

Configuring Product Families Using Design Spaces

Lars Geyer

System Software Research Group
p.o. box 3049
University of Kaiserslautern
D-67653 Kaiserslautern, Germany
geyer@informatik.uni-kl.de

Abstract. In the product family based application engineering process configuration of the reusable assets is an important issue. Variability introduced during domain engineering has to be resolved to the concrete requirements of the specific application. On the requirements level variability is typically described in feature models as introduced in the FODA approach. On lower levels variability is integrated in a code framework, which is generic enough in order to be instantiated to the different requirements covered by the product family. The gap between these levels has to be bridged by a configuration technique which allows the selection of features on the requirements level resulting in a ready configured code framework which acts as a starting point for the implementation of application specific enhancements which are not part of the product family. In this paper we present our revised notion of Design Spaces, a configuration technique specialized for software product families. The new developments were influenced by the results of an analysis of existing configuration techniques, as well as experiences in the use of the technique in product families. Design Spaces are specialized for the description of classic feature models and their usage during the configuration of generic components.

Keywords: Product Families, Configuration, Design Spaces, Variability, Feature Models

Introduction

Experiences gained in recent years have shown that software development can be done more efficient with software reuse. Similar functionality is implemented only once and the implementation is reused in different applications. On the other hand the handling of software reuse is quite difficult, if it is not done right, the economic effects are wasted.

One promising approach is the implementation of a product line, i.e., a set of systems with related functionality, on base of a product family, i.e., a common reuse infrastructure, which only needs to be adapted to specific requirements of an application. The reuse infrastructure is build out of software assets. The assets are elements on all levels of the software product model, i.e., requirement templates, an architecture, a component set, generic test cases are all assets which can be reused during the development of a family member.

In the context of a product family, typically, the related application domain is scoped, i.e., the relevant area of requirements is fixed. Later, the re-

quirements of the domain within these defined borders are determined. They are further classified into commonalities, i.e., requirements found in all systems of the family, and variabilities, i.e., requirements only found in some systems.

The assets of a product family are implemented in order to fulfill the requirements of the application domain. Therefore the assets are typically generic, offering so-called variation points. Variation points allow the integration of a solution, which implements a specific variant of an associated variability. During the application engineering process the variabilities are configured to the needs of an application. Later, the chosen variants are used for the binding of the variation points to the matching solutions. In this paper we describe a technique that supports this configuration step in the application engineering process. The technique is a further development of the design space technique, which we presented in [Baum98]. Before we go into the details of the technique, we describe the configuration step in the next section. The following section summarizes the requirements for a configuration technique usable in

this context. In section 4 we compare traditional configuration techniques to these requirements, and we describe their deficiencies, which render them rather useless in the context of product family configuration. Section 5 describes the proposed configuration technique, Design Spaces, in detail. The paper concludes with some closing remarks and a look into the future.

Configuring Product Families

The goal of the configuration is the selection of functionalities and qualities for a specific product family member. The selection chooses a subset of the functionalities and qualities integrated into the product family. A product family, thereby, is a set of systems built from a common set of software assets [Bass99]. An asset is either a partial solution, e.g., a component or a design document, or it is knowledge, e.g., a requirements database or test procedures [With96].

The reuse infrastructure of a product family implements a set of requirements, which cover the functionality and the qualities found in the application domain. These requirements are typically classified as common or variable, common requirements, or short commonalities, are part of all systems of the product family while variable requirements, short variabilities, are only found in some systems.

The variable requirements are either optional, i.e., the requirement is part of a system or not, or they comprise a set of variants, a system may contain zero, one or more of the variants. In addition, variabilities may be numeric, i.e., the range of the variability is defined by an interval, out of which the value of the variability has to be chosen. A related type to alternative variability is open variability. In this case the set of alternatives is not finite, new variants may be integrated during application development. In the extreme, no variants are pre-defined; the variability only identifies a spot in the requirements, which has to be specified individually for each family member.

Orthogonal to these types of variability, which are concerned with qualitative aspects of requirements, are quantitative variabilities. A quantitative variability is given, if a specific set of functionality may be integrated into an application more than once in different incarnations, i.e., the basic functionality is configured in different ways. All these alternatives are used in one application. E.g., in the building automation domain, a building contains a lot of rooms. The rooms can be classified into room types. For each room type, the control strategy may be different,

offices would have a strategy that allows an user to change the default values given for light or heating, in a public computer lab, these values might be given by a caretaker. So the different instances of the functionality set have to be configured independent of the others.

The configuration process of a product family is responsible for the selection of a consistent set of variable requirements for a concrete system. Consistent in this context means, that the selected requirements meet the dependencies, which exist between the variable requirements, e.g., if a variable requirement requires the inclusion of another variable requirement, the second one has to be in the selection if the first requirement is chosen.

The assets of the family realize the requirements. Common requirements are integrated into the assets; the usage of an asset in an application simply includes the contained common requirements. The variable requirements, on the other hand, are also integrated into the assets, but they are represented by variation points that allow the adaptation of an asset to the variant chosen for a concrete system. The configured assets build the core of the system. After this configuration step, the assets are further enhanced to meet application specific needs, i.e., system specific requirements, which were not taken into account during the development of the product family, have to be integrated into the assets. After that, the enhanced assets are put together in order to create the application. In a good designed product family, there are only few specific enhancements; so ideally, the product development is reduced to a selection of a consistent set of variable requirements and the instantiation of the assets according to this set of requirements.

In summary, the configuration of a product family comprises two steps. In the first step, the variable requirements of the product family are configured to the needs of the specific application. In a second step, the consistent set of requirements is mapped onto the assets of the family, thereby adapting the assets for the specific application.

In [Beck01], we presented a variability model, which builds the basis for the variability treatment in a product family. The variability model contains all variable requirements of a product family as well as the links to the associated variation points. In [Beck01], we described, how the traceability knowledge concerning variabilities and associated variation points is used for the mapping of requirements to assets. In this paper, we will concentrate on the first step, i.e., the consistent configuration of variability.

Features

The configuration of variable requirements is based upon a feature model. The variable requirements are abstracted to features. A feature is a logical unit of behavior that is specified by a set of functional and quality requirements [Bosc00], i.e., features are an abstraction of the requirements; they define keywords that represent corresponding requirements. The features are arranged in a feature model, as described, e.g., in [Czar00] or [Kang90]. The variability model contains the feature model as basis for the configuration.

A feature model orders the features in a tree hierarchy. The root is defined as a member of the product family. On the first level below the root node, the main functionality groups are placed. These functionality groups are refined and further refined in the nodes below the first level.

Features are either mandatory or optional. To an optional feature, a decision is associated whether the feature should be part of a system or not. A mandatory feature is always in a system if the superior feature in the hierarchy is part of the system. In addition, features can be numeric, i.e., a numeric feature comprises a range of integer or real numbers. During the configuration one of the values needs to be chosen.

Optional features may be combined into a feature group. A feature group is used to express variable requirements with several alternatives. Therefore, it defines a cardinality which determines whether the features of the group either have an 'at most one'-, an 'exactly one'-, or an 'at least one'-semantic, i.e., whether the group has an alternative character, or whether the set of options has a mandatory subset character, i.e., it needs the selection of at least one of the options.

Further dependencies between variable requirements need to be expressed in the feature model, too. There is a whole range of possible dependencies, the classical requires relationships, one feature requires the integration of another optional feature, or the exclude relationships, two features cannot be integrated in the same application. In addition, dependencies can be defined between more than two features combining the features in boolean expressions. Between numeric features there may be relational or functional dependencies. All these different dependencies need to be defined in the feature model in order to allow a tool supported selection of a consistent set of requirements.

The Influence of Binding Times

An important issue is the influence of binding times onto the development process. For each variation point, a binding time is defined, i.e., a process task, in which the variation point is bound to a concrete solution. Typical binding times are application design, application implementation, startup time of the application, or even runtime. The variation points are bound in the defined process task. As a consequence, the selection of a variant of the variability, which drives the binding of the variation point, needs to be done in the moment the variation point is bound to a solution.

In addition, there may be more than one variation point in the product family assets, which allow the integration of solutions according to variants of one variability. These variation points may also be spread in time, i.e., one variation point may allow the integration of a solution during application design, another one may allow the integration during application implementation or even later. As a consequence, selecting a variant of a variability can be made at every binding time, in which an associated variation point exists. Therefore, decisions may be postponed as long as there are variation points in the assets. If the decision is postponed, variation points at earlier binding times are bound to general solutions allowing a further refinement later, typically introducing overhead. Since early decisions offer the highest degree of freedom in the implementation of the solution, the number of possible variants for a variability may decrease in time, i.e., there are typically less variants left for variation points with late binding time. On the other hand, taking a decision early may be risky. Critical decisions may turn out to be wrong introducing much rework in the application engineering process.

Requirements for a Configuration Technique

In this section we summarize the requirements resulting from the intention of configuring product families on the basis of a feature model. In a short summary, configuration takes place during the whole application engineering process. In all tasks, there is a configuration step based upon the variabilities of the product family. Variabilities are presented in all process tasks in which they have an influence on variation points. On the other hand, early decisions have influence on the binding of variation points in later process steps, so they have to be remembered during the whole development process.

The configuration is based upon the variabilities in the variability model. Part of the variability model is a feature model as described in section 2. The feature model structures the features, i.e., the core of the model is build from mandatory features, which represent the common requirements of the family. Attached to this core are the variabilities in the form of optional features. They can be grouped together in order to build alternative or subset feature groups. Additional dependencies have to be expressed in order to guarantee consistent feature selections. These dependencies have to be formulated within the feature model; an inference machine has to evaluate the dependencies during the application configuration.

As a consequence, a configuration technique has to allow the description of features as basic knowledge elements. These knowledge elements have to be intergratable into a refinement hierarchy, which is basically an aggregation hierarchy. In addition, it is necessary to express the status of the features, i.e., a cardinality information has to be given, which allows the definition of options (cardinality 0..1), mandatory features (cardinality 1..1), or quantitative variabilities (cardinality > 1). It has to be possible to define groups of knowledge containers for which a cardinality can be defined. With this mechanism it is possible to define alternative or subset feature groups. The knowledge elements have to allow all types of possible ranges, i.e., besides enumerations and numerical ranges the technique has to allow the representation of open variabilities.

The technique needs a constraint mechanism, which allows the definition of dependencies between features. The mechanism needs to be sufficient for the definition of all types of dependencies described in section 2.

An important issue is the presentation of the model during modeling as well as during configuration. In both cases the aggregation hierarchy must be central to the presentation in order to make the understanding of the model and the navigation within the model as easy as possible.

The technique needs a view mechanism, which allows the separation of the feature model into the parts needed in the different tasks as described in section 2. It must be possible to insert knowledge elements into more than one view in order to reflect the property, that variation points in different process tasks may offer solutions for the implementation of a variability. Additionally, it is necessary to change the range of possible values in different views, since some alternatives may be not selectable in a specific

process task. It has to be possible to postpone decisions in one view, as well as to handle decisions already done in other views when presenting the relevant view in one process task.

Other requirements concern the support of the user during the configuration process. A strategy mechanism should help the user to select the different features in an order that allows to take advantage of the dependencies, i.e., that offers the opportunity to decide as many features automatically as possible by the inference machine. Therefore, the inference machine has to support the automatic selection of features, e.g., in a requires dependency, if the condition is true, the required feature can be included into the feature selection automatically. On the other hand in excludes dependencies, if one of the referenced feature is chosen into the feature selection the other feature can be excluded automatically.

These are basically the requirements, which have to be met by a configuration technique. In the following section, we will analyze, why classical configuration technique are not suitable for this task.

Configuration Techniques and Their Deficiencies

In an industry project, we analyzed some commercial configuration techniques and associated tool suites for their suitability in the context of the described requirements of product family configuration. All techniques are based on the same principles. The knowledge representation is done with an object oriented description technique. Knowledge is represented as concepts. A concept corresponds to an object in an object oriented programming language with the exception that it does not contain methods. A concept comprises a set of parameters of different types, concepts are arranged in an inheritance hierarchy, and they may be related by associations.

For the description of dependencies between concepts and parameters, all techniques have a constraint mechanism. A constraint typically consists of a condition and an execution part. The condition part defines a condition over existing concept instances and parameter values. If this condition is evaluated to true, the execution part is executed, typically resulting in new concept instances or new parameter values.

During the configuration, concept instances are created according to cardinalities of associations. Starting point is a goal concept that is instantiated. All connected concepts are also instantiated if the cardinality on the connecting association is larger than

zero. If not, the user is asked to select an appropriate cardinality. Furthermore, the user has to select parameter values, or, in the case of inheritance, derived concepts, which refine basic concepts to more concrete alternatives.

Feature models can be described with these modeling techniques. A feature is represented by a concept; the feature hierarchy is spanned by the association mechanism. Most dependencies can be described with constraints. A problem is the description of feature groups. They have to be described with constraints. But the constraint mechanisms are typically not well suited for this task, since the constraints needed for feature groups try to prevent the user from the selection of a wrong combination. The constraint mechanisms of the commercial techniques are typically specialized for constraints, which automatically generate instances under certain conditions. So, the definition of feature groups was a major problem we encountered during the evaluation. Especially for alternative feature groups, there are other possibilities for the definition. They can be expressed with the inheritance or the parameter mechanisms, so there is a deviation for these feature groups, but this deviation corrupts the easy and clear structure of a feature model by introducing different techniques for the representation of features. As a consequence, they should not be used under consideration of understandability and maintainability of the feature model.

Another problem of the configuration techniques is their specialization on the configuration of a component set. The basic philosophy behind the techniques is the description of a component architecture by abstract component types and associations between them. E.g., a common example is computer configuration. A computer contains a motherboard. The motherboard has one or more processors, memory chips, it is associated to a graphics adapter, and so on. The abstract component types are specialized by concrete products, e.g., an Athlon XP 1900+ processor, or an ATI graphics adaptor. Constraints describe dependencies between the concrete products, i.e., an Athlon motherboard only allows the integration of an Athlon processor. Intel processors cannot be used anymore, if the motherboard is already selected. During the configuration the user selects which component types he needs, then he selects the concrete products. The constraints are checked by a tool in order to generate a consistent configuration.

The problem of the configuration techniques is, that the tool support is specialized to this configuration

philosophy. All analyzed tools are centered around the inheritance hierarchy. None of them gave an usable presentation of the association structure, which is the core of a feature model representation. As a consequence, the models were not presented adequately for the requirements of feature modeling, making the tools unusable for this task. But this is not only a problem of the tool support. For feature modeling the feature hierarchy is the superior structure that defines the hierarchy of the model. For the analyzed configuration techniques, the association mechanism is a generic method in order to combine equal leveled knowledge elements. This has several consequences, e.g., in a feature model names can be selected in the context of the superior feature, i.e., equal names at different parts of the feature model are no problem because of their different superior feature. The analyzed configuration techniques used the names as primary keys in the knowledge base, so concepts need unique names. This demand can be a problem in large feature models, where it is difficult to keep an overview of the complete model.

In summary, there is a large gap in the underlying principles of configuration. This gap cannot easily be bridged. As a consequence, the analyzed configuration techniques do not allow an adequate description of feature models. They can be used, but there are a lot of compromises and problems that cannot easily be solved. Therefore, we propose our own configuration technique, which is an enhancement of the original design space technique [Shaw96]. We enhanced this technique in [Baum98], but experiences made in recent years proposed a major revision of the technique, which we present in the next section.

Design Spaces

A design space is spanned by a set of dimensions, i.e., a dimension is the basic knowledge element in a design space. The dimensions of a design space are ordered in a tree structure. A design space has a dedicated root dimension, which is the ancestor of all other dimensions. The other dimensions all have one superior dimension. Dimensions may be optional or mandatory. Associated to an optional dimension is a yes/no-decision, allowing to model optional aspects of a domain, e.g., an optional feature. Mandatory dimensions are used as structuring concept for the model tree, representing, e.g., mandatory features. In addition, dimensions may comprise a numeric range, allowing the representation of numeric entities in the domain, e.g., a numeric feature in a feature model.

During the configuration the dimensions are profiled, i.e., for each optional dimension the user may decide whether the represented entity is part of the configuration (decision yes) or not (no). For numeric dimensions the user has to choose a value out of the associated range.

Dependencies between dimensions can be expressed by correlations. In principle, a correlation is a boolean term that expresses a relationship between optional or numeric dimensions. It is also possible to describe functional or relational dependencies for numeric dimensions. Global correlations allow the definition of quality attributes.

Dimensions

So precisely, a dimension is an 11-tuple (*id*, *name*, *url*, *stereotype*, *range*, *default_value*, *template*, *superior*, *optional*, *default_decision*, *max_card*).

The *id* attribute contains an unique identifier of the dimension. Ideally, the identifier should be defined by a tool.

A dimension has a *name*. The name has to be unique in the context of the superior dimension, i.e., a name can be used more than once if the superior dimension is different for these dimensions.

A dimension can be associated with a detailed description of the meaning of the represented concept. The description is external to the model, it is referenced by the *url* attribute, which is an internet URL. The external representation allows a flexible support of descriptions, because there is no restriction on the format.

The *stereotype* attribute stores a list of stereotypes, i.e., simple text labels in the form of a string. Related dimensions may be grouped by associating a common stereotype to the dimensions. A dimension may have more than one stereotype. Stereotypes may comprise a single numeric value, which is used as a weight in global correlations in order to compute the quality attribute.

The optional *range* attribute allows the definition of an integer or real interval, which represents a numeric entity in the domain. During the profiling a concrete value needs to be chosen. It is furthermore possible to define a default value for the numeric range. This is done by defining a value in the optional *default_value* attribute. The default value will always be used when the dimension is not profiled.

The meaning of the *template* attribute will be described later in this section. Possible values for this attribute are *static*, *dynamic*, and *none*.

The *superior* attribute references the superior dimension in the Design Space hierarchy. The superior

dimension is referenced by the *id*. Only the root dimension and static templates do not have a superior dimension.

The *optional* flag determines whether the dimension is optional, i.e., a yes/no decision is associated to the dimension, or whether it is mandatory. Associated to the flag may be a default decision. This is specified with the optional attribute *default_decision*. The default decision is used whenever the user has not profiled the dimension himself.

The *max_card* attribute is used for dynamic templates. It is described in the template section below. The value of this attribute is either a positive integer number or *n*.

Dimension Groups

Optional dimensions with a common superior dimension may be grouped into a dimension group. A dimension group allows the definition of a cardinality for the group. Possible cardinalities are 0..1 and 1..1 which represent an alternative dimension group, i.e., at most one or exactly one of the dimensions may be profiled to yes. Another cardinality is 1..n representing a mandatory subset dimension group, i.e., at least one of the dimensions has to be profiled to yes.

A dimension group is a 4-tuple (*id*, *name*, *members*, *cardinality*). The *id* attribute has the same meaning as for dimensions; it is an unique identifier. The *name* attribute is optional and allows the definition of a name for the dimension group. The name should be unique in order to distinguish different dimension groups

In the *members* attribute a list of dimensions is stored. The dimensions are part of the dimension group. All dimensions have to be optional and they must have the same superior dimension. The dimensions are referenced by their *id*.

The *cardinality* attribute may be selected to one of three values: *0..1*, *1..1*, or *1..n*.

Templates

Templates may be used in different situations. Static templates are a mechanism, which reduces the modeling effort. In some cases, a model contains a specific part more than once. It is, therefore, favorable to define such a part only once as a static template. The template is then instantiated at the different places in the model. For each instance of a template, all included dimensions are instantiated, i.e., all associated decisions have to be done independently from the other instances of the template. A static template is defined by setting the *template* attribute of a dimen-

sion to *static*. This dimension does not have a superior dimension; instead, the dimension is the root of the template. All subordinate dimensions can be defined in the normal way. For each instance of the static template there exists a template object that is a 4-tuple (*id*, *name*, *dimension*, *superior*). The attributes *id* and *name* have the same meaning as in dimension groups. With the *dimension* attribute the root dimension of the template is referenced by the id of the dimension. The *superior* attribute references the place in the model at which the template is plugged in, i.e., it references the id of the dimension that is used as superior dimension of the template root.

Dynamic templates are used in cases where it is not known at modeling time, how many instances of a template are necessary in a concrete situation. These templates can be used in case of quantitative variability, i.e., if a functionality group is instantiated more than once in an application, and the functionality group needs to be configured for the different instances.

Dynamic templates always have a common superior dimension, so they can be handled within the dimension objects. For a dynamic template the *template* attribute of a dimension has to be set to *dynamic*. Secondly, the *max_card* attribute has to be set to a value, which indicates the maximum number of instances allowed for the template, or to *n*, which indicates no upper limit on the number. The user has to choose a number for the cardinality. He also has to choose a name for the template instance, which overwrites the original name of the dimension.

Correlations

Correlations are used in order to describe dependencies between dimensions of a design space. Correlations are boolean, relational, or functional expressions. In addition, table correlations are used to express complex relationships between larger groups of dimensions in cases, where it is easier to enumerate all possible combinations instead of defining a boolean expression that specifies the valid combinations. Global correlations allow the extraction of quality statements concerning a complete configuration.

The correlations are constantly overviewed during the configuration process. An inference machine evaluates relevant correlations after each decision done by the user. If correlations restrict the alternatives of a dimension to exactly one, the inference machine will automatically select the remaining value as profile of the dimension.

Boolean Correlations

Boolean correlations are 4-tuples (*name*, *url*, *hard*, *expression*). The *name* attribute is the unique name of the correlation. The *url* attribute has the same meaning as in dimensions. With the *hard* flag it is possible to determine whether the correlation is hard or weak. Hard correlations have to be evaluated to true for a profile of the design space. Weak correlations have not to be held in a configuration. They are used to express favorable combinations of decisions, or they specify dynamic default values, i.e., default values that are only relevant in specific situations.

The *expression* defines the condition, which needs to be held in order to satisfy the correlation. It is a boolean expression that has to be evaluated to true. Allowed operators are *and*, *or*, *not*, *implies*, *equivalence*, and *xor*. Operands are relations, which compare dimensions with a specific selection, i.e., whether an optional dimension is evaluated to yes or no, or a numeric dimension is chosen in a specified interval. It is also possible to specify a number of instances for a dynamic template as an operand.

A special treatment is necessary for operands that reference dimensions in a template. In the standard case, a correlation is instantiated for each template instance, i.e., the condition described by the correlation needs to be held for each template instance. In some cases, a correlation should only be valid for a special instance of a static template. In these cases, the operand reference needs an additional attribute, which names the corresponding template object.

Relational and Functional Correlations

Relational or functional correlations have the same form as boolean correlations. Only the correlation expression is different. The expression contains a relation that has to be held. Allowed comparators are '=', '!=', '<', '>', '<=', and '>='. The operators are either '+', '-', '*', or '/'. Operands are simple references on numeric dimensions. A correlation is functional, if the expression is an equation and if the left hand side is given by a single operand.

It is possible to combine boolean and relational expressions. In this case, the correlation expression must be an implication. On the left hand side of the implication a normal boolean expression can be used as a condition, which needs to be true in order to make the relation relevant. On the right hand side a normal relational or functional expression can be used.

Table Correlations

A table correlation can be used in cases where it is easier to enumerate all possible combinations of dimensions instead of defining a boolean expression that allows to compute all possible combinations. A correlation of this type is a 4-tuple (*name*, *url*, *hard*, *table*). The attributes *name*, *url*, and *hard* are equally defined as in boolean correlations.

The *table* attribute defines all valid combinations of profiles. The table columns, thereby, define the dimensions or alternative dimension groups that are part of the correlation. One row of the table specifies for each dimension or dimension group how the dimensions have to be profiled for a valid combination of these dimensions.

Global Correlations

Global correlations accumulate the values of a specific stereotype associated to the dimensions of a design space. The dimensions may have stereotypes with associated numeric values. A global correlation accumulates the values associated to the dimensions chosen to 'yes', i.e., dimensions, which are part of a configuration. The correlation compares this value to a global value.

Global correlations are given by a 6-tuple (*name*, *url*, *hard*, *stereotype*, *larger*, *value*). The attributes *name*, *url*, and *hard* are the same as in boolean correlations.

The global correlation accumulates the values associated to dimensions by the name of the stereotype referenced in the *stereotype* attribute. The accumulated value is compared to the value specified with the *value* attribute. The *larger* flag defines whether the accumulated value has to be larger than the correlation value or whether it has to be lesser.

Configuration Support

The previous defined concepts allow the modeling of a design space. A configuration based upon a design space is done by instantiating relevant dimensions, i.e., starting from the root dimension, all not optional dimensions will be instantiated. The user may now profile the remaining dimensions by either selecting an optional dimension as in or out, i.e., the optional dimension will be instantiated or not according to the selection of the user. Or, the user selects a value for a numeric dimension respectively chooses the number of instances of a dynamic template. If a dimension is instantiated, a configuration tool evaluates subordinate dimensions, whether they may be instantiated, too, or whether they add new decisions on the list of undone decisions. A configu-

ration stops, when all decisions are done, i.e., when the list of undone decisions is empty.

In principle, the user may do the decision in any sequence. But in many cases it makes sense to profile dimensions in a specific sequence, because of correlations, which maximize the decisions done by the inference machine. So two additional concepts allow the definition of a configuration strategy in order to ease the configuration process. Masks are used to mask out parts of a design space, which become only relevant under certain conditions. The part of the design space is masked out as long as the inference machine evaluates the condition to false. The second concept, sequences, allow the definition of a sequence between two sets of dimensions, i.e., the dimensions in the second set are masked out until all dimensions in the first set are profiled. This can be used in order to guide an user in the configuration process by hiding dimensions which are better profiled at a later point of time.

Mask

Masks are used to hide dimensions until a specific condition is held by a design space profile. In the normal case, a dimension is hidden as long as the superior dimension is not instantiated during the configuration process. So, in these simple cases, masks are not necessary, the model itself prevents the presentation of the irrelevant information. But in cases, where the condition for the presentation is more complex, a mask is used in order to define the condition.

A mask is a 4-tuple (*name*, *url*, *condition*, *dimension*). The attributes *name* and *url* have the same meaning as in correlations. The *condition* is a boolean expression, which has the same form as in boolean correlations. Only if this condition is evaluated to true, the dimensions enumerated in the *dimension* attribute are visible to an user. This is also valid for the subordinate dimensions of the listed dimensions.

Sequence

A sequence is used to mask out a set of dimensions as long as another set is not profiled. With this concept it is possible to guide an user by hiding dimensions which he should not profile until the dimensions of the first set are profiled. The dimensions in the first set carry an additional flag which indicates for a dimension whether its condition is that the dimension has to be profiled or that the dimension and all subordinate dimensions need to be profiled for the visibility of the dimensions in the second set.

A sequence is a 5-tuple (*name*, *strategy*, *url*, *set1*, *set2*). The attributes *name* and *url* have the same meaning as in correlations. The attribute *strategy* is a list of names. The names reference different strategies and the attribute defines in which strategies the sequence is member. At the beginning of a configuration, an user chooses a strategy, only the sequences, which are member of this strategy, will be used in the guidance of the user through the configuration. The attribute *set1* is the set of dimensions that needs to be profiled; *set2* references the dimensions, which are masked out as long as the dimensions of *set1* are not profiled. Associated to each dimension in *set1* is a flag which indicates whether only the dimension should be profiled, or whether all subordinate dimensions need to be profiled, too.

Views

The last concept of the design space technique is the view mechanism. Views are used in order to define different viewpoints onto a design space. The different viewpoints may be used, e.g., to define a sub design space, which contains the dimensions relevant for a binding time. A viewpoint contains only the dimensions and correlations which are needed for a given configuration scenario.

A view is a 5-tuple (*name*, *url*, *dimension*, *correlation*, *strategy*). The attributes *name* and *url* are the same as in correlations. The attributes *dimension*, *correlation*, and *strategy* are lists of dimension ids, respectively correlation, or strategy names. Only the listed dimensions are part of the configuration scenario defined by the view. In the view, only the listed correlations describe dependencies between the dimensions. And, it is only allowed to use the specified strategies in order to execute the configuration process.

If correlations contain references on dimensions, which are not part of the view, it is important to distinguish the case in which the dimension is not profiled from the case in which the dimension is already profiled in a previous configuration scenario. In the first case the correlation expression is evaluated, but the evaluation may not be complete because the correlation is neither evaluated to true or false. If this is the case, the correlation is ignored in this configuration scenario. If a dimension, not part of the view, is already profiled, the external profile is used during the evaluation of the correlation.

If sequences reference dimensions that are not part of the view, these dimensions are simply ignored.

Conclusions and Further Work

In this paper we summarized the requirements relevant in the context of product family configuration. We furthermore described our experiences with commercial configuration techniques. The result of our evaluation was that these techniques are difficult to use in the context of product family configuration. As a consequence, we defined a new configuration technique based on our previous work on design spaces. This new technique was presented in detail in this paper.

In the near future, we will implement a prototype tool support in order to evaluate the technique in case studies. Since the technique implements more or less directly the feature models known, e.g., from FODA [Kang90], we think that it will be very easy to describe feature based configuration models that can be used in the course of product family configuration.

References

- [Bass99] Bass, L.; Clements, P.; Donohoe, P.; McGregor, J.; Northrop, L.: Fourth Product Line Practice Workshop Report, <http://www.sei.cmu.edu/publications/documents/00.reports/00tr002.html>, November 1999
- [Baum98] Baum, L.; Geyer, L.; Molter, G.; Rothkugel, S.; Sturm, P.: Architecture-Centric Software Development Based on Extended Design Spaces, ARES 2nd Int'l Workshop on Development and Evolution of Software Architectures for Product Families, Feb. 1998
- [Beck01] Becker, M.; Geyer, L.; Gilbert, A.; Becker, K.: Variability Modeling as a Catalyst for Efficient Variability Treatment, Proceedings of the 4th Workshop on Product Family Engineering (PFE-04), Bilbao, Spain, 2001
- [Bosc00] Bosch, J.: Design and Use of Software Architectures - Adopting and Evolving a Product Line Approach, Addison-Wesley, 2000
- [Czar00] Czarnecki, K.; Eisenecker, U.W.: Generative Programming - Methods, Tools, and Applications, Addison-Wesley, 2000
- [Kang90] Kang, K.; Cohen, S.; Hess, J.; Nowak, W.; Peterson, S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study, Technical Report, CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, November 1990
- [Shaw96] Shaw, M.; Garlan, D.: Software Architecture - Perspectives on an Emerging Discipline, Prentice Hall 1996
- [With96] Withey, J.: Investment Analysis of Software Assets for Product Lines, <http://www.sei.cmu.edu/publications/documents/96.reports/96.tr.010.html>, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, 1996