Formalizing Cardinality-based Feature Models and their Staged Configuration

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

[1] University of Waterloo, Canada
[2] University of Applied Sciences Kaiserslautern, Zweibrücken, Germany

Abstract Feature modeling is an important approach to capture the commonalities and variabilities in system families and product lines. Cardinality-based feature modeling integrates a number of existing extensions of previous approaches. Staged configuration is a process that allows the incremental configuration of cardinality-based feature models. It is achieved by performing a step-wise specialization of the feature model.

In this paper, we argue that cardinality-based feature models can be interpreted as a special class of context-free grammars. We make this precise by specifying a translation from a feature model into a context-free grammar. Consequently, we provide a semantic interpretation for cardinality-based feature models by assigning an appropriate semantics to the language recognized by the corresponding grammar. Finally, we give an account on how feature model specialization can be formalized as transformations on the grammar equivalent of feature models.

1 Introduction

Feature modeling is a key approach to capture and manage the common and variable features 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, etc. [13].

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 [6, 10]. In application engineering, feature models can drive requirements elicitation and analysis. Knowing which features are available in the software family may help the customer to decide about the features his or her system should support. Knowing which of the 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, 7, 10, 15, 24]. 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 [10, 14], 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 in [19] and since then it has been applied in a number of domains including telecom systems [16, 20], template libraries [10], network protocols [1], and embedded systems [9]. Based on this growing experience, a number of extensions and variants of the original FODA notation have been proposed [8–11, 16, 18, 20, 22, 23].

1.1 Contributions and Overview

This paper is based on our previous work for cardinality-based feature models [11]. In that work, we introduce cardinality-based feature modeling as an integration and extension of existing approaches. We also introduce and motivate the notion of staged configuration. The present work adds a formal account of cardinality-based feature modeling and its specialization. The main contributions include a) a translation for cardinality-based feature models into context-free grammars b) a semantic interpretation of feature models c) a set of informal grammar transformation rules which mimics feature model specialization steps.

The remainder of the paper is organized as follows: in Section 2, we describe cardinality-based feature modeling as it was proposed in [11]. However, instead of using an example from operating system security profiling, we explain the feature modeling approach by means of a configurable editor. Section 3 describes the notion of staged configuration as introduced in [11]. It is motivated and applied to the configurable editor example. The interpretation of feature models as context-free grammars is then detailed in Section 4. This includes the introduction of an abstract syntax model (Section 4.1), a translation into context-free grammars (Section 4.2), and a semantics for feature models (Section 4.3). Section 5 informally describes how feature model specialization steps have equivalent grammar transformation rules. Related work is discussed in Section 6 and Section 7 concludes.

2 Cardinality-Based Feature Modeling

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 of features with the root representing a concept (*e.g.*, a software system). *Feature models* are feature diagrams plus *additional information* such as feature descriptions, binding times, priorities, stakeholders, etc.

As a practical example to demonstrate our cardinality-based language for feature modeling, consider the feature diagram of a configurable text editor in Figure 1.

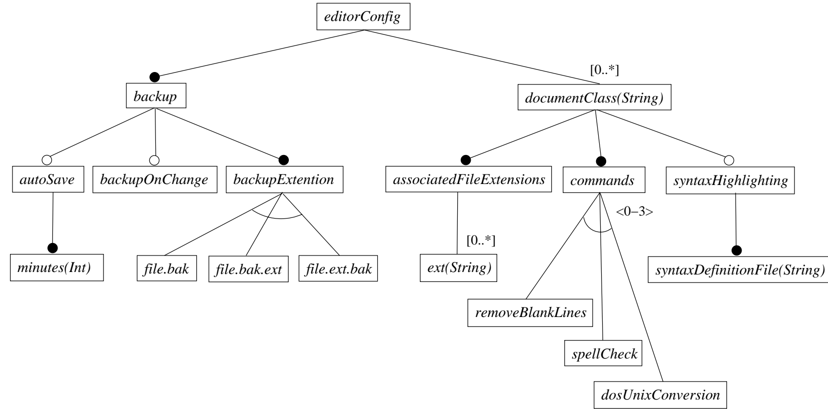


Figure 1. Text editor configuration example

The root feature of our sample diagram is `editorConfig`. 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 occurs in a *feature group*, *e.g.*, `removeBlankLines`. 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 features `backup` and `autoSave`.

The feature `minutes` is a subfeature of `autoSave` and has an attribute for specifying the time period between automatic saves. In general, any feature can be given maximum one attribute by associating it with a type, which is `Int` in the case of `minutes`. A collection of attributes can be modeled as a number of subfeatures, where each is associated with a desired type. If present, we indicate the associated type within parentheses after the feature name, *e.g.*, *myFeature (Int)*. It is also possible to specify a value of the associated type immediately with the type, *e.g.*, *myFeature (5 : Int)*.

Every solitary feature is qualified by a *feature cardinality*[3]. It specifies how often the solitary subfeature (and any possible subtree) can be duplicated as a child of its parent. For example, the features `documentClass` and `ext` have the feature cardinality $[0..*]$. This means that an editor can be configured for zero or more document classes, each with zero or more associated file extensions. The feature cardinality of the remaining solitary features are implied by the filled or empty circle above the given feature as $[1..1]$ or $[0..1]$, respectively. For example, `backup` has the cardinality $[1..1]$ (it is a mandatory feature), whereas the cardinality of `autoSave` is $[0..1]$ (it is an optional feature). In general, a feature cardinality I is a sequence of intervals of the form $[n_1..n'_1] \ldots [n_l..n'_l]$,

[3] More precisely, a feature cardinality is attached to the *relationship* between a solitary feature and its parent.

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.

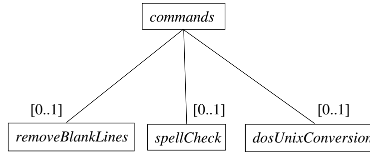
An example of a valid specification of a feature cardinality with more than one interval is [0..2][6..6], which says that we can take a feature 0, 1, 2 or 6 times. Note that we allow the last interval in a feature cardinality to have as an upper bound the Kleene star $*$. Such an upper bound denotes the possibility to take 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.

The available naming schema for back-up files (such as `file.bak` or `file.ext.bak`) are modeled as a *feature group*. In general, 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: $0 \leq n \leq n' \leq k$. In our example, since no cardinality was specified for the group of naming schemas, the cardinality $\langle 1\text{–}1 \rangle$ is assumed, *i.e.*, the naming schemas form an *exclusive-or* group. Conversely, any combination of zero or more subfeatures of `commands` are allowed thanks to the group cardinality $\langle 0\text{–}3 \rangle$.

At this point, we ought to mention that, theoretically, we could generalize the notion of group cardinality to be a sequence of intervals, similarly as 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 which was necessary for the notation in [8]. For example, in that notation, an *inclusive-or* group – which has group cardinality $\langle 1\text{–}k \rangle$ where k is the size of the group – can have both optional and mandatory subfeatures (corresponding to feature cardinalities [0..1] and [1..1], respectively). This 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 3.2). 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.

Observe that, 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 feature group with group cardinality $\langle 0\text{–}3 \rangle$ under the feature `commands` in Figure 1 can also be modeled as follows:



We keep this distinct from the representation in Figure 1 and leave it up to a tool implementer to decide how to deal with these multiple representations. 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.

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

The need for staged configuration arises in at least three contexts:

Software supply chains In software supply chains [15], a supplier can create families of software components, platforms, and/or services dedicated 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. In [11], we provide an example involving embedded software in the automotive industry.

Optimization Staged configuration offers an opportunity to perform optimizations based on partial evaluation. 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 over the net. 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.[4]

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 at the level of individual computers. A concrete example of specializing security policies is give in [11].

Configuration stages can be defined in terms of three different dimensions:

Time A configuration stage may defined in terms of the different phases in a product lifecycle (*e.g.*, development, 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 technologies (*e.g.*, compile-time, preprocessing 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 polices, 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.

3.1 Configuration vs. Specialization

So far, we have loosely defined the terms feature model specialization and feature model configuration. At this point, it is important to make 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 *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

[4] 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 (see [9, pp. 168-169] for a concrete example).

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. We now define *staged configuration* as a form of configuration, which is achieved by successive specialization steps 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 fully specialized feature diagram and then deriving the configuration (which is trivial). 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.

3.2 Specialization Steps

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

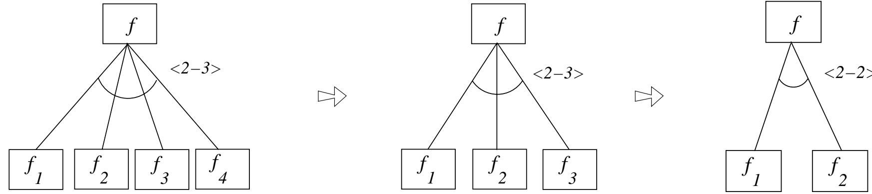
Refining feature cardinalities A feature cardinality is a sequence of intervals representing a (possibly infinite) set of distinct natural numbers. Each natural number in the cardinality stands for an accepted number of occurrences for the solitary subfeature. Refining a feature cardinality means to 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 descendents. 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 subfeatures 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 to *split* 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 subfeature from a group A feature group of size k with group cardinality $\langle n_1 - n_2 \rangle$ combines a set of k subfeatures and indicates a choice of at least n_1 and at most n_2 distinct subfeatures. A specialization step can alter a feature group by removing one of the 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 $\langle n_1 - \mathtt{min}(n_2, k-1) \rangle$, where $\mathtt{min}(n, n')$ takes the minimum of the two natural numbers n and n'. The following is an example specialization sequence where each step removes one subfeature from the group:



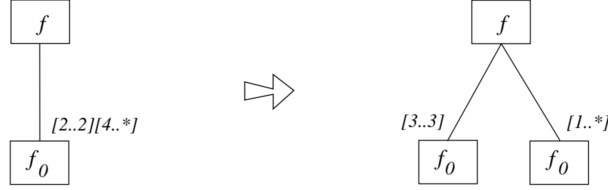
It is not possible to remove subfeatures from a feature group once $n_1 = k$. In that case, all subfeature *have* to be taken and all variability is eliminated.

Assigning an attribute value An obvious specialization step which assigns 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.

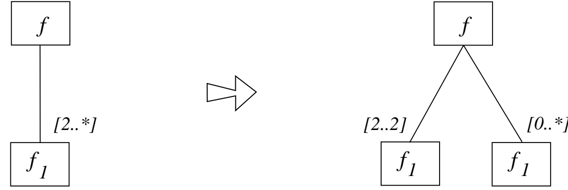
Unlike the other specialization operations, it is possible that cloning only *changes* the diagram without removing variabilities. However, the new diagram will generally be more amenable to specialization, so we consider it a specialization step nonetheless.

We explain this process with an example:



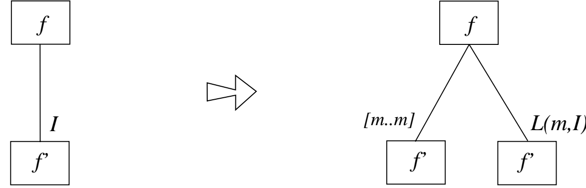
The feature cardinality $[2..2][4..*]$ of the original subfeature f_0 indicates that f_0 must occur at least 2 times or more than 3 times in a configuration. In the specialized diagram above, we have cloned f_0 (to the left) and assigned it a new feature cardinality $[3..3]$. The *original* feature f_0 (to the right) has a new cardinality that guarantees that the new diagram does not allow configurations which were 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 right-most f_0 has to be $[1..*]$. Specialization has occurred since the new diagram does not allow a user to only select 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 right-most f_1 now has cardinality $[0..*]$.

More generally, suppose $I = [n_1..n_1'] \ldots [n_l..n_l']$ 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:

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

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

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

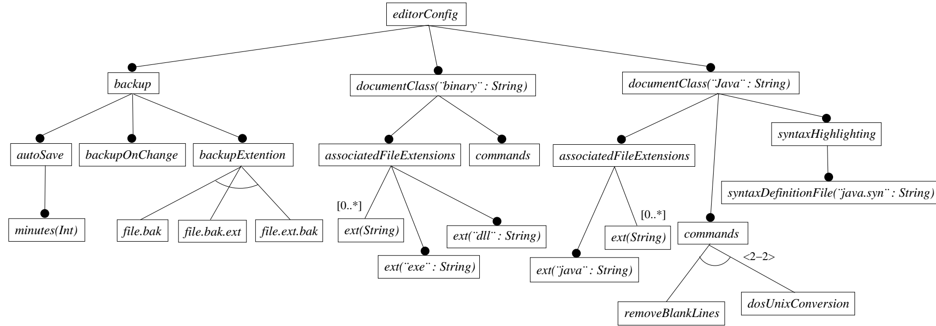


Figure 2. Sample specialization of the editor configuration

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, which we do not consider in this paper and leave for future work.

3.3 Editor Configuration Example Revisited

Coming back to our editor configuration example in Section 2, assume that the developer of the editor decides to create specialized editors for different markets. For example, a specialization for programmers might support editing Java files and viewing binary files (see Figure 2).

The specialization is achieved by a combination of steps from Section 3.2: refining the feature cardinalities for `autoSave` and `backupOnChange` to [1..1]; cloning `documentClass` twice and removing the original `documentClass` by refining its feature cardinality to [0..0]; assigning names to the cloned `documentClass`es; cloning `ext` and assigning names to each of them; deleting selected features from the groups under `commands`; and refining the group cardinality for the commands of the Java document class to $\langle 2\text{–}2 \rangle$.

The specialized feature diagram could be used as the input to a generator or as a start-up configuration file. The remaining variability could be exposed to the end-user through a properties dialog box in the generated editor.

4 Feature Models as Context-Free Grammars

Our approach to formalize cardinality-based feature models is based on a translation of feature diagrams into context-free grammars. A formal semantics for a feature model is then obtained by an appropriate interpretation of the sentences recognized by the corresponding grammar.

Before we can translate cardinality-based feature models into grammars, we need a more practical formulation for feature models. Our strategy is to define an *abstract syntax* and a corresponding meta model for the cardinality-based feature modeling notation introduced in Section 2.

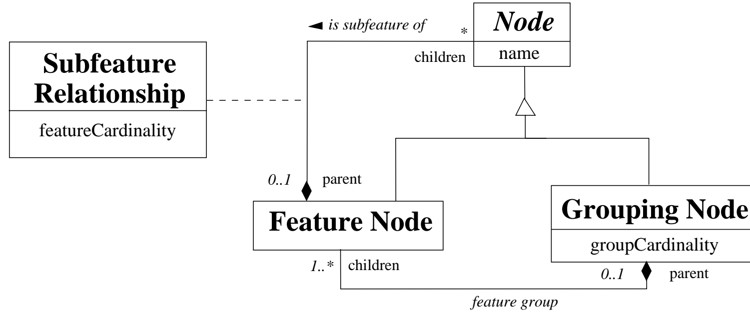


Figure 3. Abstract syntax meta model for cardinality-based feature models

4.1 Abstract Syntax for Feature Diagrams

Usually, an abstract-syntax model is engineered to accommodate the expressiveness of the meta modeling language. For example, if the meta model is expressed as a UML model, sound object-oriented design principles are used. In our case, however, we want the abstract syntax to be easily manipulable by a translation algorithm. This requires more orthogonality of the different language elements and may not necessarily lead to the most intuitive meta model.

Still, for improved readability, we formalize the abstract syntax model with a UML class diagram because this standardized notation is readily understandable. Figure 3 contains the meta model for the proposed abstract syntax of our cardinality-based feature notation[5]. An abstract syntax feature diagram consists of nodes which can be either *feature nodes* or *grouping nodes*. Each node has a name which refers to the actual feature name. Note that we differentiate between a feature name and a feature node. As is the case in the concrete syntax, feature names do not have to be unique. However, each feature node is a unique object.

A feature node may have a set of zero or more subnodes via the *subfeature relationship*, which in itself has a feature cardinality. Solitary subfeatures from the concrete syntax are thus represented in the abstract syntax as a node with an *is-subfeature-of* link.

A grouping node, on the other hand, has at least one or more subfeature nodes in its feature group. Observe that a grouping node cannot have an empty feature group because if a node has no children, it must be a feature node.

In addition to the diagram constraints of Figure 3, we have three implicit constraints: a) a grouping node always has an associated subfeature link; in other words, a diagram cannot have a grouping node as its root; b) the feature cardinality of the subfeature link associated with a grouping node is always [1..1]; c) an abstract syntax feature diagram is always a proper tree, *i.e.*, a node cannot appear as one of its descendents.

The attentive reader will notice that there is no one-to-one match between the abstract syntax model and the previously introduced concrete syntax. The

[5] Contrast this with the meta model proposed in [11].

main difference is that in the concrete syntax, we allow a feature to have a heterogeneous mix of solitary subfeatures and feature groups. In contrast, the abstract syntax insists on a homogeneous form of subfeaturing: each node either has solitary subfeatures or stands for a feature group. If we need a feature group to co-exist with solitary subfeatures, we have to introduce a grouping node, which can be treated as an anonymous solitary subfeature. This explains the requirement that the anonymous subfeature for a grouping node always has feature cardinality [1..1].

Although attributes are a useful and important aspect of the feature notation presented in this paper, we do not consider features with attributes in the formalization of feature models. This extension only clutters the presentation. In Section 4.2 we informally discuss how attributes can be added to the formalization.

Let us now consider a practical notation for the abstract syntax. A feature node and its subfeatures with associated feature cardinality have the following form:

$$f \left| \begin{array}{l} I_1\, f_1 \\ \vdots \\ I_n\, f_n \end{array} \right.$$

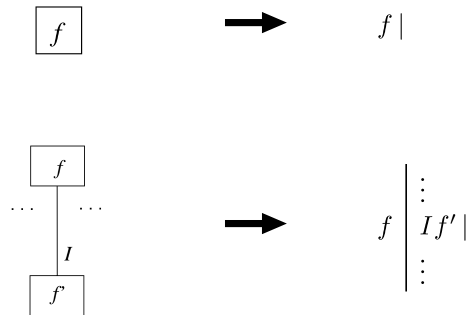
In this case, f is a feature node, whereas $f_1, \ldots, f_n$ are its subnodes, which can be either features or grouping nodes. Note that in this notation, we mix feature names and feature nodes because we can uniquely determine which node we are talking about by observing its position in the tree. The feature cardinality I_i of a subfeature relationship precedes the feature node it is qualifying.

A grouping node of size k (*i.e.*, with k subfeatures) is presented as follows:

$$g\, \langle n\text{–}n' \rangle \left\{ \begin{array}{l} f_1 \\ \vdots \\ f_k \end{array} \right.$$

The interval $\langle n\text{–}n' \rangle$ is the group cardinality.

The concrete syntax can now be easily mapped onto the abstract syntax:



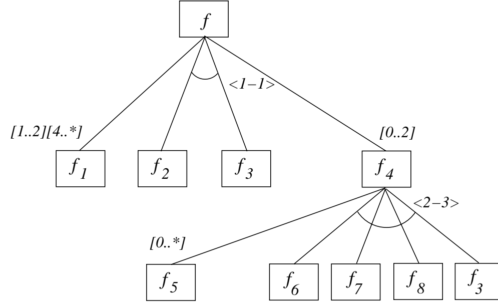
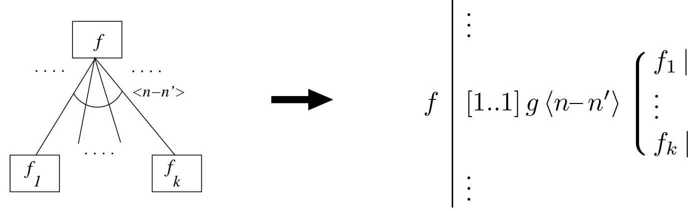


Figure 4. An example feature diagram



It is relatively straightforward to check that this mapping obeys the implicit constraints on the abstract syntax mentioned above.

We finish this section with the small example in Figure 4. Its translation into the abstract syntax is as follows:

$$
f \left|\begin{array}{l}
[1..2][4..*]\ f_1 \mid \\
[1..1]\ g\ \langle 1\text{–}1\rangle \left(\begin{array}{l} f_2 \mid \\ f_3 \mid \end{array}\right. \\
[0..2]\ f_4 \left|\begin{array}{l}
[0..*]\ f_5 \mid \\
[1..1]\ g\ \langle 2\text{–}3\rangle \left(\begin{array}{l} f_6 \mid \\ f_7 \mid \\ f_8 \mid \\ f_3 \mid \end{array}\right.
\end{array}\right.
\end{array}\right.
$$

We will use this feature diagram in the remainder of the paper to explain the formalization of cardinality-based feature models.

4.2 A Translation into Context-Free Grammars

As mentioned earlier, our approach to the formalization of a feature diagram is to translate it into a context-free grammar. Feature diagrams, as they are presented in this paper, can actually be modeled as *regular grammars*[6] and do not require

[6] Regular grammars require that the right-hand side of a production can only have one terminal possibly followed by a non-terminal.

the additional expressiveness of context-free grammars. However, because regular grammars are rather awkward to read, we will present a translation into a more convenient context-free grammar.

An informal description We first explain the translation into context-free grammars by means of the example in Figure 4. The notation for context-free grammars is standard, but for improved readability, we print non-terminals in italic and underline terminals.

The translation consists of two parts: translating feature nodes and translating grouping nodes. First, let us look at how to deal with feature nodes, for example the root feature node f of the example. It has three subfeature links with associated nodes f_1, g, and f_4. For the root feature f, we generate the following production: $F_{f^1} \rightarrow \underline{(}\,\underline{\mathrm{f}}\,\underline{,}\, N_{f^1 f_1^1} \underline{\cup} N_{f^1 g^2} \underline{\cup} N_{f^1 f_4^3} \underline{)}$.

Although feature names are generally not unique, an actual feature node occurring in a diagram is. Hence, we use an encoding that guarantees that the non-terminal corresponding to a particular node in the diagram is unique. It works as follows: Each abstract syntax node is represented by a non-terminal F_{id}. The subscript id is a sequence of node names describing the path of the root node all the way down to the current node. Moreover, each node name itself is superscripted with an index that distinguishes it from all its siblings[7].

In theory, it is sufficient to only use the superscripts without the node names to form a unique identity for the non-terminal. However, it is easier to read the productions when we add the node names to the identity. We use the same method for other non-terminals such as N and G that describe different aspects of the same diagram node.

In the above production, we see the non-terminals $N_{f^1 f_1^1}$, $N_{f^1 g^2}$, and $N_{f^1 f_4^3}$. They encode the possible number of occurrences according to the feature cardinality of the respective feature. For example, let us investigate what happens with the case for $N_{f^1 f_1^1}$. It requires four productions:

$$N_{f^1 f_1^1} \rightarrow \underline{\{}\, N^{1}_{f^1 f_1^1} \underline{\}} \qquad N_{f^1 f_1^1} \rightarrow \underline{\{}\, N^{2}_{f^1 f_1^1} \underline{\}}$$
$$N_{f^1 f_1^1} \rightarrow \underline{\{}\, N^{4}_{f^1 f_1^1} \underline{\}} \qquad N_{f^1 f_1^1} \rightarrow \underline{\{}\, N^{5*}_{f^1 f_1^1} \underline{\}}$$

Each right-hand side is a non-terminal, enclosed in curly brackets to represent the different numbers that the feature cardinality represents. Because the final interval has a Kleene star, it is possible to go to the non-terminal $N^{5*}_{f^1 f_1^1}$. Each of these *numbered* non-terminals is then elaborated in a production that goes to the non-terminal representation of the subnode. For node f_1, this implies the

[7] In many of the examples in this paper, we use subscripts of f to identify different feature names, *e.g.*, f_1. However, these subscripts are part of the actual feature name and should not be confused with the indices that identify siblings.

following productions:

$$N^{1}_{f^1 f^1_1} \to F_{f^1 f^1_1} \qquad N^{2}_{f^1 f^1_1} \to F_{f^1 f^1_1}\,\underline{,}\,F_{f^1 f^1_1}$$

$$N^{4}_{f^1 f^1_1} \to F_{f^1 f^1_1}\,\underline{,}\,F_{f^1 f^1_1}\,\underline{,}\,F_{f^1 f^1_1}\,\underline{,}\,F_{f^1 f^1_1}$$

$$N^{5*}_{f^1 f^1_1} \to F_{f^1 f^1_1}\,\underline{,}\,N^{4}_{f^1 f^1_1} \qquad N^{5*}_{f^1 f^1_1} \to F_{f^1 f^1_1}\,\underline{,}\,N^{5*}_{f^1 f^1_1}$$

The first three productions are obvious. The last two productions take care of the interval [5..∗]. The Kleene star requires a recursive production to represent an unbounded number of f_1, but always ends by taking at least 5 of those features. Note that we insert a comma terminal symbol in between occurrences of $F_{f^1 f^1_1}$. That is to make sure we can interpret the result sentence appropriately. The interpretation of resulting sentences is discussed below.

The cases for $N_{f^1 g^2}$ and $N_{f^1 f^3_4}$ are analogous. If the feature cardinality allows for 0 occurrences, as is the case for $N_{f^1 f^3_4}$, we have to include a production $N^{0}_{f^1 f^3_4} \to \epsilon$. Finally, we have to consider the case where a feature has no subfeature links at all. This is the case for feature f_1, where we have the production $F_{f^1 f^1_1} \to \underline{(}\ \underline{\mathsf{f}_1}\ \underline{,}\ \underline{\emptyset}\ \underline{)}$.

Now, let us look at the translation of a grouping node. From the example diagram in Figure 4, we take the grouping node g of the feature f_4, which has group cardinality $\langle 2\text{–}3\rangle$ and four subfeatures f_6, f_7, f_8, and f_3. This means that we have to generate productions that accept all combinations of minimum 2 and maximum 3 distinct subfeatures. As with feature cardinalities, we directly encode all possible numbers in the interval of the group cardinality with extra non-terminals. However, unlike feature nodes, we do not produce a terminal for the grouping node because it is only an artifact of the abstract syntax:

$$F_{f^1 f^3_4 g^2} \to G^{2}_{f^1 f^3_4 g^2} \qquad F_{f^1 f^3_4 g^2} \to G^{3}_{f^1 f^3_4 g^2}$$

In the next step, we have to produce all the combinations for each number in the interval. A possible choice of subnodes is comma-separated. For example, for $G^{2}_{f^1 f^3_4 g^2}$, we have the following productions:

$$G^{2}_{f^1 f^3_4 g^2} \to F_{f^1 f^3_4 g^2 f^1_6}\,\underline{,}\,F_{f^1 f^3_4 g^2 f^2_7} \qquad G^{2}_{f^1 f^3_4 g^2} \to F_{f^1 f^3_4 g^2 f^1_6}\,\underline{,}\,F_{f^1 f^3_4 g^2 f^3_8}$$

$$G^{2}_{f^1 f^3_4 g^2} \to F_{f^1 f^3_4 g^2 f^1_6}\,\underline{,}\,F_{f^1 f^3_4 g^2 f^4_3} \qquad G^{2}_{f^1 f^3_4 g^2} \to F_{f^1 f^3_4 g^2 f^2_7}\,\underline{,}\,F_{f^1 f^3_4 g^2 f^3_8}$$

$$G^{2}_{f^1 f^3_4 g^2} \to F_{f^1 f^3_4 g^2 f^2_7}\,\underline{,}\,F_{f^1 f^3_4 g^2 f^4_3} \qquad G^{2}_{f^1 f^3_4 g^2} \to F_{f^1 f^3_4 g^2 f^3_8}\,\underline{,}\,F_{f^1 f^3_4 g^2 f^4_3}$$

The productions for $G^{3}_{f^1 f^3_4 g^2}$ are analogous. The context-free grammar is completed by identifying the start symbol, which is the non-terminal representing the root feature node. In our example, this is F_{f^1}. The complete translation of the example in Figure 4 can be found in Appendix A.

A translation algorithm We now provide a complete algorithm, written in pseudo-code. It consists of a large recursive function $\mathcal{T}\left(\ldots\right)$ which takes as one of its arguments an abstract syntax node and produces a set of grammar productions.

The notation used in the pseudo-code is largely self-explaining. We use the symbol α for a sequence of grammar symbols, which may contain both terminals and non-terminals. Juxtaposition expresses construction and destruction of a sequence of objects. For example, $I = [n..n']I'$ implies that $[n..n']$ is the head of the sequence I and I' is its tail. The symbol ξ stands for a sequence of feature names.

We use pattern matching on the type of the parameter of the function $\mathcal{T}\left(\ldots\right)$. According to the meta model in Figure 3, the parameter is a ***node*** object that can be either a **feature node** or a **grouping node**. The function $\mathcal{T}$ dispatches accordingly. The translation starts with $\mathcal{T}\left(\varepsilon, 1, f\ldots\right)$, where f is the root feature of the diagram.

First, we specify the case for a feature node:

$$\mathcal{T}\left(\xi, m, f \;\middle|\; \begin{array}{l} I_1\, f_1 \ldots \\ \vdots \\ I_n\, f_n \ldots \end{array}\right) = \begin{cases} \mathit{if}\; n = 0 \;:\; \{\, F_{\xi'} \rightarrow \underline{(}\,\underline{\mathsf{f}}\,\underline{,}\,\underline{\emptyset}\,\underline{)}\,\} \\ \mathit{if}\; n > 0 \;:\; \{\, F_{\xi'} \rightarrow \underline{(}\,\underline{\mathsf{f}}\,\underline{,}\, N_{\xi' f_1^1} \,\underline{\cup} \ldots \underline{\cup}\, N_{\xi' f_n^n}\,\underline{)}\,\} \cup \\ \qquad \displaystyle\bigcup_{i=1}^{n} \left(\begin{array}{l} \mathit{if}\; I = \varepsilon \;:\; \{\, N_{\xi' f_i^i} \rightarrow \underline{\emptyset}\} \\ \mathit{if}\; I \neq \varepsilon \;:\; \mathcal{N}(\, N_{\xi' f_i^i}, I_i,\, F_{\xi' f_i^i}) \\ \qquad\qquad \cup\, \mathcal{T}\left(\xi', i, f_i \ldots\right) \end{array} \right) \end{cases}$$

$$\mathit{where}\; \xi' = \xi f^m$$

It needs a function which generates productions for each number in the feature cardinality.

$$\mathcal{N}(\, N, I,\, F) = \begin{cases} \mathit{if}\; I = \varepsilon \;:\; \varepsilon \\ \mathit{if} \begin{pmatrix} I = [n_1..n_2]I' \wedge \\ n_1 < n_2 < * \end{pmatrix} : \begin{array}{l} \{\, N \rightarrow \underline{\{}\; N^{n_1}\, \underline{\}}\} \cup \\ \mathcal{N}(\, N, [(n_1 + 1)..n_2]I',\, F) \cup \\ \mathcal{N}'(\, N^{n_1}, n_1,\, F) \end{array} \\ \mathit{if} \begin{pmatrix} I = [n..n]I' \wedge \\ n < * \end{pmatrix} : \begin{array}{l} \{\, N \rightarrow \underline{\{}\; N^{n}\, \underline{\}}\} \cup \mathcal{N}(\, N, I',\, F) \cup \\ \mathcal{N}'(\, N^{n}, n,\, F) \end{array} \\ \mathit{if} \begin{pmatrix} I = [n..*]I' \wedge \\ 0 < n \end{pmatrix} : \begin{array}{l} \{\, N \rightarrow \underline{\{}\; N^{n}\, \underline{\}},\, N \rightarrow \underline{\{}\; N^{(n+1)*}\, \underline{\}}, \\ \;N^{(n+1)*} \rightarrow\; F\,\underline{,}\; N^{n}, \\ \;N^{(n+1)*} \rightarrow\; F\,\underline{,}\; N^{(n+1)*}, \\ \;N^{n} \rightarrow \mathcal{N}''(n,\, F)\} \end{array} \\ \mathit{if}\; I = [0..*]I' \;:\; \begin{array}{l} \{\, N \rightarrow \underline{\{}\; N^{0}\, \underline{\}},\, N \rightarrow \underline{\{}\; N^{1*}\, \underline{\}}, \\ \;N^{1*} \rightarrow\; F,\, N^{1*} \rightarrow\; F\,\underline{,}\; N^{1*},\, N^{0} \rightarrow \epsilon\} \end{array} \end{cases}$$

The following functions generate a production which repeats the non-terminal F, separated by commas, as indicated by n.

$$\mathcal{N}'(\, N, n, F) = \begin{cases} \textit{if}\; n = 0 \;:\; \{\, N \rightarrow \epsilon\} \\ \textit{if}\; n > 0 \;:\; \{\, N \rightarrow \mathcal{N}''(n,\, F)\} \end{cases}$$

$$\mathcal{N}''(n,\, F) = \begin{cases} \textit{if}\; n = 1 \;:\; F \\ \textit{if}\; n > 1 \;:\; F \,\underline{,}\, \mathcal{N}''(n-1,\, F) \end{cases}$$

Now, we consider the case for a grouping node:

$$\mathcal{T}\left(\xi, m, f\,\langle n{-}n'\rangle \begin{pmatrix} f_1 \ldots \\ \vdots \\ f_k \ldots \end{pmatrix}\right) = \begin{cases} \textit{if}\; n' = 0 \;:\; \{\, F_{\xi'} \rightarrow \epsilon\} \\ \textit{if}\; 0 < n = n' \;:\; \{\, F_{\xi'} \rightarrow\; G^n_{\xi'}\} \cup \\ \qquad\qquad \mathcal{G}(\, G^n_{\xi'}, n, \xi', f_k \ldots f_1) \cup \\ \qquad\qquad \bigcup_{i=1}^{k} (\mathcal{T}\;(\xi', i, f_i \ldots)) \\ \textit{if}\; n < n' \;:\; \{\, F_{\xi'} \rightarrow\; G^n_{\xi'}\} \cup \\ \qquad\qquad \mathcal{G}(\, G^n_{\xi'}, n, \xi', f_k \ldots f_1) \cup \\ \qquad\qquad \mathcal{T}\left(\xi, m, f\,\langle n''{-}n'\rangle \begin{pmatrix} f_1 \ldots \\ \vdots \\ f_k \ldots \end{pmatrix}\right) \end{cases}$$

$$\textit{where}\; \xi' = \xi f^m \;\textit{and}\; n'' = n + 1$$

It requires two functions $\mathcal{G}$ and $\mathcal{G}'$, which are very similar. Together, they generate all possible combinations of n subfeatures for the grouping node. We need two functions because we have to deal separately with the first selected feature and any possibly remaining features because of the comma separation. Note that for the case $n = k$, we *have* to take all subfeatures. In case we have $n < k$, we can either select the subfeature or not.

$$\mathcal{G}(\, G, n, \xi, f_k \ldots f_1) = \begin{cases} \textit{if}\; n = 0 \;:\; \{\, G \rightarrow \epsilon\} \\ \textit{if}\; 0 < n < k \;:\; \mathcal{G}'(\, G,\, F_{\xi f^k_k}, n-1, \xi, f_{k-1} \ldots f_1) \cup \\ \qquad\qquad \mathcal{G}(\, G, n, \xi, f_{k-1} \ldots f_1) \\ \textit{if}\; 0 < n = k \;:\; \mathcal{G}'(\, G,\, F_{\xi f^k_k}, n-1, \xi, f_{k-1} \ldots f_1) \end{cases}$$

$$\mathcal{G}'(\, G, \alpha, n, \xi, f_k \ldots f_1) = \begin{cases} \textit{if}\; n = 0 \;:\; \{\, G \rightarrow \alpha\} \\ \textit{if}\; 0 < n < k \;:\; \mathcal{G}'(\, G, \alpha', n-1, \xi, f_{k-1} \ldots f_1) \cup \\ \qquad\qquad \mathcal{G}'(\, G, \alpha, n, \xi, f_{k-1} \ldots f_1) \\ \textit{if}\; 0 < n = k \;:\; \mathcal{G}'(\, G, \alpha', n-1, \xi, f_{k-1} \ldots f_1) \end{cases}$$

$$\textit{where}\; \alpha' = \; F_{\xi f^k_k} \,\underline{,}\, \alpha$$

4.3 Semantics of Feature Models

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

Although this description is sufficient in practice, it is helpful to provide a more 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.*, when they denote the same set of configurations.

Consider the context-free grammar generated from a cardinality-based feature model. We define the semantics of a feature model as an interpretation of the sentences from the language recognized by the grammar.

Each individual sentence of the recognized language corresponds to exactly one configuration. For example, the following string is recognized by the grammar in Appendix A and thus, represents a valid configuration of the feature diagram in Figure 4:

$$(f, \{(f_1, \emptyset), (f_1, \emptyset)\} \cup \{(f_3, \emptyset)\} \cup \{(f_4, \{(f_5, \emptyset)\} \cup \{(f_6, \emptyset), (f_3, \emptyset)\})\})$$

Although the string is nothing more than a sequence of terminals, we can assign it a semantic interpretation as follows: *"each pair is an element from the relation between features (identified by their feature names) and their multiset of subfeatures"*. The semantics of the entire feature model is now defined as the set of all such configurations.

There are a few observations to make at this point:

- The sentences representing configurations still contain the tree structure of the feature diagram via the nested pairs of features and their subfeatures. This is important because a configuration not only represents what features are selected but also how they fit in the hierarchy.
- The second element of a pair is a *multiset*, not a set. This implies that each occurrence of the same feature name is semantically relevant for a configuration. Moreover, we cannot use sequences to represent subfeatures, because the order in which features are chosen is not relevant for the result configuration.
- The semantics of feature diagrams via context-free grammars is layered: first, we interpret a feature model as a language that the corresponding grammar recognizes. Second, we interpret the recognized language in a standard mathematical way to obtain a denotation for the feature model.
- It is relatively easy to add attribute types to the generated grammar. For a feature f with an attribute of type t, we want to be able to construct a configuration triple consisting of the feature f, a value v of the type t, and a multiset of possible subfeatures. In the production of the feature f, we have a non-terminal T^t at the position for the value. The non-terminal *represents* the type t at the grammar level. This is achieved by a set of productions

with left-hand side T^t, encoding all possible values in t. The semantics of the feature diagram will now include all possible configurations with a particular value of the type t. If a feature diagram has already assigned a value v to the attribute, we can simply adapt the productions for T^t in such a way that they only recognize v. In this case, the adapted productions for T^t actually stand for the singleton type $\{v\}$.

5 Feature Model Specialization and Context-Free Grammars

In Section 3.2, we discussed the possible specialization steps to achieve staged configuration of feature models. Specialization of a feature model can also be interpreted on the level of its grammar equivalent.

5.1 Specialization as a Grammar Transformation

In this section, we give an informal discussion on how to achieve each individual specialization step of Section 3.2 by grammar transformations, where we omit the initialization of attributes. We do not provide a formal algorithm because it would unnecessarily clutter the exposition. Instead, we explain the different specialization operations by means of the example feature diagram in Figure 4.

Refining feature cardinalities Consider the feature cardinality [1..2][4..∗] of the feature node for f_1. This cardinality is represented by the production rules with left-hand side $N_{f^1 f_1^1}$.

In Section 3.2, we gave several possibilities to refine a feature cardinality, including the removal of an interval, reducing an interval and splitting it. On the level of the grammar, this amounts to removing the productions that stand for any of the numbers 1, 2, and 4 from the cardinality. In other words, delete the production $N_{f^1 f_1^1} \rightarrow \underline{\{}\, N^i_{f^1 f_1^1} \,\underline{\}}$, where $i \in \{1, 2, 4\}$. The reduction of the interval [4..∗] to [4..4] simply requires the removal of $N_{f^1 f_1^1} \rightarrow \underline{\{}\, N^{5*}_{f^1 f_1^1} \,\underline{\}}$.

Observe that the removal of these kind of productions leaves several more productions in the grammar *dead*, *i.e.*, they are never *used* whenever we try to accept a sentence starting with the start symbol. In the remaining of this section, we will assume that dead productions are automatically *garbage collected*.

The nature of the grammar has excluded one category of refinement so far: the reduction or splitting of [4..∗] into two intervals [4..m] and [m..∗], where $m > 4$. That is because the encoding of the Kleene star at the grammar level uses recursion. In order to achieve a refinement of [4..∗] involving a number $m > 4$, we have to *unroll* the productions that encode the Kleene star.

In the example, suppose we want to split [4..∗] into [4..6][7..∗], we have to replace the productions $N_{f^1 f_1^1} \rightarrow \underline{\{}\, N^{5*}_{f^1 f_1^1} \,\underline{\}}$, $N^{5*}_{f^1 f_1^1} \rightarrow F_{f^1 f_1^1} \,\underline{,}\, N^4_{f^1 f_1^1}$, and

$N^{5*}_{f^1 f^1_1} \rightarrow F_{f^1 f^1_1} \, \underline{,} \, N^{5*}_{f^1 f^1_1}$ by the following rules:

$$N_{f^1 f^1_1} \rightarrow \underline{\{} \, N^5_{f^1 f^1_1} \, \underline{\}} \qquad N_{f^1 f^1_1} \rightarrow \underline{\{} \, N^6_{f^1 f^1_1} \, \underline{\}} \qquad N_{f^1 f^1_1} \rightarrow \underline{\{} \, N^7_{f^1 f^1_1} \, \underline{\}}$$

$$N_{f^1 f^1_1} \rightarrow \underline{\{} \, N^{8*}_{f^1 f^1_1} \, \underline{\}} \qquad N^5_{f^1 f^1_1} \rightarrow F_{f^1 f^1_1} \, \underline{,} \, N^4_{f^1 f^1_1} \qquad N^6_{f^1 f^1_1} \rightarrow F_{f^1 f^1_1} \, \underline{,} \, N^5_{f^1 f^1_1}$$

$$N^7_{f^1 f^1_1} \rightarrow F_{f^1 f^1_1} \, \underline{,} \, N^6_{f^1 f^1_1} \qquad N^{8*}_{f^1 f^1_1} \rightarrow F_{f^1 f^1_1} \, \underline{,} \, N^7_{f^1 f^1_1} \qquad N^{8*}_{f^1 f^1_1} \rightarrow F_{f^1 f^1_1} \, \underline{,} \, N^{8*}_{f^1 f^1_1}$$

The translation process might want to expand the productions with left-hand side $N^5_{f^1 f^1_1}$, $N^6_{f^1 f^1_1}$, and $N^7_{f^1 f^1_1}$ as well. This will guarantee that the new grammar would be identical to the grammar that the translation of the specialized feature diagram generates.

Refining group cardinalities Because the refinement of a group cardinality is a simplification of the refinement of a feature cardinality, we can use the same process as above. However, we are only allowed to remove productions in such a way that we reduce the interval. In the example, we can reduce the group cardinality $\langle 2\text{–}3 \rangle$ of the feature group of f_4 by removing either the production $F_{f^1 f^3_4 g^2} \rightarrow G^2_{f^1 f^3_4 g^2}$ (constructing the group cardinality $\langle 3\text{–}3 \rangle$) or the production $F_{f^1 f^3_4 g^2} \rightarrow G^3_{f^1 f^3_4 g^2}$ (constructing the group cardinality $\langle 2\text{–}2 \rangle$).

Removing a subfeature from a group To remove a subfeature node with identification *id* from a feature group, we have to eliminate the productions with left-hand side G^i_{id} that have the non-terminal F_{id} in their right-hand side.

For example, suppose we want to remove the feature node for f_8 from the feature group of f_4 in Figure 4, we have to eliminate the following productions:

$$G^2_{f^1 f^3_4 g^2} \rightarrow F_{f^1 f^3_4 g^2 f^1_6} \, \underline{,} \, F_{f^1 f^3_4 g^2 f^3_8} \qquad G^2_{f^1 f^3_4 g^2} \rightarrow F_{f^1 f^3_4 g^2 f^2_7} \, \underline{,} \, F_{f^1 f^3_4 g^2 f^3_8}$$

$$G^2_{f^1 f^3_4 g^2} \rightarrow F_{f^1 f^3_4 g^2 f^3_8} \, \underline{,} \, F_{f^1 f^3_4 g^2 f^4_3} \qquad G^3_{f^1 f^3_4 g^2} \rightarrow F_{f^1 f^3_4 g^2 f^1_6} \, \underline{,} \, F_{f^1 f^3_4 g^2 f^2_7} \, \underline{,} \, F_{f^1 f^3_4 g^2 f^3_8}$$

$$G^3_{f^1 f^3_4 g^2} \rightarrow F_{f^1 f^3_4 g^2 f^2_7} \, \underline{,} \, F_{f^1 f^3_4 g^2 f^3_8} \, \underline{,} \, F_{f^1 f^3_4 g^2 f^4_3}$$

$$G^3_{f^1 f^3_4 g^2} \rightarrow F_{f^1 f^3_4 g^2 f^1_6} \, \underline{,} \, F_{f^1 f^3_4 g^2 f^3_8} \, \underline{,} \, F_{f^1 f^3_4 g^2 f^4_3}$$

However, as we noted in Section 3.2, the new upper bound of the group cardinality is $n'' = \mathtt{min}(n', k-1)$ where n' is the old upper bound and k the old size of the group. If $n' = n''$, nothing else has to be done. However, if $n'' < n'$, then we also have to remove all the productions of the form $F_{id} \rightarrow G^i_{id}$, where $i \in \{n''+1, \ldots, n'\}$. In the example, this means that we have to eliminate $F_{f^1 f^3_4 g^2} \rightarrow G^3_{f^1 f^3_4 g^2}$ once we remove another subfeature from the group.

Cloning a solitary subfeature Cloning a solitary subfeature on the grammar level involves several different operations. Suppose we want to clone a subfeature with original feature cardinality I and give it a fixed feature cardinality $[m..m]$.

First, we have to make sure that we split I sufficiently so that at least one interval $[n..n']$ in I has $m < n$. For example, if we have $m = 3$ for feature f_1 in the example, nothing has to happen. However, if we have $m = 3$ for feature f_5, we have to unroll the Kleene star sufficiently to turn $[0..*]$ into $[0..3][4..*]$. This step guarantees that we have a production in the grammar which has at least m non-terminals F_{id}, where id is the identity of the to-be-cloned subfeature.

For the second step, we have to perform a process called *production splitting*. To explain this, let us clone f_1 from the example with a fixed feature cardinality [3..3]. The production $N^4_{f^1 f^1_1} \rightarrow F_{f^1 f^1_1} \underline{,} F_{f^1 f^1_1} \underline{,} F_{f^1 f^1_1} \underline{,} F_{f^1 f^1_1}$ represents the number 4 in the original feature cardinality, so we turn it into two productions $N^1_{f^1 f^1_1} \rightarrow F_{f^1 f^1_1}$ and $N^3_{f^1 f^4_1} \rightarrow F_{f^1 f^4_1} \underline{,} F_{f^1 f^4_1} \underline{,} F_{f^1 f^4_1}$. The new identity $f^1 f^4_1$ stands for the unique feature node that now occurs as a new sibling of the old node for f_1. The superscript 4 indicates that it is the 4th sibling.

Next, we simply duplicate all the production rules that are reachable from the non-terminal $F_{f^1 f^1_1}$, but rename each non-terminal in a unique way. The occurrences of $F_{f^1 f^1_1}$, however, have to be substituted for $F_{f^1 f^4_1}$. This process obviously does not change the semantics of the grammar as it is the equivalent of duplicating the descendent nodes of the node for f_1 in the actual feature diagram.

Because we have split the production for $N^4_{f^1 f^1_1}$ into two productions with left-hand side non-terminals $N^1_{f^1 f^1_1}$ and $N^3_{f^1 f^4_1}$, we have to reflect this change also upwards in the grammar. To this end, we split $N_{f^1 f^1_1} \rightarrow \underline{\{} N^4_{f^1 f^1_1} \underline{\}}$ into $N_{f^1 f^1_1} \rightarrow \underline{\{} N^1_{f^1 f^1_1} \underline{\}}$ and $N_{f^1 f^4_1} \rightarrow \underline{\{} N^3_{f^1 f^4_1} \underline{\}}$. The production for the parent of the to-be-cloned feature has to be updated to reflect that $N_{f^1 f^4_1}$ represents the feature cardinality of the new subfeature node, so we change it into $F_{f^1} \rightarrow \underline{(} \underline{\mathsf{f}} \underline{,} N_{f^1 f^1_1} \underline{\cup} N_{f^1 f^4_1} \underline{\cup} N_{f^1 g^2} \underline{\cup} N_{f^1 f^3_4} \underline{)}$.

We are not done because we still have to reflect the changes incurred by $L(3, [1..2][4..*])$ on the feature cardinality of the original node. This is done by following an analogous process as in the definition $L(m, I)$, but this time on the superscripts of the non-terminal $N_{f^1 f^1_1}$. The rules where the superscript is smaller than m are eliminated. For rules where the superscript is greater or equal to m, we simply reduce the superscripts with m. Here too, the change in the grammar will at most imply that less sentences are recognized because we only eliminate grammar productions (the renaming of non-terminals does not affect the sentences being recognized).

In the example, this means that we get rid of $N_{f^1 f^1_1} \rightarrow \underline{\{} N^1_{f^1 f^1_1} \underline{\}}$ and $N_{f^1 f^1_1} \rightarrow \underline{\{} N^2_{f^1 f^1_1} \underline{\}}$. The rules $N_{f^1 f^1_1} \rightarrow \underline{\{} N^{5*}_{f^1 f^1_1} \underline{\}}$, $N^{5*}_{f^1 f^1_1} \rightarrow F_{f^1 f^1_1} \underline{,} N^4_{f^1 f^1_1}$, and $N^{5*}_{f^1 f^1_1} \rightarrow F_{f^1 f^1_1} \underline{,} N^{5*}_{f^1 f^1_1}$ are renamed into $N_{f^1 f^1_1} \rightarrow \underline{\{} N^{2*}_{f^1 f^1_1} \underline{\}}$, $N^{2*}_{f^1 f^1_1} \rightarrow F_{f^1 f^1_1} \underline{,} N^1_{f^1 f^1_1}$, and $N^{2*}_{f^1 f^1_1} \rightarrow F_{f^1 f^1_1} \underline{,} N^{2*}_{f^1 f^1_1}$ respectively.

5.2 Specialization does not add Configurations

The informal transformation rules described above are designed in such a way that the resulting *specialized grammar* recognizes a language which, when interpreted, is a subset of the interpretation of the language recognized by the original non-specialized grammar. This is not very difficult to understand, because, with the exception of production splitting and production unfolding, all the previously described transformations only eliminate productions and hence, only remove sentences from the recognized language.

Moreover, it is also relatively straightforward to see that the unrolling of the Kleene star preserves any recognized language, *i.e.*, it does not remove or add sentences. Production splitting also preserves any recognized language by the grammar, but it is slightly more involved to prove this fact, even though it is fairly intuitive to understand.

The main observation from these properties is that feature diagram specialization does not increase the possible set of configurations, which is exactly what we expect from such a process.

6 Related Work

Since its initial introduction in [19], several extensions and variants of the original FODA notation have been proposed [8–10, 16, 18, 20, 22, 23]. We introduced the cardinality-based notation for feature modeling described in Section 2 in a previous paper [11]. The notation integrates four earlier extensions: feature cardinalities [9], group cardinalities [22], feature diagram references [3], and attributes [3, 9]. This mix of concepts has been determined by our experience in applying feature modeling in practice (*e.g.*, see [9]) and has been motivated in [11]. Other extensions such as *consists-of* or *is-generalization-of* relationships between features [16, 20, 23], as well as additional information on features such as descriptions, categories, binding time, priorities, stakeholders, exemplar systems, etc., are left to be handled as user-defined, structured annotations. Such annotations are also supported through an extensible meta model [3, 9].

In this paper, we propose a formalization of feature diagrams, which is not the first attempt to do so. Van Deursen and Klint [14] use algebraic specifications to formally define FODA-style feature diagrams with exclusive-or groups. However, that work does not consider feature or group cardinalities. Moreover, we also give a formal characterization of the specialization of cardinality-based feature models.

The correspondence between feature diagrams and grammars has been suggested in a position paper by de Jonge and Visser [12]. While they only illustrate the idea using a simple FODA-like feature diagram and a grammar of a possible configuration language corresponding to the sample diagram, we give a full and precise semantics for cardinality-based feature diagrams and their specialization.

The correspondence between feature diagrams and grammars can actually be traced back at least to early 70s, when Hall presented a formal connection between *and-or graphs* and context-free grammars in [17]. And-or graphs are used

in artificial intelligence as a representation for problem decomposition. Although not quite the same as tree-based feature diagram, and-or graphs do bear some interesting similarities: And-nodes correspond to a feature with only mandatory subfeatures and or-nodes correspond to a feature with an inclusive-or group. Hall explicitly shows that and-or graphs are equivalent to context-free grammars. In this work, we only consider a uni-directional translation of an extended feature diagram notation into context-free grammars. The grammar representation of the feature diagram is then used to define a semantic interpretation of feature models.

An alternative approach to formalize feature diagrams is to map them to some logic, such as first-order predicate logic or Horn clauses. The advantage of the latter is the possibility of obtaining an executable semantics. In fact, some feature-based configuration tools (e.g., Pure::Consul [5, 21]) translate feature models into Prolog in order to provide support for constraint checking. Although not directly based on feature modeling, a related approach was proposed by Zeller [25], in which *feature logic* is used for the constraint-based configuration of software variants. Feature logic is specifically designed to express logical assertions over nested attribute-value pairs.

While the existing logic-based approaches handle global constraints (e.g., constraints between features in different parts of a feature diagram), they do not consider cloning, or more generally, specialization.

7 Conclusion

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. We provided a formalization of cardinality-based feature modeling because it casts light on the expressiveness and nature of feature models. We also believe that it gives a solid foundation for further tool development.

The study of specialization at the grammar level has helped to better understand possible specialization steps at the diagram level. In fact, this analysis has revealed other specialization steps than those described in Section 3.2. Some of them can be quite involved and some cannot even be be translated back into the concrete syntax of the current feature diagram notation. The proposed steps of this paper (Section 3.2) 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 more 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 [9] and CaptainFeature [3, 4]. 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.

In future work, we plan to extend our model to handle global constraints. Adequate tool support for the newly introduced cardinality-based feature notation as well as its specialization is under way.

A Grammar Translation Example

This is the complete context-free grammar generated from the feature diagram in Figure 4. It is based on the algorithm presented in Section 4.2. The order in which the productions appear is arbitrary.

$$\{\ F_{f^1} \rightarrow \underline{(}\,\underline{\mathsf{f}}\,\underline{,}\, N_{f^1 f^1_1} \underline{\cup} N_{f^1 g^2} \underline{\cup} N_{f^1 f^3_4}\, \underline{)}, \quad N_{f^1 f^1_1} \rightarrow \underline{\{}\, N^1_{f^1 f^1_1}\, \underline{\}},$$

$$N_{f^1 f^1_1} \rightarrow \underline{\{}\, N^2_{f^1 f^1_1}\, \underline{\}}, \quad N_{f^1 f^1_1} \rightarrow \underline{\{}\, N^4_{f^1 f^1_1}\, \underline{\}}, \quad N_{f^1 f^1_1} \rightarrow \underline{\{}\, N^{5*}_{f^1 f^1_1}\, \underline{\}},$$

$$N_{f^1 g^2} \rightarrow \underline{\{}\, N^1_{f^1 g^2}\, \underline{\}}, \quad N_{f^1 f^3_4} \rightarrow \underline{\{}\, N^0_{f^1 f^3_4}\, \underline{\}}, \quad N_{f^1 f^3_4} \rightarrow \underline{\{}\, N^1_{f^1 f^3_4}\, \underline{\}},$$

$$N_{f^1 f^3_4} \rightarrow \underline{\{}\, N^2_{f^1 f^3_4}\, \underline{\}}, \quad N^1_{f^1 f^1_1} \rightarrow F_{f^1 f^1_1}, \quad N^2_{f^1 f^1_1} \rightarrow F_{f^1 f^1_1} \underline{,} F_{f^1 f^1_1},$$

$$N^4_{f^1 f^1_1} \rightarrow F_{f^1 f^1_1} \underline{,} F_{f^1 f^1_1} \underline{,} F_{f^1 f^1_1} \underline{,} F_{f^1 f^1_1}, \quad N^{5*}_{f^1 f^1_1} \rightarrow F_{f^1 f^1_1} \underline{,} N^4_{f^1 f^1_1},$$

$$N^{5*}_{f^1 f^1_1} \rightarrow F_{f^1 f^1_1} \underline{,} N^{5*}_{f^1 f^1_1}, \quad N_{f^1 g^2} \rightarrow F_{f^1 g^2}, \quad N^0_{f^1 f^3_4} \rightarrow \epsilon, \quad N^1_{f^1 f^3_4} \rightarrow F_{f^1 f^3_4},$$

$$N^2_{f^1 f^3_4} \rightarrow F_{f^1 f^3_4} \underline{,} F_{f^1 f^3_4}, \quad F_{f^1 f^1_1} \rightarrow \underline{(}\,\underline{\mathsf{f_1}}\,\underline{,}\,\underline{\emptyset}\,\underline{)}, \quad F_{f^1 g^2} \rightarrow G^1_{f^1 g^2},$$

$$G^1_{f^1 g^2} \rightarrow F_{f^1 g^2 f^1_2}, \quad G^1_{f^1 g^2} \rightarrow F_{f^1 g^2 f^2_3}, \quad F_{f^1 g^2 f^1_2} \rightarrow \underline{(}\,\underline{\mathsf{f_2}}\,\underline{,}\,\underline{\emptyset}\,\underline{)},$$

$$F_{f^1 g^2 f^2_3} \rightarrow \underline{(}\,\underline{\mathsf{f_3}}\,\underline{,}\,\underline{\emptyset}\,\underline{)}, \quad F_{f^1 f^3_4} \rightarrow \underline{(}\,\underline{\mathsf{f_4}}\,\underline{,}\, N_{f^1 f^3_4 f^1_5} \underline{\cup} N_{f^1 f^3_4 g^2}\, \underline{)},$$

$$N_{f^1 f^3_4 f^1_5} \rightarrow \underline{\{}\, N^0_{f^1 f^3_4 f^1_5}\, \underline{\}}, \quad N_{f^1 f^3_4 f^1_5} \rightarrow \underline{\{}\, N^{1*}_{f^1 f^3_4 f^1_5}\, \underline{\}}, \quad N_{f^1 f^3_4 g^2} \rightarrow \underline{\{}\, N^1_{f^1 f^3_4 g^2}\, \underline{\}},$$

$$N^0_{f^1 f^3_4 f^1_5} \rightarrow \epsilon, \quad N^{1*}_{f^1 f^3_4 f^1_5} \rightarrow F_{f^1 f^3_4 f^1_5}, \quad N^{1*}_{f^1 f^3_4 f^1_5} \rightarrow F_{f^1 f^3_4 f^1_5} \underline{,} N^{1*}_{f^1 f^3_4 f^1_5},$$

$$N^1_{f^1 f^3_4 g^2} \rightarrow F_{f^1 f^3_4 g^2}, \quad F_{f^1 f^3_4 f^1_5} \rightarrow \underline{(}\,\underline{\mathsf{f_5}}\,\underline{,}\,\underline{\emptyset}\,\underline{)}, \quad F_{f^1 f^3_4 g^2} \rightarrow G^2_{f^1 f^3_4 g^2},$$

$$F_{f^1 f^3_4 g^2} \rightarrow G^3_{f^1 f^3_4 g^2}, \quad G^2_{f^1 f^3_4 g^2} \rightarrow F_{f^1 f^3_4 g^2 f^1_6} \underline{,} F_{f^1 f^3_4 g^2 f^2_7},$$

$$G^2_{f^1 f^3_4 g^2} \rightarrow F_{f^1 f^3_4 g^2 f^1_6} \underline{,} F_{f^1 f^3_4 g^2 f^3_8}, \quad G^2_{f^1 f^3_4 g^2} \rightarrow F_{f^1 f^3_4 g^2 f^1_6} \underline{,} F_{f^1 f^3_4 g^2 f^4_3},$$

$$G^2_{f^1 f^3_4 g^2} \rightarrow F_{f^1 f^3_4 g^2 f^2_7} \underline{,} F_{f^1 f^3_4 g^2 f^3_8}, \quad G^2_{f^1 f^3_4 g^2} \rightarrow F_{f^1 f^3_4 g^2 f^2_7} \underline{,} F_{f^1 f^3_4 g^2 f^4_3},$$

$$G^2_{f^1 f^3_4 g^2} \rightarrow F_{f^1 f^3_4 g^2 f^3_8} \underline{,} F_{f^1 f^3_4 g^2 f^4_3},$$

$$G^3_{f^1 f^3_4 g^2} \rightarrow F_{f^1 f^3_4 g^2 f^1_6} \underline{,} F_{f^1 f^3_4 g^2 f^2_7} \underline{,} F_{f^1 f^3_4 g^2 f^3_8},$$

$$G^3_{f^1 f^3_4 g^2} \rightarrow F_{f^1 f^3_4 g^2 f^1_6} \underline{,} F_{f^1 f^3_4 g^2 f^2_7} \underline{,} F_{f^1 f^3_4 g^2 f^4_3},$$

$$G^3_{f^1 f^3_4 g^2} \rightarrow F_{f^1 f^3_4 g^2 f^1_6} \underline{,} F_{f^1 f^3_4 g^2 f^3_8} \underline{,} F_{f^1 f^3_4 g^2 f^4_3},$$

$$G^3_{f^1 f^3_4 g^2} \rightarrow F_{f^1 f^3_4 g^2 f^2_7} \underline{,} F_{f^1 f^3_4 g^2 f^3_8} \underline{,} F_{f^1 f^3_4 g^2 f^4_3},$$

$$F_{f^1 f_4^3 g^2 f_6^1} \rightarrow \underline{(}\,\underline{\mathsf{f}_6}\,\underline{,}\,\underline{\emptyset}\,\underline{)}, \quad F_{f^1 f_4^3 g^2 f_7^2} \rightarrow \underline{(}\,\underline{\mathsf{f}_7}\,\underline{,}\,\underline{\emptyset}\,\underline{)},$$
$$F_{f^1 f_4^3 g^2 f_8^3} \rightarrow \underline{(}\,\underline{\mathsf{f}_8}\,\underline{,}\,\underline{\emptyset}\,\underline{)}, \quad F_{f^1 f_4^3 g^2 f_3^4} \rightarrow \underline{(}\,\underline{\mathsf{f}_3}\,\underline{,}\,\underline{\emptyset}\,\underline{)}\ \}$$

References

1. Michel Barbeau and Francis Bordeleau. A protocol stack development tool using generative programming. In D. Batory, C. Consel, and W. Taha, editors, *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02), Pittsburgh, October 6-8, 2002*, LNCS 2487, pages 93–109. Springer-Verlag, 2002.
2. Don Batory, Clay Johnson, Bob MacDonald, and Dale von Heeder. Achieving extensibility through product-lines and domain-specific languages: A case study". *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):191–214, April 2002.
3. Thomas Bednasch. Konzept und implementierung eines konfigurierbaren metamodells für die merkmalmodellierung. Diplomarbeit, Fachbereich Informatik, Fachhochschule Kaiserslautern,Standort Zweibrücken, Germany, October 2002. Available from `http://www.informatik.fh-kl.de/~eisenecker/studentwork/dt_bednasch.pdf` (in German).
4. Thomas Bednasch, Christian Endler, and Markus Lang. CaptainFeature, 2002-2004. Tool available on SourceForge at `https://sourceforge.net/projects/captainfeature/`.
5. Danilo Beuche. *Composition and Construction of Embedded Software Families*. PhD thesis, Otto-von-Guericke-Universität Magdeburg, Germany, Dec 2003. Available from `http://www-ivs.cs.uni-magdeburg.de/~danilo`.
6. Jan Bosch. *Design and Use of Software Architecture: Adopting and evolving a product-line approach*. Addison-Wesley, 2000.
7. Craig Cleaveland. *Program Generators with XML and Java*. Prentice-Hall, 2001.
8. Krzysztof Czarnecki. *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, October 1998. Available from `http://www.prakinf.tu-ilmenau.de/~czarn/diss`.
9. Krzysztof Czarnecki, Thomas Bednasch, Peter Unger, and Ulrich W. Eisenecker. Generative programming for embedded software: An industrial experience report. In D. Batory, C. Consel, and W. Taha, editors, *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02), Pittsburgh, October 6-8, 2002*, LNCS 2487, pages 156–172. Springer-Verlag, 2002.
10. Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.
11. Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged configuration using feature models. In Robert Nord, editor, *Third Software Product-Line Conference*. Springer-Verlag, September 2004. `http://www.ece.uwaterloo.ca/~kczarnec/splc04.pdf`.
12. Merijn de Jonge and Joost Visser. Grammars as feature diagrams. In *ICSR7 Workshop on Generative Programming (GP2002), Austin (Texas), April 15, 2002*, pages 23–24. online proceedings, 2002. Available from `http://www.cwi.nl/events/2002/GP2002/GP2002.html`.

13. Jean-Marc DeBaud and Klaus Schmid. A systematic approach to derive the scope of software product lines. In *Proceedings of the 21st International Conference on Software Engineering (ICSE)*, pages 34–43. IEEE Computer Society Press, 1999.
14. Arie van Deursen and Paul Klint. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10(1):1–17, 2002.
15. Jack Greenfield and Keith Short. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools.* Wiley, 2004. To be published.
16. Martin Griss, John Favaro, and Massimo d' Alessandro. Integrating feature modeling with the RSEB. In *Proceedings of the Fifth International Conference on Software Reuse (ICSR)*, pages 76–85. IEEE Computer Society Press, 1998.
17. Patrick A. V. Hall. Equivalence between AND / OR graphs and context-free grammars. *Communications of the ACM*, 16(7):444–445, July 1973.
18. Andreas Hein, Michael Schlick, and Renato Vinga-Martins. Applying feature models in industrial settings. In P. Donohoe, editor, *Proceedings of the Software Product Line Conference (SPLC1)*, pages 47–70. Kluwer Academic Publishers, 2000.
19. Kyo Kang, Sholom Cohen, James Hess, William Nowak, and Spencer Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90TR -21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, November 1990.
20. Kwanwoo Lee, Kyo C. Kang, and JaeJoon Lee. Concepts and guidelines of feature modeling for product line software engineering. In Cristina Gacek, editor, *Software Reuse: Methods, Techniques, and Tools: Proceedings of the Seventh Reuse Conference (ICSR7), Austin, USA, Apr.15-19, 2002*, LNCS 2319, pages 62–77. Springer-Verlag, 2002.
21. pure-systems GmbH. Variant management with pure::consul. Technical White Paper. Available from `http://web.pure-systems.com`, 2003.
22. Matthias Riebisch, Kai Böllert, Detlef Streitferdt, and Ilka Philippow. Extending feature diagrams with UML multiplicities. In *6th Conference on Integrated Design & Process Technology (IDPT 2002), Pasadena, California, USA*, 2002.
23. Mikael Svahnberg, Jilles van Gurp, and Jan Bosch. On the notion of variability in software product lines. In *Proceedings of The Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 45–55, August 2001.
24. David M. Weiss and Chi Tau Robert Lai. *Software Product-Line Engineering: A Family-Based Software Development Process.* Addison-Wesley, 1999.
25. Andreas Zeller and Georg Snelting. Unified versioning through feature logic. *ACM Transactions on Software Engineering and Methodology*, 6(4):398–441, October 1997.