Configurable Software Product Families

Tomi Männistö[1], **Timo Soininen**[1] and **Reijo Sulonen**[1]

Abstract. Product configuration is a specific area of research and business for mechanical (and electrical) products. However, configurable software products have not attracted as much interest. This paper outlines the concept of configurable software product families covering millions of variants from which product individuals are configured to meet particular customer needs. Solutions to managing such software products are sought from experiences with mechanical products and expressed here in the form of a research agenda.

1 INTRODUCTION

This paper investigates software as a configurable product family that may include millions of variants. Thus, we aim at providing methods and tools for a software engineering paradigm in which instances of software are created in a routine manner. This creation would be based on a predefined model (or architecture) that describes all variants and the required knowledge for selecting functional combinations of components. Such paradigm becomes important, for example, when software is embedded in a product that is configurable and the software must adapt to the hardware configuration. If the available memory is limited, the loaded software cannot simply include all possible variability and dynamically adapt to the hardware. Examples of such products are tomorrow's mobile terminals. Similar strategies are also currently sought for enterprise resource planning (ERP) systems, which are large configurable information system packages [1].

In this paper, we chart a research agenda starting from the experiences in product configuration of mechanical and electrical products, which are called *traditional products* in the following. We begin with a brief introduction to product configuration and software engineering, especially software architectures. However, we assume that the reader is familiar with product configuration and thus the introduction to it is minimal.

Our goal for the research work is to find description methods for software product families. The methods should be understandable to software engineers with no special skills in formal methods or logic. In addition, the description of a software product family of an industrial company should be manageable both in complexity and in size. This restricts the modeling essentially to the design level, as very deep models tend to be extremely large. Furthermore, the used description language should allow the models to be processed by computers, which requires strict tradeoffs between the expressivity and complexity of the underlying concepts, as the more powerful logical formalisms may in practice be infeasible.

With respect to previous work in software engineering, a natural counterpart to which this research should be contrasted is modeling and applying software architectures. We aim at rather limited models if compared to the most general approaches in software engineering but, on the other hand, we aim at general concepts that are not specific to any particular software domain.

2 PRODUCT CONFIGURATION

A *configurable product* is adapted to the needs of a particular customer in a *configuration process* using predesigned *components* and a predefined *configuration model.* A *configuration task* is thus to find a suitable variant from the search space defined by the configuration model. The output of a *configuration process* is a *configuration,* which is an adequate description of the *product individual* so that it can be manufactured (see Figure 1) [2,3,4,5,6,7].

3 SOFTWARE ENGINEERING

In this section, a brief look is taken at the basic concepts in software engineering, in particular those of software architectures. We begin briefly with component based software, move on to software architectures and concentrate there on software architecture description languages and domain specific software architectures, which provide the central point of reference for this paper.

Within component-based software engineering (CBSE), definitions of a *software component* include [8]:

- nearly independent and replaceable part of a system with a clear function in the context of a well-defined architecture
- dynamically bindable package that is accessed through documented interfaces
- unit of composition with contractually specified interfaces
- business component representing an autonomous business concept or process.

In many cases, components are seen as rather independent units and it is required that truly composable systems allow connecting system components into a whole in ways not foreseen by the original developers of the components [9].

Software architecture is a term understood in many different ways, typically meaning the structure of components of a software system, including the relationships and guidelines for design and management of evolution [10]. According to Moriconi et al. [11], a software architecture is represented by the following concepts:

- component
- *interface* that denotes a logical point of interaction between a component and its environment
- *connector,* relating interface points, components or both
- *configuration,* which is a collection of constraints that wire objects into a specific architecture
- *mapping* from the language of an abstract architecture to the language of concrete architecture
- *architectural style*

Architectural style is defined by a collection of conventions for a class of software architectures. Style is thus more a general theory for a subfield of software engineering. Common architectural styles

[1] Helsinki University of Technology, TAI Research Centre and Laboratory of Information Processing Science, P.O. Box 9555, FIN-02015 HUT, Finland. Email: {Tomi.Mannisto, Timo.Soininen, Reijo.Sulonen}@hut.fi

include pipe-filter, batch-sequential, blackboard, implicit invocation (event-based) and client-server [11]. Formal methods allow analyzing properties of styles and result to a set of general theorems about all systems in the family [12].

Software architecture description languages (ADL) are used to support architecture-based system development. *A system architecture* or *architectural model* is specified by a set of components, connectors, a configuration and a set of constraints and is written in an architecture description language [13]. An architectural model may apply to a single system or to a family of systems in a domain; the latter is referred to as a *generic architecture* or *domain specific software architecture* (DSSA*)* [14], which comprises of [15]:

- *reference architecture* describing a computational framework for a domain of applications
- *component library* of reusable chunks of domain expertise
- *application configuration method* for configuring components to meet particular application requirements

The focus of this paper is on software product families that include large variation. Variety in software product families can be implemented by an approach called *customization,* in which a 'universal' software product is adapted to behave as any specific variant [16]. However, the size of the software product increases because it contains all the variability of the product.

An alternative approach is to use preprocessor directives to optionally include pieces of source code. In this approach, however, the big picture is easily lost, as the representation of variation is not explicit but is embedded in the source code. Variability may also be achieved by *modularization,* in which the variants are produced by selecting appropriate components to the family architecture [15,16].

When large software products are adapted to different customers, a typical approach is "copy and paste", also called *cloning*. That is, an existing variant of the software product is taken as a basis and then modified accordingly. This approach has its drawbacks since in duplication and ad doc modification of architectural components the original ideas behind the architecture are easily lost, which in consequence deteriorates the overall product architecture [17].

4 CONFIGURING SOFTWARE

4.1 Feasibility of Configuring Software

Some issues specific to software might make configurable software products infeasible. For example, including extraneous components does not typically increase the cost of a product individual. This enables in many cases the selling of software product individuals that contain all possible features. Sometimes, however, the available memory is a limiting resource. In such case, it may be necessary to carefully select the components in a particular configuration simply because otherwise the product individuals would not fit into the memory.

In mobile communication, there are also limitations in bandwidth, which may become an important factor if the software product individual is transferred via a wireless communication channel. Therefore, it may be important to load the exact software variant for the hardware in question and take into account the hardware and other software options already installed into the product individual to avoid unnecessary usage of bandwidth.

The current generation of ERP systems relies on a monolithic software architecture in which customer requirements are met by a large number of parameters, options and configurable functionality. However, a minimalist strategy based on components is an alternate way to meet situation-specific requirements [1].

The common point in all these cases is that customer specific variation of software is needed but it should be implemented by other means than single monolithic software product.

4.2 Comparison to Mechanical Products

There are three major product processes: development, order-delivery and after-sales. The processes and their results are illustrated in Figure 1. For configurable products the order delivery process is required to be smooth and not to include any design work, which is carried out in the development process. Many companies manufacturing project-like traditional products have recently sought ways towards this kind of operation, i.e., product configuration. Similarly, software engineering can find ways towards order-delivery process that would provide customer specific solutions without programming.

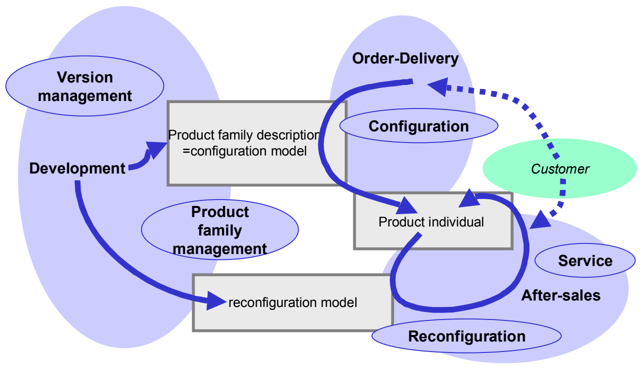

Figure 1. Basic product processes.

The generic software architectures, especially DSSAs, are close relatives to product configuration of traditional products. In both fields the term component means very much the same and also concepts 'port' and 'connector' are used, for example. In component based software, the components are very often seen as independent entities used in manner not thought at the time they were designed. In traditional products there are also such components, e.g., screws, bolts, capacitors and resistors. These, however, are typically not relevant to product configuration, in which components are larger chunks, also called modules, designed with the particular product family in mind. Of approaches to software variety, the closest to product configuration is modularization.

Furthermore, in DSSAs, the reference architecture with component library corresponds to configuration model and application configuration method to configuration algorithm of traditional products. Configuration models of traditional products are expressed by special languages designed for configuration purposes, which have the similar basis as ADLs of software architectures. Configuration languages, however, are typically applicable to all configurable products; they are not targeted to specific type of configurable products. It is not clear what are the right concepts for modeling software product families in general. Are components and first-class components the right concepts [18]; or should the configuration model capture the dynamic behavior of the system [19]; or should the configuration model be based on business processes [20]?

It would be in principle possible to model the physics behind the design of a traditional product. For example, one could include kinematics and fluid or energy flow equations and constraints in the configuration model. This, however, is hardly ever done, as most companies do not have resources to build configuration models from physical principles Consequently, the configuration knowledge is typically at surface, design level. Thus, the fact that a configuration model describes only correct product individuals cannot be derived from the model—it is based on the designers' capabilities of designing functioning product families. The development of some software architecture styles in the form of general theory of software engineering resembles the approaches to develop a general theory of design.

5 PROPOSAL FOR A RESEARCH AGENDA

Our proposal is based on lesson learned in research with traditional products. The modeling and management of configuration knowledge of traditional products is difficult even with a static information. That is, for example, without analyzing whether a product individual fulfils some kinematic conditions. For configuring software, capturing the behavior of software has been proposed [19]. That is an important line of research, but unlike it, we begin here with a static situation. Regarding configuration of software product families, this means that we do not suggest starting from the general theory of software architectures. We thus intend to investigate a small part of the software architectures, which includes research on architecture description languages and domain specific software architectures in a context where large variety is central. Our work belongs to an emerging research area of software product lines in which the first international conferences are currently being organized (see http://www.sei.cmu.edu/plp/conf/SPLC.html). Our idea is to approach software configuration with the methods and tools developed for mechanical products. We aim to analyze how software products can be treated with them.

Our approach to managing software product families assumes

- a need for customer specific adaptation in a relatively routine configuration process. For example, because of restrictions in the size of the available memory.
- the software product to consist of components (or modules) that have clear interfaces
- existence of domain specific software architecture, or in other words, a configuration model, that describes the variants of the software product family
- a language for modeling the above mentioned components and the architecture, i.e., a configuration modeling language.

The management of a software product family would be done independently of the details of the process producing the software. This process may include selection of correct software modules, setting values for pre-processor directives, textual means (e.g., macros) for modifying the source code, selecting module versions from version management tools, creation of scripts for compiling and linking the executable, etc. The point is that the management is based on a configuration model at the architectural level. That model serves as a tool for the development and management of product family and for the actual configuration of software product individuals. The research tries to provide answers to, e.g., the following questions:

- How should the architectures and components of software product families and their evolution be modeled?
- What kind of intelligent support for re-using architectures and components and configuring software can be offered?
- How does the (dynamic) reconfiguration affect the situation? What does it enable?

ACKNOWLEDGEMENTS

We gratefully acknowledge the financial support from Technology Development Centre of Finland (Tekes) and Helsinki Graduate School of Computer Science and Engineering (HeCSE).

REFERENCES

[1] K. Kumar and J. van Hillegersberg, 'Enterprise resource planning—experiences and evolution', *Communications of the ACM,* **43**, 22–26, (2000).

[2] S. Mittal and F. Frayman, 'Towards a generic model of configuration tasks', *in: Proc. of the 11th International Joint Conference on Artificial Intelligence (IJCAI),* 1395–1401, 1989.

[3] T. Männistö, H. Peltonen, and R. Sulonen, 'View to product configuration knowledge modelling and evolution', *in: Configuration—papers from the 1996 AAAI Fall Symposium (AAAI technical report FS-96–03)*, B. Faltings and E.C. Freuder, eds. AAAI Press, 111–118, 1996.

[4] T. Darr, D. McGuinness, and M. Klein, Special Issue on Configuration Design. *AI EDAM* **12,** (1998).

[5] B. Faltings and E.C. Freuder, Special Issue on Configuration. *IEEE intelligent systems & their applications* **13,** (1998).

[6] H. Peltonen, T. Männistö, T. Soininen*, et al,* 'Concepts for modelling configurable products', *in: Proc. of the Product Data Technology Days,* Quality Marketing Services, 189–196, 1998.

[7] T. Soininen, J. Tiihonen, T. Männistö, and R. Sulonen, 'Towards a General Ontology of Configuration', *AI EDAM,* **12**, 357–372, (1998).

[8] A.W. Brown and K.C. Wallnau, 'The current state of CBSE', *IEEE Software,* **15**, 37–46, (1998).

[9] M. Shaw, R. DeLine, D.V. Klein*, et al,* 'Abstractions for software architecture and tools to support them', *IEEE Transactions on software engineering,* **21**, 314–335, (1995).

[10] D. Garlan and D.E. Perry, 'Introduction to the special issue on software architecture', *IEEE Transactions on software engineering,* **21**, 269–274, (1995).

[11] M. Moriconi, X. Qian, and R.A. Riemenschneider, 'Correct architecture refinement', *IEEE Transactions on software engineering,* **21**, 356–372, (1995).

[12] G.D. Abowd, R. Allen, and D. Garlan, 'Formalizing style to understand descriptions of software architecture', *ACM Transactions on software engineering and methodology,* **4**, 319–364, (1995).

[13] J.J.P. Tsai, A. Liu, E. Juan, and S. Avinash, 'Knowledge-based software architectures: acquisition, specification, and verification', *IEEE Transactions on knowledge and data engineering,* **11**, 187–201, (1999).

[14] P. Kogut and P. Clements, 'Features of architecture description languages', *in: Proceedings of Software Technology Conference,* 1995.

[15] B. Hayes-Roth, K. Pfelger, P. Lalanda, P. Morignot, and M. Blabanovic, 'A domain-specific software architecture for adaptive intelligent systems', *IEEE Transactions on software engineering,* **21**, 288–301, (1995).

[16] A. Karhinen, A. Ran, and T. Tallgren, 'Configuring design for reuse', *in: Proceedings of International Conference on Software Engineering, ICSE'97,* 701–710, 1997.

[17] D. Dikel, D. Kane, S. Ornburn, W. Loftus, and J. Wilson, 'Applying software product-line architecture', *Computer,* **30**, 49–61, (1997).

[18] J. Bosch, 'Evolution and composition of resusable assets in product-line architectures: a case study', *in: Software architecture*, P. Donohoe, ed. Kluwer Academic Publishers, 321–339, 1999.

[19] C. Kühn, 'Requirements for configuring complex software-based systems', *in: Configuration—Papers from the 1999 AAAI workshop*, B. Faltings, E.C. Freuder, G.E. Friedrich and A. Felfernig, eds. AAAI Press, 11–16, 1999.

[20] A.-W. Scheer and F. Habermann, 'Making ERP a Success', *Communications of the ACM,* **43**, 57–61, (2000).