Kumbang Configurator–A Configuration Tool for Software Product Families[*]

Varvana Myllärniemi and **Timo Asikainen** and **Tomi Männistö** and **Timo Soininen**
Helsinki University of Technology
Software Business and Engineering Institute (SoberIT)
varvana.myllarniemi@hut.fi

Abstract

This paper presents Kumbang Configurator, a prototype system for deriving product individuals from configurable software product families. Configurable software product families resemble configurable products in that they have a pre-defined structure and can be customised according to customer requirements in a routine manner. The conceptual basis underlying the configurator is Kumbang, a language for modelling configurable software product families from the feature and architectural points of view. Features represent the family from a requirements or functional point of view, and architecture from a technical or structural one. The configurator has been implemented in the Java programming language, and validated with two examples, one of which is based on an industrial case.

1 Introduction

Software product families, or *software product lines*, as they are also called, have emerged as an important paradigm for software reuse [Clements and Northrop, 2001; Bosch, 2000]. A software product family is commonly defined to consist of a *common architecture*, a *set of reusable assets* used in systematically producing individual products, and the *set of products thus produced* [Bosch, 2000].

The most systematic class of software product families can be called *configurable software product families* [Bosch, 2002; Raatikainen *et al.*, 2005]. A configurable software product family has the property that individuals of the family can be deployed in a systematic manner; there is no need for coding within components, and not much need for adding glue code between them; hence, configurable software product families resemble configurable, non-software products. The deployment of individuals is called the *configuration task*. It has been noted that the configuration of software product families task can be burdensome and error-prone [Clements and Northrop, 2001]. Thus there is a need for concrete tool support.

[*]The support of National Technology Agency of Finland is acknowledged.

Product configurators, or *configurators* for short, are tools that support the configuration task in producing a correct product individual. Configurators prevent configuration errors and automate routine tasks. A large number of configurators have been developed, e.g. [Tiihonen *et al.*, 2003], mainly for the purpose of configuring non-software products.

The question whether configurators developed for non-software configurable products can be applied to software product families has been studied in [Asikainen *et al.*, 2003; 2004]. The outcome from these studies has been that the existing tools cover important parts of the configuration models of software product families, but lack support for architectural connections. Hence, new tools are needed.

In this paper, we introduce Kumbang Configurator, a tool supporting the configuration task for configurable software product families. Kumbang Configurator utilises Kumbang modelling language. Kumbang is based on modelling a configurable software product family from two independent, yet mutually related points of view. *Features* are abstractions from the requirements set on the product family. A feature may include a number of *subfeatures* as its constituent elements, and be characterised by a number of *attributes*. The technical aspects of the product family are captured by its *architecture* that is defined in terms of *components*. A component may have a number of other components as its *parts*, and be characterised by a number of attributes. Further, the possibilities for interaction of a component are specified by its *interfaces* and *bindings* between them.

Kumbang Configurator supports the user in the configuration task by providing a graphical user interface through which the user can enter his specific requirements for a product individual. Further, the configurator checks the configuration for *consistency* and *completeness* after each step, and deduces the consequences of the selections made so far. The necessary inferences are implemented using *smodels* [Simons *et al.*, 2002], which is a general-purpose inference tool based the stable model semantics of logic programs.

To the best of our knowledge, a tool supporting modelling concepts similar to Kumbang has not been previously implemented. Hence, Kumbang Configurator is the main contribution of this paper.

The remainder of the paper is organised as follows. Section 2 discusses Kumbang language. Section 3 presents Kumbang Configurator. Further, Section 4 discusses how Kumbang Configurator has been validated with example cases. Thereafter, Section 5 compares Kumbang Configurator with related work. Finally, Section 6 draws conclusions and suggestions for future work.

2 Kumbang Language

In this section, we discuss the Kumbang language and its background. First, we will iterate on *feature modelling* and Forfamel, the feature modelling method in Kumbang. Second, we will give similar treatment to architecture description and Koalish. Finally, we will discuss how Forfamel and Kumbang are integrated in Kumbang.

Feature modelling has become a popular method for modelling requirements of software product families [Kang *et al.*, 1990; Czarnecki and Eisenecker, 2000]. A feature has been defined as a characteristic of a system that is visible to the end-user [Kang *et al.*, 1990], and as a logical unit of behaviour that is specified by a set of functional and quality requirements [Bosch, 2000]. Feature modelling methods are based on organising features into *feature models* that typically take the form of a tree. Such a tree captures the variability of the software product family modelled in terms of its features.

Forfamel [Asikainen, 2004] is a feature modelling method that synthesises existing feature modelling methods with configuration modelling concepts stemming from the product configuration domain, more specifically from [Soininen *et al.*, 1998]. The fundamental modelling element of Forfamel is *feature type*; each feature type intentionally defines the properties of its instances, i.e., *features*. A feature type may include a *subfeature* definition that specifies the number and types of possible feature instances, thus defining the compositional structure of features. Further, a feature type may define *attributes* that characterise its instances. The value range for an attribute instance is attained by defining *attribute value types*. Moreover, a feature type may be defined one or more *supertypes*; a feature type inherits the property definitions of its supertypes. Finally, *constraints* between different combinations of features may be stated, thus restricting the valid combinations of features.

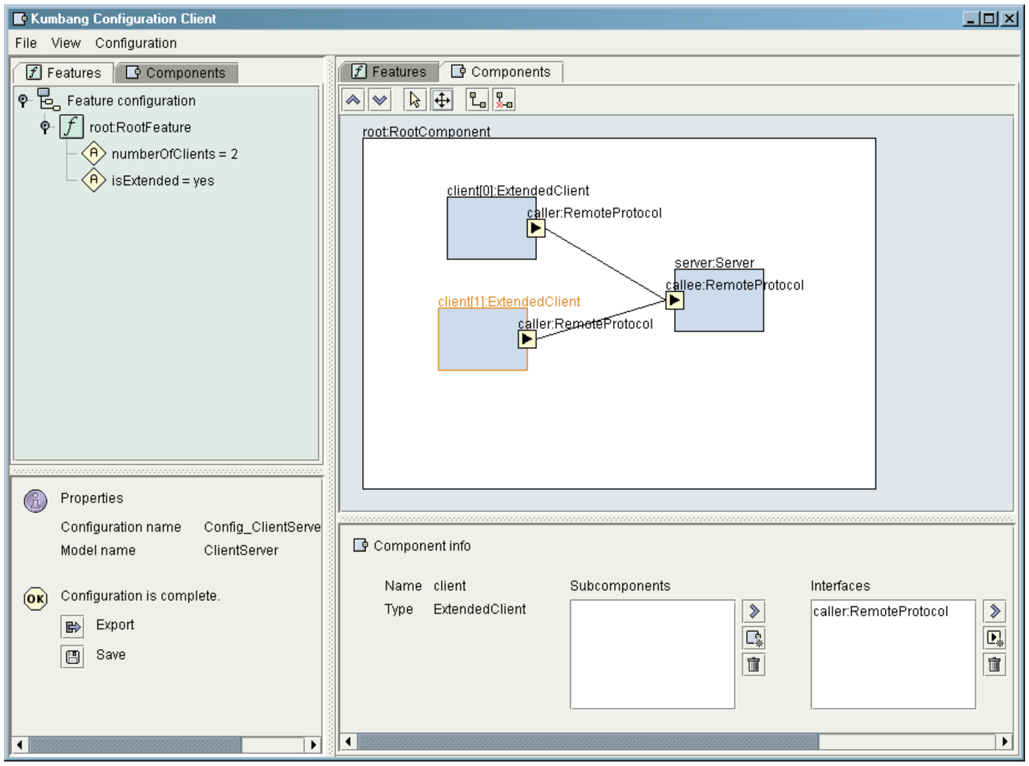
Example. In order to clarify the concepts presented in this paper, we provide a running example. The configuration model in Figure 1 depicts a small client-server system with varying number of clients. The model contains only one feature type *RootFeature*. However, feature type *RootFeature* contains two attribute definitions. Attribute *numberOfClients* is of type *Int2*, which means that its value can be either one or two. Attribute *isExtended* is of type *Boolean* and thus its value can be either *yes* or *no*. □

However, features as are such not sufficient for describing all the relevant aspects of software product families. In more detail, means for describing the technical aspects of the product family are needed. Towards this end, Koalish [Asikainen *et al.*, 2003; Asikainen, 2004] is a method for modelling *software product family architectures*. In other words, Koalish can be used to describe the overall structure of the software product family in terms of its architectural elements. Koalish is based on Koala [van Ommering *et al.*, 2000], a component model and an architecture description language developed and used at Philips Consumer Electronics. Koalish shares its conceptual basis with Koala, but adds several variability mechanisms familiar from the product configuration domain, see e.g. [Soininen *et al.*, 1998]. In more detail, a Koalish model can contain *component* types that are instantiated as component instances. A component type can define the number and types of components as parts under the corresponding component instance. Further, a component type can define the type and direction of *interface* instances contained in the component instance. A *required* interface signals that the functions enlisted in its interface type must be implemented by a *provided* interface with such functions. In order to satisfy required interfaces, interface instances can be *bound* with each other. Also, similarly as in Forfamel, constraints concerning the combinations of architectural elements may be stated. Finally, in a manner similar to Forfamel attribute mechanism, attributes can be used for characterising properties of component instances.

```
Kumbang model ClientServer
  root feature RootFeature
  root component RootComponent

feature type RootFeature {
  attributes
    Int2 numberOfClients;
    Boolean isExtended;
  implementation
    cardinality($.client) = value(numberOfClients);
    value(isExtended) = yes <=>
      for_all(X : $.client) instance_of(X, ExtendedClient);
}

component type RootComponent {
  contains
    (Client, ExtendedClient) client[1-2];
    Server server;
}

component type Client {
  requires RemoteProtocol caller;
}

component type ExtendedClient {
  requires RemoteProtocol caller;
}

component type Server {
  provides RemoteProtocol callee { grounded };
}

interface type RemoteProtocol {
  sendData; checkStatus;
}

attribute type Boolean = { yes, no }
attribute type Int2    = { 1, 2 }
```

Figure 1: A sample Kumbang model that describes a small client-server system.

Example. The running example defines a two-level compositional structure for component instances. Namely, the root component type *RootComponent* defines parts *server* and *client*. Part definition *server* states that a *RootComponent* instance must contain one component of type *Server*, while part definition *client* states that a *RootComponent* instance must contain one or two *Client* or *ExtendedClient* components. Further, due to the interface definition *caller*, a component of type *Client* or *ExtendedClient* contains one required interface of type *RemoteProtocol*. In a similar manner, a *Server*



root:RootFeature
numberOfClients = 2
isExtended = yes

Configuration name	Config_ClientServe
Model name	ClientServer

Configuration is complete.

Name	client
Type	ExtendedClient

Interfaces: caller:RemoteProtocol

Figure 2: A screenshot from Kumbang Configurator user interface. This screenshot illustrates how the configuration model presented in our running example can be configured.

component contains a provided interface of type *RemoteProtocol*. Interface type *RemoteProtocol* consists of two functions, *sendData* and *checkStatus*. □

The Kumbang language combines Forfamel and Koalish into a single modelling language for configurable software product families. Hence, Kumbang enables modelling a software product family simultaneously from a feature and architectural point of view. However, these two views are often related to each other, just like system requirements and architecture are related. Therefore, Kumbang enables specifying how features are *implemented* by architectural entities. In more detail, feature types may include *implementation constraints* that must hold for the architecture in order for the product individual to provide the specific feature. A model of a configurable software product family presented in Kumbang is a *configuration model*, in the sense of [Soininen *et al.*, 1998], of the product family.

Example. The configuration model in our running example is a Kumbang model, and it exemplifies how features and components can be related to each other. The implementation constraints defined in feature type *RootFeature* relate selected attribute values to component configuration. The value of attribute *numberOfClients* is related to the number of client components as parts under *RootComponent*, while value *yes* for attribute *isExtended* is equivalent to all client components being of type *ExtendedClient*. □

3 Kumbang Configurator

This section provides an overview of Kumbang Configurator. Subsection 3.1 presents a brief overview of the functionality provided by the system, while Subsection 3.2 discusses the system architecture. Finally Subsection 3.3 discusses how configuration reasoning has been implemented.

3.1 Provided Functionality

As discussed earlier, Kumbang Configurator is based on Kumbang language; thus it can be used for deriving configurations that contain both features and components. However, the tool also supports Forfamel and Koalish models (see Section 2) as special cases. This means that the tool can be used for deriving configurations with features only, or configurations with components only.

Kumbang Configurator takes a configuration model of a software product family represented in Kumbang as input. The configuration model is used as the basis of the configuration task. The configurator offers a graphical user interface (see Figure 2) through which the user can make *selections* that modify the configuration. In more detail, selections can add or remove features, components, interfaces or bindings and set attribute values.

The graphical user interface provided by Kumbang Configurator visualises the configuration in a way that resembles existing notations known in the software product family community. For example, the visualisation of the component configuration (right side of Figure 2) resembles the graphical notation of Koala [van Ommering *et al.*, 2000]. The user interface has been designed to show all available selections explicitly. Thus the tool can be used without additional in-depth knowledge of the configuration model.

Kumbang Configurator checks whether the configuration is *consistent* and *complete*. A consistent configuration is such that no rules of the configuration model have been violated. In contrast, a complete configuration is such that all necessary selections have been made. Further, Kumbang Configurator deduces the direct consequences of the configuration selections made so far. This means that the tool can automatically add selections implied by previous selections, identify selections conflicting with previous selections, and illustrate these selections in the user interface. Especially, selections in the feature hierarchy may have implications on the architecture, as specified by the implementation constraints in feature types. Consequently, the configuration model may be such that the architecture of the individual is completely deduced from the selections made in the feature hierarchy. This situation corresponds to the typical assumption that requirements are used as a basis for specifying software architecture. On the other hand, the constraints may take the form of equivalence constraints, implying that selections about the architecture may have implications in the feature hierarchy.

Finally, when the configuration is complete, the user of the tool can request a description of the product individual. This description can be used as a basis for building the product from existing software assets. However, Kumbang Configurator does not itself build the product; a separate builder is needed for this purpose.

Example. Figure 2 shows a screenshot from Kumbang Configurator applied to the running example. The current feature configuration is shown on the left as a tree, while a diagram of the architectural configuration is shown on the right. The user of the tool has set attribute *isExtended* to value *yes* and attribute *numberOfClients* to *2*. Based on these two selections, Kumbang Configurator has automatically deduced the entire component configuration. In the component configuration, root component *RootComponent* is composed of two components of type *ExtendedClient* and one of type *Server*. The interfaces of these components have been bound accordingly. □

3.2 System Architecture

Kumbang Configurator follows the distributed client-server architectural style (see Figure 3). The communication between the client and the server happens through Java Remote Method Invocation (RMI). In particular, the server provides a RMI interface for clients. A server may serve multiple clients simultaneously.

The client is implemented in the Java programming language. It provides a graphical user interface and interacts with the user of the tool during the configuration task.

The server is likewise implemented in the Java programming language. It takes care of the configuration reasoning (see Subection 3.3). Since this reasoning utilises *smodels* module, and since *smodels* has been implemented in C++, the

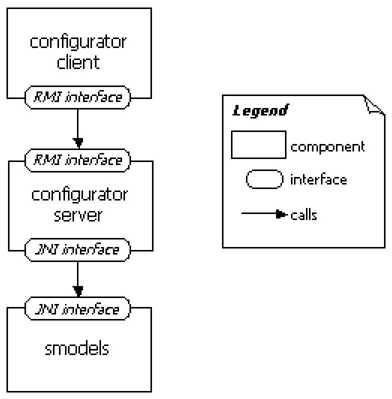


Figure 3: Kumbang Configurator follows the distributed client-server architectural style.

server has to connect to *smodels* module through Java Native Interface (JNI).

There are several reasons for following the client-server style. Firstly, as configuration reasoning can be computationally expensive, the server can run on a dedicated machine. Secondly, the server provides centralised model management. The server stores configuration models in a repository that is accessed by clients. Centralised model management enables separation between domain engineering and application engineering activities: those who perform the configuration task (for example, sales personnel) can use configuration models directly from the repository.

3.3 Configuration Reasoning

The term *configuration reasoning* refers to the inferences required to implement the configurator. In more detail, this includes checking configurations for consistency and completeness with respect to a configuration model and deducing the consequences of the selections made.

To implement the reasoning, the Kumbang model is translated to a form of logic programs; the translation is illustrated in the upper part of Figure 4. In more detail, the Kumbang model is first translated into Weight Constraint Rule

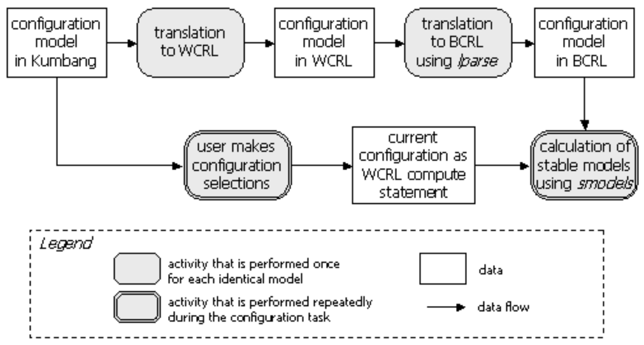

configuration model in Kumbang
translation to WCRL
configuration model in WCRL
translation to BCRL using lparse
configuration model in BCRL
user makes configuration selections
current configuration as WCRL compute statement
calculation of stable models using smodels
Legend
activity that is performed once for each identical model
activity that is performed repeatedly during the configuration task
data
data flow

Figure 4: How Kumbang Configurator uses *smodels* for configuration reasoning.

Language (WCRL) [Simons *et al.*, 2002], a general-purpose knowledge representation language; details of this translation can be found in [Asikainen, 2004]. The resulting WCRL program is further translated into a more restricted form of weight constraint rules, namely Basic Constraint Rule Language (BCRL). This latter translation is carried out by the *lparse* module of the *smodels* system [Simons *et al.*, 2002]. It should be noted that the translation from WCRL to BCRL can be time-consuming, but it only needs to be done after a configuration model is created or changed.

Example. Before the configuration task begins, Kumbang Configurator translates the running example to WCRL. The resulting WCRL model contains 110 weight constraint rules that specify the configuration model. For example, a rule

$$1 \{ \mathit{hasattr}(X, \mathit{isExtended}, V) : \mathit{attrBoolean}(V) \} 1$$
$$\text{:- } \mathit{in}(X), \mathit{instance}(X, \mathit{featRootFeature}).$$

states that if an instance *X* of type *RootFeature* is currently in the configuration, it must have exactly one attribute named *isExtended*, and this attribute must have a value *V* from the range specified by attribute value type *Boolean*. □

The inference steps during the configuration task are illustrated in the lower part of Figure 4. In short, the selections made by the user are translated into *compute statements* that can be combined with the BCRL representation of the configuration model. Using this combination, the *smodels* [Simons *et al.*, 2002] system can be used to compute the consequences, which are in turn fed back in the client and represented in the user interface.

For further information on this topic, please refer to [Myllärniemi, 2005; Asikainen, 2004].

4 Validation

Kumbang Configurator has been validated using two sample cases. The first case represents a significant portion of real life case, and the second is an invented toy example that demonstrates additional aspects of the tool.

The first case originates from Robert Bosch GmbH [MacGregor, 2004]. Robert Bosch GmbH is a company developing various automotive systems containing embedded software. A distinguishing characteristic of automotive industry is the large number of variants. The case used for validation is a part of a model for a car periphery system (CPS), and it has been obtained from a presentation by John MacGregor [MacGregor, 2004]. The original presentation included a demonstration that showed how this particular system has been configured. The configurator tool used in the demonstration has been presented in [Hotz *et al.*, 2004].

However, the first case does not utilise interfaces or bindings. Since these are major contributions of Kumbang Configurator, another case was constructed for this purpose. Although the details of the case were invented for evaluation purposes, the case was motivated by a real-life system. The case describes a distributed weather station network that is used for measuring various weather-related quantities at multiple physical locations.

The authors modelled and configured both cases. Modelling included writing the cases into a Kumbang configuration model. As a result, the first case yielded a configuration model with dozens of feature, component, and attribute types, while the second configuration model was somewhat smaller. The configuration task included deriving various different configurations using Kumbang Configurator. Although systematic performance tests have not yet been run, these cases indicate that the system performance is adequate. (For example, checking the configuration state for the weather station configuration model takes approximately 60ms on a 750MHz Pentium PC.) However, it is yet unknown how Kumbang Configurator can handle very large configurations.

5 Related Work

There exists a number of configurator tools that have been designed to support configuration task. However, only few have been designed for the software domain. To some extent, it is possible to use a configurator designed for traditional mechanical products for configuring software [Asikainen *et al.*, 2004]. However, to our knowledge, none of the existing configurators currently provides the same set of functionality as Kumbang Configurator.

WeCoTin [Tiihonen *et al.*, 2003] is a configurator designed mainly for traditional, mechanical products. It employs many techniques that are also used in Kumbang Configurator, e.g., it uses *smodels* to implement the necessary reasoning. However, there are several differences between WeCoTin and Kumbang Configurator. First and foremost, WeCoTin doesn't recognise connectors or interfaces, which are an essential part of Kumbang Configurator. Secondly, WeCoTin lacks the separation between features and architectural elements. It can be used for modelling both of them, but not in parallel in a same configuration model.

There are a few configurator tools that have been designed for software domain. One of them is presented in [Hotz *et al.*, 2004]; it is built on top of existing product configurators. When comparing the approach in [Hotz *et al.*, 2004] with Kumbang Configurator, one can find several similarities. Both approaches support the separation between features and components, and both provide many similar variability mechanisms. However, there are also several differences. For example, Kumbang Configurator supports connectors and interfaces, whereas [Hotz *et al.*, 2004] does not.

Further, there are software configuration tools that do not originate from the field of configurable product research. Mae [Roshandel *et al.*, 2004] is a system that combines architectural description languages (ADL) with software configuration management (SCM) principles. Due to the fact that both Mae and Kumbang Configurator bring variability mechanisms to ADLs, the architecture-related modelling concepts in these two approaches bear many similarities. However, Mae provides several capabilities, such as *evolution* and *anytime variability*, which are lacking from Kumbang Configurator. In contrast, Mae does not include features. Further, the configuration reasoning mechanisms in Mae are rather limited compared to Kumbang Configurator; for example, validity checking is performed only after the configuration has been constructed.

6 Conclusions and Future Work

This paper presented Kumbang Configurator, which is a prototype tool for configuring product individuals from configurable software product families. Kumbang Configurator is based on modelling language Kumbang, which combines feature-based and architecture-based modelling methods. Thus Kumbang language has been designed for software product family domain. Further, Kumbang Configurator utilises existing inference engine *smodels* for configuration reasoning.

However, we have identified areas that require further research.

In order to ease the modelling task, we need to provide a graphical modelling tool that enables easy creation of Kumbang configuration models. We are currently working on such modelling tool. Further, Kumbang Configurator does not currently support activities after the configuration task, that is, building the source code into an executable artifact. We are investigating on how to relate architecture to actual implementation entities, and how to provide tool support for the build task.

Finally, further empirical knowledge is needed on the performance of Kumbang Configurator. For this purpose, systematic empirical tests are needed. It would be especially interesting to study how connectors affect the performance of the system.

References

[Asikainen *et al.*, 2003] T. Asikainen, T. Soininen, and T. Männistö. A Koala-based ontology for configurable software product families. In *IJCAI 2003 Configuration workshop*, 2003.

[Asikainen *et al.*, 2004] Timo Asikainen, Tomi Männistö, and Timo Soininen. Using a configurator for modelling and configuring software product lines based on feature models. In *Proceedings of the Workshop on Software Variability Management for Product Derivation, at Software Product Line Conference (SPLC3)*, 2004.

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

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

[Bosch, 2002] Jan Bosch. Maturity and evolution in software product lines: Approaches, artefacts and organization. In Gary J. Chastek, editor, *Proceedings of the Second Software Product Line Conference (SPLC2)*, pages 257–271, 2002.

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

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

[Hotz *et al.*, 2004] Lothar Hotz, Thorsten Krebs, and Katharina Wolter. Combining software product lines and structure-based configuration—methods and experiences. In *Proceedings of the Workshop on Software Variability Management for Product Derivation, at Software Product Line Conference (SPLC3)*, 2004.

[Kang *et al.*, 1990] K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak, and A.S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, ADA 235785, Software Engineering Institute, 1990.

[MacGregor, 2004] John MacGregor. CONIPF—configuration in industrial product families. Presentation in Workshop on Software Variability Management for Product Derivation, at Software Product Line Conference (SPLC3), 2004.

[Myllärniemi, 2005] Varvana Myllärniemi. Kumbang Configurator—a tool for configuring software product families. Master's thesis, Helsinki University of Technology, Department of Computer Science and Engineering, 2005.

[Raatikainen *et al.*, 2005] Mikko Raatikainen, Timo Soininen, Tomi Männistö, and Antti Mattila. Characterizing configurable software product families and their derivation. *Software Process: Improvement and Practice*, 10(1), 2005.

[Roshandel *et al.*, 2004] Roshanak Roshandel, Andre van der Hoek, Marija Mikic-Rakic, and Nenad Medvidovic. Mae—a system model and environment for managing architectural evolution. *ACM Transactions on Software Engineering and Methodology*, 18(2):240–276, 2004.

[Simons *et al.*, 2002] Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138:181–234, 2002.

[Soininen *et al.*, 1998] T. Soininen, J. Tiihonen, T. Männistö, and R. Sulonen. Towards a general ontology of configuration. *AI EDAM (Artificial Intelligence for Engineering Design, Analysis and Manufacturing)*, 12(4):357–372, 1998.

[Tiihonen *et al.*, 2003] Juha Tiihonen, Timo Soininen, Ilkka Niemelä, and Reijo Sulonen. A practical tool for mass-customising configurable products. In *Proceedings of the 14th International Conference on Engineering Design (ICED'03)*, 2003.

[van Ommering *et al.*, 2000] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala component model for consumer electronics software. *IEEE Computer*, 33(3):78–85, 2000.