrik, Niemelä Ilkka, & Soininen Timo. 2002. Extending and Implementing the Stable Model Semantics. *Artificial Intelligence*, **138**(1-2), 181–234.

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

Ylinen Katariina, Männistö Tomi, & Soininen Timo. 2002. Configuring Software Product with Traditional Methods - Case Linux Familiar. *In: Configuration workshop of the 15th European Conference on Artificial Intelligence (ECAI 2002)*.

References

Allen Robert, & Garlan David. 1997. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology,* **6**(3), 213–249.

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, Soininen Timo, & Männistö Tomi. 2002. Representing Software Product Family Architectures Using a Configuration Ontology. *In: Configuration workshop of the 15th European Conference on Artificial Intelligence (ECAI 2002)*.

Asikainen Timo, Soininen Timo, & Männistö Tomi. 2003a. A Koala-Based Approach for Modelling and Deploying Configurable Software Product Families. *In: Proceedings of the 5th International Workshop on Product Family Engineering (PFE-5). Lecture Notes in Computer Science 3014.*

Asikainen Timo, Soininen Timo, & Männistö Tomi. 2003b. A Koala-Based Ontology fro Configurable Software Product Families. *In: Configuration Workshop of 18th International Conference on Artificial Intelligence (IJCAI-03)*.

Asikainen Timo, Soininen Timo, & Männistö Tomi. 2003c. Towards Managing Variability Using Software Product Family Architecture Models and Product Configurators. *In: Proceedings of Software Variability Management Workshop. IWI preprint 2003-7-01*.

Asikainen Timo, Männistö Tomi, & Soininen Timo. 2004a. Forfamel: Feature Modelling for Configurable Software Product Families. *In: 2nd Groningen Workshop on Software Variability Management (submitted)*.

Asikainen Timo, Männistö Tomi, & Soininen Timo. 2004b. Representing Feature Models of Software Product Families Using a Configuration Ontology. *In: Configuration workshop of the 16th European Conference on Artificial Intelligence (ECAI 2004)*.

Asikainen Timo, Männistö Tomi, & Soininen Timo. 2004c. Using a Configurator for Modelling and Configuring Software Product Lines Based on Feature Models. *In: Proceedings of Software Variability Management for Product Derivation—Towards Tool Support, a workshop in SPLC 2004.*

Bosch Jan. 2000. *Design and Use of Software Architecture*. Addison-Wesley.

Clements Paul, & Northrop Linda M. 2001. *Software Product Lines: Practices and Patterns*. Addison-Wesley.

Cohen Sholom. 2002. *Product Line State of the Practice Report*. Tech. rept. CMU/SEI-2002-TN-017. Software Engineering Institute (SEI), Carnegie Mellon University.

Conradi R., & Westfechtel B. 1996. Configuring Versioned Software Products. *Pages 88–109 of: Proceedings of the International Conference on Software Engineering (ICSE-18)*.

Eisenhardt Kathleen M. 1989. Building Theories from Case Study Research. *Academy of Management Review*, **14**(4), 532–550.

Frakes William B., & Fox Christopher J. 1995. Sixteen Questions About Software Reuse. *Communications of the ACM*, **38**(7), 75–87.

Garlan David. 2001. Software Architecture. *In: Encyclopedia of Software Engineering*. Wiley & Sons.

Garlan David, Monroe Robert T., & Wile David. 1997. Acme: An Architecture Description Interchange Language. *In: Proceedings of CASCON'97*.

Knauber Peter, Bermejo Jesus, Böckle Günter, do Prado Leite Julio Cesar Sampaio, van der Linden Frank, Northrop Linda M., Stark Michael, & Weiss David. 2002. Quantifying Product Line Benefits. *Pages 155–163 of: Proceedings of the 4th International Workshop on Product Family Engineering (PFE-4). Lecture Notes in Computer Science 2290*.

Kojo Tero, Soininen Timo, & Männistö Tomi. 2003. Towards Intelligent Support for Managing Evolution of Configurable Software Product Families. *Pages 86–101 of: Proceedings of the 11th International Workshop on Software Configuration Management (SCM-11). Lecture Notes in Computer Science 2469*.

Männistö Tomi, & Bosch Jan (eds). 2004. *Software Variability Management for Product Derivation—Towards Tool Support*. SoberIT Technical Reports.

Männistö Tomi, Soininen Timo, & Sulonen Reijo. 2000. Configurable Software Product Families. *In: Configuration workshop of the 14th European Conference on Artificial Intelligence (ECAI 2000)*.

Männistö Tomi, Soininen Timo, & Sulonen Reijo. 2001a. Modelling Configurable Products and Software Product Families. *In: Configuration Workshop of 17th International Joint Conference on Artificial Intelligence (IJCAI-01)*.

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

Medvidovic Nenad, & Taylor Richard M. 2000. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, **26**(1), 70–93.

Morisio Maurizio, Ezran Michel, & Tully Colin. 2002. Success and Failure Factors in Software Reuse. *IEEE Transactions on Software Engineering*, **28**(4), 340–357.

Myllärniemi Varvana, Asikainen Timo, Männistö Tomi, & Soininen Timo. 2004. Tool for Configuring Product Individuals from Configurable Software Product Families. *In: Proceedings of Software Variability Management for Product Derivation—Towards Tool Support, a workshop in SPLC 2004*.

Parnas David L. 1976. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, **17**(4), 40–52.

Raatikainen Mikko. 2003. *A Research Instrument for an Empirical Study of Software Product Families*. M.Sc.Tech. thesis, Helsinki University of Technology.

Raatikainen Mikko, Soininen Timo, Männistö Tomi, & Mattila Antti. 2004a. A Case Study of Two Configurable Software Product Families. *In: Proceedings of the 5th International Workshop on Product Family Engineering (PFE-5). Lecture Notes in Computer Science 3014.*

Raatikainen Mikko, Männistö Tomi, & Soininen Timo. 2004b. CASFIS—Approach for studying software product families in industry. *In: 2nd Groningen Workshop on Software Variability Management (submitted)*.

Raatikainen Mikko, Soininen Timo, & Männistö Tomi. 2005. Characterizing Product Derivation in the Configurable Software Product Family. *Software Process: Improvement and Practices*, **to appear**.

Simons Patrik, Niemelä Ilkka, & Soininen Timo. 2002. Extending and Implementing the Stable Model Semantics. *Artificial Intelligence*, **138**(1-2), 181–234.

Soininen Timo, Tiihonen Juha, Männistö Tomi, & Sulonen Reijo. 1998. Towards a General Ontology of Configuration. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing (AIEDAM)*, **12**(4), 357–372.

Strauss Anselm, & Corbin Juliet. 1998. *Basics of Qualitative Research*. 2 edn. Sage.

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

Tracz Will. 1988. Software Reuse Myths. *ACM SIGSOFT Software Engineering Notes*, **13**(1), 17–21.

van der Hoek A, Hall R. S., Heimbigner D., & Wolf A. L. 1997. Software Release Management. *Pages 159–175 of: Proceedings of the European Software Engineering Conference ESEC/FSE 1997*.

van der Linden Frank. 2002. Software Product Families in Europe: The Esaps and Cafe projects. *IEEE Software*, **19**(4), 41–49.

van Ommering Rob, van der Linden Frank, Kramer Jeff, & Magee Jeff. 2000. The Koala Component Model for Consumer Electronics Software. *IEEE Computer*, **33**(3), 78–85.

Yin Robert K. 1994. *Case study Research*. 2nd edn. Sage.

Ylinen Katariina, Männistö Tomi, & Soininen Timo. 2002. Configuring Software Product with Traditional Methods - Case Linux Familiar. *In: Configuration workshop of the 15th European Conference on Artificial Intelligence (ECAI 2002)*.