Synchronizing Cardinality-Based Feature Models and Their Specializations

Chang Hwan, Peter Kim, and Krzysztof Czarnecki

University of Waterloo, Canada
{chpkim, kczarnec}@swen.uwaterloo.ca

Abstract. A software product line comprises a set of products implementing different configurations of features. The set of valid feature configurations within a product line can be described by a feature model. In some practical situations, a feature configuration needs to be derived in stages by creating a series of successive specializations of the initial feature model. In this paper, we consider the scenario where changes to the feature model due to, for example, the evolution of the product line, need to be propagated to its existing specializations and configurations. After discussing general dimensions of model synchronization, a solution to synchronizing cardinality-based feature models and their specializations and configurations is presented.

1 Introduction

Feature modeling is a systematic way of describing variabilities and commonalities of systems in a software product line [1,2]. A feature model describes a set of possible configurations or combinations of features. In this paper, we focus on a particular style of feature models referred to as *cardinality-based feature models* [3].

A configuration can be arrived at in stages, where at each stage some choices are made [3]. The outcome of each stage is a feature model which is a specialization of the input feature model for that stage. A specialization of a feature model describes a subset of the configurations represented by that model. The need for staged configuration arises in several practical situations, such as in

– *software supply chains,* e.g., a platform vendor may need to make some configuration choices to a platform before releasing it to a specific customer, and the customer may need to provide further settings for individual applications;
– *optimization,* e.g., certain configuration choices could be made at compile time, while remaining ones are decided at runtime; the application code could be optimized based on the compile-time choices;
– *multi-level policies,* e.g., security policies may be specialized at different levels of an organization.

In any realistic setting, the variabilities and commonalities in a product-line will evolve. Inevitably, a feature model will also have to change, and the existing specializations of multiple stages will have to be synchronized in order to reflect the change in the feature model. The interesting challenge is to perform synchronization with the intent of preserving the choices made in the stages of specializations. For simplicity, we refer to the synchronization of cardinality-based feature models and their specializations and configurations as *feature-model synchronization*.

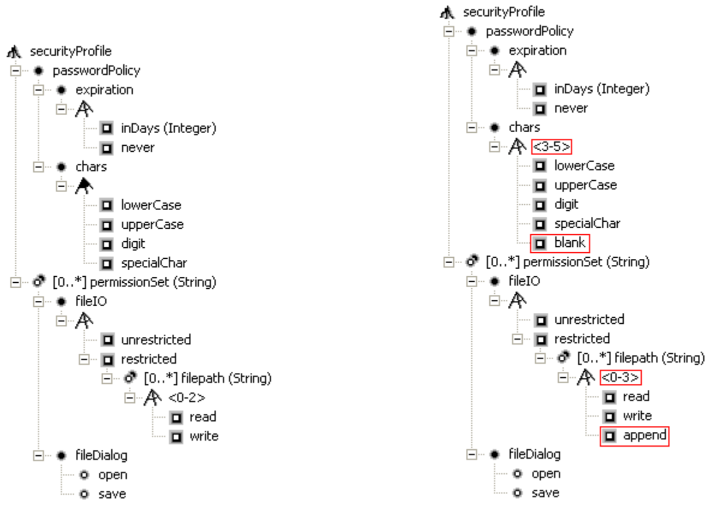
In this paper, we first characterize feature-model synchronization according to dimensions that are applicable to other model synchronization problems. Then we give a solution to the feature-model synchronization problem. The solution and the significant issues surrounding the problem are explained in natural language. The main characteristic of the solution is that it treats feature models, specializations, and configurations in a uniform way. Additionally, we describe how the solution can be specified using the *Relations* language from the latest submission for the Object Management Group's MOF 2.0 Query/View/ Transformation standard [4]. To our knowledge, feature model synchronization in a multi-staged configuration setting is a problem unexplored until now. We believe that the results we present are novel contributions with relevance to both the software product-line community and the model-based development community.

In Section 2, the synchronization problem is motivated through an example. In Section 3, general dimensions of model synchronization are discussed. In Section 4, feature model synchronization is characterized and a technique to achieve it is described. Section 5 discusses related work. Section 6 concludes the paper.

2 Background and Motivating Example

A *feature model* is a hierarchy of features plus constraints describing valid *configurations* of features. Features are used to model functional and non-functional characteristics of systems, but for the purpose of our discussion, they are just symbols with no further semantics. A feature model always has a *root feature*. The remaining features are either *grouped* or *solitary*, i.e., they are either part of a *feature group* or not. Each solitary feature is annotated with a *feature cardinality*, which is an interval constraining how many times the feature has to be included in a configuration if its parent is also included.[1] Each feature group is annotated with a *group cardinality*, which is an interval constraining how many features from that group have to be included in a configuration if the parent feature of the group is also included. Additional constraints on possible configurations may also need to be expressed, such as *requires* and *excludes* constraints. Constraints may be specified using XPath, as explained elsewhere [5]. A feature may be associated with an attribute type, in which case an attribute value can be specified during configuration.

[1] As explained later in Section 4.2, we also associate feature cardinalities with grouped features. However, the only possible values are [0..0], [0..1], and [1..1]. Normally, a grouped feature has [0..1] as its default cardinality. The cardinalities [1..1] and [0..1] are used for features that were selected or eliminated, respectively, from a group during specialization.



(a) Initial feature model (b) Modified feature model

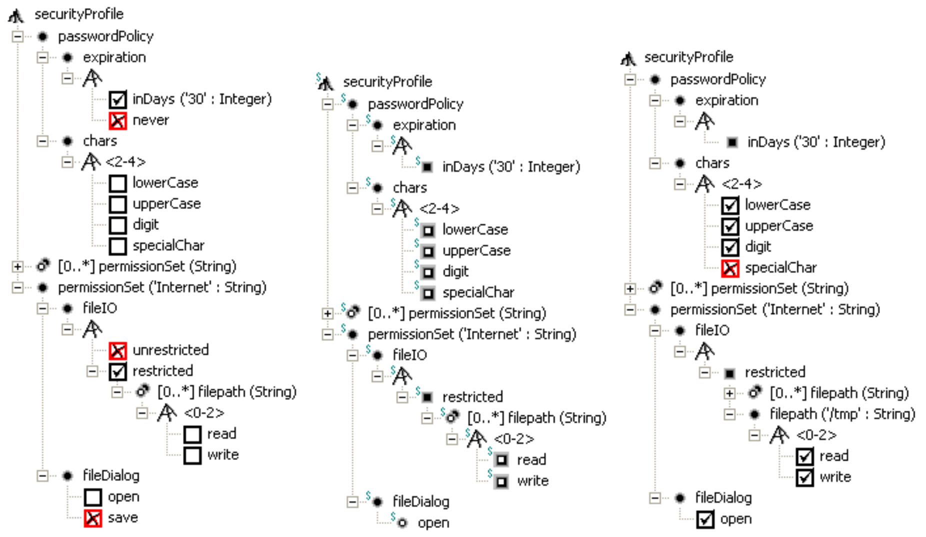
Fig. 1. Security profile feature model before and after changes

Table 1. Symbols used in cardinality-based feature modeling

Symbol	Explanation
⋔	Root feature
●	Solitary feature with cardinality [1..1], i.e., *mandatory* feature
○	Solitary feature with cardinality [0..1], i.e., *optional* feature
○ [0..m]	Solitary feature with cardinality [0..m], $m > 1$, i.e., *optional clonable* feature
● [n..m]	Solitary feature with cardinality [n..m], $n > 0 \wedge m > 1$, i.e., *mandatory clonable* feature
▣	Grouped feature
f(‘$value'$: T)	Feature f with attribute of type T and value of ‘$value'$
⋀	Feature group with cardinality $\langle 1\text{–}1 \rangle$, i.e. *xor*-group
▲	Feature group with cardinality $\langle 1\text{–}k \rangle$, where k is the group size, i.e. *or*-group
⋀ $\langle i\text{–}k \rangle$	Feature group with cardinality $\langle i\text{–}k \rangle$

Figure 1(a) shows a sample feature model describing the configuration choices available in a security profile of an operating system. The notation is explained in Table 1. The profile contains a policy for password expiration, which can be never or after a specific number of days, and for the kind of characters required in a password. The permission set specifies allowable operations on various resources such as files, file dialogs, and environment variables. A configuration can have multiple permission sets, e.g., a permission set defined for code downloaded from the Internet and another for code coming from the intranet of an organization.

A *specialization* of a feature model is another feature model which describes a subset of the configurations represented by the original feature model.



(a) Specialization of 1(a) in check-box view (b) Specialization of 1(a) in feature model view (c) Configuration of 2(b) in check-box view

Fig. 2. Specializations of the initial feature model from Figure 1(a)

A specialization can be created by first copying the original model and then applying *specialization steps* to the copy [6]: select or eliminate an optional solitary feature, select or eliminate a grouped feature from a group, refine feature cardinality, refine group cardinality, clone a clonable solitary feature, and assign value to a feature attribute.[2] Figure 2(a) shows a specialization of the feature model from Figure 1(a) in the *check-box view*, which supports the application of the specializations steps. Check boxes are shown for optional solitary features and grouped features. Placing a check on a check box corresponds to selecting a feature. A cross corresponds to eliminating the feature. An empty check box means no change. Cardinalities can be refined by editing them. The clone operation can be invoked on clonable features through the context menu. Values can be assigned to feature attributes. The resulting specialization rendered in the *feature model view* is shown in Figure 2(b). Note that we use a filled square to indicate a selected grouped feature, e.g., *inDays* and *restricted*. Compared to the original model, in the specializations shown in the check-box view in 2(a) and in the feature model view in 2(b), `never` is eliminated, `inDays` is assigned an attribute value of 30, the cardinality of the group under `chars` is refined to $\langle 2–4 \rangle$, `permissionSet` has a clone with the attribute value `Internet` assigned to it. Furthermore, `restricted` is selected in the clone, and `unrestricted` and `save`

[2] The original description of specialization steps [6] includes unfolding feature model references. We do not consider the latter in this paper for simplicity. Furthermore, selecting or eliminating an optional solitary feature is equivalent to refining its cardinality to [1..1] or [0..0], respectively.

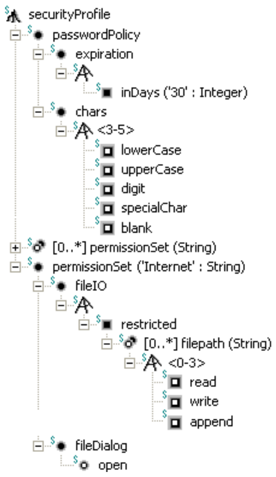
are eliminated. The *prototype* feature for the clone, i.e., `permissionSet` with cardinality [0..∗], is shown in collapsed form for brevity.[3] An alternative rendering could suppress the display of the groups under `expiration` and `fileIO` and show *inDays* and *restricted* as mandatory subfeatures of `expiration` and `fileIO`, respectively.

The specialization in Figure 2(b) could have been created by an organization to take their special security requirements into account. Further specializations could be created based on the presented one, for example, in order to satisfy the security constraints of individual departments. Finally, the department-level specializations could be used as a basis for creating configurations for individual computers.

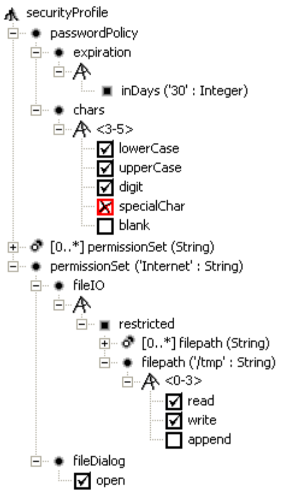
The relationship between a feature model and its *configuration* corresponds to the one between a type and its instance. A feature model in which there is no variability represents exactly one configuration. Such a feature model, which is comparable to the notion of a singleton type, can also be used in place of the configuration it describes. This observation allows us to treat configurations as feature models, too. As a consequence, a tool can use the same interface for creating specializations and configurations, e.g., the check-box view, and configurations and specializations can be also rendered in the feature-model view. Figure 2(c) shows a sample configuration, which was created based on the specialized feature model from Figure 2(b). Note some of the choices made, including the elimination of `specialChar` and the selection of `open`. Figure 2(c) shows a configuration even though it still contains `filepath` with cardinality [0..∗]. This is because we assume that features with cardinality $[0..n]$, where $n > 1$, are by default not part of the configuration. Similarly, undecided optional features, i.e., those with empty check boxes, could also be considered as by default not part of the configuration. We keep features such as `filepath` with cardinality [0..∗] in the check-box view as prototypes, should the user decide to create more clones. If desired, the prototype `filepath` can be eliminated explicitly by refining its cardinality to [0..0].

Any changes made to a feature model need to be propagated to all its specializations and configurations, which is an example of model synchronization. For example, the available security settings described by the feature model in Figure 1(a) could change in the next release of the operating system to those described by the feature model in Figure 1(b) (the changes are highlighted). Comparing the models reveals that `blank` character type has been added, the cardinality of the group containing `blank` has been changed to $\langle 3\text{–}5 \rangle$, `append` has been added to the group under `filepath`, and that group's cardinality has changed to $\langle 0\text{–}3 \rangle$. All these changes need to be propagated to the specialization and configuration in Figure 2, whose updated versions are shown in Figure 3. Please note that the configuration from 2(c) became a specialization in Figure 3(b) due to the newly added features `blank` and `append`, which are undecided. Furthermore, the organization could decide to change the specialization in Figure 2(a) by applying further specialization steps or undoing some of the previously applied ones. These changes would also need to be propagated to the configuration in Figure 2(c).

[3] The feature from which a clone is created is referred to as the *prototype* of the clone.



(a) Specialization shown in 2(a) and 2(b) updated



(b) The resulting specialization after updating the configuration in 2(c)

Fig. 3. Specializations updated according to the changed feature model from Figure 1(b)

3 General Dimensions of Model Synchronization

In this section, we analyze dimensions of model synchronization in general and classify feature model synchronization in terms of these dimensions. This exercise will help us to better understand the characteristics of feature model synchronization and devise a solution in Section 4.

Model synchronization is concerned with maintaining consistency among two or more models in the presence of changes to one or more of them. Synchronization includes both the detection of inconsistencies among the models and modification of one or more models in order to re-establish consistency. The modification is referred to as *reconciliation*.[4]

Structural characteristics. Synchronization problems can be characterized through their structural properties:

– **Number of models and synchronization direction.** Synchronization can be thought of as a procedure with two or more input parameters, some or all of which are also output parameters. Even more generally, different subsets of the input parameters could be additionally marked as output parameters for different individual invocations of synchronization. A common case is *unidirectional* synchronization between two models, where a

[4] There is no fundamental difference between synchronizing a set of models and synchronizing different parts of one model because the set of models could be viewed as parts of one large model. However, in order to keep our discussion more clear, we take the former rather than the latter view.

change to a source model needs to be propagated to a target model. Unidirectional synchronization corresponds to making the source immutable and target mutable for the synchronization procedure. In contrast, *bidirectional* synchronization between two models treats both models as mutable.
- **Types of models.** Each model involved in synchronization conforms to a metamodel. Each model could conform to a different metamodel, or some or all of the models could conform to the same metamodel.
- **Multiplicity of relationships among elements.** The consistency among the involved models can be expressed as relationships among their elements. The instances of these relationships are traceability links. An important characteristic of the relationships is their multiplicities, such as *one-to-one*, *one-to-many*, and *many-to-many*. Models involving one-to-one or one-to-many relationships are usually easier to synchronize than those involving many-to-many relationships.

User-facing characteristics. Several characteristics are related to how synchronization is triggered and performed from the user viewpoint:

- **Point in time and frequency of triggering.** Synchronization could be performed *continuously*, where every change to a model is immediately propagated to the other models, or could be triggered *on demand*. The continuous mode is inappropriate in most practical cases. The continuous mode would slow down editing because every editing operation would trigger change propagation. Also, we usually do not want to keep models synchronized at all times; in particular, we do not want to synchronize with all intermediate states of a model while the model is edited. A more practical choice is to allow the user to invoke synchronization when he deems it appropriate.
- **Scope of synchronization.** Synchronization can be performed in a *push* or *pull* mode. In the push mode, a change to a model is pushed to all dependent models. In the pull mode, each dependent model can be synchronized individually on demand. An important consideration is whether all the models to be synchronized are available within a single development environment, or whether they are distributed among different physical locations. In the latter case, the push mode would usually involve pushing change notifications only, while the user at the receiving side would decide whether and when to perform synchronization.
- **Reconciliation strategy.** In general, different reconciliation strategies may involve different degrees of *automation* versus *interaction with the user*, *preservation of the existing model structure*, and *completeness*. During reconciliation, some model changes may be determined as necessary. They can be performed automatically. However, different alternative changes may be available in order to achieve further progress with reconciliation. The selection of alternative changes could be automated by making some default choices, or the alternatives could be presented to the user so that choices can be made interactively. Furthermore, the different choices may involve different degrees of modification to the existing elements of the models being

reconciled. If the model to be changed by reconciliation has been automatically generated, overriding existing structures may not be an issue; however, if the model contains manual modifications, they should be preserved as much as possible. Finally, the reconciliation process could be partial. The remaining inconsistencies would be marked as such and left to the user to be resolved manually.

Implementation Choices. Two main issues need to be considered when implementing model synchronization:

– **Representation of the synchronization logic.** The synchronization logic could be expressed in *action-* or *state-oriented* style. In the action-oriented style, edit operations on one model are mapped to edit operations on the dependent models. Such a mapping could be used to implement the continuous mode of synchronization. On-demand mode could be achieved by mapping whole histories of operations. The history of the source model could be recorded during editing, or it could be created by comparing the version of the model before the edits with the version after the edits. In the state-oriented style, the models to be synchronized are analyzed, and one or more of them are transformed to re-establish consistency. The state-oriented approach may sometimes require access to the source model versions before and after editing in order to properly propagate changes. The necessary model transformation can be expressed algorithmically or as a set of transformation rules. The transformation rules can address synchronization explicitly. Alternatively, synchronization can be defined implicitly by model consistency rules. In the former case, the transformation rules will read source and target models and modify the target models to establish consistency. In the latter case, the consistency between the models is specified as rules, and a rule engine will attempt to fix any violations of these rules by modifying the target models. The engine may need to interact with the user, apply some heuristics, and use additional problem-specific strategies to perform the reconciliation.
– **Representation of traceability links.** Traceability links can be explicit or implicit in the synchronization rules. Implicit links are established as instances of pattern matching. They are created and maintained by the rule engine. Explicit links need to be taken care of explicitly by the user. They are not automatically created and maintained by the rule engine. Traceability links can be implemented using in-memory pointers and/or globally unique identifiers of model elements.

4 Feature Model Synchronization

In the following three subsections, we first characterize feature model synchronization using the dimensions introduced in Section 3, then specify the metamodel for representing feature models, and, finally, describe how feature model synchronization can be performed.

4.1 Classification of Feature Model Synchronization

Feature model synchronization can be characterized in terms of the general dimensions for model synchronization as follows:

Structural characteristics. Assuming that specializations and configurations are feature models (as explained in Section 2), feature model synchronization is a unidirectional synchronization between two feature models. In other words, both models conform to the same metamodel. Furthermore, because of cloning, the relationship between elements in a feature model and the elements in one of the feature model's specializations or configurations, is in general, one to many rather than one to one.

User-facing characteristics. Feature model synchronization requires activation on demand. Both pull and push modes are of interest. We also need to consider both synchronization among feature models loaded within one development environment and those distributed to different users. Preservation of existing user choices in specializations and configurations is absolutely important. Therefore, the synchronization process will usually be only partially automatic. Remaining inconsistencies, such as violated cardinality or additional constraints, will be marked. The user can resolve them by using the constraint-solving facilities normally available in feature model specialization and configuration [7]. In other words, an automatic synchronization phase is followed by a tool-supported, interactive phase, in which alternative conflict resolution choices and completions based on defaults are presented to the user.

Implementation Choices. Feature model synchronization can be implemented using the variety of implementation choices listed in Section 3. In the further discussion, we only consider the state-oriented approach. Since continuous update is inappropriate for feature model synchronization, the action-based approach would need to operate on histories. We find the state-oriented approach simpler and more robust since it does not need to consider histories. In Section 4.3 and Appendix A, we will discuss an algorithmic solution and a rule-based one using consistency rules, respectively. Only the latest versions of the source and target models are needed for feature model synchronization. The presented solutions use explicit links, which are established automatically when a copy of a feature model to be specialized is first created. In the case of feature model synchronization, the additional complexity of maintaining the links by the synchronization algorithm explicitly is minimal.

4.2 Metamodel and Renderings for Cardinality-Based Feature Models and Their Specializations

Before getting into the specifics of synchronization, the underlying representation of cardinality-based feature models and their specializations needs to be understood. Figure 4 shows the metamodel for representing feature models and their specializations and configurations. A feature model is a hierarchy of `Nodes` modeled by the `parent-child` composition. A `RootFeature` is the only `Node` without a `parent`. A `Feature` or a `RootFeature` may have `Features`

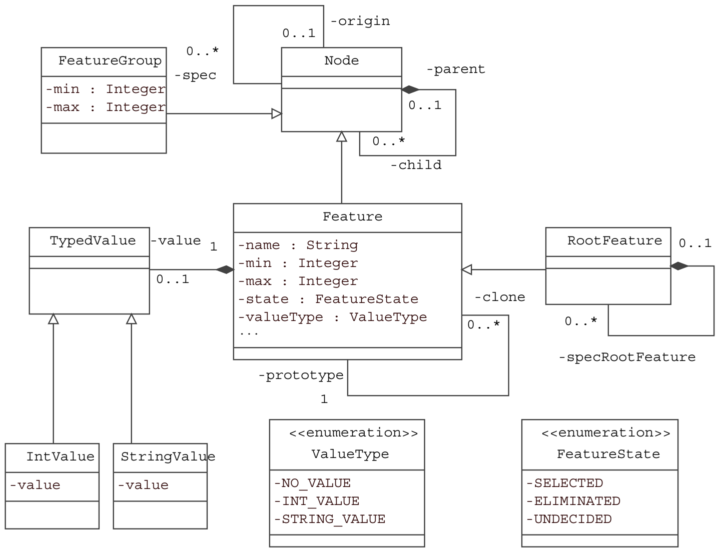


Fig. 4. Metamodel for cardinality-based feature models and their specializations

and/or `FeatureGroups` as children. Each child of a `FeatureGroup` must be a `Feature`. `Features` and `FeatureGroups` use `min` and `max` to represent cardinalities. The cardinality of root and grouped features is always $\langle 1\text{–}1 \rangle$. A `Feature` may have an attribute, in which case its type is indicated by `valueType`. A `Feature` with an attribute may contain a `TypedValue`. The `specRootFeature` composition on `RootFeature` is used to represent a hierarchy of specializations. The `RootFeature` of a feature model contains the `RootFeatures` of its direct specializations and configurations, which are feature models too. Traceability links between `Nodes` of a feature model and its specialization are modeled by the `origin-spec` association. Group and feature cardinalities in the nodes of a specialization are used to override cardinalities of their `origin` nodes. A specialization cannot add an attribute, but only an attribute value. Adding an attribute value is only possible if a value has not been yet assigned in the previous stage. `Feature` has the field `state`, which is used to represent specialization choices on optional solitary features and grouped features, which are made in the check-box view, as shown in Figure 2(a). The allowed values for `state` are `SELECTED`, `UNDECIDED`, or `ELIMINATED`. The meaning of these values will be explained shortly. The relationship `clone-prototype` is used to relate clones to its prototype feature, i.e., the feature from which they were cloned.

When a feature model is first created in the feature model editor as in Figure 1(a), none of the nodes in the model have a corresponding `origin` node. A specialization of the feature model is created by first creating a copy of it and then setting the origin of each node in the specialization to the corresponding node in the original feature model. The feature model editor showing the original feature model renders the model according to Table 1, where the group and feature

Table 2. Interpretation and rendering of `min`, `max`, and `state`

Stored cardinality [`min`..`max`]	Value of state	Effective cardinality	Rendering in check-box view	Rendering in feature-model view
`min`$-c \leq 0\wedge$ `max`$-c \leq 0$	—	[0..0]	✗ or feature not shown	
`min`$-c \leq 0\wedge$ `max`$-c = 1$	`ELIMINATED`	[0..0]	☒	✗ or feature not shown
	`UNDECIDED`	[0..1]	☐	○ for solitary and ▫ for grouped feature
	`SELECTED`	[1..1]	☑	● for solitary and ■ for grouped feature
`min`$-c = 1\wedge$ `max`$-c = 1$	—	[1..1]	● for solitary and ■ for grouped feature	
`min`$-c \leq 0\wedge$ `max`$-c > 1$	—	[0..`max`$-c$]	○● [0..`max`$-c$]	
`min`$-c \geq 1\wedge$ `max`$-c > 1$	—	[`min`$-c$..`max`$-c$]	●● [`min`$-c$..`max`$-c$]	

cardinalities shown in this rendering are those *stored* in the model instance, i.e., [`min`..`max`] and ⟨`min`–`max`⟩, respectively. Rendering of specializations is different. First, specializations are not to be shown using the feature model editor, as free editing is not allowed for a specialization. Instead, a specialization can be shown in (1) the check-box view (e.g., Figure 2(a)), which allows the application of specialization steps, or (2) the feature model view (e.g., Figure 2(b)), which renders the result of the specialization as a feature model and does not allow any editing. Second, the views do not display the cardinalities stored in the specialization instance, i.e., [`min`..`max`] and ⟨`min`–`max`⟩, directly. Instead, both views render the model nodes according to their *effective cardinality*. The effective cardinality for a feature and its corresponding rendering in the check-box and feature model views are given in Table 2. In that table, c refers to the number of clones of that feature. An entry of "—" in the second column means that the value of `state` is insignificant for the case represented by the corresponding row. Feature groups are rendered according to Table 1 and their effective cardinality. The effective cardinality of a feature group with the group cardinality ⟨`min`–`max`⟩ and k features is defined as $\langle \texttt{min}-min(\texttt{max}, k-e)\rangle$, where e is the number of features with the effective feature cardinality of [0..0] that are contained in the group. The specialization step of feature or group cardinality refinement can be achieved in the check-box view by directly editing the cardinality stored in a node, with the only constraint that the refined *stored* cardinality is a subinterval of the *effective* cardinality of the node's `origin`.

4.3 Feature Model Synchronization Steps

Synchronization between a feature model and a specialization starts with an automatic phase, possibly followed by an interactive phase. During the the automatic phase, synchronization steps are applied to the specialization, such as adding, deleting, and relocating nodes, adding and removing attributes, and adjusting attribute values and cardinalities. The interactive phase may be necessary in order to enforce new cardinality values by deleting or creating clones and changing feature selections within groups and in order to re-establish global constraints (such as implies and requires constraints) by further reconfiguration. The changes in the interactive phase cannot be fully automated because there may be many different ways to enforce cardinalities and global constraints and the user may need to be consulted.

In the case of multiple specializations in a multi-stage scenario, the synchronization steps need to be repeated for each stage. The direct specializations of the feature model that was modified are synchronized first. Then the direct specializations of the newly synchronized specializations are synchronized. This process continues recursively until all models are synchronized.

The automatic synchronization steps are described in the remainder of this section in an algorithmic style. In Appendix A, we indicate how these steps can be represented as consistency rules between a model and its specializations in the OMG QVT *Relations* language [4].

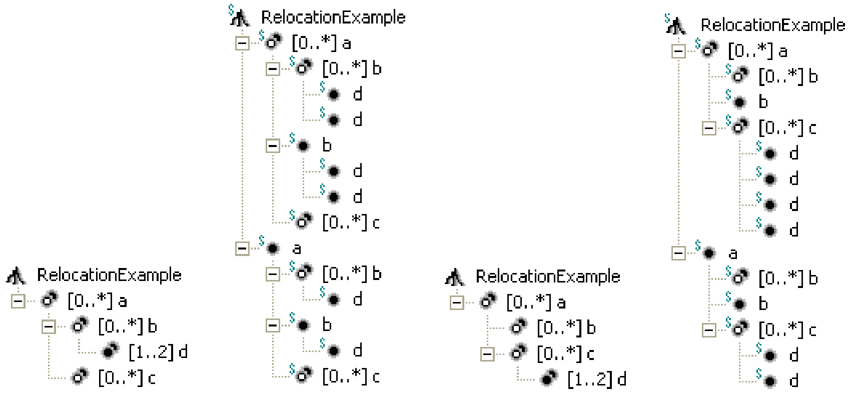
Addition. Node instances that are missing in the specialization are added. For every node `m` in the original model and for every node `s` in `m.spec`, if `m` has a child `mChild` for which there is no child `sChild` of `s` such that `sChild.origin` = `mChild`, such an `sChild` is created. Note that since all the cloned features in the specialization have a traceability link back to the original-model feature, adding nodes underneath that feature means that the specializations of the nodes will be added to each clone.

Removal. Node instances in the specialization whose `origin` attribute is null are removed.

Changing attribute. If a feature `f` in the original model has an attribute type, i.e., `valueType` $\neq$ `NO_VALUE`, every feature in `f.spec` must have the same attribute type. Furthermore, if `f` has an attribute value, every feature in `f.spec` must also have the same attribute value.

Relocation. A model node and everything below may be moved from one parent to another parent. As a result, the specializations of the changed model are out of sync with the model. In particular, the parent of the model node and the parent of a corresponding specialization node will not be connected by the `origin-spec` traceability link. Due to the possible presence of clones, relocating out-of-place specialization nodes is a challenge for synchronizing cardinality-based feature models. Consider a simple feature model in Figure 5(a) and its specialization in 5(b). Imagine that `d` is moved under `c`, as shown in Figure 5(c). There are different ways to synchronize. One way is simply to take all the specialization nodes of `d` and copy them for each specialization node of `c`. There are six `d` specialization nodes in 5(b). That means there would be six `d` specialization nodes in each of the two `c` specialization nodes. In general, because clones are collected throughout the hierarchy, this solution tends to produce an excessive number of clones. Also, the high number of clones will often lead to cardinality violations. The model allows a maximum of two clones of `d`, but the relocation, as described, yields six clones.

Another way is to perform relocation under a well-defined scope, as shown in Figure 5(d). We could take the source container node, or `b`, in this case, and the target container node, or `c`, and find the common ancestor node between the two in the model, which is `a`. We could perform the relocation under the scope of the common ancestor, by taking all the nodes of `d` under the ancestor and copying them under each target node of `c` under the ancestor. In this case, four `d` nodes would end up under the first `c` node and two `d` nodes would end up under the second `c` node. Still, there is a cardinality violation, as four `d` clones are not allowed according to the feature model. However, the move performed against the second `c` clone is fine. This relocation method attempts to reduce the probability and extent of cardinality violation. Nodes violating cardinality are marked as such to allow the user to fix them manually.



(a) Model (b) Specialization (c) Modified model (d) Updated specialization

Fig. 5. Feature relocation example

The synchronization mechanism described can also be used for moving a solitary feature into a feature group in the model. Moving the solitary feature into the feature group this way would mean that both the corresponding prototypes and their clones in the specialization would all end up in the corresponding feature group. Alternatively, only the prototypes can be moved into the feature group, discarding the clones. In any case, the prototypes and clones will be marked for the user to be inspected manually. The user will have to choose exactly one prototype or clone to keep among the prototypes and clones with the same origin and delete the remaining ones.

Cardinality changes. The stored cardinality of every feature group and feature prototype in a specialization has to be a subinterval of the effective cardinality of the corresponding origin group or feature. If this is not the case, `min` and/or `max` of the feature group or feature prototype need to be adjusted to enforce this constraint. Finally, the number of clones in the specialization may violate the stored cardinality of their corresponding prototype feature. Also, the number of selected features in a group may violate the stored group cardinality Both kinds of violations have to be resolved by the user, as they may involve deciding which nodes to delete and which nodes to create.

5 Related Work

We are not aware of any previous work on feature model synchronization. The closest bodies of work are that on model synchronization and schema evolution.

There has been a considerable amount of effort in the model-driven development community to provide generic frameworks for model transformation that support synchronization. An OMG standard for model transformations called "MOF 2.0 Query/View/ Transformation (QVT)" [4] is under development. In Appendix A, QVT *Relations* language is demonstrated through the specification of synchronization rules. Although there does not yet exist a publicly available implementation of QVT, a prototype implementing some of its ideas was developed by IBM under the name *Model Transformation Framework (MTF)* [8]. MTF provides a concise language for specifying equivalence relations for models represented using the Eclipse Modeling Framework (EMF) and a transformation engine. We have experimented with MTF as an infrastructure to implement feature model synchronization for FeaturePlugin [5], a feature modeling tool for Eclipse. The prototype implementation was able to synchronize node additions and removal. Unfortunately, in its current state, MTF simply lacks support for defining the custom constraints that are required to fully achieve synchronization. A comparable approach to MTF is *ModelWeaver*, as proposed by Bezivin et al. [9]. However, in contrast to MTF, which is EMF-centric, the ModelWeaver approach focuses on relationships between different technical spaces, such as MOF, XML, EMF, GrammarWare, etc.

Other works in the model transformation area include the efforts of Ivkovic and Kontogiannis on synchronizing software artifacts across levels of abstraction [10]. In their approach, model dependencies are implicitly encoded using transformation rules and an equivalence relation is used to evaluate when two models become synchronized. Furthermore, Mens et al. use description logics to synchronize between UML models [11].

The synchronization problem has also been explored in the context of schema evolution, for example, in object-oriented databases [12,13,14]. Bernstein et al. describe a general model management framework for schema evolution in which mappings between models are treated as first class objects and operators are defined for common operations on models, such as merging, matching, and differencing [15]. They provide a concrete implementation of the framework called *Rondo* that implements some model operators [16]. Also, Sprinkle et al. describe the use of a graphical model transformation language to migrate models form one version of a metamodel to another version [17]. The relationship between a feature model and its configurations is comparable to that of a schema and the data conforming to that schema. However, in contrast to schema evolution, feature model evolution considers multiple stages of specialization.

Mens et al. [18] propose a taxonomy for software evolution, in which they also discuss dimensions relevant to model transformation such as the time and frequency of updates. However, they are much less detailed and at a much higher abstraction level with respect to model synchronization, compared to the dimensions in Section 3.

6 Conclusion and Future Work

In this paper, we characterize the feature model synchronization problem according to dimensions that are applicable to other model synchronization problems and devise a solution to this problem. Important characteristics of our solution are (1) the use of a uniform metamodel for representing feature models, specializations, and configurations; (2) a two-phase approach automatically synchronizing node hierarchy, attributes, and cardinalities, and leaving the violations of cardinalities and additional constraints (such as requires and excludes constraints) to be resolved using the constraint-solving facilities normally available in feature model specialization and configuration. As a bonus, the unification of the metamodel for feature models, specializations, and configurations lead to the development of the check-box view as a uniform interface for editing both specializations and configurations. This is in contrast to our prior work [5], in which the specialization interface was different from the configuration interface.

We have explored two styles of expressing the synchronization logic. A rule-based solution using the *Relations* language of QVT is presented in Appendix A. As experience with this emerging standard and technology is still very scarce, preparing these rules has been an interesting and useful exercise. Unfortunately, we could not test them because no implementation of the language is publicly available as of writing. However, we currently have a Java implementation of our approach to feature model synchronization in an algorithmic style. The implementation is a part of FeaturePlugin [5]. The synchronization facility in FeaturePlugin is used to synchronize feature models with their specializations and configuration that are loaded into the tool. It is also used to synchronize feature models with their metamodels after the metamodels have been modified The use of our synchronization technique is possible in the latter case since metamodels in FeaturePlugin are feature models, too.

In future work, we plan to explore different practical application scenarios for feature model synchronization, including the case where the feature models to be synchronized are distributed. Furthermore, we would like to better understand the kind of changes needing synchronization that are common in practice. On this basis, we would like to explore practical evolution and synchronization strategies. For example, synchronizing feature additions is usually easier than synchronizing removals and relocations. Thus, a conservative evolution strategy could be to avoid changes other than extensions. Alternatively, modifications could be achieved by including both the old and new parts in the new version of a feature model, with the old part marked as obsolete. The feature model could contain constraints and scripts to configure the new parts of a configuration based on the old parts of the configuration. Furthermore, we would like to understand what factors should be looked at to determine when and how often to synchronize. Finally, we plan to explore the application of the constraint based configuration facilities in FeaturePlugin to increase the automation level and support the user in resolving cardinality and other constraint violations during the interactive phase.

Acknowledgements

We would like to thank Ulrich Eisenecker and the anonymous reviewers for their valuable comments on earlier drafts of this paper.

References

1. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley, Boston, MA (2001)
2. Kang, K., Cohen, S., Hess, J., Nowak, W., Peterson, S.: Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (1990)
3. Czarnecki, K., Helsen, S., Eisenecker, U.: Staged configuration through specialization and multi-level configuration of feature models. Software Process Improvement and Practice **10** (2005) 143–169 Available from `http://swen.uwaterloo.ca/~kczarnec/spip05b.pdf`.
4. Object Management Group, Inc.: Revised submission for MOF 2.0 Query/View/Transformation RFP (ad/2002-04-10). (2005) QVT-Merge Group, version 2.0, ad/2005-03-02.
5. Antkiewicz, M., Czarnecki, K.: FeaturePlugin: Feature modeling plug-in for Eclipse. In: OOPSLA'04 Eclipse Technology eXchange (ETX) Workshop. (2004) Paper available from `http://www.swen.uwaterloo.ca/~kczarnec/etx04.pdf`. Software available from `gp.uwaterloo.ca/fmp`.
6. Czarnecki, K., Helsen, S., Eisenecker, U.: Formalizing cardinality-based feature models and their specialization. Software Process Improvement and Practice **10** (2005) 7–29
7. Batory, D.: Feature Models, Grammars, and Propositional Formulas. Technical Report TR-05-14, University of Texas at Austin, Texas (2005)
8. Griffin, C.: Model Transformation Framework (2000-2004) Tool available at `http://www.alphaworks.ibm.com/tech/mtf`.
9. Bézivin, J., Jouault, F., Valduriez, P.: First experiments with a ModelWeaver. In: Proceedings of the OOPSLA/GPCE'04 Workshop on Best Practices for Model-Driven Software Development. (2004)
10. Ivkovic, I., Kontogiannis, K.: Tracing evolution changes of software artifacts through model synchronization. In: ICSM. (2004) 252–261
11. Straeten, R.V.D., Mens, T., Simmonds, J., Jonckers, V.: Using description logic to maintain consistency between UML models. In: Proceedings of UML 2003. Volume 2863 of LNCS., Heidelberg, Germany, Springer-Verlag (2003) 326–340

12. Monk, S., Sommerville, I.: Schema evolution in OODBs using class versioning. SIGMOD Rec. **22** (1993) 16–22
13. Ra, Y.G., Rundensteiner, E.A.: A transparent schema-evolution system based on object-oriented view technology. IEEE Transactions on Knowledge and Data Engineering **9** (1997) 600–624
14. Rashid, A.: A database evolution approach for object-oriented databases. In: ICSM. (2001) 561–564
15. Bernstein, P.A., Levy, A.Y., Pottinger, R.A.: A Vision for Management of Complex Models. Technical Report MSR-TR-2000-53, Microsoft Research, Redmond, WA (2000)
16. Melnik, S., Rahm, E., Bernstein, P.A.: Rondo: A programming platform for generic model management. In: Proceedings of ACM SIGMOD, San Diego, California, USA (2003)
17. Sprinkle, J., Karsai, G.: Model migration through visual modeling. In: Proceedings of 3rd OOPSLA Workshop on Domain-Specific Modeling, Anaheim, CA (2003)
18. Mens, T., Buckley, J., Zenger, M., Rashid, A.: Towards a taxonomy of software evolution. In: Proceedings of FWO Network Meeting. Foundations of Software Evolution, Vienna, Austria (2002)

A Synchronization Rules in the QVT Relations Language

The following text shows how the synchronization rules can be specified in the *Relations* language that is described in the latest approved version of the MOF 2.0 QVT proposed standard [4].

The synchronization is expressed as a transformation between a model and its specialization (line 1). When the transformation is called with a specialization as its target, the contained relations (lines 3 and 11) will be enforced (as indicated on lines 6 and 14). The relation `ModelRootFeatureToSpecializationRoot-Feature` requires that if a model root `m` (line 5) and a specialization root `s` (line 6) are connected by a `origin-spec` link (line 8), both roots will also satisfy the `ModelNodeToSpecializationNode` relation (line 9). The latter relation states that for every three matching nodes `m`, `mChild`, and `s`, such that `m.child` = `mChild` and `s` is in `m.spec` (line 13), another node `sChild` must exist (line 14–15). That node `sChild` must be a child of `s`, have `mChild` as its origin, and satisfy `ModelNodeToSpecializationNode` together with `mChild` (lines 17–19). If such a node does not exist, it will be created automatically. The rule also states implicitly that any node in specialization that does not participate in the relationship will be deleted.

```
 1 transformation synchronization(model:FeatureMetamodel, specialization:FeatureMetamodel)
 2 {
 3   relation ModelRootFeatureToSpecializationRootFeature
 4   {
 5     checkonly domain model m:RootFeature{spec=aSpec:RootFeature{}}
 6     enforce domain specialization s:RootFeature{}
 7
 8     when { s=aSpec; }
 9     where{ ModelNodeToSpecializationNode(m, s); } }
10
11   relation ModelNodeToSpecializationNode
12   {
    checkonly domain model m:Node{child=mChild:Node{}, spec=s:Node{}};
    enforce domain specialization sChild:Node{parent=sChildParent:Node{},
                                               origin=sChildOrigin:Node{}};

    where { sChildParent = s;
            sChildOrigin = mChild;
            ModelNodeToSpecializationNode(mChild, sChild); } }
}
```

It is relatively easy to extend the above code to handle the synchronization of node names, attribute types, and attribute values. However, handling node relocation and cardinality changes requires calls to imperative functions, which can be provided as part of the metamodel.