Using a Configurator for Modelling and Configuring Software Product Lines based on Feature Models

Timo Asikainen, Tomi Männistö, and Timo Soininen

Helsinki University of Technology, Software Business and Engineering Institute
P.O. Box 9210, FIN-02015 HUT, Finland
{timo.asikainen, tomi.mannisto, timo.soininen}@hut.fi
http://www.soberit.hut.fi/

Abstract. We show how WeCoTin, an academic prototype product configurator originally designed for non-software products, can be used to create and edit feature models of software product lines. Further, we show that WeCoTin enables the easy configuration of software product lines, i.e., generating descriptions of valid products in the product line.

1 Introduction

A software product line may comprise a very large number of different individual systems, and means to distinguish between these systems are required. One such means is to create a *feature model* that describes all the possible combinations of *features* that products in a specific product line may deliver [1]. A large number of feature modelling methods exists [2-5]. They give somewhat divergent definitions for feature, ranging from "attribute of a system that directly affects end-users" [2] to "distinguishable characteristic of a concept that is relevant to some stakeholder" [5].

Feature models of industrial software product lines can be very large [6-8]: e.g., [6] mentions a feature model with about 500 features. Consequently, creating and managing such models can become burdensome. Further, the task of selecting a valid and suitable set of features for a single system can become very difficult to solve; we call this task the *configuration task*. Some attempts have been made towards solving these problems [7, 9], but no generally applicable and accepted solution has been found. Hence, additional research is needed.

Problems similar to the above-mentioned ones have been previously encountered in the context of *configurable* (non-software) *products*. These problems have been studied in the *product configuration* domain, a sub-domain of artificial intelligence [10-12]. The studies have resulted in a rough consensus about the concepts relevant for describing configurable products [13, 14], and a number of supporting systems, *product configurators*, have been developed and deployed in companies [15].

In the remainder of this paper, we will show how a particular product configurator, WeCoTin, can be used to support the tasks of creating and maintaining feature models of software product lines, and that of configuring individual systems in software product lines based on their features.

The remainder of this paper is structured as follows. Next, in Section 2 we will briefly discuss feature modelling in general and in particular the feature modelling concepts that are used as the baseline in this paper. Thereafter, in Section 3, we will provide an overview of the product configuration domain, including the most important results achieved in it, and describe the functionality of WeCoTin. A translation from the feature modelling concepts to the concepts of WeCoTin in follows in Section 4. Discussion and comparison to previous work follows in Section 5. Conclusions and an outline for further in Section 6 round up the paper.

2 Features and Feature Modelling

In this section, we describe the feature modelling concepts that is used as the baseline in this paper. The concepts are based on a number of feature modelling methods: the basis of these concepts is FODA [2], the first and still widely-cited feature modelling method, and a feature modelling method introduced by Czarnecki et al. [16], which introduces some interesting extensions to feature modelling.

There is no single, commonly accepted definition for feature. However, according to [5], the two most popular definitions are: 1) an end user visible characteristic of a system, and 2) a distinguishable characteristic of a concept (e.g., system, component, and so on) that is relevant to some stakeholder of the concept.

A feature model is a description of the commonalities and differences between the individual software systems in a software product family. In other words, a feature model defines a set of valid feature combinations. Each such valid feature combination can serve as a specification of a software system.

Structurally, feature model is a rooted tree. The nodes of the tree are features. Each feature is identified by a *name*. The root of the tree is called *root feature*. Each feature may have a number of other feature as its *subfeatures*, each of which must have a unique name. Each subfeature has a *cardinality* that specifies how many instances of the subfeature may occur in a valid feature combination. Syntactically, cardinality is a set of integer values. There are two important special cases: a *mandatory subfeature* has cardinality {1}, and an *optional subfeature* cardinality {0, 1}.

Example. Throughout the paper, we will use a running example introduced in Fig. 1 (a); Fig. 1 (b) contains a legend of the notation used. The example is a feature model of an advanced text editor: in addition to the standard functionality of a text editor, our editor includes sophisticated features, namely equation editing and optionally importing data from SQL databases. ■

A subfeature may have the form of an *alternative feature*. The difference with ordinary subfeatures is that instead of a single feature, there are multiple features that form a group. The semantics of alternative features is the same as that of ordinary subfeatures, with the extension that any feature in the group may be used in a valid feature combination, as long as the bounds specified by the cardinality are obeyed. Notice that at this point, our notion of cardinality is different from that in [16]; however, in [17] the authors of [16] have changed their notion of cardinality to that used in this paper.

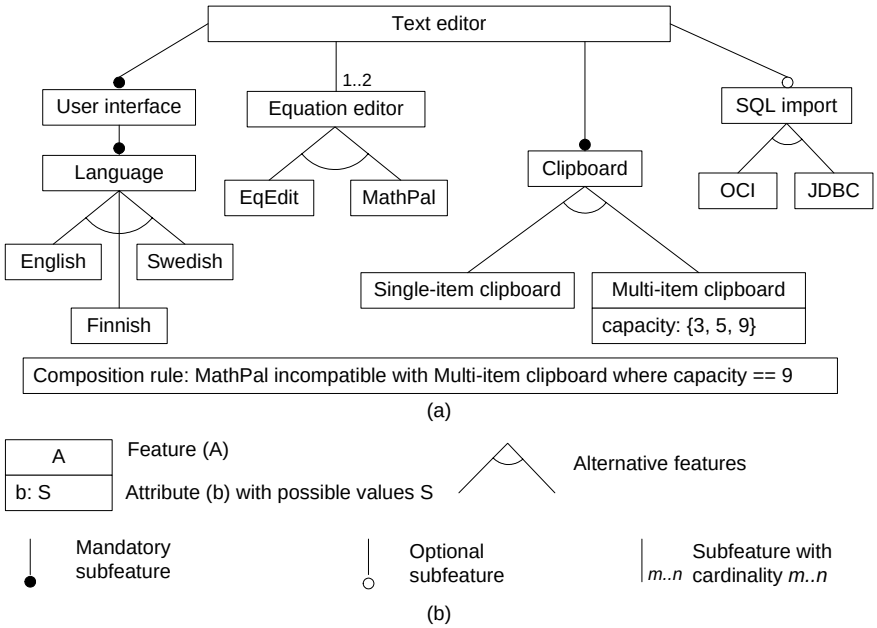

Fig. 1 (a) A feature model for an advanced text editor. (b) Legend of the notation used.

Example. There are four occurrences of alternative features in our example. **Language** defines an alternative feature with three alternatives, **English**, **Finnish**, and **Swedish**. The intuition is that the language in the user interface must be one of these three languages. Similarly, a **Clipboard** is either a **Single-item clipboard**, or a **Multi-item clipboard**. Further, **OCI** and **JDBC** are the two alternative means to implement **SQL import**. However, the alternative subfeature of **Equation editor** has different semantics: the choice is not exclusive, but both subfeatures can be selected; this is denoted by the cardinality 1..2 of the subfeature. Intuitively, a **Text editor** may contain either one or both of the two possible equation editors.■

A feature may define a number of *attributes*. An attribute represents a characteristic of a feature, and is identified by a *name*. Each attribute has a *value type*, which specifies the values the attribute may take in a valid feature combination.

Example. In our running example, feature **Multi-item clipboard** has an attribute name **capacity**. The possible values for this attribute are 3, 5, and 9; the intuition is that a multi-item clipboard may hold a maximum of either 3, 5, or 9 items at a time. ■

Finally, feature models may be augmented with *composition rules*. Composition rules come in two forms, namely *requires* and *incompatible with*. Both take two operands, which may be references to the presence of a feature, or attribute values of features operated on a comparison operator.

Example. There is one composition rule in our example: if there is a **MathPal** in a text editor, there may not be a **Multi-item clipboard** with **capacity** equal to 9. ■

3 Product Configuration and Product Configurators

In this section, we provide an overview of *product configuration* research, a subfield of artificial intelligence [10] that has inspired the approach presented in this paper.

3.1 Overview of Product Configuration Research

Research in product configuration domain is based on the notion of *configurable product*: a configurable product is such a product that each product individual is adapted to the needs of a particular customer order. Historically, the configurable products studied in the domain have been non-software products, typically mechanical and electronics products. A fundamental characteristic of a configurable product is that it has a modular structure: product individuals consist of pre-designed components, and different product variant can be produced by selecting different components. [12]

The possibilities for adapting the configurable product are predefined in a *configuration model* that specifies the entities that may appear in a configuration and the rules on how the entities can be combined. A specification of a product individual, *configuration*, is produced in the *configuration task* based on the configuration model and a set of customer needs.

Efficient knowledge-based information systems, *product configurators*, have become an important and successful application of artificial intelligence techniques for companies selling products adapted to customer needs [10]. The basic functionality of a configurator is to support a user in generating a valid and suitable configuration with respect to a given configuration model matching his specific needs. Examples of the kinds of support provided are: a configurator represents the choices provided by the underlying configuration model in a way that enables the user to easily enter his needs. Further, the configurator makes deductions based on the needs the user has entered so far: it automatically makes choices required by earlier choices, prevents the user from making incompatible choices, and is able to generate a configuration based on choices made so far, if such a configuration exists. The above-described functionality is based on representing configuration model using knowledge representation languages with declarative, formal semantics, and efficient, sound, and complete inference tools operating on these.

Product configurators are not merely a theoretical endeavour: they have been applied to a number of different kinds of products; perhaps the most challenging kinds of products have been telephone switching systems at Siemens [15], and other kind of telecommunication products at AT&T [18]. At Siemens, the problem instances have been considerably large: typically, a configuration has included tens of thousands of components with hundreds of thousands of attributes, and over 100,000 connection points. Finally, product configurators have become parts of ERP (Enterprise Resource Planning) systems, such as SAP [19], and Baan [20], and are available as embeddable products (see, e.g., http://www.ilog.fr/products/configurator).

3.2 Product Configuration Modelling Language (PCML) and WeCoTin

WeCoTin [21] operates on PCML (Product Configuration Modelling Language), a language for modelling configurable products. The conceptual basis of PCML, in turn, is a subset of a conceptualisation of configuration knowledge represented in [13].

A PCML model is a description of a configurable product. Such a model has a number of configurations, and each of these describes a valid and suitable instance of the configurable product.

The most important modelling concept in PCML is *component type*. A component type intentionally specifies the properties of their *instances*; each component instance is directly of exactly one type, and indirectly of all the supertypes of this type. Component types can be defined a compositional structure using *part definition*s. A part definition consists of a part name, a (nonempty) set of possible component types, and a cardinality. In addition, component types can be defined *properties* and *constraints*. A *property definition* contains *property name*, a *property value type*. A constraint, in turn, specifies a condition that must hold for the instances of the type. Finally, the component types are organised in a taxonomy, where the subtypes of a type inherit the properties (part definitions, property definitions, and constraints) of their supertype. A *configuration* consists of a set of component instances and relations between them.

Fig. 2 depicts the one view of the architecture of WeCoTin. The configurator consists of two main subsystems, one of which is used for modelling, and the other for configuration. Using the terms in the software product line domain, the modelling support can be used as part of the domain engineering phase; deployment support is intended for the application engineering phase. The idea is that the result from the configuration task is an abstract description of an individual system that can be used as input to realisation tools, such as make. In the following, we will describe the two subsystems of WeCoTin; for more details, the reader should refer to [21].

Modelling Support. The purpose of the modelling support is to enable the easy creation and maintenance of configuration models. WeCoTin supports this in various ways. Foremost, WeCoTin includes a graphical modelling tool, called ModellingTool, for creating and editing taxonomies of component types and the compositional structure of components in terms of part definitions.

Once a model has been created, WeCoTin can be used to translate the model into Weigh Constraint Rule Language (WCRL) [22], a general-purpose knowledge representation language similar to logic programs. The translation can be time-consuming, but can be done offline: that is, the model needs to be retranslated only when it is changed, not between executions of successive configuration tasks.

Configuration Support. When using WeCoTin, the user performs the configuration task using a web-based *configuration interface* specific to the configuration model at hand. WeCoTin generates such an interface automatically; no programming is required. The main idea is that each property and part definition is used to generate a *question* that goes into the configuration interface.

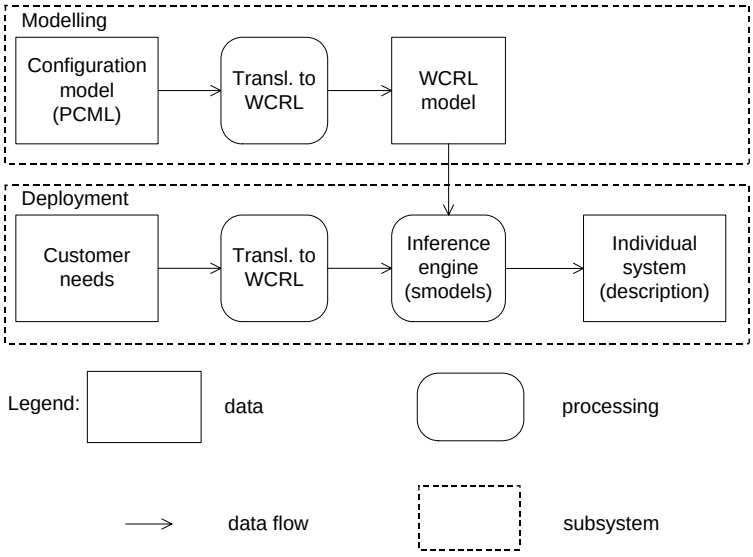

Fig. 2 The basic data flows and processing elements of WeCoTin

An inference tool *smodels* [22] operating on WCRL is used to support the user in the configuration task. First, given a set of customer needs, the inference engine can be used to calculate a *partial stable model*. A partial stable model describes what must be true, what must not be true, and what is still unknown of the configuration satisfying the customer requirements entered so far. The partial stable model can be used, e.g., to prevent a customer from making incompatible choices by disabling alternatives in the configuration interface. Second, at any point, the inference engine can be used to find a configuration that satisfies the customer needs entered so far.

The configuration task is completed when a valid configuration satisfying the needs of the user has been generated.

4 Translating Features into WeCoTin Concepts

In this section, we suggest a translation from the feature modelling concepts of Section 2 to those of the configuration ontology. In more detail, we will show how different concepts and relations in feature modelling methods are translated into configuration modelling concepts. An overview of the translation is presented in Table 1.

Features are translated into component type; the root feature is mapped to be the configuration type of the configuration model. The name of the feature becomes the name of the type.

Subfeatures are translated into part definitions. We use the terms *whole-feature* and *subfeature* to refer to the feature containing a subfeature and the subfeature, respectively. The part definition is located in the component type corresponding to the

whole-feature. The name of the part definition is the name of the subfeature. The cardinality of the feature becomes the cardinality of the part definition. The set of possible part types contains a single type, namely the component type to which the subfeature is translated.

Alternative features are likewise translated into part definitions. At this point, there is no obvious choice for the name of the part definition; let us name all such part definitions arbitrarily as **Choice**. The set of possible types consists of the types corresponding to the features in the set. Cardinality is the cardinality of the alternative feature.

Attributes are translated into property definitions in component types. The name of the attribute definitions is simply the name of the attribute.

Example. Fig. 3 illustrates how the WeCoTin ModellingTool can be used to create a configuration model corresponding to the feature model of our running example. As can be seen, the window has been divided into three panes. The upper-left pane contains the component types. The lower-left pane, in turn, contains the part structure of component types. Finally, the right pane, in turn, illustrates detailed information about the currently active element; in this case, it is the subfeature named **Choice** that corresponds to the alternative feature of **Language** in Fig. 2.

Table 1 The translation from feature modelling concepts to PCML concepts

Feature modelling concept	PCML / WeCoTin concept
Feature	Component type
Name	Name
Root feature	Configuration type
Subfeature	Part definition
Cardinality	Cardinality
Alternative feature	Part definition
Alternatives	Possible part type
Cardinality	Cardinality
Attribute	Property definition
Name	Name
Possible values	Possible values
Composition rule	Constraint

Creating the kind of Fig. 3 is easy. First, one creates component types corresponding to the features; this amount to giving type a name and adding the attributes, if any. Second, one nominates the root feature types as *configuration type*. Finally, the subfeature structure can be created simply by dragging the different types into the hierarchy in the lower-left pane; some extra effort is needed to handle the alternative feature groups. ■

Further, Fig. 4 illustrates a configuration interface corresponding to our running example. Such an interface is generated automatically based on the configuration model; hence, there is no additional effort required once the configuration model has been created using the above-described process.

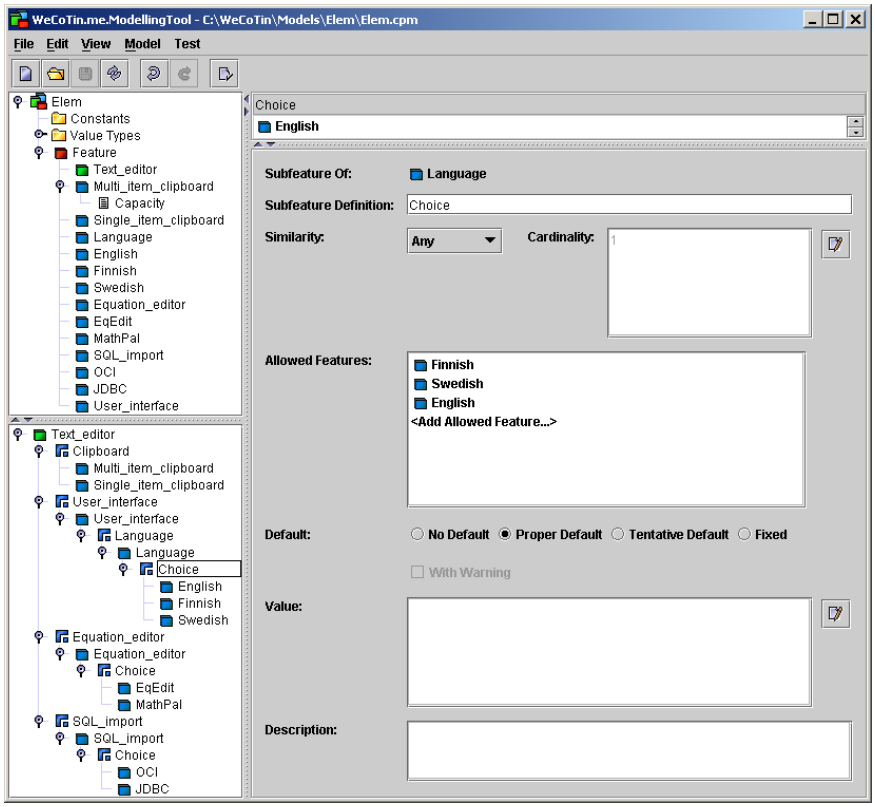

Fig. 3 WeCoTin ModellingTool. In this version of the modelling tool, some concepts have been renamed: component types are called features, part definitions are called subfeatures, and properties are called attributes.

The configuration interface is divided into three panes. The left pane contains the compositional structure and illustrates the choices made so far, and the choices still to be made. In the figure, all the necessary choices have already been made: e.g., it has been decided that the **Clipboard** will be **Multi-item clipboard** with **capacity** 9.

The fact that all the choices has been made is illustrated with a full circle containing an OK sign in the lower-right pane. The same pane also contains a field for a price of the currently made choices; this is zero due to the fact that we have not entered pricing information for our example.

Finally, the upper-right pane contains an example of a configuration question. In more detail, previously it has been decided that there will be one equation editor in the text editor, and now it is to decided whether an **EqEdit** or **MathPal** should be chosen. However, the choice for **MathPal** has been greyed. This is due to the fact that a **Multi-item clipboard** with **capacity** 9 has already been selected, and there is a constraint saying that the **MathPal** is incompatible with this choice, see Fig. 1.

MathPal could, nevertheless be selected, but this would lead to an inconsistent configuration. ■

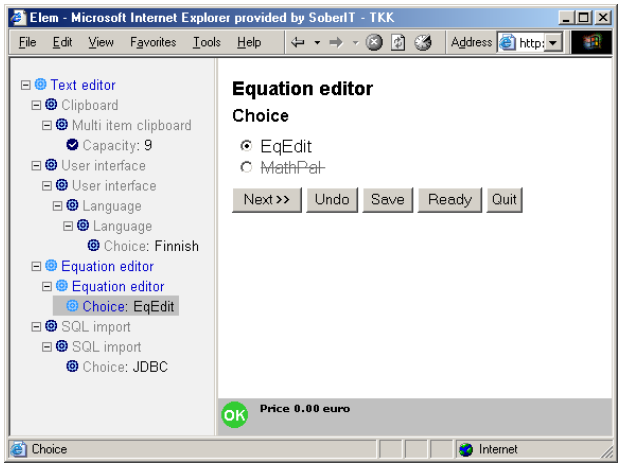

Fig. 4 WeCoTin configuration interface.

5 Discussion and Comparison with Previous Work

In this section, we will first iterate on some issues arising from the translation presented above, and thereafter contrast this paper with previous work.

The central observation to be made from the above-presented description of WeCoTin and translation from feature models to WeCoTin is that it is feasible to create feature models using WeCoTin and to use it in the deployment process to come up with valid and suitable feature combinations. This is encouraging, as feature models have been a prominent method for describing software product lines [2].

However, WeCoTin is not perfectly suited for modelling and configuring features. Perhaps the most important reason for this is that WeCoTin distinguishes between the definition of component types and their use in composition hierarchy, whereas feature modelling methods make no such distinction. An implication of this that there is a need to separately create the component types corresponding to features, and to organise them into the composition hierarchy; in the configuration interface, the same phenomenon is manifested as the feature names appearing twice, first as the name of the part and thereafter as the (only possible) type for that part, see Fig. 4.

However, the above-discussed distinction may also be used to derive advantages. It is not too difficult to imagine situations in which it would be beneficial to distinguish between the role of a subfeature from the particular subfeature filling in the role. Also, the fact that features are made into types enables the easy and consistent

reuse of features at multiple places in the same feature model, or in different feature models.

There is a number of problematic issues inherent to product configuration. The most important such problem is that the complexity of the computational tasks related to configuration is potentially very high [22], which may result in intolerable running times and memory consumption. However, the high complexity has been successfully managed for large products previously [15, 23], which suggests that the complexity is not necessarily a problem in the context of software product families either.

Research closely related to the work presented in this paper has been conducted earlier. However, to the best of our knowledge, the idea of applying existing product configurators to feature models has not been considered previously. Hence, we consider this the main contribution of our paper.

In [24], Krebs and Wolter iterate on the idea of modelling evolving product families using feature models. Their work is similar to ours in that it uses feature modelling concepts as its conceptual basis. However, unlike we, they do not suggest that an existing configurator could be used to carry out the configuration task.

Beuche et al. have introduced an approach called CONSUL for creating and configuring feature models [12]. In their approach, software product families are modelled not only using features, but components as well. However, what seems to differentiate their work from knowledge-based configuration is the lack of automated inference and its advantages.

In [25], Mannion shows how requirement models (and feature models) can be encoded as propositional formulae. A single system is represented as a set of requirements that the system fulfils. A *valid* system is such a system that satisfies the formula representing the product line. Further, a product line is defined to be valid if the line contains at least one valid system. He provides tool support for checking the validity of single systems and product line models using Prolog and for counting the number of valid single systems and enumerating all of them. What differentiates our approach from that of Mannion is that ours includes extensive support for creating feature models, whereas he describes no such support. The same applies to the configuration task: WeCoTin is designed to support it in various ways, while Mannion seems to provide no support for the task.

In addition to WeCoTin, a large number of product configurators have been created, e.g. [19, 20]. Many of these configurators would probably have been equally well suited for our purposes as WeCoTin; the reason for using WeCoTin was that it was freely and easily available to us for this purpose.

6 Conclusions and Further Work

Above, we have shown how WeCoTin, an existing product configurator designed originally for non-software products, can be used to create feature models of software product lines and to generate valid and suitable (with respect to a given set of customer needs) feature combinations based on these models. The fact that WeCoTin was originally designed to support a concept set different from feature modelling concepts resulted in some anomalies.

More research is required to assess the practical applicability of the results presented in this paper. A natural first step would be to try using WeCoTin in industrial contexts, and thereby empirically assess its applicability. Based on the findings from the assessments, the modelling concepts and their supporting tools should be further improved. Finally, as the computational tasks related to configuration are potentially very complex, the computational feasibility of configuring software should be analysed through practical experiments and theoretical complexity analysis.

Acknowledgements

We gratefully acknowledge the financial support from the Academy of Finland (project number 51394) and National Technology Agency of Finland (Tekes). We also thank Mr. Andreas Anderson for his assistance in installing and running WeCoTin.

References

1. Kang, K., Lee, J., Donohoe, P.: Feature-oriented Product Line Engineering. IEEE Software 19(4) (2002) 58-65
2. Kang, K., Cohen, S. G., Hess, J. A., Novak, W. E., Peterson, S. A.: Feature-Oriented Domain Analysis (FODA) - Feasibility Study . CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University (1990)
3. Kang, K., Kim, S., Lee, J., Kim, K., Shin, E., Huh, M.: FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. Annals of Software Engineering 5(1998) 143-168
4. Griss, M., Favaro, J., d'Alessandro, M.: Integrating Feature Modelling with the RSEB. In: Proceedings of the Fifth International Conference on Software ReuseIEEE Computer Society (1998) 76-85
5. Czarnecki, K. and Eisenecker, U. W.: Generative Programming. Addison-Wesley, Boston (MA) (2000)
6. Lee, K., Kang, K., Lee, J.: Concepts and Guidelines of Feature Modeling for Product Line Software Engineering. In: Proceedings of the 7th International Conference on Software Reuse. LNCS 2319Springer, Berlin-Heidelberg (2002) 62-77
7. von der Maßen, T., Lichter, H.: RequiLine: A Requirements Engineering Tool for Software Product Lines. In: Prodeedings of the 5th International Workshop on Product Family Engineering. LNCS 3014Springer, Berlin Heidelberg (2003)
8. Fey, D., Fajta, R., Boros, A.: Feature Modeling: A Meta-model to Enhance Usability and Usefulness. In: Chastek, Gary J.: Proceedings of Second International Software Product Line Conference (SPLC2). Lecture Notes in Computer Science 2379, Berlin-Heidelberg (2002) 198-216
9. Beuche, D., Papajewski, H. , Schröder-Preikschaft, W.: Variability Management with Feature Models. In: van Gurp, Jilles and Bosch, Jan: Proceedings of Software Variability Management Workshop. IWI preprint 2003-7-01University of Groningen, Groningen, The Netherlands (2004) 72-82
10. Faltings, B., Freuder, E. C.: Special Issue on Configuration. IEEE Intelligent Systems 14(4) (1998) 29-85

11. Darr, T., Klein, M., McGuinness, D. L.: Special Issue on Configuration Design. Artificial Intelligence for Engineering Design, Analysis and Manufacturing 12(4) (1998) 293-397
12. Soininen, T., Stumptner, M.: Introduction to Special Issue on Configuration. Artificial Intelligence for Engineering Design, Analysis and Manufacturing 17(1-2) (2003) 1-2
13. Soininen, T., Tiihonen, J., Männistö, T., Sulonen, R.: Towards a General Ontology of Configuration. Artificial Intelligence for Engineering Design, Analysis and Manufacturing 12(4) (1998) 357-372
14. Felfernig, A., Friedrich, G., Jannach, D.: UML as Domain Specific Language for the Construction of Knowledge-Based Configuration Systems. International Journal of Software Engineering and Knowledge Engineering 10(4) (2000) 449-469
15. Fleischanderl, G., Friedrich, G., Haselböck, A., Schreiner, H., Stumptner, M.: Configuring Large Systems Using Generative Constraint Satisfaction. IEEE Intelligent Systems 13(4) (1998) 59-68
16. Czarnecki, K., Bednasch, T., Unger, P., Eisenecker, U. W.: Generative Programming for Embedded Software: An Industrial Experience Report. In: ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component EngineeringSpringer-Verlag, Berlin-Heidelberg (2002) 156-172
17. Czarnecki, K., Helsen, S. , Eisenecker, U. W.: Staged Configuration Using Feature Models. In: Nord, Robert L.: Proceedings of Third Software Product Line Conference (SPLC-3)Springer, Berlin-Heidelberg (2004)
18. McGuinness, D. L., Wright, J. R.: An Industrial-Strength Description Logic-Based Configurator Platform. IEEE Intelligent Systems 14(4) (1998) 69-77
19. Haag, A.: Sales Configuration in Business Processes. IEEE Intelligent Systems 13(4) (1998) 78-85
20. Yu, B., Skovgaard, J.: A Configuration Tool to Increase Product Competitiveness. IEEE Intelligent Systems 13(4) (1998) 34-41
21. Tiihonen, J., Soininen, T., Niemelä, I., Sulonen, R.: A Practical Tool for Mass-Customising Configurable Products. In: Proceedings of the International Conference on Engineering Design (ICED'03), Stockholm, Sweden (2003)
22. Simons, P., Niemelä, I., Soininen, T.: Extending and Implementing the Stable Model Semantics. Artificial Intelligence 138(1-2) (2002) 181-234
23. Kojo, T., Männistö, T., Soininen, T.: Towards Intelligent Support for Managing Evolution of Configurable Software Product Families. In: Proceedings of 11th International Workshop on Software Configuration Management (SCM-11), LNCS 2649Springer, Berlin Heidelberg (2003) 86-101
24. Krebs, T., Wolter, K.: Mass Customization for Evolving Product Families. In: Proceedings of PETO, Copenhagen, Denmark (2004)
25. Mannion, M.: Using First-Order Logic for Product Line Model Validation. In: Chastek, Gary J.: Proceedings of the Second International Conference on Software Product Lines (SPLC2). Lecture Notes in Computer Science 2379.Springer, Berlin-Heidelberg (2002) 176-187