

Staged Configuration Using Feature Models

Krzysztof Czarnecki¹, Simon Helsen¹, and Ulrich Eisenecker²

¹ University of Waterloo, Canada

² University of Applied Sciences Kaiserslautern, Zweibrücken, Germany

Abstract. Feature modeling is an important approach to 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. This is important because in a realistic development process, different groups and different people eliminate product variability in different stages. We also indicate how cardinality-based feature models and their specialization can be given a precise formal semantics.

1 Introduction

Feature modeling is a key approach to capturing and managing the common and variable features of systems in a system family or a product line. In the early stages of software family development, feature models provide the basis for scoping the system family by recording and assessing information 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 [1]. Later, feature models play a central role in the development of a system family architecture, which has to realize the variation points specified in the feature models [2, 3]. In application engineering, feature models can drive requirements elicitation and analysis. Knowing which features are available in the software family may help customers decide which features their system should support. Knowing which desired features are provided by the system family and which have to be custom-developed helps to better estimate the time and cost needed for developing the system. A software pricing model could also be based on the additional information recorded in a feature model.

Feature models also play a key role in generative software development [2, 4–7]. 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). In this context, feature models are used to scope and develop DSLs [2, 8], which may range from simple parameter lists or feature hierarchies to more sophisticated DSLs with graph-like structures.

Feature modeling was proposed as part of the Feature-Oriented Domain Analysis (FODA) method [9], and since then, it has been applied in a number of domains including telecom systems [10, 11], template libraries [2], network

protocols [12], and embedded systems [13]. Based on this growing experience, a number of extensions and variants of the original FODA notation have been proposed [10, 11, 13–17].

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 based on specializing feature models and illustrate how specialization can be achieved in a sound way. Finally, we briefly indicate how a cardinality-based feature model can be formalized. The details of this formalization are elaborated elsewhere [18].

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 gives a glimpse of an approach to formalize feature models. Appendix A gives a comparison of three different notations for feature modeling.

2 Background and Related Work

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 between systems. 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 descendent nodes are features. In the FODA feature diagram notation (see the leftmost column of Table 1 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 and associates [9], 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 [13] (after they were initially rejected [2]).
- *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 thesis [14]: the inclusive-or group and the inclusive-or group with optional subfeatures (see the middle column of Table 1 in Appendix A).³ The concept of groups was further generalized in [17] 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 1 in Appendix A).
- *attributes*. Attributes were introduced by Czarnecki and associates [13] 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 Bednash [19] 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 a desired type.
- *relationships*. Several authors [10, 11, 16] 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. FeatuRSEB [10] proposes functional, architectural, or 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, and exemplar systems [2, 14].

³ Inclusive-or features were introduced independently by Czarnecki [14] and Griss and associates [10]. However, inclusive-or features in Griss and associates’ work [10] imply reuse-time binding, whereas inclusive-or features in Czarnecki’s work [14] are independent of binding time.

- *modularization*. A feature diagram may contain one or more special leaf nodes, each standing for a separate feature diagram [19]. This mechanism allows the breaking up of large diagrams into smaller ones and the reuse of common parts in several places. This is an important mechanism because, in practice, feature diagrams can 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 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 [19]. In contrast to Bednasch’s work [19], 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 [13] 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 (e.g., [17] and the example 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 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 [19, 20] 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 and associates [13], we think that they are better handled as user-defined, structured annotations. Such annotations are also supported through an extensible metamodel [19].

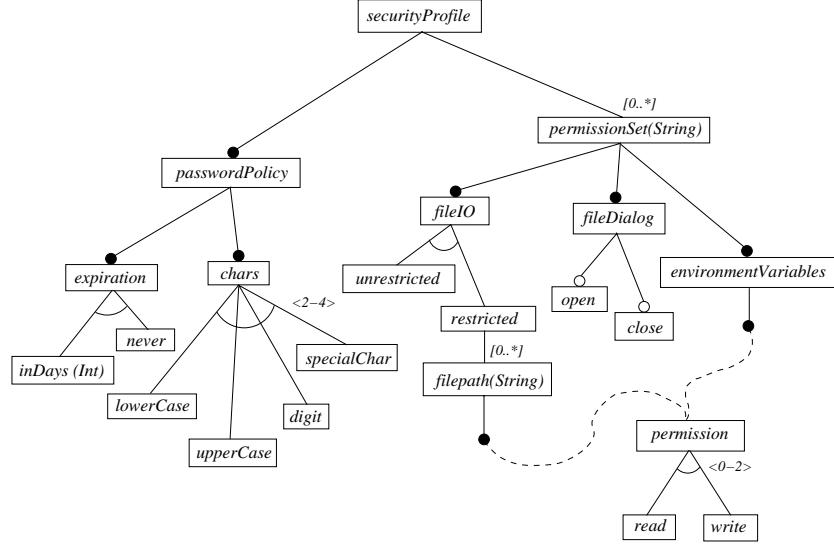


Fig. 1. Security profile example

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 Fig. 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 only 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; for example, `myFeature (Int)`. It is also possible to specify a value of the associated type immediately with the type; for example, `myFeature (5 : Int)`. The constraints on the kind of characters required in a password are specified by a feature group with cardinality $\langle 2-4 \rangle$. This means that any actual password policy must specify between two and four (different) requirements on the kind of characters.

In our example, since no cardinality was specified for the group of **expiration** policies, the cardinality $\langle 1-1 \rangle$ 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 a feature cardinality.

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

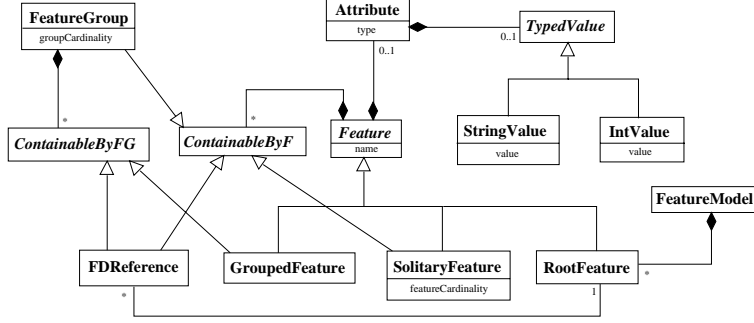


Fig. 2. UML metamodel for cardinality-based feature models

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 Fig. 1 for clarification.

Consider the Unified Modeling Language (UML) metamodel for cardinality-based feature models in Fig. 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 kind 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, the feature **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.

Fig. 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*.⁴ It specifies how often the solitary subfeature (and any possible subtree) can be duplicated as a child of f . A feature cardinality I is a sequence of intervals of the form $[n_1..n'_1] \dots [n_l..n'_l]$, where we assume the following invariants:

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

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* $\langle n-n' \rangle$, 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: $0 \leq n \leq n' \leq k$.

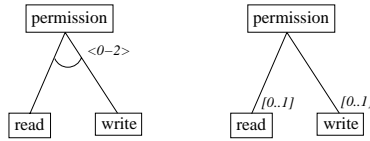
At this point, we ought to mention that, theoretically, we could generalize the notion of group cardinality to be a sequence of intervals as we did for feature cardinalities. However, we have found no practical applications of this usage, and it would only clutter the presentation.

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 work [14]. For example, in that notation, an inclusive-or group

⁴ More precisely, a feature cardinality is attached to the solitary subfeature *relationship*. This relationship is implicit in the metamodel.

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 4.3). For example, the duplication 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 of *preferred form* which is shown by default.

In Appendix A, we discuss a comparison of the new cardinality-based notation with the FODA notation and the notation introduced in [2, 17]. 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.

4 Staged Configuration

4.1 Motivation

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 group 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 *proper* subset of the systems described by the feature model to be specialized.

The need for staged configuration arises in the context of *software supply chains* [7]. 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. Basic services such as tasking support, network drivers, network management, flash support, diagnosis, and so forth, are

implemented as components by different software vendors. A software vendor may deliver different configurations of its component to different car manufacturers to reflect the differing requirements of the individual manufacturers (such as different terminologies or provided interfaces). Doing so constitutes the first configuration stage. In a second stage, each car manufacturer has to further configure the components 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 second configuration stage may be even more complex, as some global settings and available options may be determined by the manufacturer, while other settings may be provided by the suppliers of the control functions. Finally, given a concrete configuration, the code implementing the basic services is generated. Based on the outlined requirements, it should be possible to perform configuration in stages and to compose (possibly specialized) feature models.

In general, supply chains require staged configuration of platforms, components, and services. However, staged configuration may be required even within one organization. For example, security policies could be configured in stages for an entire enterprise, its divisions, and the individual computers. The enterprise-level configuration would determine the choices available to the divisions, and the divisions would determine the choices available to the individual computers.

4.2 Configuration vs. Specialization

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 *configuration*. *Specialization* 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 true 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. Finally, *staged configuration* is a form of configuration achieved by successive specialization followed by deriving a configuration from the most specialized feature diagram in the specialization sequence.

In general, we can have the following two extremes when performing configuration: a) deriving a configuration from a feature diagram *directly* and b) specializing a feature diagram down to a full specialization and then deriving the configuration (which is trivial). Please note that sometimes we might not be interested in arriving at one specific configuration. For example, a feature diagram that still contains unresolved variability could be used as an input to a generator. This could be 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.

4.3 Specialization Steps

In Section 4.1, we described the process of staged configuration as the removal of possible configuration choices. In this section, we discuss in more detail what kind of configuration choices can be eliminated. We will call the removal of a certain configuration choice a *specialization step*.

There are six categories of specialization steps: a) refining a feature cardinality, b) refining a group cardinality, c) removing a grouped feature from a feature group, d) assigning a value to an attribute which only has been given a type, e) *cloning* a solitary subfeature, and f) unfolding a feature reference. We discuss each of these possibilities in more detail below.

Refining feature cardinalities. A feature cardinality is a sequence of intervals representing a (possibly infinite) set of distinct natural numbers. Each natural number in the cardinality stands for an accepted number of occurrences for the solitary subfeature. Refining a feature cardinality means to eliminate elements from the subset of natural numbers denoted by the cardinality. This can be achieved as follows:

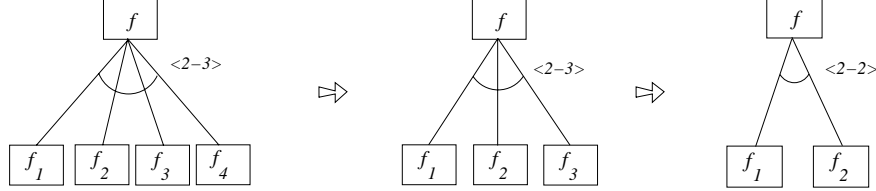
1. remove an interval from the sequence; or
2. if an interval is of the form $[n_i..n'_i]$ where $n_i < n'_i$,
 - (a) reduce the interval by increasing n_i to n''_i or decrease the n'_i to n'''_i as long as $n''_i \leq n'''_i$. If $n'_i = *$, then it is possible to replace $*$ with a number m such that $n_i \leq m$; or
 - (b) split the interval in such a way that the new sequence still obeys the feature cardinality invariant and then reduce the split intervals.

A special case of refining a feature cardinality is to refine it to $[0..0]$ or ε . In either case, it means we have removed the entire subfeature and its descendants. We leave it up to a tool to decide whether to visually remove features with feature cardinality $[0..0]$ or ε .

Refining group cardinalities. A group cardinality $\langle n_1 - n_2 \rangle$ is an interval indicating a minimum and maximum number of distinct grouped features to be chosen from the feature group. Its form is a simplification of a feature cardinality and can only be refined by reducing the interval (i.e., by increasing n_1 to n'_1 and decreasing n_2 to n'_2 as long as $n'_1 \leq n'_2$). Currently, we do not allow *splitting* an interval because we have no representation for such a cardinality. Of course, such an operation should be incorporated whenever we allow sequences of intervals for group cardinalities.

Removing a grouped feature from a feature group. A feature group of size k with group cardinality $\langle n_1 - n_2 \rangle$ combines a set of k grouped subfeatures and indicates a choice of at least n_1 and at most n_2 distinct grouped features. A specialization step can alter a feature group by removing one of the grouped

subfeatures with all its descendents, provided that $n_1 < k$. The new feature group will have size $k - 1$, and its new group cardinality will be $\langle n_1 - \min(n_2, k - 1) \rangle$, where $\min(n, n')$ takes the minimum of the two natural numbers n and n' . The following is an example specialization sequence where each step removes one grouped subfeature from the group:



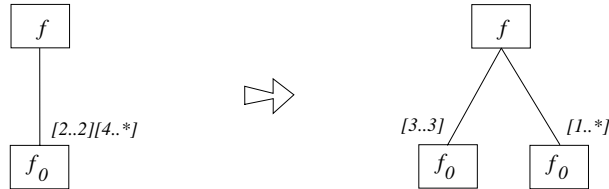
It is not possible to remove grouped subfeatures from a feature group once $n_1 = k$. In that case, all subfeatures *have* to be taken, and all variability is eliminated.

Assigning an attribute value. An obvious specialization step is to assign a value to an uninitialized attribute. The value has to be of the type of the attribute.

Cloning a solitary subfeature. This operation makes it possible to *clone* a solitary subfeature and its entire subtree, provided the feature cardinality allows it. Moreover, the cloned feature may be given an arbitrary, but fixed, feature cardinality by the user, as long as it is allowed by the original feature cardinality of the solitary subfeature.

Unlike the other specialization operations, cloning may *change* the diagram without removing variabilities. However, the new diagram will generally be more amenable to specialization, so we consider it a specialization step nonetheless.

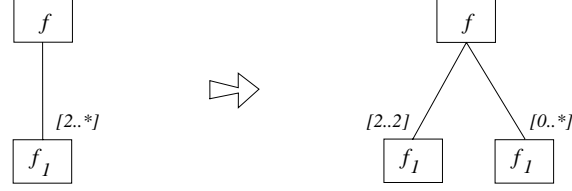
We explain this process with an example:



The feature cardinality $[2..2][4..*]$ of the original subfeature f_0 indicates that f_0 must occur at least two times or more than three times in a configuration. In the specialized diagram above, we have cloned f_0 (to the left) and assigned it a new feature cardinality $[3..3]$. The *original* feature f_0 (to the right) has a new cardinality that guarantees the new diagram does not allow configurations

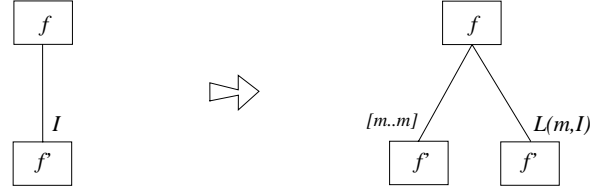
previously forbidden. In the example, because we have chosen to give the left f_0 the fixed feature cardinality [3..3], the feature cardinality of the rightmost f_0 has to be [1..*]. Specialization has occurred since the new diagram does not allow a user to select only f_0 two times.

Consider another example:



In this case, no actual specialization took place. The cloned feature f_1 to the left has been given the fixed cardinality [2..2]. However, because the original feature cardinality was [2..*], the rightmost f_1 now has cardinality [0..*].

More generally, suppose $I = [n_1..n'_1] \dots [n_l..n'_l]$ and $0 < m$. Provided we have $(n'_l = *) \vee (m \leq n'_l < *)$, it is possible to clone the solitary subfeature f' of f with feature cardinality I as follows:



Given that we always have $* - n = *$ for any $n \in \mathbb{N}$, we can define the function $L(m, I)$ as follows:

$$L(m, \varepsilon) = \varepsilon$$

$$L(m, [n..n']I) = \begin{cases} \text{if } (m \leq n) : [(n - m)..(n' - m)]L(m, I) \\ \text{if } (n < m) \wedge (m \leq n') : [0..(n' - m)]L(m, I) \\ \text{if } n' < m : L(m, I) \end{cases}$$

The reader may wonder why we only allow the cloned feature to be given a *fixed* feature cardinality. This is because, in general, it is impossible to construct a correct feature diagram by cloning a feature and giving it an arbitrary interval or sequence of intervals without some more expressive form of additional constraints. However, there are a few useful special cases for this situation that we do not consider in this paper and leave for future work.

Unfolding a feature diagram reference. Finally, we have a specialization step that allows the user to *unfold* a feature diagram reference. This basically means that we substitute the reference for the entire feature diagram it refers to by means of its root feature. Although this operation never removes variability, we consider it a specialization step for the same reasons as mentioned above for

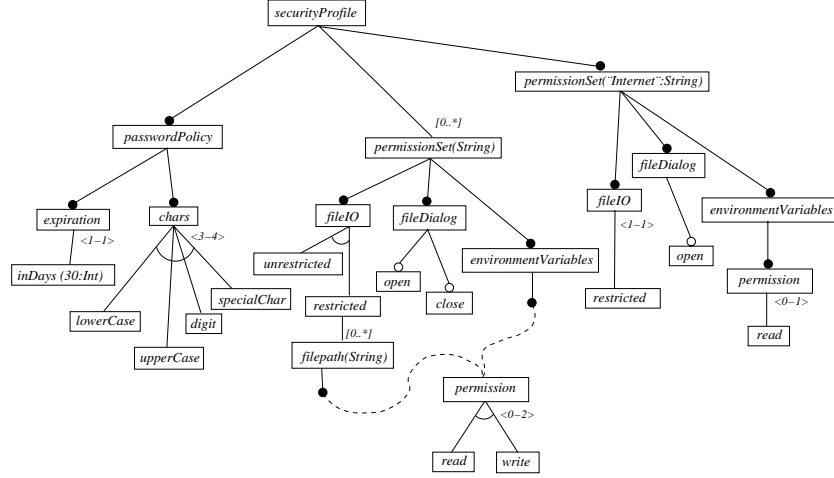


Fig. 3. Sample specialization of the security profile

the cloning of a solitary feature: it makes the new feature model potentially more amenable for specialization because each unfolded feature diagram can now be specialized differently.

4.4 Security Profiling Example Revisited

Let us now apply the notion of staged configuration to the security profile example from Section 3.1. Assume, for instance, that the IT infrastructure of a company supports the security profile from Fig. 1. The company then decides to specialize this profile to define a standard enterprise-level security profile as depicted in Fig. 3.

The specialization is achieved by a combination of steps from Section 4.3. 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. Moreover, 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.

5 Feature Models as Context-Free Grammars

The semantics of a feature model can be defined by the set of *all possible configurations*. A configuration itself is denoted by a structured set of features chosen according to the informal rules of interpretation of feature diagrams.

Although this description is sufficient in practice, it is helpful to provide a formal semantics to improve the understanding of some of the intricacies of feature modeling. For example, a formal semantics allows us to define exactly what it means when two apparently different feature models are *equivalent* (i.e., denote the same set of configurations).

More interestingly, if feature model specialization maps to the semantic interpretation of a feature model, it would be possible to formally establish that feature model specialization reduces the set of possible configurations.

Our approach to obtaining such a semantics is to cast feature models back into a well-known formalism: context-free grammars. The semantic interpretation of a feature diagram then coincides with a natural interpretation of the sentences recognized by the grammar. It is beyond the scope of this paper to provide a detailed account of exactly how a feature model can be translated into a context-free grammar, but the interested reader is invited to examine the details in an accompanying technical report [18].

We have to mention that feature diagrams, as they are presented in this paper, can actually be modeled as *regular grammars*⁵ and do not require the additional expressiveness of context-free grammars. The main reason to use context-free grammars instead is purely for convenience.

It is not only possible to specify cardinality-based feature models as a context-free grammar. Feature model specialization can be mapped onto a set of operations on the context-free grammar. Those operations are such that it is possible to determine that the set of all configurations of a specialized feature model is never greater than the set of configurations of the original feature model.

6 Conclusion and Future Work

Cardinality-based feature modeling provides an expressive way to describe feature models. Staged configuration of such cardinality-based feature models is a useful and important mechanism for software supply chains based on product lines.

AmiEddi [21,22] was the first editor supporting the feature modeling notation from [2]. As a successor of the first prototype, CaptainFeature [19,20] implements a cardinality-based notation that is similar to the one described in this paper. In future work, we plan to extend our model with external constraints and reference attributes [13]. Meanwhile, commercial tools supporting variant configuration for product lines are starting to emerge (e.g., GEARS [23] and Pure::Consul [24, 25]). Pure::Consul is even directly based on FODA-like feature modeling. More

⁵ Regular grammars require that the right-hand side of a production can only have one terminal possibly followed by a nonterminal.

advanced capabilities such as staged configuration and cardinality-based feature modeling still need to be addressed by these tools.

The study of specialization at the grammar level [18]) has helped to better understand possible specialization steps at the diagram level. In fact, such an analysis has revealed specialization steps other than those described in Section 4.3. Some of them can be quite involved, and some cannot even be translated back into the concrete syntax of the proposed feature diagram notation. The current specialization steps in this paper (Section 4.3) are an attempt to balance simplicity and practical relevance.

We think that specialization and direct configuration should be two distinct procedures. Although any desired configuration can be achieved through specialization, specialization offers finer grained steps that would be unnecessarily tedious for direct configuration. We already have experience with tool support for configuration based on existing tool prototypes: ConfigEditor [13] and CaptainFeature [19, 20]. Both support configuration in a strictly top-down manner. This contrasts with our approach where staged configuration of a feature model can be achieved in an arbitrary order.

Adequate tool support for the newly introduced cardinality-based feature notation as well as its specialization is under way.

Acknowledgements

We would like to thank Michal Ankiewicz for fruitful discussions about the meta-model 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.

References

1. DeBaud, J.M., Schmid, K.: A systematic approach to derive the scope of software product lines. In: Proceedings of the 21st International Conference on Software Engineering (ICSE), IEEE Computer Society Press (1999) 34–43
2. Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley (2000)
3. Bosch, J.: Design and Use of Software Architecture: Adopting and evolving a product-line approach. Addison-Wesley (2000)
4. Weiss, D.M., Lai, C.T.R.: Software Product-Line Engineering: A Family-Based Software Development Process. Addison-Wesley (1999)
5. Cleaveland, C.: Program Generators with XML and Java. Prentice-Hall (2001)
6. Batory, D., Johnson, C., MacDonald, B., von Heeder, D.: Achieving extensibility through product-lines and domain-specific languages: A case study”. ACM Transactions on Software Engineering and Methodology (TOSEM) **11** (2002) 191–214
7. Greenfield, J., Short, K.: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. Wiley (2004) To be published.
8. Deursen, A.v., Klint, P.: Domain-specific language design requires feature descriptions. Journal of Computing and Information Technology **10** (2002) 1–17

9. Kang, K., Cohen, S., Hess, J., Nowak, W., Peterson, S.: Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90TR -21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (1990)
10. Griss, M., Favaro, J., d' Alessandro, M.: Integrating feature modeling with the RSEB. In: Proceedings of the Fifth International Conference on Software Reuse (ICSR), IEEE Computer Society Press (1998) 76–85
11. Lee, K., Kang, K.C., Lee, J.: Concepts and guidelines of feature modeling for product line software engineering. In Gacek, C., ed.: Software Reuse: Methods, Techniques, and Tools: Proceedings of the Seventh Reuse Conference (ICSR7), Austin, USA, Apr.15-19, 2002. LNCS 2319, Springer-Verlag (2002) 62–77
12. Barbeau, M., Bordeleau, F.: A protocol stack development tool using generative programming. In Batory, D., Consel, C., Taha, W., eds.: Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02), Pittsburgh, October 6-8, 2002. LNCS 2487, Springer-Verlag (2002) 93–109
13. Czarnecki, K., Bednasch, T., Unger, P., Eisenecker, U.W.: Generative programming for embedded software: An industrial experience report. In Batory, D., Consel, C., Taha, W., eds.: Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02), Pittsburgh, October 6-8, 2002. LNCS 2487, Springer-Verlag (2002) 156–172
14. Czarnecki, K.: Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models. PhD thesis, Technical University of Ilmenau, Ilmanau, Germany (1998) Available from <http://www.prakinf.tu-ilmenau.de/~czarn/diss>.
15. Hein, A., Schlick, M., Vinga-Martins, R.: Applying feature models in industrial settings. In Donohoe, P., ed.: Proceedings of the Software Product Line Conference (SPLC1), Kluwer Academic Publishers (2000) 47–70
16. Svahnberg, M., van Gurp, J., Bosch, J.: On the notion of variability in software product lines. In: Proceedings of The Working IEEE/IFIP Conference on Software Architecture (WICSA). (2001) 45–55
17. Riebisch, M., Böllert, K., Streitferdt, D., Philippow, I.: Extending feature diagrams with UML multiplicities. In: 6th Conference on Integrated Design & Process Technology (IDPT 2002), Pasadena, California, USA. (2002)
18. Czarnecki, K., Helsen, S., Eisenecker, U.: Formalizing cardinality-based feature models and their staged configuration. Technical Report 04-11, Departement of Electrical and Computer Engineering, University of Waterloo, Canada (2004) <http://www.ece.uwaterloo.ca/~kczarnec/TR04-11.pdf>.
19. Bednasch, T.: Konzept und implementierung eines konfigurierbaren meta-modells für die merkmalsmodellierung. Diplomarbeit, Fachbereich Informatik, Fachhochschule Kaiserslautern, Standort Zweibrücken, Germany (2002) Available from http://www.informatik.fh-kl.de/~eisenecker/studentwork/dt_bednasch.pdf (in German).
20. Bednasch, T., Endler, C., Lang, M.: CaptainFeature (2002-2004) Tool available on SourceForge at <https://sourceforge.net/projects/captainfeature/>.
21. Selbig, M.: A feature diagram editor — analysis, design, and implementation of its core functionality. Diplomarbeit, Fachbereich Informatik, Fachhochschule Kaiserslautern, Standort Zweibrücken, Germany (2000)
22. Mario Selbig: AmiEddi (2000-2004) Tool available at <http://www.generative-programming.org>.
23. Krueger, C.W.: Software mass customization. White paper. Available from <http://www.biglever.com/papers/BigLeverMassCustomization.pdf> (2001)

24. pure-systems GmbH: Variant management with pure::consul. Technical White Paper. Available from <http://web.pure-systems.com> (2003)
25. Beuche, D.: Composition and Construction of Embedded Software Families. PhD thesis, Otto-von-Guericke-Universität Magdeburg, Germany (2003) Available from <http://www-ivs.cs.uni-magdeburg.de/~danilo>.

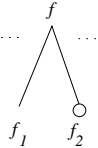
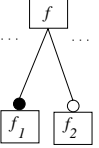
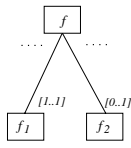
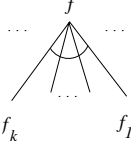
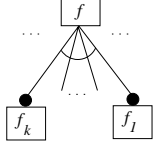
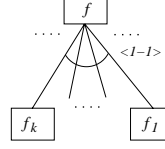
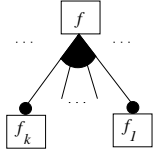
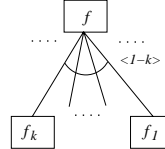
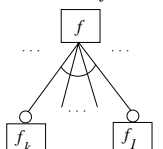
A Overview of Feature Modeling Notations

The cardinality-based feature modeling notation of this paper is a continuation of existing modeling notations [2, 17]. Table 1 compares the current proposal with the extended notation from [2, 17] and the FODA notation.

Table 1 compares three different notations for feature diagrams. The left-most column shows the FODA notation [9] which has mandatory (f_1), optional (f_2), and alternative subfeatures ($f_k \dots f_1$). In the extended notation [2, 14], depicted in the middle column, alternative groups come in two flavors: inclusive-or and exclusive-or groups. Moreover, 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.

Table 1. Comparison of feature modeling notations

FODA notation in [9]	Extended notation in [2, 14]	Cardinality-based notation
<p><i>mandatory and optional subfeatures</i></p> 	<p><i>mandatory and optional subfeatures</i></p> 	<p><i>mandatory and optional subfeatures</i></p> 
<p><i>alternative subfeatures</i></p> 	<p><i>exclusive-or group</i></p> 	<p><i>group with cardinality <1-1></i></p> 
<p>n/a</p>	<p><i>inclusive-or group</i></p> 	<p><i>group with cardinality <1-k></i></p> 
<p>n/a</p>	<p><i>exclusive-or group with optional subfeatures</i></p> 	<p><i>group with cardinality <0-1></i></p> 