Staged Configuration Through Specialization and Multi-Level Configuration of Feature Models

Krzysztof Czarnecki[1], Simon Helsen[2⋆], and Ulrich Eisenecker[3]

[1] University of Waterloo, Canada
[2] Interactive Objects Software GmbH, Germany
[3] University of Leipzig, Germany

Abstract Feature modeling is a key technique for capturing commonalities and variabilities in system families and product lines. In this paper, we propose a cardinality-based notation for feature modeling, which integrates a number of existing extensions of previous approaches. We then introduce and motivate the novel concept of staged configuration. Staged configuration can be achieved by the stepwise specialization of feature models or by multi-level configuration, where the configuration choices available in each stage are defined by separate feature models. Staged configuration is important because in a realistic development process, different groups and different people make product configuration choices in different stages. Finally, we also discuss how multi-level configuration avoids a breakdown between the different abstraction levels of individual features. This problem, sometimes referred to as "analysis paralysis", easily occurs in feature modeling because features can denote entities at arbitrary levels of abstractions within a system family.

Key words: Software product lines, system families, domain analysis, software configuration

1 Introduction

Software product-line engineering (also known as system-family engineering) seeks to exploit the commonalities among systems from a given problem domain while managing the variabilities among them in a systematic way (Clements and Northrop, 2001; Weiss and Lai, 1999; Parnas, 1976). In product-line engineering, new product variants can be rapidly created based on a common set of reusable assets such as a common architecture, components, models, development processes, etc.[4]

Feature modeling is an important technique for capturing and managing commonalities and variabilities in product lines throughout all stages of product-line engineering. At an early stage of product-line development, feature modeling is used for product-line scoping, i.e., deciding which features should be supported by a product line and which should not. Scoping involves recording and assessing information in feature models such as which features are important to enter a new market or remain in an existing market, which features incur a technological risk, what is the projected development cost of each feature, and so forth (DeBaud and Schmid, 1999). In product-line design, the points and ranges of variation captured in feature models need to be mapped to a common product-line architecture (Czarnecki and Eisenecker, 2000; Bosch, 2000). In actual product development, feature models can drive requirements elicitation and analysis, help in estimating development cost and effort, and provide a basis for automated product configuration.

Feature models also play a key role in generative software development (Czarnecki and Eisenecker, 2000; Greenfield and Short, 2004). Generative software development aims at automating application engineering based on system families: a system is generated from a specification written in one or more textual or graphical domain-specific languages (DSLs) (Weiss and Lai, 1999; Czarnecki and Eisenecker, 2000; Cleaveland, 2001; Batory, Johnson, MacDonald and von Heeder, 2002; Greenfield and Short, 2004). In this context, feature models are used to scope and develop DSLs (Czarnecki and Eisenecker, 2000;

⋆ Work done while at the University of Waterloo.

[4] System-family engineering is mainly concerned with building systems from common assets, whereas software product-line engineering additionally considers scoping and managing common product characteristics from the marketing perspective. In this paper, we use both terms interchangeably.

Deursen and Klint, 2002), which may range from simple parameter lists or feature hierarchies to more sophisticated DSLs with graph-like structures.

Feature modeling was originally proposed as part of the Feature-Oriented Domain Analysis (FODA) method (Kang, Cohen, Hess, Nowak and Peterson, 1990), and since then, it has been applied in a range of domains including telecom systems (Griss, Favaro and d'Alessandro, 1998; Lee, Kang and Lee, 2002), template libraries (Czarnecki and Eisenecker, 2000), network protocols (Barbeau and Bordeleau, 2002), and embedded systems (Czarnecki, Bednasch, Unger and Eisenecker, 2002; Lohmann, Schröder-Preikschat and Spinczyk, 2005). Based on this growing experience, a number of extensions and variants of the original FODA notation have been put forward (Griss et al., 1998; Czarnecki, 1998; Hein, Schlick and Vinga-Martins, 2000; van Gurp, Bosch and Svahnberg, 2001; Lee et al., 2002; Riebisch, Böllert, Streitferdt and Philippow, 2002; Czarnecki et al., 2002; Beuche, 2003; Cechticky, Pasetti, Rohlik and Schaufelberger, 2004).

1.1 Contributions and Overview

In this paper, we make the following contributions: we present a cardinality-based notation for feature models, which integrates and adapts four existing extensions to the FODA notation—namely feature cardinalities, group cardinalities, feature diagram references, and attributes. We also propose the novel concept of staged configuration, which can be achieved either by stepwise specialization of feature models or by multi-level configuration, where the configuration choices available in each stage are defined by separate feature models. Finally, we discuss how the concept of multi-level configuration helps avoiding the problem of "analysis paralysis" in feature modeling, which amounts to a breakdown between the different abstraction levels of individual features.

The remainder of the paper is organized as follows. Section 2 reviews background concepts and related work on feature modeling. Our cardinality-based notation for feature modeling is presented in Section 3. Staged configuration is described in Section 4. Section 5 describes how "analysis paralysis" in feature modeling can be avoided thanks to multi-level configuration. A larger example of multi-level configuration including some observations is presented in Section 6. Related work on incorporating time in variability modeling and configuration is discussed in Section 7. Appendix A gives a comparison of three different notations for feature modeling.

Note: A shorter version of this paper was presented at the Software Product Line Conference (Czarnecki, Helsen and Eisenecker, 2004). The current paper extends the earlier version with new material on multi-level configuration (major parts of Section 4), process of feature modeling (Section 5), and electronic commerce example (Section 6).

2 Background

2.1 Features, Feature Diagrams, and Feature Models

A *feature* is a system property that is relevant to some stakeholder and is used to capture commonalities or discriminate among systems in a family (Czarnecki and Eisenecker, 2000). Note that while the original definition by Kang et al. (Kang et al., 1990) has defined features as "user-visible" properties, we allow features with respect to any stakeholder, including customers, analysts, architects, developers, system administrators, etc. Consequently, a feature may denote any functional or non-functional characteristic at the requirements, architectural, component, platform, or any other level. Features are organized in *feature diagrams*. A feature diagram is a tree with the root representing a concept (e.g., a software system) and its descendant nodes being features. In the FODA feature diagram notation (see the leftmost column of Table 7 in Appendix A), features can be *mandatory*, *optional*, or *alternative*. *Feature models* are feature diagrams plus *additional information* such as feature descriptions, binding times, priorities, stakeholders, and so forth.

Feature diagrams offer a simple and intuitive notation to represent variation points in a way that is independent of implementation mechanisms such as inheritance or aggregation. It is important not to confuse feature diagrams with part-of hierarchies or decompositions of software modules. Features may or may not correspond to concrete software modules. In general, we distinguish the following four cases:

– *Concrete* features such as data storage or sorting may be realized as individual components.
– *Aspectual* features such as synchronization or logging may affect a number of components and can be modularized using aspect technologies.
– *Abstract* features such as performance requirements usually map to some configuration of components and/or aspects.
– *Grouping* features may represent a variation point and map to a common interface of plug-compatible components, or they may have a purely organizational purpose with no requirements implied.

2.2 Summary of Existing Extensions

Since its initial introduction in the technical report by Kang et al. (1990), several extensions and variants of the original FODA notation have been proposed. In the following summary, we abstract from variations in concrete syntax and focus on the conceptual extensions.

– *feature cardinalities.* Features can be annotated with cardinalities, such as [1..$*$] or [3..3]. Mandatory and optional features can be considered special cases of features with the cardinalities [1..1] and [0..1], respectively. Feature cardinalities were motivated by a practical application (Czarnecki et al., 2002), after being initially discouraged (Czarnecki and Eisenecker, 2000, p. 117).
– *groups and group cardinalities.* Alternative features in the FODA notation can be viewed as a grouping mechanism. Two further kinds of groups were proposed in Czarnecki's (1998) thesis: the inclusive-or group and the inclusive-or group with optional subfeatures (see the middle column of Table 7 in Appendix A).[5] The concept of groups was further generalized by Riebisch et al. (2002) as a set of features annotated with a cardinality specifying an interval of how many features can be selected from that set. The previous kinds of groups become special cases of groups with cardinalities (see the rightmost column of Table 7 in Appendix A).
– *attributes.* Attributes were introduced by Czarnecki et al. (2002) as a way to represent a choice of a value from a large or infinite domain such as integers or strings. An elegant way to model attributes proposed by Bednasch (2002) is to allow a feature to be associated with a type, such as integer or string. A collection of attributes can be modeled as a number of subfeatures, where each is associated with the desired type.
– *relationships.* Several authors (e.g., Griss et al., 1998; van Gurp et al., 2001; Lee et al., 2002) proposed to extend feature models with different kinds of relationships such as *consists-of* or *is-generalization-of.*
– *feature categories and annotations.* FODA distinguishes among context, representation, and operational features. Griss et al. (1998) propose functional, architectural, and implementation feature categories. Section 2.1 gives yet another categorization. Additional information on features suggested in FODA include descriptions, constraints, binding time, and rationale. Other examples are priorities, stakeholders, default selections, open-or-closed-for-extensions attribute, and exemplar systems (Czarnecki, 1998).
– *modularization.* A feature diagram may contain one or more special leaf nodes, each representing a separate feature diagram (Bednasch, 2002). This mechanism allows breaking up large diagrams into smaller ones and reusing common parts in several places. This is an important mechanism because, in practice, feature diagrams often become too large to be considered in their entirety.

3 Cardinality-Based Feature Modeling

This section proposes a *cardinality-based notation for feature modeling*, which is based on modest changes to the previously introduced concepts of feature cardinalities, group cardinalities, and diagram modularization (see Section 2.2).

In particular, a feature cardinality specification may consist of a sequence of intervals. Furthermore, our notation does not allow features that are members of a feature group to be qualified with feature

[5] Inclusive-or features were introduced independently by Czarnecki (1998) and Griss et al. (1998). However, inclusive-or features as proposed by the latter authors imply reuse-time binding, whereas inclusive-or features as proposed by the former author are independent of binding time.

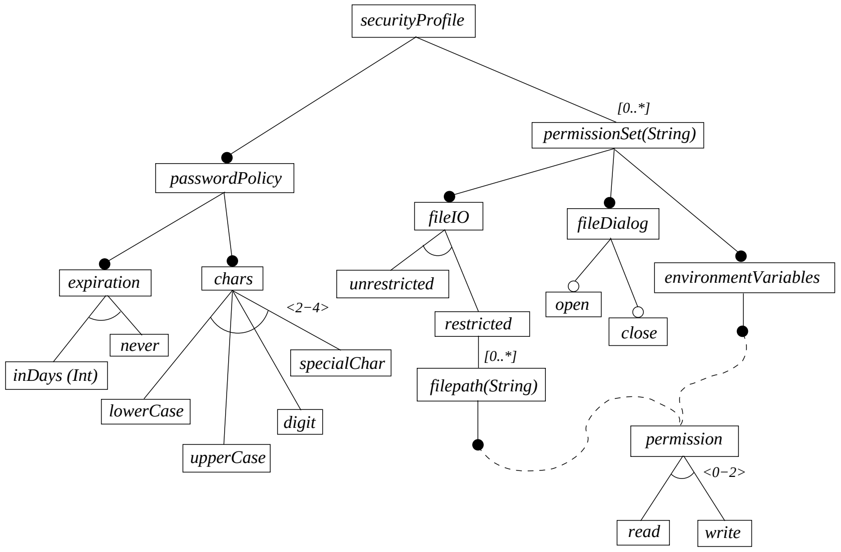


Figure 1. Security profile example

cardinalities. This is because a feature cardinality is a property of the relationship between a single subfeature and its parent. Similarly, a group cardinality is a property of the relationship between a parent and a set of subfeatures. Next to cardinalities, we have the notion of *feature diagram references*, which allow us to reuse or modularize feature models in a similar fashion as described by Bednasch (2002). In contrast to Bednasch's (2002) work, feature diagram references allow *recursion*, which may be either direct or indirect. Direct recursion occurs when a feature diagram reference refers to the feature diagram in which it resides, while indirect recursion involves more than one diagram.

The chosen set of conceptual extensions and their adaptations are motivated both by practical applications and the urge to achieve a balance between simplicity and conceptual completeness. Feature cardinalities and attributes are common in modeling both embedded software (Czarnecki et al., 2002) and enterprise software (see our example in Section 3.1). The primary motivation for including group cardinalities is elegance and completeness. Although our experience so far shows that the vast majority of groups are either exclusive-or or inclusive-or, other group cardinality values may still be useful in more exotic situations, as shown by Riebisch et al. (2002) and in Section 3.1. Compared to the more profound semantic implications of feature cardinalities, the addition of group cardinalities is semantically relatively straightforward.

In our notation, we do not consider relationships between features such as *consists-of* or *is-generalization-of* because we think that they are better modeled using other notations such as entity-relationship or UML class diagrams. In general, we believe that a feature modeling notation should focus purely on capturing commonality and variability. However, if necessary, a tool with an extensible metamodel (Bednasch, 2002; Bednasch, Endler and Lang, 2002-2004) may allow the user to introduce additional kinds of relationships. Finally, feature categories and other additional information are domain dependent, and as previously argued by Czarnecki et al. (2002), we think that they are better handled as user-defined, structured annotations. Such annotations are also supported through an extensible metamodel (Bednasch, 2002).

3.1 A Security Profiling Example

As a practical example to demonstrate the expressiveness of our feature modeling language, consider a feature model of an operating system security profile in Figure 1.

The password policy of the security profile has an expiration time and possible requirements on the kind of characters to be used. Passwords can be set to never expire, or to expire after a given number of days. The number of days a password remains valid can be set in the integer attribute of the `inDays` feature. Generally, whenever a feature has an attribute, we indicate its type within parentheses after the feature name, e.g., `inDays(Int)`. It is also possible to specify a value of the associated type immediately

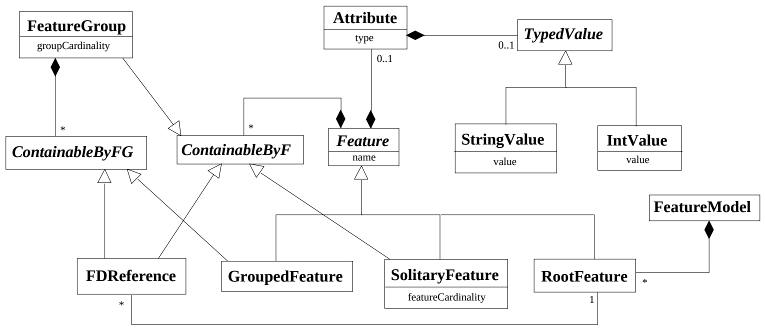


Figure 2. UML metamodel for cardinality-based feature models

with the type, e.g., `inDays(30:Int)`. The constraints on the kind of characters required in a password are specified by a feature group with cardinality ⟨2–4⟩. This means that any actual password policy must specify between two and four (different) requirements on the kind of characters.

In our example, because no cardinality was specified for the group of `expiration` policies, the cardinality ⟨1–1⟩ is assumed (i.e., the `expiration` policies form an *exclusive-or* group).

The security profile also has zero or more permission sets. This is indicated with the feature cardinality [0..∗]. If a feature cardinality is [1..1], we draw a little filled circle above the feature. Observe that features belonging to a group do not have feature cardinalities.

A permission set determines various permissions for executing code. In our simple model, a permission set has a string attribute to specify its name, and it allows us to specify permissions with respect to file IO, file dialogs, and environment variables. Other examples would be permissions to access a database, invoke reflection, access a Web address, etc. According to our model, file IO can be restricted to a list of file paths, or it is unrestricted. For each file path, we can specify its name and associated read/write permissions.

Notice that we use a feature diagram reference for the permission model because we want to reuse it for environment variables. In this paper, we use a dashed line to represent a feature diagram reference, but it should be noted that, in practice, a different representation may be necessary to avoid a convoluted diagram. This is especially important if the purpose of the feature diagram reference is to modularize a large feature model over different diagrams.

Finally, the permission to open a file dialog and to close it can be selected independently. The empty circle above the features `open` and `closed` indicates that those features are optional, i.e., they have the feature cardinality [0..1].

3.2 A Metamodel

Now that we have seen an example of a cardinality-based feature model, we explain the available concepts more accurately by means of an abstract syntax model, where we will refer to the example in Figure 1 for clarification.

Consider the Unified Modeling Language (UML) metamodel for cardinality-based feature models in Figure 2. A feature model consists of any number of *root features*, which form the root of the different feature diagrams in the model. In the security profile example, both the features `securityProfile` and `permission` are root features.

A root feature is only one of three different kinds of features. The other two are the *grouped feature* and the *solitary feature*. The former is a feature which can only occur in a *feature group*. For example, `never` is a grouped feature in a feature group, which is contained by the feature `expiration`. A solitary feature is a feature which is, by definition, not grouped in a feature group. Many features in a typical feature model are solitary; for example, the feature `passwordPolicy` and `permissionSet`.

Features can have an optional attribute of a certain type, and those attributes can have an optional value. In this simplified model, we only have integer and string attributes.

Figure 2 also has a class named *FDReference* that stands for a feature diagram reference. It can refer to only one root feature, but a root feature can be referred to by several references. In the example, the feature `permission` is referred to by two references.

The abstract classes *ContainableByFG* and *ContainableByF* stand for those kind of objects that can be contained by a feature group and a feature, respectively. A feature group can contain only grouped features or feature diagram references, whereas a feature can contain only solitary features, feature groups, and references.

A solitary subfeature of a feature f is qualified by a *feature cardinality*.[6] It specifies how often the solitary subfeature (and any possible subtree) can be cloned as a child of f. A feature cardinality I is a sequence of intervals of the form $[n_1..n'_1]\ldots[n_l..n'_l]$, where we assume the following invariants:

$$\begin{array}{ccc} \forall\, i \in \{1,\ldots,l-1\} : n_i, n'_i \in \mathbb{N} & n_l \in \mathbb{N} & n'_l \in \mathbb{N} \cup \{*\} \\ \forall\, n \in \mathbb{N} : n < * & & 0 \leq n_1 \\ \forall\, i \in \{1,\ldots,l\} : n_i \leq n'_i & & \forall\, i \in \{1,\ldots,l-1\} : n'_i < n_{i+1} \end{array}$$

An empty sequence of intervals is denoted by ε.

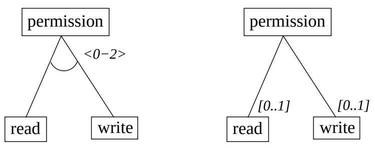
An example of a valid specification of a feature cardinality is $[0..2][6..6]$, which says that we can take a feature 0, 1, 2, or 6 times. Note that we allow the last interval in a feature cardinality to have as an upper bound the Kleene star $*$. Such an upper bound denotes the possibility of taking a feature an unbounded number of times. For example, the feature cardinality $[1..2][5..*]$ requires that the associated feature is taken 1, 2, 5, or any number greater than 5 times. Semantically, the feature cardinality ε is equivalent to $[0..0]$ and implies that the subfeature can never be chosen in a configuration.

A feature group expresses a choice over the grouped features in the group. This choice is restricted by the *group cardinality* $\langle n\text{–}n'\rangle$, which specifies that one has to select at least n and at most n' distinct grouped features in the group. Given that $k > 0$ is the number of grouped features, we assume that the following invariant on group cardinalities holds: $0 \leq n \leq n' \leq k$.

At this point, we ought to mention that, theoretically, we could generalize the notion of group cardinality to be a sequence of intervals as we did for feature cardinalities. However, we have found no practical applications of this usage, and it would only clutter the presentation.

Grouped features do not have feature cardinalities because they are not in the solitary subfeature relationship. This avoids redundant representations for groups and the need for group normalization that was necessary for the notation in Czarnecki's (1998) work. For example, in that notation, an inclusive-or group with both optional and mandatory subfeatures (corresponding to feature cardinalities $[0..1]$ and $[1..1]$ respectively), would be equivalent to an inclusive-or group in which all the subfeatures were optional. Keeping feature cardinalities out of groups also avoids problems during specialization (see Section 4.2). For example, the cloning of a subfeature with a feature cardinality $[n..n']$, where $n' > 1$, within a group could potentially increase its size beyond the upper bound of the group cardinality.

Even without the redundant representations for groups, there is still more than one way to express the same situation in our notation. For example, the following two different diagrams are identical in their semantics.



Conceptually, we keep these two diagrams distinct and leave it up to a tool implementer to decide how to deal with them. For instance, it might be useful to provide a conversion function for such diagrams. Alternatively, one could decide on one type as the *preferred form*, which is shown by default.

An alternative approach to integrate feature and group cardinality in an attempt to reduce the number of redundant representations was proposed by Cechticky et al. (2004).[7] The notation presented in that work does not support the concept of a solitary feature: a group with one grouped feature is used instead.

[6] More precisely, a feature cardinality is attached to the solitary subfeature *relationship*. This relationship is implicit in the metamodel.

[7] This work was done and published independently and in parallel with our SPLC3 paper (Czarnecki et al., 2004).

Furthermore, in contrast to our notation, grouped features have feature cardinalities, and their lower bound has to be larger than 0. During configuration, group and feature cardinalities are interpreted in a top down fashion: first, the desired group members need to be selected from a group in consistence with its group cardinality; second, a selected member can be cloned if its feature cardinality permits it.

The notation by Cechticky et al. (2004) does not eliminate all redundant representations. For example, a group with cardinality $\langle k{-}k \rangle$ and k members can also be represented as k groups, each with cardinality $\langle 1{-}1 \rangle$ and one member. Most importantly, however, attaching feature cardinalities to grouped features makes the notation by Cechticky et al. (2004) awkward for specialization because cloning the members of a group may invalidate its cardinality.

In Appendix A, we compare our new cardinality-based notation with the FODA notation and the notation introduced by Czarnecki and Eisenecker (2000). For the sake of readability, we will keep using the latter notation whenever an appropriate equivalent exists. However, because the cardinality-based notation has no feature cardinality in groups, we will not use the filled circle on top of grouped features.

Cardinality-based feature models can be given formal semantics via translation to context-free grammars. The details of this formalization are elaborated elsewhere (Czarnecki, Helsen and Eisenecker, 2005).

4 Staged Configuration

A feature model describes the configuration space of a system family. An application engineer may specify a member of a system family by selecting the desired features from the feature model within the variability constraints defined by the model (e.g., the choice of exactly one feature from a set of alternative features).

The process of specifying a family member may also be performed in stages, where each stage eliminates some configuration choices. We refer to this process as *staged configuration*. Each stage takes a feature model and yields a specialized feature model, where the set of systems described by the specialized model is a subset of the systems described by the feature model to be specialized.

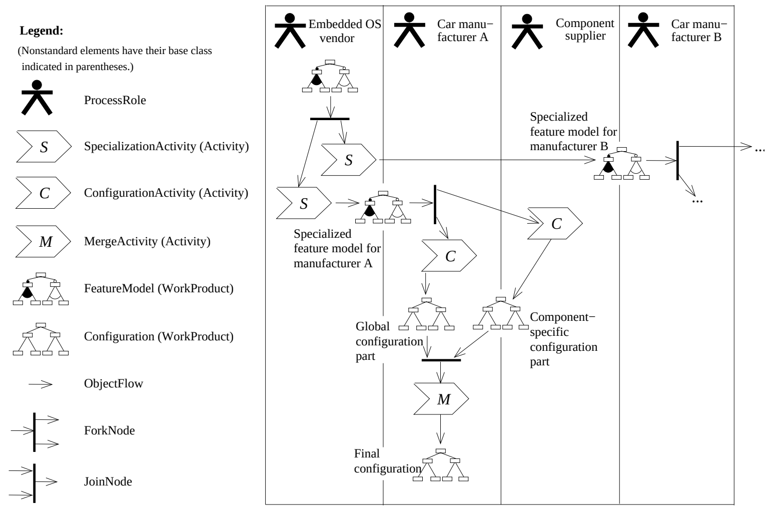

Legend:
(Nonstandard elements have their base class indicated in parentheses.)
ProcessRole
S
SpecializationActivity (Activity)
C
ConfigurationActivity (Activity)
M
MergeActivity (Activity)
FeatureModel (WorkProduct)
Configuration (WorkProduct)
ObjectFlow
ForkNode
JoinNode
Embedded OS vendor
Car manu−facturer A
Component supplier
Car manu−facturer B
Specialized feature model for manufacturer B
...
...
Specialized feature model for manufacturer A
Global configuration part
Component−specific configuration part
Final configuration

Figure 3. Workflow model of a staged configuration scenario from automotive software engineering

The need for staged configuration arises in at least three contexts (Czarnecki et al., 2005):

Software supply chains In software supply chains (Greenfield and Short, 2004), a supplier can create families of software components, platforms, and/or services specialized for different system integrators based on one common system family, which covers all the variability needed by the individual families. In general, a similar relationship can exist between tier-n and tier-n+1 suppliers of the system integrator. This creates the need for multi-stage configuration, where each supplier and system integrator is able to eliminate certain variabilities and to compose specialized families. Let us take a look at an example based on a real scenario involving the configuration and generation of basic services for electronic control units (ECUs) embedded in an automobile in Figure 3 (the modeling notation conforms to the Software Process Engineering Metamodel (SPEM) Specification (OMG, 2002)). An embedded operating system (EOS) offers basic services such as tasking support, network drivers, network management, flash support, diagnosis, and so forth. An EOS vendor may deliver different specialized EOS variants to different car manufacturers in order to reflect their different requirements, such as different terminologies or required interfaces. Doing so constitutes the first configuration stage. After the first stage, the EOS needs to be further configured for each different ECU in a car, depending on the needs of the control functions, such as break control or engine management, to be installed on the given ECU. The latter configuration step is split between the car manufacturer and the component vendors supplying the manufacturer with the control functions. The manufacturer determines various general EOS settings, whereas the component vendors are responsible for control-function-specific settings. The different partial configurations need to be merged into a final configuration. The EOS code specific to a given ECU is then generated based on that final configuration.

Optimization Staged configuration offers an opportunity to perform optimizations based on partial evaluation (Jones, Gomard and Sestoft, 1993). When certain configuration information becomes available at some stage and it remains unchanged thereafter, the software can be optimized with respect to this information. For example, configuration information available at compile-time can be used to eliminate unused code from a component and to perform additional compile-time optimizations. Similarly, the component could be further specialized based on configuration information available at deployment time. Optimizations are especially interesting for embedded software and software that needs to be distributed through a network connection. Note that optimization stages may coincide with the configuration stages within a supply chain, but this is not always the case. Many suppliers decide to deploy runtime keys to enable or disable certain product features without physically eliminating the unused code.[8]

Policy standards Infrastructure policies may be defined at different levels of an organization with the requirement of hierarchical compliance. For example, security policy choices provided by the computing infrastructure of an enterprise can be specialized at the enterprise level, then further specialized at the level of each department, and finally configured at the level of individual computers (see Figure 4). We will come back to this example in Section 4.2.

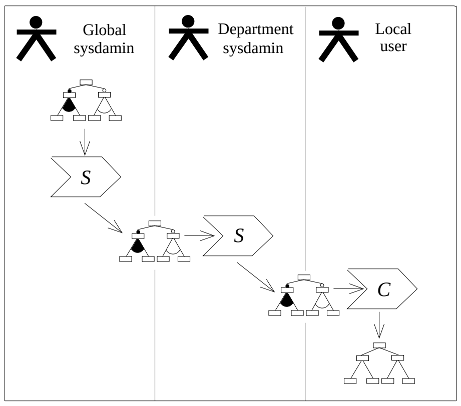


Figure 4. SPEM model for staged configuration of policy standards

Configuration stages can be defined in terms of three different dimensions (Czarnecki et al., 2005):

[8] The choice of employing runtime techniques for static configuration may sometimes be dictated by the limitations of the implementation technology being used. However, there may be more profound reasons, such as logistics requiring packaging all variants into a single distribution; a concrete example for the latter is discussed by Czarnecki et al. (2002, pp. 168-169).

Time A configuration stage may be defined in terms of the different phases in a product lifecycle, e.g., product design, testing, roll-out, deployment, normal operation, maintenance, etc. These times are, of course, product-specific. They will have to be mapped to the different binding times offered by the utilized implementation technologies, e.g., preprocessing time, compile-time, template-instantiation time, load-time, runtime, etc.

Roles In staged configuration scenarios, eliminating different variabilities may be the responsibility of different parties, which can be assigned different roles. For example, a central security group may be responsible for defining enterprise-wide security policies, while the configuration of the individual computers may be done by the departmental system administrators. Similarly, multiple configuration roles will be typically required in a supply-chain scenario.

Targets A target system or subsystem for which a given software needs to be configured may also define possible configuration stages. For example, a given component may be deployed several times within a given software system. In this case, the component could first be specialized to eliminate functionality not needed within the system and then be further configured for each deployment context within that system.

4.1 Configuration vs. Specialization

So far, we have loosely defined the terms *feature model specialization* and *feature model configuration.* At this point, we need to review the terminology from our previous paper (Czarnecki et al., 2005), which makes these notions more precise. A *configuration* consists of the features that were selected according to the variability constraints defined by the feature diagram. The relationship between a feature diagram and a configuration is comparable to the one between a class and its instance in object-oriented programming. The process of deriving a configuration from a feature diagram is also referred to as the *configuration process.* The *specialization process* is a transformation process that takes a feature diagram and yields another feature diagram, such that the set of the configurations denoted by the latter diagram is a subset of the configurations denoted by the former diagram. We also say that the latter diagram is a *specialization* of the former one. A *fully specialized* feature diagram denotes only one configuration.

In general, we can have the following two extremes when performing configuration:[9] a) deriving a configuration from a feature diagram *directly* and b) specializing a feature diagram down to a fully specialized feature diagram and then deriving the configuration (which is trivial). Thus, staged configuration can be achieved by specialization, where at each stage some specialization steps are applied and the last stage is followed by deriving a configuration from the most specialized feature diagram in the specialization sequence.

Observe that we might not always be interested in one specific configuration. For example, a feature diagram that still contains unresolved variability could be used as an input to a generator. This is useful when generating a specialized version of a framework (which still contains variability) or when generating an application that should support the remaining variability at runtime.

4.2 Specialization Steps

As previously stated, specialization allows the fine-grained elimination of variabilities and can be used to perform configuration in multiple stages. This fine-grained elimination of variability can be achieved using *specialization steps.* The available specialization steps are a) refining a feature cardinality, b) refining a group cardinality, c) removing a grouped feature from a group, d) selecting a grouped feature from a group, e) assigning a value to an attribute which only has been given a type, f) cloning a solitary subfeature, and g) unfolding a feature-diagram reference. They are illustrated in Figure 5. The precise definition of each step is given elsewhere (Czarnecki et al., 2005). Specialization steps can be offered in a tool simply as operations within the context menu of a selected feature, group, or reference.

Let us now apply the notion of staged configuration using specialization to the security profile example from Section 3.1. Assume, for instance, that the IT infrastructure of a company supports the security profile from Figure 1. The company then decides to specialize this profile to define a standard enterprise-level security profile as depicted in Figure 6.

[9] We usually drop the term “process” from “configuration process” and “specialization process” when the intended meaning is clear from the context.

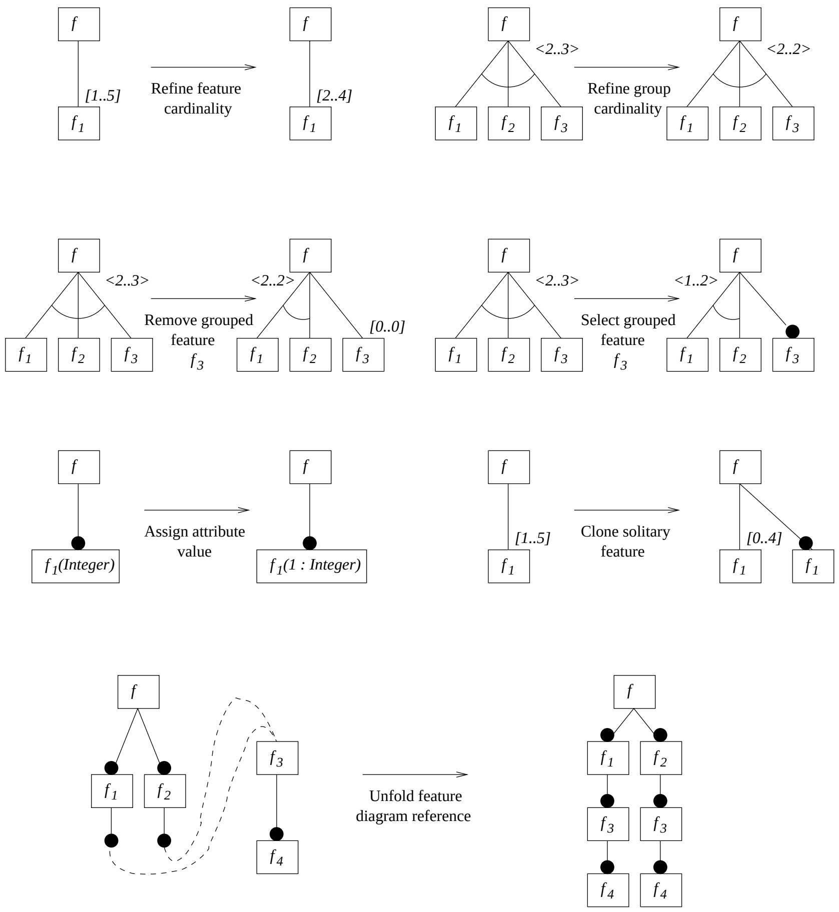


Figure 5. Specialization steps

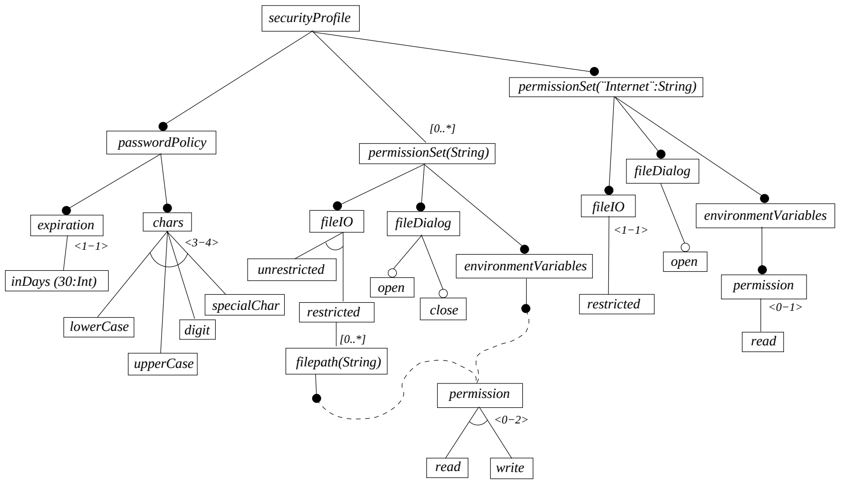


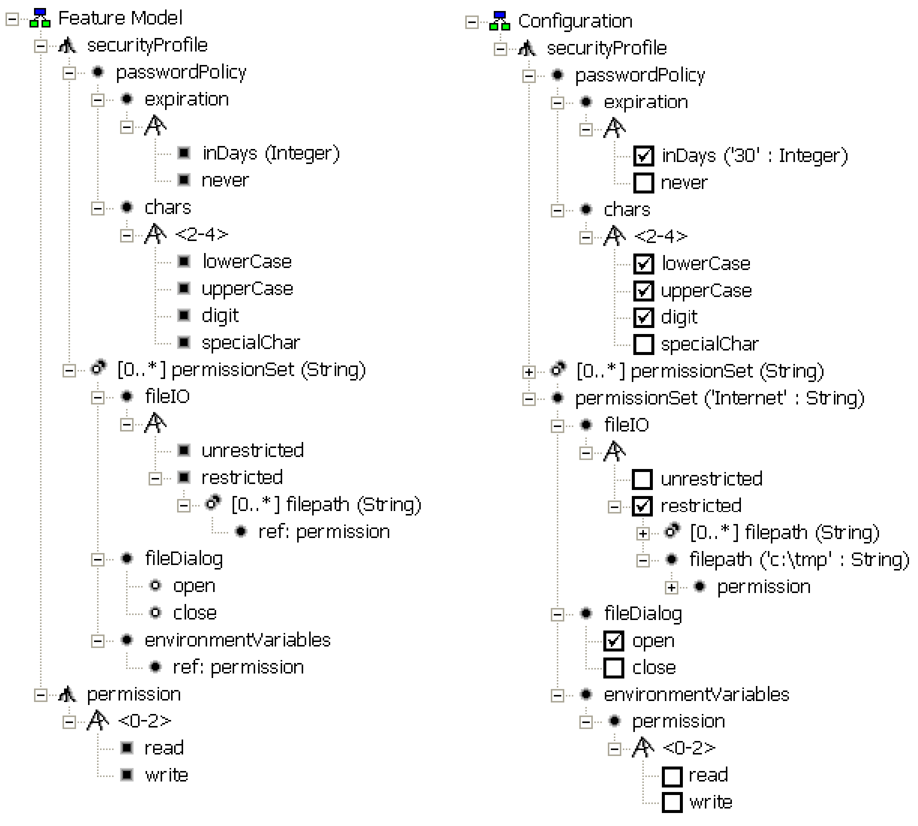
Figure 6. Sample specialization of the security profile

The specialization is achieved by a combination of the previously described steps. In the feature `passwordPolicy`, we have eliminated the ability to have a non-expiring password and set the expiry time at 30 days. On top of that, the company requires at least three different kind of characters to be used instead of two.

Moreover, we clone the feature `permissionSet` and assign it the name "Internet" because we want to specify a specific permission set for programs running from the Internet. In particular, we want file IO to be restricted and allow no access to local file paths. Furthermore, file dialogs may be allowed only to be opened, but Internet programs may be given the permission to read environment variables. Observe that we unfolded the `permission` feature diagram reference under the `environmentVariables` feature. If desired, the enterprise-level security profile could be further specialized by individual departments and then for individual computers within the departments.

4.3 Configuration Interface

If the primary goal is to create a configuration in a single stage, specialization steps listed in the previous section are too general for this task. In particular, arbitrary refinements of feature and group cardinalities and removing grouped features are not needed in single-stage configuration. For example, while during specialization we might want to produce a specialized inclusive-or group by removing some features from it, we are only interested in selecting one feature from that group during single-stage configuration. Similarly, we are not interested in arbitrarily refining the cardinality of a feature, but we want to pick exactly one number within the cardinality interval and include the feature in the configuration the corresponding number of times.



```
Feature Model
  securityProfile
    passwordPolicy
      expiration
        
          inDays (Integer)
          never
      chars
        <2-4>
          lowerCase
          upperCase
          digit
          specialChar
    [0..*] permissionSet (String)
      fileIO
        
          unrestricted
          restricted
            [0..*] filepath (String)
              ref: permission
      fileDialog
        open
        close
      environmentVariables
        ref: permission
  permission
    <0-2>
      read
      write
```

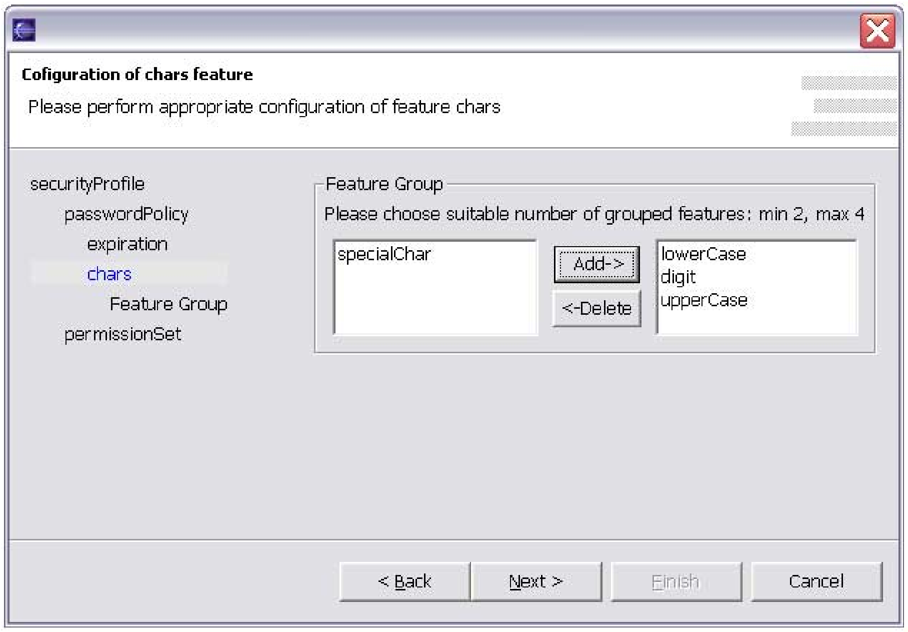
Editing

```
Configuration
  securityProfile
    passwordPolicy
      expiration
        
          ☑ inDays ('30' : Integer)
          ☐ never
      chars
        <2-4>
          ☑ lowerCase
          ☑ upperCase
          ☑ digit
          ☐ specialChar
    [0..*] permissionSet (String)
    permissionSet ('Internet' : String)
      fileIO
        
          ☐ unrestricted
          ☑ restricted
            [0..*] filepath (String)
            filepath ('c:\tmp' : String)
              permission
      fileDialog
        ☑ open
        ☐ close
      environmentVariables
        permission
          <0-2>
            ☐ read
            ☐ write
```

Configuration

Figure 7. Example of feature model editing and configuration interface

Single-stage configuration is very tightly related to the interface exposed to the user, and it is difficult to give a set of conceptual steps for single-stage configuration without discussing concrete user interfaces. As an example, consider the user interface shown in Figure 7, which is a screenshot from the feature modeling tool FeaturePlugin (Antkiewicz and Czarnecki, 2004). The left part shows the feature model from Figure 1 using a tree explorer view. The right part shows a configuration interface, where the user can select an optional solitary feature (i.e., a feature with cardinality [0..1]) or a grouped feature by ticking off a check box next to the feature. There is no need for a separate remove feature operation as in specialization; if a check box has not been ticked off, the corresponding feature is assumed to be excluded from the configuration being built (e.g., `close`). Cloning is available as an operation on a feature with a cardinality whose upper bound is larger than one through a context menu. For example, `permissionSet` and `filepath` were cloned once. Features with attribute types need to be assigned values, e.g., 30 for `inDays`. Finally, feature diagram references are unfolded lazily and transparently for the user. A feature diagram reference is automatically replaced by a copy of the root feature of the diagram it points to. If the original root feature has subfeatures, the copy will be displayed as if it had a collapsed subtree. An example for the latter case is `permission` under `filepath`. Expanding the subtree triggers further unfolding, as illustrated by `permission` under `environmentVariables`. Lazy unfolding avoids infinite looping in the case of recursive feature diagram references.


Cofiguration of chars feature

Please perform appropriate configuration of feature chars

- securityProfile
  - passwordPolicy
    - expiration
    - chars
      - Feature Group
  - permissionSet

Feature Group

Please choose suitable number of grouped features: min 2, max 4

specialChar

lowerCase
digit
upperCase

Figure 8. Configuration wizard in FeaturePlugin

A rendering option for the described user interface is *top-down configuration*, where the subtrees that have not been selected are not visible. Alternatively, these subtrees can be shown in a different color. Other possible configuration interfaces are forms or wizards, which can be generated from feature models. For example, FeaturePlugin also offers a configuration wizard (see Figure 8).

4.4 Configuration Choices within a Stage

Some multi-staged configuration scenarios assume that any configuration choice available within a feature model could be made at a given configuration stage, unless it was not made in a previous stage. In other words, configuration choices are not assigned to be made in a specific stage. An example of such a scenario is the staged configuration of security policies in Section 4.2. This kind of multi-staged configuration is achieved simply through specialization. For example, at the enterprise-level configuration stage, any

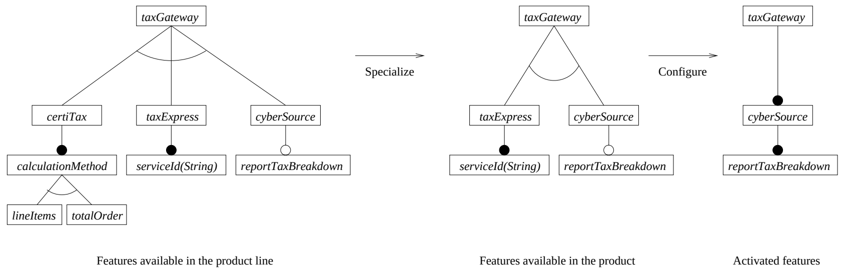


Figure 9. Tax-gateway configuration through specialization

desired specialization steps can be applied to security policies provided by the computing infrastructure in order to produce the desired enterprise-level security profile.

Other multi-staged configuration scenarios may require certain configuration choices to be made within certain stages. This is usually the case in the context of software supply chains, where stages are performed by parties playing different roles, with each role being responsible for a certain set of configuration choices. Typically, one role would make all the choices it is responsible for, before the next role would take over. Furthermore, the choices one role makes may influence which choices will be available to the next role.

As an example, consider a product line of electronic shops, which is configured in two stages. The first stage is the product-line configuration stage, where the desired features of the shop to be instantiated from the product line are specified. This could be done, for example, through a web interface offered at the web site of the product-line vendor. The desired system could then be generated based on the configuration and posted for download. The second stage is system-level configuration, when the generated system is configured at the customer's installation site. In each stage, a different set of configuration choices may need to be resolved. As an example, consider the configuration of tax gateways. A tax gateway is an electronic service (e.g., a web service) offered by a third party and used by an electronic shop to calculate the taxes due for the products being sold. At the product-line configuration stage, support for several gateways could be available, say, CertiTax, TaxExpress, and CyberSource. At that stage, one or more gateways can be selected to include the necessary support code in the system to be generated. However, at the system-level configuration stage, only one gateway from those available in the installation (which depends on the configuration in the first stage) has to be selected as the currently active one. The active gateway handles all tax calculations. Furthermore, the activation of a tax gateway requires further parameters such as the service subscription identification, whether taxes should be calculated per line item or for the total purchase, or whatever other parameters a given gateway provider may require. Specifying latter configuration parameters is clearly the responsibility of whoever is managing a given shop installation.

The two-staged configuration of tax gateways can be expressed using specialization, as illustrated in Figure 9.[10] In this approach, the product-line configuration role starts with the product-line-level feature model and specializes it into the system-level feature model. However, the problem with this approach is that there is no explicit indication in the model which configuration choices need to be performed by which role. When using specialization, the first role could already specify gateway parameters such as the service subscription identification or the calculation method, although these choices should be left to the second configuration role.

One way of assigning configuration choices to stages is the idea of annotating features with *binding times*, as suggested by Kang et al. (1990), where binding times are essentially *stages* in our model. Unfortunately, assigning stages to features turns out to be quite limiting. In order to see the limitations, let us analyze different situations. First of all, assigning just one stage to a feature is too limiting because the selection of that feature may need to be done within two or more stages. In our example, tax gateways are available for selection in both stages, except those eliminated in the first stage. In other words, we may need to annotate a feature with more than one stage. Furthermore, we may want different variabilities

[10] Note that we could also have used an inclusive-or group in the initial feature model without invalidating the specialization sequence; however, using an exclusive-or group in that model is more adequate as it reflects that the final configuration should include only one tax gateway.

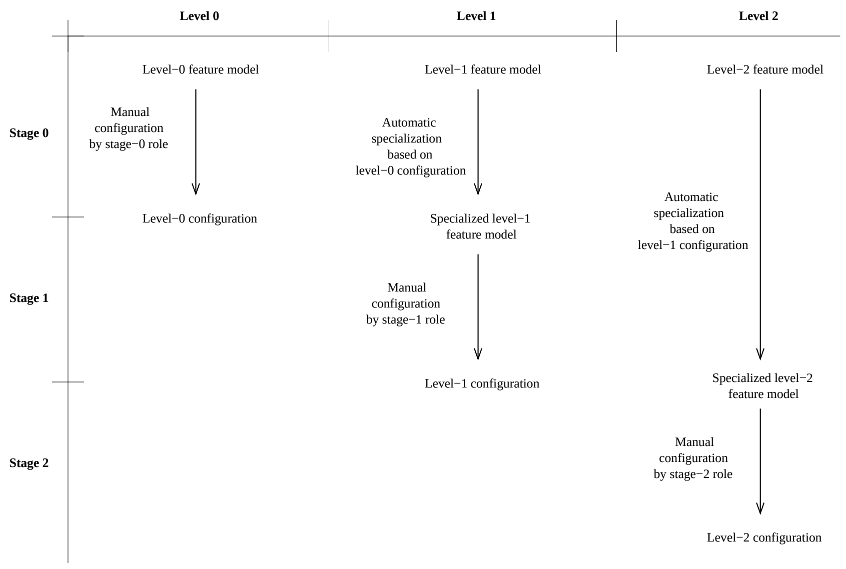

Level 0
Level 1
Level 2
Stage 0
Stage 1
Stage 2
Level−0 feature model
Manual configuration by stage−0 role
Level−0 configuration
Level−1 feature model
Automatic specialization based on level−0 configuration
Specialized level−1 feature model
Manual configuration by stage−1 role
Level−1 configuration
Level−2 feature model
Automatic specialization based on level−1 configuration
Specialized level−2 feature model
Manual configuration by stage−2 role
Level−2 configuration

Figure 10. Three-level configuration

to exist at different stages. In our tax gateway example, we really want tax gateways to be grouped in a inclusive-or group at the first stage, and then we want the tax gateways selected in the first stage to be grouped in an exclusive-or group in the second stage. This would allow us to use the more convenient and intuitive configuration operations rather than the specialization steps in the first stage. In general, we might want features and groups to have different cardinalities at different stages, i.e., we might want to have several cardinalities annotated with different stages for the same feature or group. Finally, the same features can have different subfeature relationships or can be grouped differently in different stages. We will see concrete examples for the latter situation in Figure 14 in Section 6. A global requirement on these annotations is that all the elements annotated with a given stage form a proper feature model. Consequently, in the remainder of this paper, we will consider developing separate feature models for each stage, rather than trying to annotate one model with stages. This leads us to the concept of *multi-level configuration*.

4.5 Multi-Level Configuration

Multi-level configuration is a form of staged configuration where the choices available to each stage are represented by separate feature models. The concept is illustrated in Figure 10, which shows a three-level arrangement. Each of the level-n feature diagrams (at the top of Figure 10) is created manually, then their staged configuration proceeds top-down in the figure. The main idea is that the configuration choices available at stage n are represented by the (possibly specialized) level-n feature model available at the beginning of that stage. For example, the configuration choices available at stage 0 are represented by the level-0 feature model. Similarly, the configuration choices available at stage 1 are represented by the *specialized* level-1 feature model available at the beginning of stage 1. The manual configuration of the level-n feature model at stage n triggers an automatic specialization of the level-$n+1$ feature model (by a mechanism that is explained later). That specialization represents then the configuration choices available at stage $n+1$. Observe that each stage entails manual configuration (rather than specialization) by the corresponding role.

Figure 11 shows a two-level arrangement for the tax-gateway configuration example from the previous section. The level-0 diagram describes the configuration choices available at the product-line level, whereas the level-1 diagram describes the configuration choices available at the system level. The figure corresponds to the specialization scenario from Figure 9. Note that, whereas the initial diagram in Figure 9 has an exclusive-or group, the corresponding group in the level-0 diagram in Figure 11 is inclusive-or. This is because the selection of one active gateway has been postponed to the system-level configuration, i.e., the level-1 diagram.

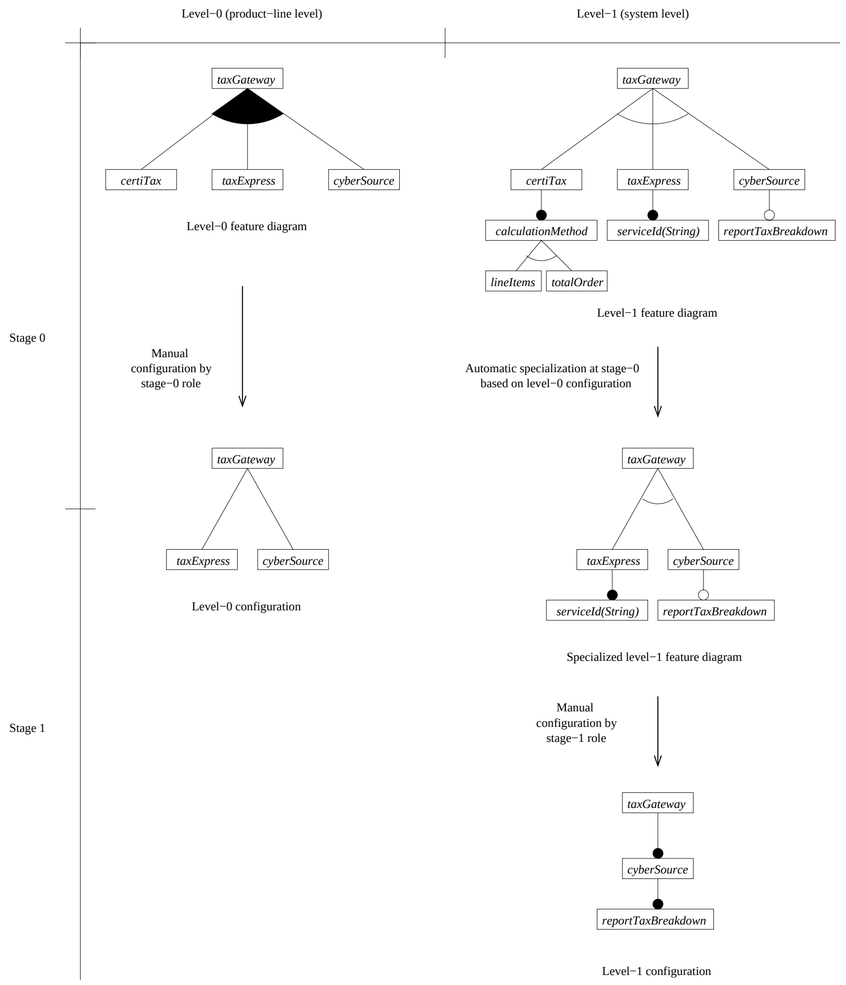


Figure 11. Two-level tax-gateway configuration

In addition to providing a separate feature model for each level, we also need a mechanism to specify how a level-n feature diagram should be automatically specialized based on a level-$n-1$ configuration. In the remainder of this section, we propose one such mechanism, which is both simple and expressive.

The automatic specialization of a feature model at level n is defined by *specialization annotations* attached to features in that model. A specialization annotation consists of two elements:

– a *condition*, which is a Boolean algebra formula over features from the level-$n-1$ feature model, e.g., $f_1 \wedge (\overline{f_2} \vee f_3)$; the formula can be interpreted w.r.t. a level-$n-1$ configuration: f_i is *true* if feature $\texttt{f}_i$ is part of the configuration or *false* otherwise.
– a *specialization type*, which is either *negative*, *positive*, or *complete*; the specialization type controls which of the specialization steps from Table 1 is applied to the annotated feature according to Table 2.

Table 1. Specialization operations *remove* and *select*

Specialization operation	**Precondition**	**Effect**
remove(f,m)	f is a solitary feature with cardinality $[0..n]$ in feature model m	refine feature cardinality of f to $[0..0]$
remove(f,m)	f is a grouped feature in a group with k features and cardinality $\langle n_1 - n_2 \rangle$, where $n_1 < k$, in feature model m	remove grouped feature (see Section 4.2)
select(f,m)	f is a solitary feature with cardinality $[0..n]$, where $n > 0$, in feature model m	refine feature cardinality of f to $[1..n]$
select(f,m)	f is a grouped feature in a group with cardinality $\langle n_1 - n_2 \rangle$, where $0 < n_2$, in feature model m	select grouped feature (see Section 4.2)

Table 2. Meaning of specialization types *negative*, *positive*, and *complete*

Specialization type	**Condition value**	**Specialization operation applied**
negative	*true*	(no operation)
negative	*false*	*remove*
positive	*true*	*select*
positive	*false*	(no operation)
complete	*true*	*select*
complete	*false*	*remove*

At most one annotation may be attached to a single feature and we have to ensure that the preconditions of Table 1 are not violated. These conditions help avoiding conflicts during specialization, such as trying to remove a mandatory feature.

The specialization annotations for a feature model at level-n can be represented as a function s^n that maps an annotated feature to a tuple consisting of a condition and a specialization type:

$$s^n : \mathcal{F}^n \rightarrow (\mathcal{B}(\mathcal{F}^{n-1}) \times \mathcal{T})$$

where

– $\mathcal{F}^n$ is the set of features in level-n feature model,
– $\mathcal{F}^{n-1}$ is the set of features in level-$n-1$ feature model,
– $\mathcal{B}(\mathcal{F}^{n-1})$ is the set of all Boolean formulas over the features in $\mathcal{F}^{n-1}$, and

Table 3. Specialization annotations of the level-1 feature diagram from Figure 11

Annotated feature	Condition	Specialization type
`certiTax`^{1}	$certiTax^0$	*negative*
`taxExpress`^{1}	$taxExpress^0$	*negative*
`cyberSource`^{1}	$cyberSource^0$	*negative*

– $\mathcal{T} = \{negative, positive, complete\}$.

Given a level-n feature model m^n with specialization annotations defined by s^n and a level-$n-1$ configuration c^{n-1}, the automatic specialization of m^n at the end of stage $n-1$ is performed as follows (in pseudocode):

```
for each $F \in \mathcal{F}^n$ such that $s^n(F) = (B, T)$
    if $eval(B, c^{n-1})$ then
        if $T = positive \vee T = complete$ then
            $select(F, m^n)$
    else
        if $T = negative \vee T = complete$ then
            $remove(F, m^n)$
```

where $eval(B, c^{n-1})$ is a function evaluating the Boolean formula B w.r.t. the configuration c^{n-1}.

Observe that executing this specialization algorithm may potentially violate group or other constraints in the feature model being specialized. For example, a given configuration at level $n-1$ may imply the selection of two alternatives from one group of alternative features. Such violations could be reported during the algorithm's execution, or static analysis could be used to derive a set of constraints for the level-$n-1$ feature model, such that valid configurations at level $n-1$ can only lead to valid specializations at level n.

The necessary specialization annotations for the level-1 feature diagram of the tax-calculation example in Figure 11 are given in Table 3. Feature names have their level specified as superscript for clarity.

Let us take a look how these annotations control the automatic specialization of the level-1 feature diagram in Figure 11. After the stage-0 role has created the level-0 configuration, the previously given specialization algorithm is applied to the level-1 feature diagram. According to Table 3, each of the features `certiTax`, `taxExpress`, and `cyberSource` is annotated with an equally named level-0 feature as condition and the `negative` configuration type. Because the corresponding level-0 features are absent from the level-0 configuration, the level-1 diagram is specialized by removing all three annotated features from it, that is, `certiTax`, `taxExpress`, and `cyberSource`.

Observe that the presented mechanism utilizes only a limited set of specialization operations, i.e., select and remove, and cannot clone features, unfold feature diagram references, or arbitrarily refine feature or group cardinalities. This makes the mechanism simple, but also sufficiently expressive for a large class of practical problems, in particular when mapping product-line-level to system-level feature models (see Section 6). If necessary, it could be extended or replaced by a more sophisticated mechanism.

5 Process and Applicability of Feature Modeling

A common pitfall in feature modeling is "analysis paralysis", which amounts to a breakdown between the different abstraction levels of individual features. This often happens while specifying common and variable features of a system without having clear guidelines on what level of abstraction is relevant for the current analysis. Because discovering one feature may lead to the discovery of other features, it is easy to create large feature models with lots of features at different levels of abstractions, some of which may end up being irrelevant for the analysis. For example, when creating a feature model of a product line of electronic shops, we could include optional features such as weight and size for product information in the model. Alternatively, the feature model could only distinguish among the choices of whether the product information should be appropriate for either physical or electronic products or both and leave the details of what exact information should be recorded for each product type to ordinary data modeling. The features weight or size are simply not relevant for analyzing the product line. Guidelines to evaluate such modeling alternatives are needed.

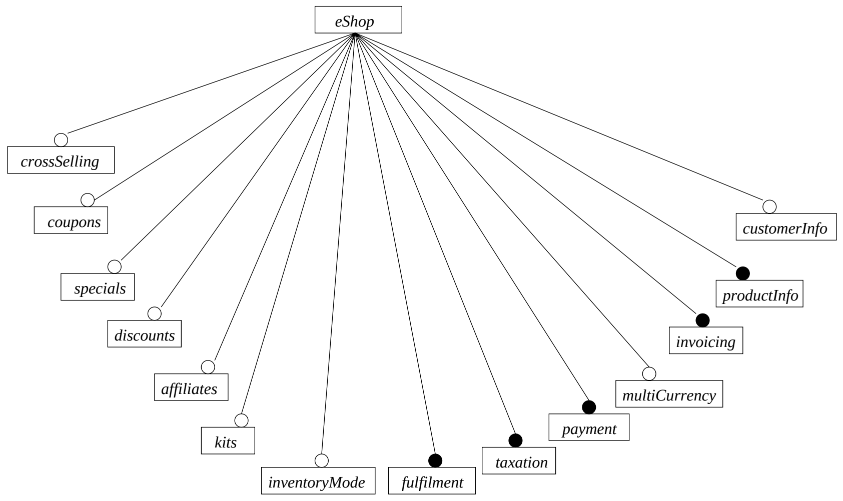
Multi-level configuration promotes the idea of modeling the configuration choices available in each configuration stage separately. This idea can help us to avoid "analysis paralysis" because each stage focuses on a particular level of abstraction. Before starting with feature modeling, the relevant roles that will be responsible for making configuration choices need to be identified. For example, for our product line of electronic shops, there is the role of specifying the desired shop that has to be generated, the role of a system administrator specifying security settings, gateway and database connections, and other settings for a shop installation, and the role of a shop owner that needs to specify in which currencies product prices should be displayed and what shipment methods are available.

Once the configuration roles are determined, feature models for each role need to be created. Creating a feature model from the viewpoint of a configuration role provides the necessary focus to avoid the breakdown of different feature modeling abstraction levels. The appropriateness of having a certain variability in a feature model can be tested by answering the following questions:

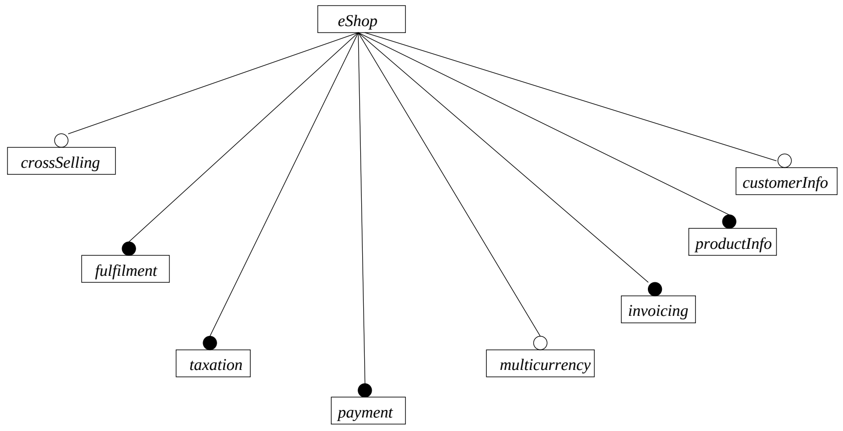
1. *Is the configuration choice that the variability affords the responsibility of the role?* Only a positive answer to this question warrants the inclusion of a given variability in a feature model. This test ensures that only the variabilities that a given role cares about will be included. Observe that multi-level configuration allows assigning responsibility for a given configuration choice to more than one role.
2. *Is the configuration choice implied by some other choices (either by other roles or by the same role)?* If a given choice is implied by other choices, it should not be included in the feature model. For example, deciding whether selling electronic and/or physical products will be supported by a shop implies the necessary information that needs to be stored for each product. Thus, there is no need to include individual features such as optional weight and size in the feature model.
3. *Is the variability more part of runtime behavior and data variability than a configuration choice?* Variability is an inherent part of any computation and data. Algorithms consist of choice and iteration statements, and data varies at runtime. Feature modeling is not appropriate for capturing all kinds of variability, but it is meant for modeling configuration choices. The distinction between configuration choices and the runtime variability in algorithms and data which we do not consider configuration choices is not always clear-cut. However, configuration is typically one meta-level above the subject matter being configured, and configuration data is more static than the primary runtime data being processed. For example, the configuration decision on which tax-gateway connection components will be included in an installation is clearly a "meta-level" decision with respect to the code. Also, the tax-gateway configuration decision—which may even be a runtime one if the components can be downloaded and dynamically linked into the system during its operation—is usually more static than, say, the price of a given product. In other words, we expect prices to change more frequently than the set of tax-gateway connection components included in the system. As another example, consider the list of predefined countries and their provinces which are available when specifying an address. Although the meta-level criterion seems less applicable in this case, the list is likely to be relatively static, and it would not be unusual to consider it as configuration data. Finally, feature modeling is particularly well suited for representing configuration information that benefits from its tree-like representation. Although graph-like structures can also be expressed in feature models using reference attributes (Czarnecki et al., 2002), runtime data typically contains complex relationships, and the latter are more appropriately modeled using UML class diagrams or ER diagrams.[11]

Up to this point, we assumed the ideal situation in which the configuration roles are entirely known before creating feature models. In practice, this might not be the case. Some configuration roles might be

[11] It should be noted that configuration information can sometimes also be inherently graph-like, e.g., when directly representing an arrangement of components and connectors. The latter is the domain of architecture description languages (ADLs) (e.g., Medvidovic and Taylor, 2000) rather than feature modeling. However, a feature model may be used to describe a configuration in terms of high-level features to be mapped to a realization in an ADL.



Level-0 feature diagram (product-line level)



Level-1 feature diagram (system level)

Figure 12. Feature diagrams of an electronic shop

discovered during feature modeling. They may come from considering the product configuration workflow, which may be imposed by a supply chain scenario, or they may be introduced purely in order to improve the modularity of the feature models created so far. In general, feature modeling and role determination is an iterative and incremental process, during which new roles can be introduced, existing roles can be removed or replaced, and features may need to moved from one role to another.

6 Larger Example and Some Observations

Figures 12 trough 14 show a larger two-level example for the electronic shop. Figure 12 shows the upper part of the product-line-level and system-level feature diagrams. Further details are shown for fulfilment and taxation in Figure 13 and Figure 14, respectively. The corresponding annotations of system-level diagrams are given in Tables 4 through 6. The example is based on a larger case study, which involved the analysis of several shopping-cart software packages.

Based on the electronic-shop case study and two other case studies described by Czarnecki et al. (2002), which involve embedded satellite and automotive software, we can make the following observations:

Table 4. Specialization annotations of the level-1 feature diagram from Figure 12

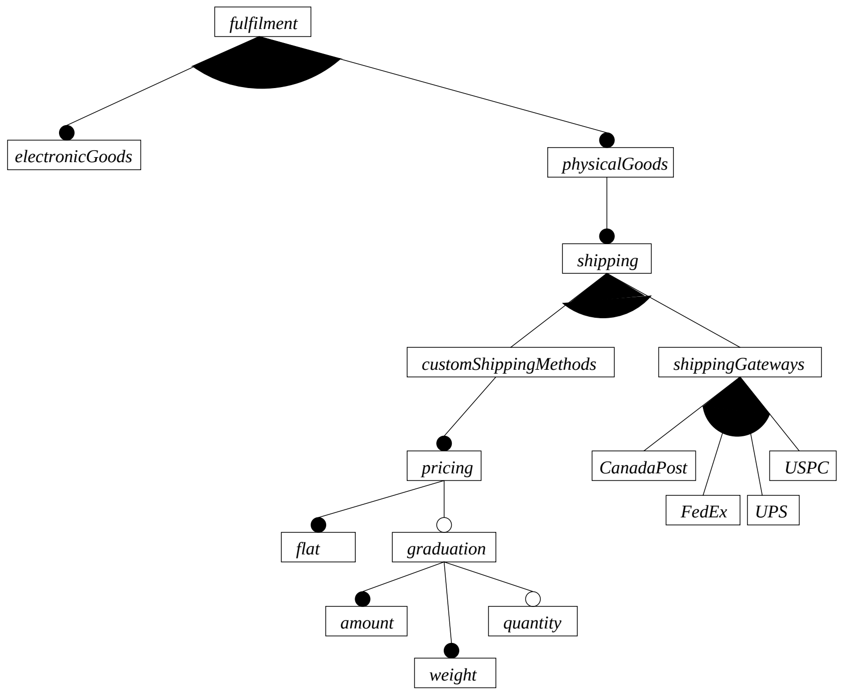
Annotated feature	**Condition**	**Specialization type**
`crossSelling`^{1}	$crossSelling^0$	*negative*
`multicurrency`^{1}	$multicurrency^0$	*negative*
`customerInfo`^{1}	$customerInfo^0$	*negative*

Table 5. Specialization annotations of the level-1 feature diagram from Figure 13

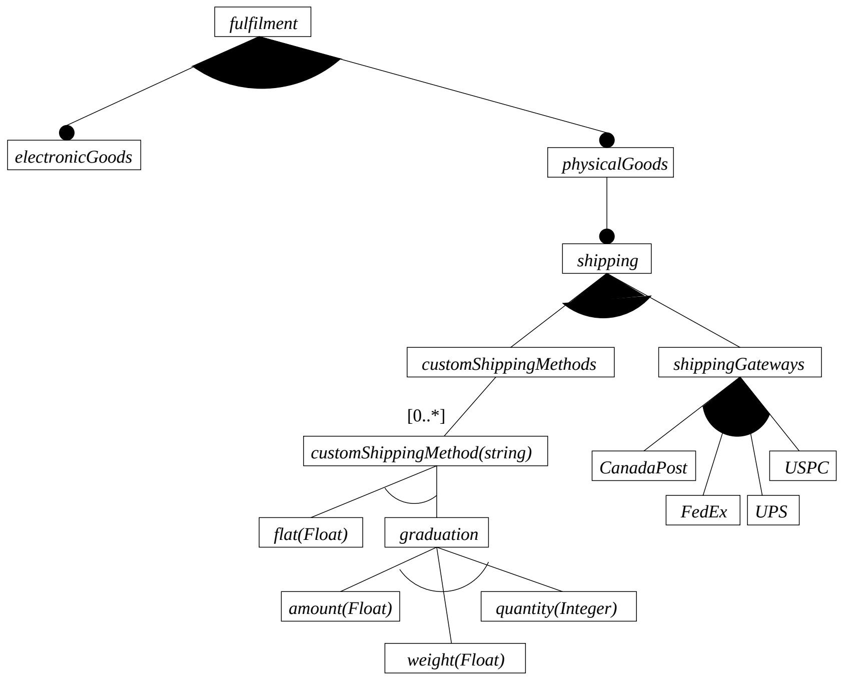
Annotated feature	**Condition**	**Specialization type**
`electronicGoods`^{1}	$electronicGoods^0$	*negative*
`physicalGoods`^{1}	$physicalGoods^0$	*negative*
`customShippingMethods`^{1}	$customShippingMethods^0$	*negative*
`graduation`^{1}	$graduation^0$	*negative*
`quantity`^{1}	$quantity^0$	*negative*
`shippingGateways`^{1}	$shippingGateways^0$	*negative*
`CanadaPost`^{1}	$CanadaPost^0$	*negative*
`FedEx`^{1}	$FedEx^0$	*negative*
`UPS`^{1}	UPS^0	*negative*
`USPS`^{1}	$USPS^0$	*negative*

Table 6. Specialization annotations of the level-1 feature diagram from Figure 14

Annotated feature	**Condition**	**Specialization type**
`customTaxation`^{1}	$customTaxation^0$	*negative*
`taxRules`^{1}	$taxRules^0$	*negative*
`address`^{1}	$address^0$	*negative*
`billing`^{1}	$billing^0$	*negative*
`postalCode`^{1}	$postalCode^0 \wedge streetAddress^0$	*negative*
`streetAddress`^{1}	$streetAddress^0$	*negative*
`taxGateways`^{1}	$taxGateways^0$	*negative*
`certiTax`^{1}	$certiTax^0$	*negative*
`taxExpress`^{1}	$taxExpress^0$	*negative*
`cyberSource`^{1}	$cyberSource^0$	*negative*

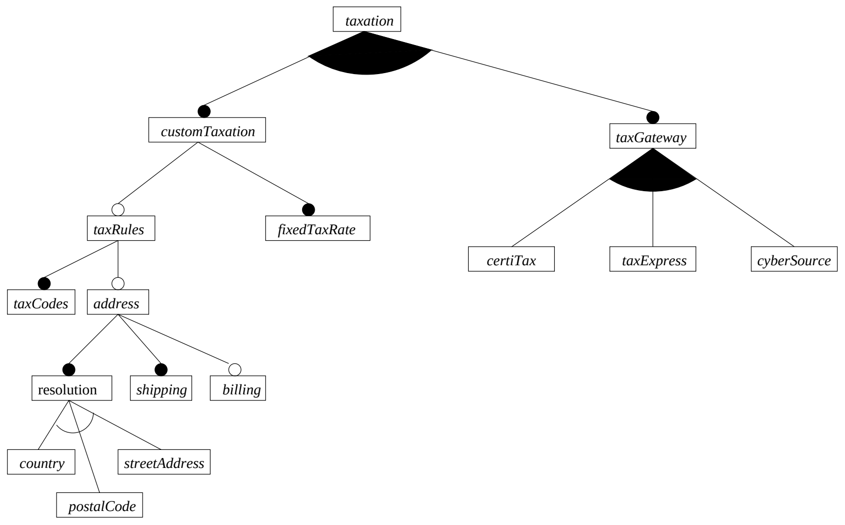


Level-0 feature diagram (product-line level)

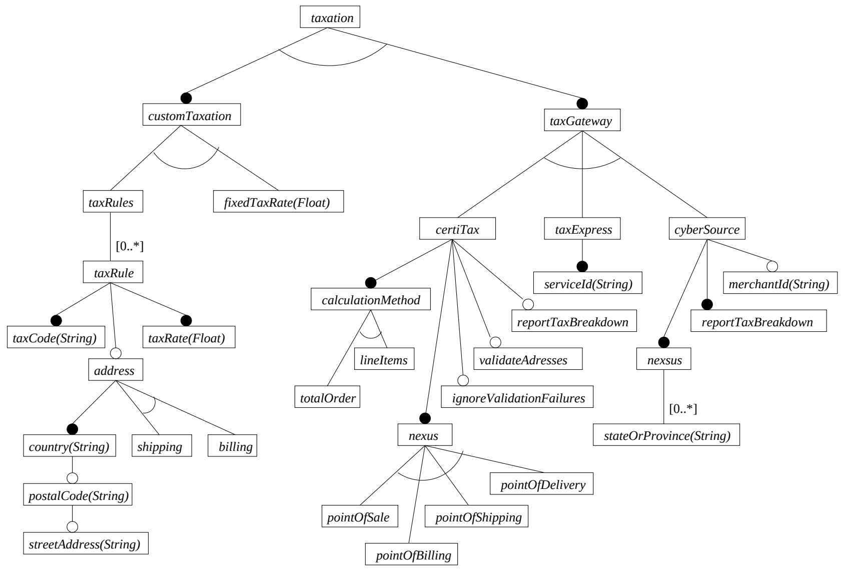


Level-1 feature diagram (system level)

Figure 13. Feature diagrams of fulfilment



Level-0 feature diagram (product-line level)



Level-1 feature diagram (system level)

Figure 14. Feature diagrams of taxation

– Product-line-level variability typically includes optional features and inclusive-or groups. Exclusive-or groups are sometimes also used. However, since the choice of one feature from a set of alternatives can always be delegated to the system level (such as in our tax gateway example in Figure 11), inclusive-or groups are more common at the product-line level than exclusive-or ones. Feature cardinalities with an upper bound greater than one are quite rare at the product-line level because product-line level configuration is primarily concerned with making choices among existing structures rather than creating new structures. Feature attributes are also quite rare at the product-line level.
– System-level variability typically includes exclusive-or and inclusive-or groups, and optional features. Inclusive-or groups at the product-line level often become exclusive-or groups at the system level. Feature cardinalities with an upper bound greater than one and feature attributes are also common at the system level, whose configuration often entails the creation of new structures. In our electronic shop example, attributes and cardinalities are needed to create multiple shipping methods and tax rules. This kind of use of feature models is close to modeling runtime data. However, because shipping methods and tax rules are relatively static and also have a simple structure, it is appropriate to represent them using feature models. Another example that requires attributes and cardinalities is component configuration, as described by Czarnecki et al. (2002). In that example, a user-defined number of tasks or services with differently configured properties need to be created during system configuration.
– The vast majority of specialization annotations controlling the specialization of system-level diagrams based on product-line-level diagrams are of negative type and use one feature, typically with the same name as the annotated feature, as a condition. Again, the reason is that the product-line level is mainly concerned with selecting existing structures at the system level, rather than constructing new ones. However, extending our specialization annotation mechanism to cover cloning of features and unfolding of feature diagram references could be useful for describing mappings between different system- and component-level models. Furthermore, although the user should be able to see a separate feature diagram for each level, a feature modeling tool could enable internal sharing of identical substructures across feature diagrams for different levels and assist the user with maintaining consistency.
– While the presented multi-level examples have only two levels, more levels may occur in practice. A typical example is a three-level arrangement with multiple-product-line level, product-line level, and system level. A big product line with a vast number of parameters may be organized into multiple product lines by selecting a set of high-level criteria such as the geographical area (e.g., Asia, Europe, and US) or the market segment (e.g., lower or upscale market segment) being targeted. Such multiple product lines are common, for example, in the automotive industry (e.g., Bühne, Lauenroth, Pohl and Weber, 2004). The criteria used to distinguish between the multiple product lines can be captured in a level-0 feature model, with a level-1 feature model used for product-line configuration and a level-2 feature model used for system configuration.
– In the discussion so far, we only considered a sequence of configuration stages. However, as shown in Figure 3, stages can sometimes be carried out in parallel, which requires a merge operation of the results. Such a merge is trivial if the different parts (Figure 3) are separate subtrees that are not connected by constraints; otherwise, conflicts have to be reconciled during the subsequent merge or during the parallel configuration stages.

7 Related Work

We already reviewed related work on feature modeling notations in Section 2. Related work on configuration in Artificial Intelligence is discussed elsewhere (Czarnecki et al., 2005, Section 6). A brief overview of existing tools for feature modeling is given in another paper (Antkiewicz and Czarnecki, 2004, Section 6). In the reminder of this section, we point to related work on incorporating time into variability modeling and configuration.

The original FODA report suggested annotating features with binding time, being either compile-time, activation-time, or runtime (Kang et al., 1990). Simos, Creps, Klinger, Levine and Allemang (1996) pointed out that binding occurs relative to development and operational processes or settings of a system. They generalized the notion of binding time to "binding site", which is any distinct point in a setting where a feature is bound. The three dimensions of configuration stages in Section 4, namely time, target (or context), and role, correspond to Czarnecki's (1998) characterization of binding site as "when, where, and by whom a feature is bound." In this work, we argue that creating separate feature models for different stages seems more practical than annotating a single feature model with different binding times or sites (Section 4.4).

Van Gurp et al. (2001) observe that variability has its own lifecycle that needs to be managed and binding time is just one point within that lifecycle. Their model includes four steps:

1. *identification of variability.* The need for variability is identified as part of requirements.
2. *constraining variability.* This step involves deciding when a variability will be introduced into design and implementation, when and how variants will be added to a system, and when a particular variant will be bound (*binding time*).
3. *implementing variability.* Based on the constraints from the previous step, a given variability needs to be implemented using a suitable variability implementation technology.
4. *managing variability.* Managing variability involves perfective and corrective maintenance, adding new variability, pruning old, no longer used variants, distribution of new variants to an already installed customer base, etc.

A related issue is the ability to configure the binding time in an implementation, which has been referred to as "parameterizing binding time of a composition mechanism" (Czarnecki and Eisenecker, 2000, p. 272) or "any-time variability" (van der Hoek, 2004). For example, a variation point introduced during design can be bound or configured for runtime binding at compile-time (Czarnecki and Eisenecker, 2000, Figure 8-6).

Dolstra, Florijn, de Jonge and Visser (2003) proposed *timeline variability*, which captures the evolution of system configuration as a transition system. The nodes of the transition system represent valid configurations, with a configuration being a set of values assigned to a set of variation points. Transitions may be triggered or enabled by time and changes to other context variables. In this model, variation points can be bound and re-bound at specific times, as specified by the transition system. Although timeline variability is more general than multi-level configuration, the latter model seems easier to expose to end-users performing staged configuration. However, timeline variability seems more appropriate for systems that need to maintain proper configuration over time. An example would be context-aware systems, in which automatic reconfiguration needs to be triggered based on some context variables such as network connectivity, battery level, location, etc.

8 Summary and Conclusions

Cardinality-based feature modeling provides an expressive way to describe configuration choices in product lines and systems. Our proposal integrates a number of concepts from existing notations such as group and feature cardinalities, attributes, and feature diagram references, and attempts to strike a balance among expressiveness, generality, and simplicity. Compared to the original FODA notation, feature cardinalities seem to be the extension with the largest semantic impact because they enable feature cloning.

Staged configuration of cardinality-based feature models is a useful and important mechanism for software supply chains based on product lines. We show that configuration stages may reflect different times, contexts, and roles in a configuration process, which can be represented as a workflow model. Furthermore, we propose specialization and multi-level configuration as two different mechanisms for performing staged configuration. In multi-level configuration, each stage has its own set of configuration choices that need to be performed in that stage, whereas configuration choices in specialization are not confined to any particular stage. We show that creating separate feature models for different binding times is more general and practicable than annotating a single feature model with binding times. We propose a simple mechanism to define mappings between levels. More elaborate mapping mechanisms are left for future work.

Finally, creating separate feature models for different stages allows us to avoid "analysis paralysis", a common pitfall in feature modeling, which amounts to a breakdown between the different abstraction levels of individual features. Performing feature analysis from the viewpoint of a particular configuration role allows us to stay focused and test whether a particular variability needs to be recorded.

Acknowledgements

We would like to thank Michal Antkiewicz for fruitful discussions about the metamodel presented in this paper. We also would like to thank Thomas Bednasch for his work on CaptainFeature, which helped us to advance our understanding of feature modeling.

This research has been in part supported by NSERC Discovery Grant #262100-03.

References

Antkiewicz, M. and Czarnecki, K. (2004). FeaturePlugin: Feature modeling plug-in for Eclipse, *OOPSLA'04 Eclipse Technology eXchange (ETX) Workshop*. Paper available from `http://www.swen.uwaterloo.ca/~kczarnec/etx04.pdf`. Software available from `gp.uwaterloo.ca/fmp`.

Barbeau, M. and Bordeleau, F. (2002). A protocol stack development tool using generative programming, *in* D. Batory, C. Consel and W. Taha (eds), *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02), Pittsburgh, October 6-8, 2002*, Vol. 2487 of *Lecture Notes in Computer Science*, Springer-Verlag, Heidelberg, Germany, pp. 93–109.

Batory, D., Johnson, C., MacDonald, B. and von Heeder, D. (2002). Achieving extensibility through product-lines and domain-specific languages: A case study", *ACM Transactions on Software Engineering and Methodology (TOSEM)* **11**(2): 191–214.

Bednasch, T. (2002). *Konzept und Implementierung eines konfigurierbaren Metamodells für die Merkmalmodellierung*, Diplomarbeit, Fachbereich Informatik und Mikrosystemtechnik, Fachhochschule Kaiserslautern, Standort Zweibrücken, Germany. Available from `http://www.informatik.fh-kl.de/~eisenecker/studentwork/dt_bednasch.pdf` (in German).

Bednasch, T., Endler, C. and Lang, M. (2002-2004). CaptainFeature. Tool available on SourceForge at `https://sourceforge.net/projects/captainfeature/`.

Beuche, D. (2003). *Composition and Construction of Embedded Software Families*, PhD thesis, Otto-von-Guericke-Universität Magdeburg, Germany. Available from `http://www-ivs.cs.uni-magdeburg.de/~danilo`.

Bosch, J. (2000). *Design and Use of Software Architecture: Adopting and evolving a product-line approach*, Addison-Wesley, Harlow, England.

Bühne, S., Lauenroth, K., Pohl, K. and Weber, M. (2004). Modelling features for multi-criteria product-lines in automotive industry, *Workshop on Software Engineering for Automotive Systems (SEAS), co-located at ICSE 2004, Edinburgh*.

Cechticky, V., Pasetti, A., Rohlik, O. and Schaufelberger, W. (2004). XML-based feature modelling, *in* J. Bosch and C. Krueger (eds), *Software Reuse: Methods, Techniques and Tools: 8th International Conference, ICSR 2004, Madrid, Spain, July 5-9, 2009. Proceedings*, Vol. 3107 of *Lecture Notes in Computer Science*, Springer-Verlag, Heidelberg, Germany, pp. 101–114.

Cleaveland, C. (2001). *Program Generators with XML and Java*, Prentice-Hall, Upper Saddle River, NJ.

Clements, P. and Northrop, L. (2001). *Software Product Lines: Practices and Patterns*, Addison-Wesley, Boston, MA.

Czarnecki, K. (1998). *Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models*, PhD thesis, Technische Universität Ilmenau, Ilmenau, Germany. Available from `http://www.prakinf.tu-ilmenau.de/~czarn/diss`.

Czarnecki, K., Bednasch, T., Unger, P. and Eisenecker, U. W. (2002). Generative programming for embedded software: An industrial experience report, *in* D. Batory, C. Consel and W. Taha (eds), *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02), Pittsburgh, October 6-8, 2002*, Vol. 2487 of *Lecture Notes in Computer Science*, Springer-Verlag, Heidelberg, Germany, pp. 156–172.

Czarnecki, K. and Eisenecker, U. W. (2000). *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, Boston, MA.

Czarnecki, K., Helsen, S. and Eisenecker, U. (2004). Staged configuration using feature models, *in* R. L. Nord (ed.), *Software Product Lines: Third International Conference, SPLC 2004, Boston, MA, USA, August 30-September 2, 2004. Proceedings*, Vol. 3154 of *Lecture Notes in Computer Science*, Springer-Verlag, Heidelberg, Germany, pp. 266–283.

Czarnecki, K., Helsen, S. and Eisenecker, U. (2005). Formalizing cardinality-based feature models and their specialization, *Software Process Improvement and Practice* **10**(1).

DeBaud, J.-M. and Schmid, K. (1999). A systematic approach to derive the scope of software product lines, *Proceedings of the 21st International Conference on Software Engineering (ICSE)*, IEEE Computer Society Press, Los Alamitos, CA, pp. 34–43.

Deursen, A. v. and Klint, P. (2002). Domain-specific language design requires feature descriptions, *Journal of Computing and Information Technology* **10**(1): 1–17. Available from `http://homepages.cwi.nl/~arie/papers/fdl/fdl.pdf`.

Dolstra, E., Florijn, G., de Jonge, M. and Visser, E. (2003). Capturing timeline variability with transparent configuration environments, *in* J. Bosch and P. Knauber (eds), *ICSE Workshop on Software Variability Management (SVM'03)*, Portland, Oregon. Available from `http://www.cs.uu.nl/research/techreps/UU-CS-2003-051.html`.

Greenfield, J. and Short, K. (2004). *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*, Wiley, Indianapolis, IN.

Griss, M., Favaro, J. and d'Alessandro, M. (1998). Integrating feature modeling with the RSEB, *Proceedings of the Fifth International Conference on Software Reuse (ICSR)*, IEEE Computer Society Press, Los Alamitos, CA, pp. 76–85.

Hein, A., Schlick, M. and Vinga-Martins, R. (2000). Applying feature models in industrial settings, *in* P. Donohoe (ed.), *Proceedings of the Software Product Line Conference (SPLC1)*, Kluwer Academic Publishers, Norwell, MA, pp. 47–70.

Jones, N. D., Gomard, C. K. and Sestoft, P. (1993). *Partial evaluation and automatic program generation*, Prentice-Hall, Inc., Englewood Cliffs, NJ. Electonic version available from `http://www.dina.dk/~sestoft/pebook/pebook.html`.

Kang, K., Cohen, S., Hess, J., Nowak, W. and Peterson, S. (1990). Feature-oriented domain analysis (FODA) feasibility study, *Technical Report CMU/SEI-90-TR-21*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

Lee, K., Kang, K. C. and Lee, J. (2002). Concepts and guidelines of feature modeling for product line software engineering, *in* C. Gacek (ed.), *Software Reuse: Methods, Techniques, and Tools: Proceedings of the Seventh Reuse Conference (ICSR7), Austin, USA, Apr. 15-19, 2002*, Vol. 2319 of *Lecture Notes in Computer Science*, Springer-Verlag, Heidelberg, Germany, pp. 62–77.

Lohmann, D., Schröder-Preikschat, W. and Spinczyk, O. (2005). Functional and non-functional properties in a family of embedded operating systems, *Proceedings of the Tenth IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2005), February 2-4, 2005, Sedona, AZ, USA*. Available from `http://www4.informatik.uni-erlangen.de/~spinczyk/pubs.shtml`.

Medvidovic, N. and Taylor, R. N. (2000). A classification and comparison framework for software architecture description languages, *IEEE Transactions on Software Engineering* **26**(1): 70–93.

OMG (2002). *Software Process Engineering Metamodel Specification.* Adopted Specification, formal/02-11-14, Object Management Group, Inc. Available from `http://www.omg.org`.

Parnas, D. (1976). On the design and development of program families, *IEEE Transactions on Software Engineering* **SE-2**(1): 1–9.

Riebisch, M., Böllert, K., Streitferdt, D. and Philippow, I. (2002). Extending feature diagrams with UML multiplicities, *6th Conference on Integrated Design & Process Technology (IDPT 2002), Pasadena, California, USA*.

Simos, M., Creps, D., Klinger, C., Levine, L. and Allemang, D. (1996). Organization Domain Modeling (ODM) Guidebook, Version 2.0, *Technical Report STARS-VCA025/001/00*, Lockheed Martin Tactical Defence Systems, Manassas, VA. Informal Technical Report for Software Technology for Adaptable, Reliable Systems (STARS).

van der Hoek, A. (2004). Design-time product line architectures for any-time variability, *Science of Computer Programming* **53**(3): 285–304.

van Gurp, J., Bosch, J. and Svahnberg, M. (2001). On the notion of variability in software product lines, *Proceedings of The Working IEEE/IFIP Conference on Software Architecture (WICSA)*, IEEE Computer Society Press, Washington, DC, pp. 45–55.

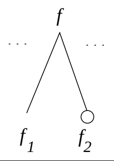
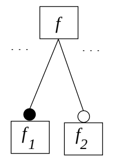
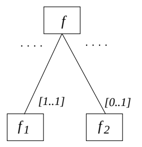
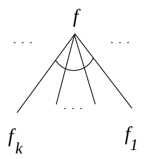
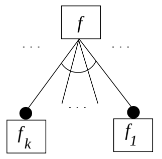
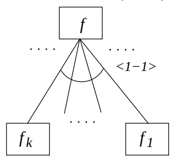
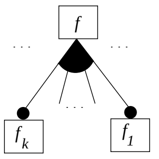
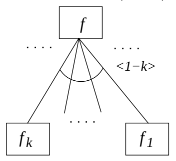
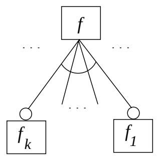
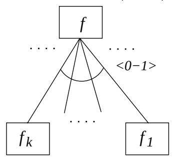
Weiss, D. M. and Lai, C. T. R. (1999). *Software Product-Line Engineering: A Family-Based Software Development Process*, Addison-Wesley, Boston, MA.

A Overview of Feature Modeling Notations

The cardinality-based feature modeling notation of this paper is a continuation of existing modeling notations (Czarnecki and Eisenecker, 2000; Riebisch et al., 2002). Table 7 compares the current proposal with the extended notation by Czarnecki (1998) and the FODA notation.

Table 7 compares three different notations for feature diagrams. The leftmost column shows the FODA notation (Kang et al., 1990) which has mandatory (f_1), optional (f_2), and alternative subfeatures ($f_k \dots f_1$). In the extended notation (Czarnecki, 1998; Czarnecki and Eisenecker, 2000), depicted in the middle column, alternative groups come in two flavors: inclusive-or and exclusive-or groups. Moreover,

Table 7. Comparison of feature modeling notations

FODA notation (Kang et al., 1990)	**Extended notation (Czarnecki, 1998; Czarnecki and Eisenecker, 2000)**	**Cardinality-based notation**
mandatory and optional subfeatures 	*mandatory and optional subfeatures* 	*mandatory and optional subfeatures* 
alternative subfeatures 	*exclusive-or group* 	*group with cardinality* $\langle 1\text{-}1\rangle$ 
n/a	*inclusive-or group* 	*group with cardinality* $\langle 1\text{-}k\rangle$ 
n/a	*exclusive-or group with optional subfeatures* 	*group with cardinality* $\langle 0\text{-}1\rangle$ 

an exclusive-or group can also have optional subfeatures. The right column shows some possibilities of the cardinality-based notation that have an equivalent diagram in the extended notation. Of course, the cardinality-based notation allows for many additional features and feature groups that cannot be expressed in either the FODA notation or the extended notation.

However, to improve readability, we suggest using the extended notation of the middle column whenever possible, except for the use of filled circles above grouped features that belong to a feature group. A feature modeling tool may provide the appropriate *syntactic sugar* for those cases.