GPCE 2006

Verifying Feature-Based Model Templates Against Well-Formedness OCL Constraints

Krzysztof Czarnecki Krzysztof Pietroszek
University of Waterloo, Canada
{kczarnec,kmpietro}@swen.uwaterloo.ca

Abstract

Feature-based model templates have been recently proposed as a approach for modeling software product lines. Unfortunately, templates are notoriously prone to errors that may go unnoticed for long time. This is because such an error is usually exhibited for some configurations only, and testing all configurations is typically not feasible in practice. In this paper, we present an automated verification procedure for ensuring that no ill-structured template instance will be generated from a correct configuration. We present the formal underpinnings of our proposed approach, analyze its complexity, and demonstrate its practical feasibility through a prototype implementation.

Categories and Subject Descriptors D.2.1 [*Software Engineering*]: Requirements/Specifications—Tools; D.2.2 [*Software Engineering*]: Design Tools and Techniques—Computer-aided software engineering (CASE)

General Terms Design

Keywords Model-driven development, software-product lines, formal verification, configuration, feature modeling, model templates, metaprogramming, feature interaction, UML, OCL

1. Introduction

Recently, there has been an increasing interest of the software-engineering community in generative and model-based software development paradigms, such as OMG's Model-Driven Architecture (MDA) [12] or Microsoft's Software Factories [8]. Additionally, there has been a trend in the model-driven software engineering community, as exemplified by the Software Factories approach in particular, towards supporting product-line practices in the modeling context.

One approach that is geared towards modeling software product lines is *feature-based model templates*, which we proposed in our earlier work [3]. A feature-based model template consists of a feature model and an annotated model expressed in some general modeling language such as UML or a domain-specific modeling language. A feature model contains a set of features, or system requirements, that are common or variable among the systems in some application domain. A feature model also defines what combinations of features are allowed in a correct system specification. The annotated model is a semantic specification of the features in some appropriate modeling language. The annotations refer to the features in the feature model, but they may have different forms. In this paper, we consider annotating individual model elements with presence conditions, which are analogous to the `#ifdef` directive of the C preprocessor. Based on a particular configuration of features, a template processor creates an instance of the template by evaluating the presence conditions in the model and removing elements whose presence conditions evaluate to false. A key strength of model templates is that they allow us to maintain several model variants, such as variants of business or design models for different product-line members, in a superimposed form within a single artifact. Furthermore, the model annotations establish the traceability between features and their realizations in the model.

Unfortunately, in our experience, creating and evolving model templates has been an error-prone process because, for example, it is easy to forget a necessary constraint in the feature model or an annotation in the annotated model. While particular instances of the template that are being currently used may be correct, instantiating the template for other configurations, which one would expect to be correct, could lead to incorrect template instances.

While there are many different notions of correctness, in this paper we only concentrate on the well-formedness of the resulting instances. In particular, we give an automatic verification procedure which can establish that no ill-formed template instances will be produced given a correct configuration of the template's feature model. The approach allows us to express the desired well-formedness constraints in the Object-Constraint Language (OCL) [14] with respect to the metamodel of the target modeling language of the template instances. This key capability is achieved through a new semantics of OCL for templates. The semantics maps OCL constraints to propositional formuls, which are then fed into a SAT solver. We present the formal underpinnings of our proposed approach, analyze its complexity, and demonstrate its practical feasibility through a prototype implementation.

As we are not aware of any other verification approaches for feature-based model templates, we believe that our work is both novel and an important step towards model-driven software product lines.

The remainder of this paper is structured as follows. Section 2 presents the necessary background for feature-based model templates. Section 3 elaborates on the verification problem for model templates. Section 4 gives an informal summary of the verification procedure using an example. Formal definitions of the concepts involved in the verification procedure, including model templates and the template interpretation for OCL, are given in Appendices A and B. Section 5 analyzes the computational complexity of the verification procedure. A prototype implementation and some experience

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
GPCE'06 October 22–26, 2006, Portland, Oregon, USA.
Copyright © 2006 ACM 1-59593-237-2/06/0010…$5.00
Reprinted from GPCE'06, [Unknown Proceedings], October 22–26, 2006, Portland, Oregon, USA., pp. 1–**??**.

with it are described in Sections 6 and 7. We report on the related work in Section 8. Finally, Section 9 concludes the paper.

2. Feature-Based Model Templates

A *feature-based model template* consists of a feature model and a model template. The feature model defines the structure of the input parameters for the template. The model template is an annotated model expressed in the target notation defined by a metamodel. We assume that the metamodel is defined using the Meta-Object Facility (MOF) formalism [11], but the approach could be also adapted for other metamodeling formalisms. Thus, the model template can be a UML model, but it can also conform to a domain-specific notation defined using MOF. The elements of the model template can be associated with annotations referring to the features in the feature model. In general, we distinguish between three kinds of annotations: *presence conditions*, *iteration directives*, and *meta-expressions*. In this paper, we only consider presence conditions, which is the most basic and useful of the three annotation kinds. Presence conditions define which elements of the annotated model should be present in an instance of the template and which not.

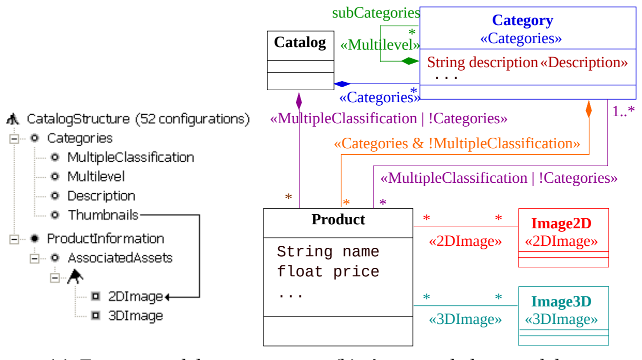


Figure 1. Example of a UML class model template

An example of a feature-based model template is given in Figure 1. The example is an excerpt from a larger model of an e-commerce platform, and it models various options for the catalog structure of the platform.

Figure 1(a) shows a feature model in the cardinality-based feature modeling notation [4]. A feature model is a tree structure with a *root feature* (⋔) as its root and the other nodes being *solitary features*, *grouped features*, or *feature groups*. In our example, the root feature denotes the concept of a catalog structure, with two solitary features as its direct children, namely, `Categories` and `ProductInformation`. Each solitary feature has a *feature cardinality*, which is an interval of the form $[m..n]$, where $m \in \mathbb{Z} \wedge n \in \mathbb{Z} \cup \{*\}$ and $0 \leq m \leq n$, assuming that $\forall m \in \mathbb{Z} : m < *$. Feature cardinality denotes how many copies of the feature with its entire subtree can be included as children of the feature's parent when specifying a concrete configuration. In this paper, we only consider cardinality-based modeling notation *without cloning*, i.e., a subset of the general notation for which the only allowed feature cardinalities are $[0..0]$, $[0..1]$, and $[1..1]$. Features with the cardinality $[1..1]$ are referred to as *mandatory* (●), whereas features with the cardinality $[0..1]$ are called *optional* (○). For example, `ProductInformation` is mandatory, whereas `Categories` is optional. In other words, all catalogs described by the feature model support storing product information, but they may or may not organize products into categories. `Categories` has four optional subfeatures: `MultipleClassification` allows a product to belong to multiple categories; `Multilevel` denotes support for nested categories; `Description` stands for categories supporting category descriptions; and `Thumbnails` stands for categories showing thumbnail images of the contained products. Furthermore, `ProductInformation` has the optional subfeature `AssociatedAssets`, which denotes support for storing various media files. In our example, `AssociatedAssets` has an *or-group* (⋔) with two grouped features, namely, `2DImage` and `3DImage`. In other words, product information may have support for storing two-dimensional images, or three-dimensional images, or both. In general, a feature group gathers k features and has a *group cardinality*, which is an interval of the form $\langle m\text{–}n \rangle$, where $m, n \in \mathbb{Z}$ and $0 \leq m \leq n \leq k$. Group cardinality denotes how many group members can be selected. The group cardinality of an or-group is $\langle 1\text{–}k \rangle$, which is $\langle 1\text{–}2 \rangle$ for the or-group in Figure 1(a). In addition to the node hierarchy and the cardinalities, a feature model may also contain additional constraints, such as *requires* and *excludes* constraints. In our example, selecting `Thumbnails` for categories requires selecting `2DImage` for product information.

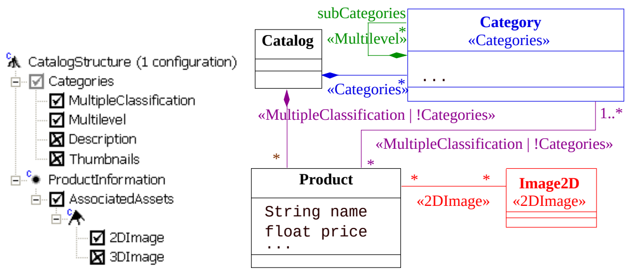


Figure 2. Feature configuration and the corresponding template instance

A feature model denotes a set of configurations. A *configuration* of a feature model is a subset of its features. A *correct* configuration is one that satisfies (i) the constraints represented by the feature hierarchy, (ii) the constraints represented by the cardinalities, and (iii) the additional constraints. The first set of constraints (i) contains the root feature of the feature model and, for every other feature in the model, an implication from the feature to its parent feature. The second set (ii) contains implications of the form $f \Rightarrow choice_{m,n}(f_1, \ldots, f_k)$ for every feature group $f_1, \ldots, f_k$ and its parent feature f and implications of the form $f_1 \Rightarrow f_2$ for every mandatory feature f_2 and its parent f_1. The meaning of $choice_{m,n}$ is as follows. Given the propositional logic formulas $p_1, \ldots, p_k$ and $0 \leq m \leq n \leq k$, $choice_{m,n}(p_1, \ldots, p_k)$ is true iff at least m and at most n of $p_1, \ldots, p_k$ are true. The set of additional constraints (iii) for the model in Figure 1(a) consists of the implication `Thumbnail` $\Rightarrow$ `2DImage`. Thus, the set of correct configurations of a feature model without cloning can be captured by a propositional logic formula, which is the conjunction of all the constraints in these three sets. For the feature model in Figure 1(a) and assuming the abbreviations in Table 1, this formula q_{FM} is as follows:

$$\begin{array}{lll}
q_{FM} & = & \\
\textit{root:} & & \mathtt{cs}\wedge \\
\textit{child-parent:} & & (\mathtt{ct} \Rightarrow \mathtt{cs}) \wedge (\mathtt{mc} \Rightarrow \mathtt{ct}) \wedge (\mathtt{ml} \Rightarrow \mathtt{ct})\wedge \\
& & (\mathtt{ds} \Rightarrow \mathtt{ct}) \wedge (\mathtt{tn} \Rightarrow \mathtt{ct}) \wedge (\mathtt{pi} \Rightarrow \mathtt{cs})\wedge \\
& & (\mathtt{aa} \Rightarrow \mathtt{pi}) \wedge (\mathtt{i2} \Rightarrow \mathtt{aa}) \wedge (\mathtt{i3} \Rightarrow \mathtt{aa})\wedge \\
\textit{group:} & & (\mathtt{aa} \Rightarrow choice_{1,2}(\mathtt{i2}, \mathtt{i3}))\wedge \\
\textit{mandatory:} & & (\mathtt{cs} \Rightarrow \mathtt{pi})\wedge \\
\textit{additional:} & & (\mathtt{tn} \Rightarrow \mathtt{i2})
\end{array} \qquad (1)$$

Table 1. Abbreviations of feature names from Figure 1(a)

Feature	Abbr.	Feature	Abbr.
CatalogStructure	cs	Thumbnails	tn
Categories	ct	ProductInformation	pi
MultipleClassification	mc	AssociatedAssets	aa
Multilevel	ml	2DImage	i2
Description	ds	3DImage	i3

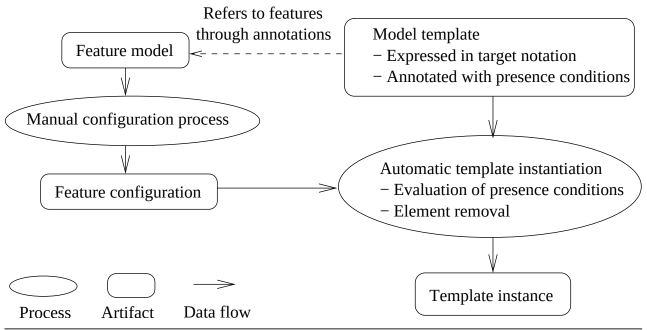

Refers to features
through annotations
Feature model
Model template
− Expressed in target notation
− Annotated with presence conditions
Manual configuration process
Feature configuration
Automatic template instantiation
− Evaluation of presence conditions
− Element removal
Template instance
Process
Artifact
Data flow

Figure 3. Model template instantiation

where

$$choice_{1,2}(\texttt{i2},\texttt{i3}) = \texttt{i2} \wedge \neg\texttt{i3} \vee \neg\texttt{i2} \wedge \texttt{i3} \vee \texttt{i2} \wedge \texttt{i3} = \texttt{i2} \vee \texttt{i3}$$

The feature model in Figure 1(a) denotes 52 correct configurations. A sample correct configuration is shown in Figure 2(a) using a so-called *check-box rendering* of a configuration tool [5]. In this rendering, optional features are shown as check boxes. The boxes of selected features are checked. The boxes of eliminated features are crossed.

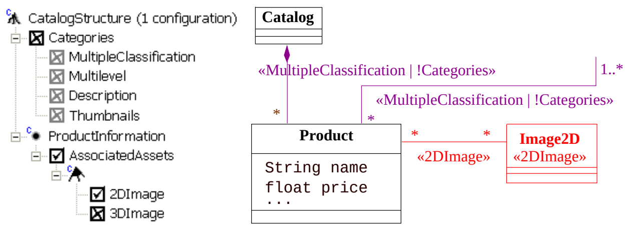
Figure 1(b) shows an example of a UML class model template, which is a UML class model annotated with presence conditions. In this paper, we assume that presence conditions are propositional formulas over the features from the feature model. A presence condition defines for which combinations of features a given model element is required. In Figure 1(b), presence conditions are shown as UML stereotypes. For example, the class `Image2D` and the adjacent association are annotated with the feature `2DImage`, meaning that they are only needed if `2DImage` is selected. Although the majority of elements is annotated with single features, more complex presence conditions are sometimes required, such as in the case of the containment relationship between `Catalog` and `Product`. The latter presence condition is required since products need to be contained directly in the catalog if categories are not supported or if multiple classification is allowed. The presence condition of `Product`, which is not annotated, is assumed to be true by default. As realized in a prototype implementation [3], the display of the stereotype labels next to the annotated elements can be optionally suppressed, and a color encoding can be applied to the elements based on the stereotypes.

The key idea of a feature-based model template is that, given a particular feature configuration, an instance of the template can be automatically created by removing the model elements whose presence conditions evaluate to false. An overview of this process is presented in Figure 3. Figure 2(b) shows an instance of the template in Figure 1(b) for the configuration in Figure 2(a). For example, the class `Image3D` is not present in the instance because the feature `3DImage` was eliminated.

3. Well-Formedness Problem of Model Templates

An important concern of template development is ensuring its correctness. A feature-based template is correct iff every correct configuration results in a correct template instance. Although, template users can apply verification techniques directly to the template instances they create, the template developer ideally should ensure the correctness of a template before it is passed on to the users. Template correctness will save the users from potentially having to deal with the template developer's bugs.

An important aspect of template correctness is well-formedness. In our experience, it is easy to make a mistake in a model template that will lead to an ill-formed instance for a correct configuration. A similar problem often occurs in practice with other kinds of templates, such as server-side templates for producing web pages or C pre-processor directives, too. In our case, mistakes could be made in the feature model, the annotated model, and/or the annotations. For example, the template developer could forget to include an additional constraint in the feature model or attach an appropriate annotation to an element. Such mistakes are particularly likely when new features and variants are added to the template, e.g., during an extension of a product line with new products. In fact, the template in Figure 1 has an annotation error, which will lead to a dangling association for any configuration with `Categories` being false. The resulting malformed instance for one such correct configuration is shown in Figure 4. The annotation error in Figure 1(b) can be corrected by changing the annotation of the non-aggregate association between `Category` and `Product` from `MultipleClassification | !Categories` to just `MultipleClassification`. The example in Figure 1 is based on a previous paper [3]. Early drafts of that paper had exactly the erroneous example from Figure 1, which went unnoticed for several revisions of the paper.



(a) Correct configuration (b) Malformed template instance

Figure 4. Sample configuration leading to a dangling association

In this paper, we assume that the abstract syntax of the target modeling notation is given by a metamodel expressed in MOF, which is essentially a combination of a class model and additional well-formedness constraints expressed in OCL. A model is well-formed iff it conforms to the metamodel, i.e., it satisfies the multiplicities and the OCL constraints of the metamodel. Figure 5 shows a fragment of the UML 2.0 metamodel for class modeling. A sample class model conforming to the metamodel is shown in Figure 6. The sample model is actually an excerpt from Figure 1(a) containing the incorrectly annotated association. Figure 6(a) shows the sample model using the UML concrete syntax. The corresponding abstract syntax using the UML object diagram notation is shown in Figure 6(b). Note that the association between `Category` and `Product` is represented by three objects: an instance of `Association` and two instances of `Property`. The properties, which play the role of `memberEnds`, are contained by the respective instances of `Class`. For simplicity, we do not show the abstract syntax objects representing the stereotypes attached to `Category` and the association, but we only indicate them using concrete syntax in Figure 6(b).

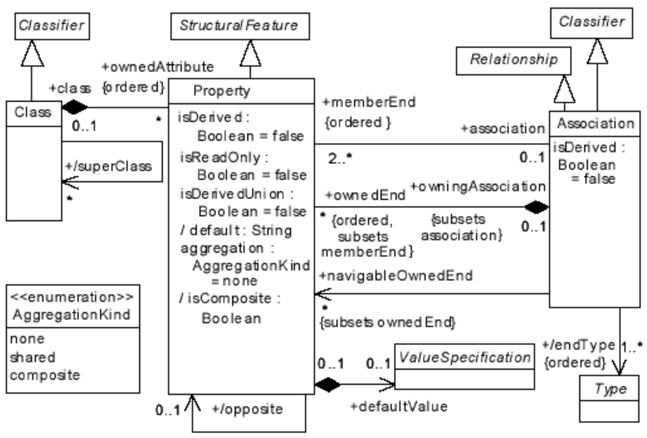
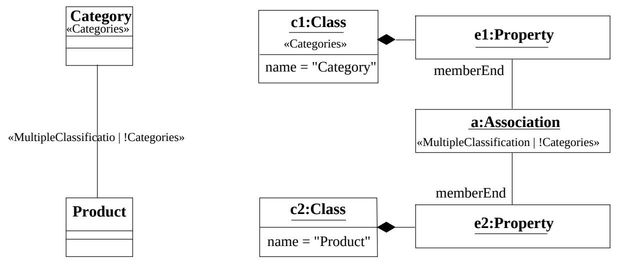


Figure 5. Fragment of the UML 2.0 metamodel for class modeling



(a) Concrete syntax (b) Abstract syntax

Figure 6. Sample class model conforming to Figure 5

Well-formedness constraints may come from different sources. One such source is the specification of the language being used, i.e., the UML standard [13] in our example. Some well-formedness constraints are stated in the UML standard explicitly. For example, the standard contains the following constraint expressed as an OCL invariant on `Association` (see Figure 5) and stating that only binary associations can be aggregations [13, p. 55]:

```
context Association inv:
  self.memberEnd->exists
      (aggregation<>AggregationKind::none)
    implies self.memberEnd->size() = 2
```

Unfortunately, many well-formedness constraints need to be inferred from the informal text of the UML specification. For example, the requirement in the standard [13, p. 56] that “a part instance be included in at most one composite at a time,” implies that a class cannot participate in more than one composition at the part end if any of these compositions has the multiplicity `[1..1]` at the aggregate end. The latter can be stated as the following OCL constraint, which is not given in the specification explicitly:

```
context Class inv:
  let oppositeAggEnds = self.attribute->collect
    (opposite)->select(isComposite)
  in  oppositeAggEnds->size() = 1 or
      oppositeAggEnds->forAll
        (lowerBound()=0 and upperBound()=1)
```

In general, well-formedness constraints may be more complex than the two previous examples, such as a constraint ensuring that a classifier realizing an interface has to implement all operations of the interface. Furthermore, they may span different subnotations of

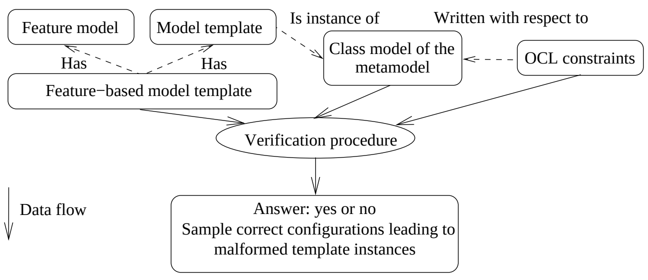


Figure 7. Context of the verification procedure

UML, such as a constraint ensuring that an operation called in an activity model is actually present in the class model.

Additional well-formedness constraints may be needed when the modeling language is specialized for a specific application domain or for the purposes of a particular project, e.g., to enforce a particular set of modeling style guidelines. In UML, such specializations are represented as profiles, which contain the additional constraints. For example, a profile for business entity models such as the one in Figure 2(b) could include the stereotype ≪`root`≫ with the constraint that any class not marked by that stereotype must participate in at least one composition at the part end.

The need for the template instance to satisfy the multiplicities in the class diagram can be stated as a set of additional OCL constraints. Thus, a verification procedure that can ensure the well-formedness of all template instances for all correct configurations against a set of OCL constraints can also be used to ensure the well-formedness of the instances with respect to the multiplicities of the metamodel. Consequently, any multiplicity `[`m`..`n`]` in the class model of the metamodel that is other than `[0..*]` can be relaxed to `[0..*]` by extending the set of well-formedness rules with an OCL constraint of the following form:

```
context c inv:                                          (2)
  let s:Integer = self.r->size() in m<=s and s<=n
```

In the above constraint, r is the role name at the binary association end with the the multiplicity `[`m`..`n`]` and c is a class at the opposite end of the association. Interestingly, there are relatively few non-composite association ends with multiplicity other than `[0..*]` in the UML 2.0 metamodel. For example, the metamodel part covering class modeling has fewer than ten such cases, which, except for `[2..*]` on `memberEnd` (see Figure 5), involve only multiplicities of the form `[0..1]`, `[1..1]`, and `[1..*]`.

While the well-formedness of a model is relatively easy to establish by simply evaluating the well-formedness constraints on the model, establishing well-formedness for model templates is not so easy. The reason is that a template may have a huge number of correct configurations and creating and individually checking an instance for each correct configuration is usually not feasible. While the small feature model in Figure 1(a) already has 52 correct configurations, that number can easily reach astronomic dimensions for practical feature models. Therefore, we need an efficient automatic verification procedure for establishing the well-formedness of model templates.

4. Verification Approach

The context of the verification procedure is shown in Figure 7. The procedure takes a feature-based model template, which consists of a feature model and an annotated model (i.e., template), the class model of which the annotated model is an instance of, and a set of well-formedness rules in OCL. The OCL rules are written with respect to the class model, and the class model and the rules form the metamodel of the target language, to which all template instances should conform. Consequently, in addition to being part of the metamodel of the target language, the class diagram is also the metamodel for the annotated model, i.e., the template.

4.1 Template Interpretation of OCL

Verifying that for all correct configurations of a feature model the resulting template instances are well-formed with respect to a set of OCL constraints can be achieved through an alternative semantics of OCL, which we refer to as the *template interpretation*. In a nutshell, while OCL expressions are normally evaluated over object structures, the template interpretation allows us to evaluate an OCL expression over a template, i.e., an object structure whose objects have presence conditions, as shown in Figure 6(b). The result of evaluating an expression over a template using the template interpretation is a set of all the values that could be obtained from evaluating the same expression on all instances of the template using the standard OCL semantics. Furthermore, each value in that set is annotated with its own presence condition, which is a propositional formula over the features from the feature model. The condition is such that, given a template instance that was created from a feature configuration satisfying the condition, evaluating the OCL expression using the usual OCL semantics over the template instance will yield the associated value. In other words, the template interpretation of OCL involves operations over sets of value-condition pairs. While the precise definition of the template interpretation is given in Appendix B, we illustrate its main idea in this section using an example.

Each OCL constraint that we want to check has the form of an invariant on some class c, with the constraint's body being a Boolean OCL expression e:

```
context c inv:
  e                                                    (3)
```

The standard OCL interpretation of this invariant with respect to a model is that e should evaluate to true for every object $\underline{c}$ of the class c. The template interpretation of e, on the other hand, yields $\{(\textit{true}, p_t), (\textit{false}, p_f), (\bot, p_\bot)\}$, where p_t, p_f, and $p_\bot$ are propositional formulas over features. Note that OCL uses a three-value logic and $\bot$ denotes "undefined".

As an example, let us consider a constraint requiring that every association has at least two ends:

```
context Association inv:
  memberEnd->size() > 1                                (4)
```

Without considering its presence conditions, the model in Figure 1(b) satisfies this constraint since every association in that model has at least two ends. In particular, evaluating `memberEnd ->size() > 1` on `a1` in Figure 6 using the standard OCL interpretation clearly yields true since `a1` is associated with two ends, namely `e1` and `e2`. More precisely, navigating to `memberEnd` from `a1` yields $\{\texttt{e1}, \texttt{e2}\}$, applying `size()` to this set yields 2, and comparing $2 > 1$ yields true.

Let us now consider the template interpretation of `memberEnd ->size() > 1` on `a1`. First, navigating to `memberEnd` from `a1` yields $\{(\texttt{e1}, p^*(\texttt{e1})), (\texttt{e2}, p^*(\texttt{e2}))\}$, where $p^*(\texttt{e1})$ and $p^*(\texttt{e2})$ denote the *accumulated presence conditions* of `e1` and `e2`, respectively. The accumulated presence condition of an object $\underline{c}$ is the conjunction of the presence condition of $\underline{c}$ and all its container objects, if any. Thus, given that the presence condition of `e1` is true and the presence condition of its containing object, namely `c1` (see Figure 6), is `ct`, $p^*(\texttt{e1})$ is $ct \wedge \textit{true}$, i.e., ct. A similar calculation for $p^*(\texttt{e2})$ yields true.

The next step, which is the application of `size()` to the result of `memberEnd`, i.e., $\{(\texttt{e1}, p^*(\texttt{e1})), (\texttt{e2}, p^*(\texttt{e2}))\}$, yields the set of three possible values, i.e., 0, 1, or 2, with their corresponding conditions: $\{(0, \neg p^*(e_1) \wedge \neg p^*(e_2)), (1, p^*(e_1) \wedge \neg p^*(e_2) \vee \neg p^*(e_1) \wedge p^*(e_2)), (2, p^*(e_1) \wedge p^*(e_2))\}$.

Finally, we need to apply the binary comparison operator `>`. While the standard interpretation of this operator takes two numbers and returns true or false, the template interpretation takes two sets, e.g., $\{(1, p_1), (2, p_2)\}$ and $\{(0, p_3), (3, p_4)\}$, where $p_1, \ldots, p_4$ are conditions, and yields a set that contains the results for all possible combinations of the arguments, i.e., $\{(1 > 0, p_1 \wedge p_3), (1 > 3, p_1 \wedge p_4), (2 > 0, p_2 \wedge p_3), (2 > 3, p_2 \wedge p_4)\}$. The latter set is easily simplified to $\{(\textit{true}, p_1 \wedge p_3 \vee p_2 \wedge p_3), (\textit{false}, p_1 \wedge p_4 \vee p_2 \wedge p_4)\}$.

Coming back to our original example, we need to compare the result of `self.memberEnds->size()`, namely $\{(0, \neg p^*(e_1) \wedge \neg p^*(e_2)), (1, p^*(e_1) \wedge \neg p^*(e_2) \vee \neg p^*(e_1) \wedge p^*(e_2)), (2, p^*(e_1) \wedge p^*(e_2))\}$, and $\{(1, \textit{true})\}$. The latter set is the template interpretation of the OCL literal `1`. The result of this comparison is the following set: $\{(0 > 1, \neg p^*(e_1) \wedge \neg p^*(e_2) \wedge \textit{true}), (1 > 1, p^*(e_1) \wedge \neg p^*(e_2) \vee \neg p^*(e_1) \wedge p^*(e_2) \wedge \textit{true}), (2 > 1, p^*(e_1) \wedge p^*(e_2) \wedge \textit{true})\}$, which can be easily simplified to the following set:

$$\{(\textit{true}, p^*(e_1) \wedge p^*(e_2)), (\textit{false}, \neg(p^*(e_1) \wedge p^*(e_2)))\} \tag{5}$$

4.2 Verification Steps

Given the OCL constraint (3), we can verify a template against the constraint for every object of class c in the template. The verification for a given object $\underline{c}$ can be achieved in two steps.

1. We evaluate e for $\underline{c}$ (i.e., assuming that `self` refers to $\{(\underline{c}, \textit{true})\}$) using the template interpretation. The template interpretation of e yields $\{(\textit{true}, p_t), (\textit{false}, p_f), (\bot, p_\bot)\}$. The resulting formula p_t is needed as an input for the next step. According to our calculation in the previous section, the template interpretation of the body expression of the constraint (4) on the object `a` (see Figure 6) is $\{(\textit{true}, p^*(e_1) \wedge p^*(e_2)), (\textit{false}, \neg(p^*(e_1) \wedge p^*(e_2))), (\bot, \textit{false})\}$ (see set (5)). Thus, in our example, p_t is $p^*(e_1) \wedge p^*(e_2)$, which becomes `ct` after substituting the values for the accumulated presence conditions.
2. We check the validity of the propositional formula $q_{FM} \Rightarrow (p^*(\underline{c}) \Rightarrow p_t)$. If the formula is valid, the template is well-formed with respect to (3) for $\underline{c}$. In that formula, q_{FM} is the propositional formula representing the set of correct feature configurations of the feature model *FM*. In our example, after substituting the values for $p^*(\texttt{a})$ and p_t, the formula that we need to check for validity is $q_{FM} \Rightarrow ((\texttt{mc} \vee \neg\texttt{ct}) \Rightarrow \texttt{ct})$, where q_{FM} is given in the equation (1).

Let us explain these two steps. The template interpretation of e in step one yields a set of value-condition pairs, where each condition represents a family of template instances for which e will evaluate to the corresponding value using the standard OCL semantics. According to the set, the standard OCL interpretation of e for $\underline{c}$ in the context of any template instance created from a feature configuration for which p_t is true will yield *true*. However, we only care about ensuring e for $\underline{c}$ in a template instance if $\underline{c}$ is present in that instance, i.e., $p^*(\underline{c})$ evaluates to true for the configuration from which the template instance was created. Consequently, the constraint (3) will not be violated for $\underline{c}$ in a template instance created from any configuration for which $p^*(\underline{c}) \Rightarrow p_t$ is true. Thus, the desired correctness of template instances for correct configurations is established iff the latter condition is implied by the feature model, i.e., $q_{FM} \Rightarrow (p^*(\underline{c}) \Rightarrow p_t)$ is valid, meaning that it is true for any configuration, correct or not. Consequently, step two verifies the well-formedness of the template against the OCL constraint (3) for $\underline{c}$ by checking the validity of the formula representing the correctness condition.

4.3 Error Detection, Reporting, and Resolution

Step two of our verification approach requires verifying the validity of a propositional formula. In practice, this can be achieved by checking the satisfiability of the negation of the formula using a SAT solver. Although the satisfiability problem is NP-complete, there are SAT-solvers that can very efficiently check satisfiability for practical cases. In particular, our experience with solvers based on Binary-Decision Diagrams (BDDs) has been very positive. We comment on the latter in Section 5. If the negation is satisfiable, we have just found an error in our feature-based model template. A SAT solver will usually also give sample valuations, which correspond to configurations in our context, for which the resulting template instances will not be well-formed. Furthermore, since we check each constraint for each instance of the context class separately, an error can be pinpointed to a particular object in the template. In our example, the negation of the correctness formula, i.e., $\neg(q_{FM} \Rightarrow ((\mathtt{mc} \vee \neg\mathtt{ct}) \Rightarrow \mathtt{ct}))$, is satisfiable, with a sample solution being the configuration in Figure 4(a). Thus, for that configuration, the constraint (4) is violated for the association `a`.

The cause for the invalidity of the correctness formula may be located in the feature model, the annotations, and/or the structure of the annotated model. Checking the individual constituents of the correctness formula may be helpful in locating that cause. Above all, we want to check the satisfiability of q_{FM} to make sure that the set of correct configurations is non-empty. Furthermore, if the formula $p^*(\underline{c}) \Rightarrow p_t$ is not satisfiable, we know that there is a guaranteed error in the template (its annotations and/or model structure), while the feature model may be correct. Finally, when none of the two previous cases is true but the correctness formula is not valid and the template developer determines by inspecting the template that the annotations and the structure of the annotated model is correct, we may have a case that a constraint is missing in the feature model. The missing constraint could come either from the problem space (domain analysis) or it could be necessary because of the particular implementation of the feature model in the model template (i.e., constraint determined by the solution space). In that case, we can apply one of the existing formula simplifiers from the field of hardware synthesis on the negation of the correctness formula and add the result to the feature model as an additional constraint. The latter action corresponds to propagating a constraint from the solution space to the problem space.

5. Computational Complexity of the Approach

In this section, we characterize the complexity of our verification procedure by analyzing both verification steps from Section 4.

In the case of step one, we need to only look at the space complexity of evaluating an OCL expression according to its template semantics since the size of the collections is the main performance bottleneck of the evaluation. This complexity can be characterized in terms of the sizes of the collections (e.g., sets of alternatives) involved in the computation. In particular, the complexity is heavily dependent on the kinds of operations and their usage patterns in the expression. We classify some OCL operations that routinely occur in well-formedness constraints (such as those for UML) in terms of their worst-case space complexity in Table 2.

Looking at Table 2, we should be only concerned about the last two entries. Binary operations such as $+$ and $-$ can be a concern, but only if they are used in certain undesirable patterns, such as chaining them. For example, an expression of the form $x_1 + \ldots + x_n$ has the complexity $O(m^n)$ (assuming m is the size of x_i), i.e, it is only tractable for a small n. Fortunately, we have not come across the need to write such expression in well-formedness constraints based on our analysis of the UML specification. The operation *size* on collections has the worst complexity. Again, we found that we only needed to apply *size* on collections of relatively small sizes in well-formedness constraints. For example, *size* is often used to enforce multiplicities as in constraint (2). Since the upper bound of the multiplicities that need to be enforced is usually small (such as 1 or 2; note that * does not need to be enforced), the corresponding collections of optional elements in the template are not likely to be large.

Table 2. Space complexity of OCL operations according to template semantics

Operations	**Result collection size**[a]
nullary: 12, etc.	$O(1)$
unary: *not*, $-$, *toUpper, toLower, length(String), isEmpty, notEmpty, isDefined*	$O(n)$
binary: *union, include*	$O(n+m)$
binary: $+$, $-$, $*$, *and, or, implies, xor, concat*, $=$, $<$, $>$, $<=$, $>=$, *includes*; iteration: *exists, forAll, select, reject, collect*	$O(n*m)$
unary: *size(Collection(t))*	$O(2^n)$

[a] For unary operations, n is the size of the input collection. For binary operations, the input collections have sizes n and m, respectively. For iteration operations, the size of the input collection is n and the size of each collection resulting from evaluating the iteration expression on each input element is assumed to be m.

The complexity of the second step when using a BDD-based SAT solver is usually dependent mainly on the number and the ordering of variables in the formula to be checked. Although the worst-case complexity for this step can be exponential, experience shows that such solvers can handle many practical problems efficiently.

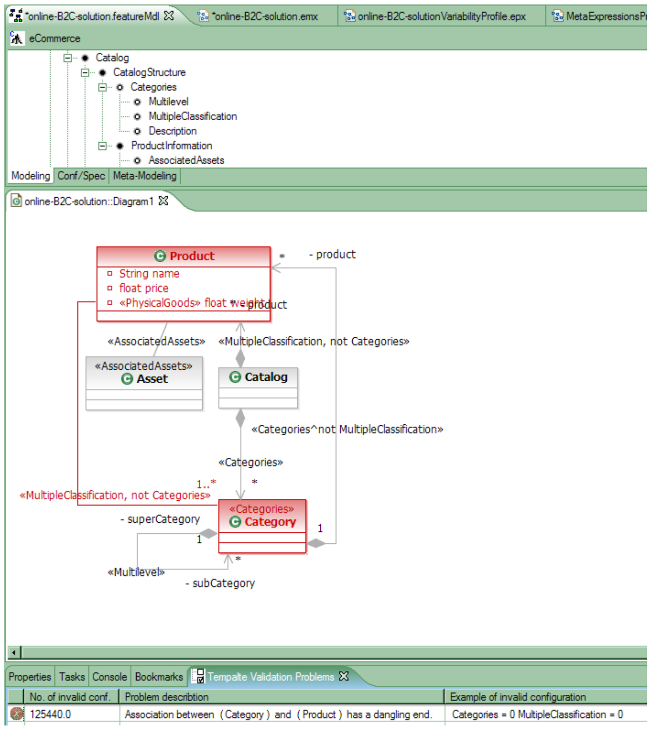
6. Prototype Implementation

In order to establish the practical feasibility of our approach, we have implemented a template verifier according to the procedure presented in this paper as part of *fmp2rsm* [3], which is a plug-in extending the IBM Rational Software Modeler (RSM) with feature-based model templates. RSM is an Eclipse based environment for UML 2.0 modeling. Support for feature modeling within *fmp2rsm* is provided through another Eclipse-based plug-in, namely Feature Modeling Plugin [1], which can run inside of RSM.[1] Figure 8 shows a screen shot of our prototype after detecting the dangling association between `Product` and `Category`. The template shown is the one from Figure 1. The verifier reports the error and pinpoints it by highlighting the association and the two classes. A sample configuration that will result in the dangling association is also given. The well-formedness constraints and the text to be shown if they are not met can be given to the tool in a XML file. The complete OCL constraint expressing the absence of dangling associations is as follows:

```
context Association inv:
  memberEnd->size() > 1 and
  memberEnd->forAll(type.isDefined)                    (6)
```

We have implemented the template interpretation for OCL by modifying an existing OCL evaluator, namely the *Dresden OCL Toolkit* [16]. We reuse the OCL parser generated by Ansgar Konnermann using the LALR(1) parser generator *SableCC* [7]. Our current implementation of the evaluator using the template interpretation supports object navigation; arithmetic, logic, and relational operations; and collection operations including `size`, `isEmpty`, `notEmpty`, `exists`, `forAll`, `collect`, `select`, `reject`, and

[1] Online demonstrations of the fmp2rsm including the verifier are available at `http://gp.uwaterloo.ca/fmp/`.



No. of invalid conf.	Problem describtion	Example of invalid configuration
125440.0	Association between (Category) and (Product) has a dangling end.	Categories = 0 MultipleClassification = 0

Figure 8. Screenshot of the fmp2rsm tool with the integrated template verifier

`includes`. We check the satisfiability of the propositional formula $\neg(q_{FM} \Rightarrow (p^*(\underline{c}) \Rightarrow p_t))$ using the JavaBDD package [18].

The current implementation of the verifier works as follows. Before checking any constraints, the propositional formula q_{FM} describing the correct configurations of the feature model is created as a BDD. This step is done only once, as the result is reused when checking the individual constraints. Next, the OCL constraints are evaluated using the modified OCL evaluator. Currently, the p_t formula that results from each evaluation is represented as a string, which is then parsed and converted into a BDD. The string representation simplifies interfacing with different BDD packages. However, better performance could potentially be achieved by having the OCL evaluator build and manipulate BDDs directly, as the intermediate propositional formulas would be possibly simplified immediately. The BDD of each formula to be checked for satisfiability is obtained by appropriately merging the BDDs of the constituent formulas, namely q_{FM}, p_t, and $p^*(\underline{c})$. Finally, each resulting BDD is checked if it has any solutions.

7. Experimental Evaluation

We have tested our prototype on business model templates for an e-commerce platform, which were developed by a member of our group who was not involved in the research on template verification. A detailed description of the templates is available [10]. The templates include a feature model with 214 features, 5 annotated class diagrams with 38 classes (28 annotated), and 17 activity diagrams with 238 nodes (31 annotated). The models contain a total of 138 annotations, which use 79 features. The templates are based on the business models of the IBM Websphere Commerce product, which are part of the product's documentation. Therefore, we believe that the templates are representative of practical business templates in terms of their structure. Our verifier found 7 dangling associations in the templates as given in [10].

In our experiments, the average time for creating the BDD for the feature model from the case study on a Pentium IV M 1.7GHz with 1 GB RAM was less than a minute. Checking for dangling associations on the annotated class model took less than a second. Checking for potential composition sharing (see the constraint in Section 3) took about 10 seconds. Verifying the constraints for the absence of dangling associations and composition sharing is linear with the number of associations in the model the constraint (6) accesses only the close neighbors of a context object.[2] Constraints that navigate over large portions of a model are usually more expensive. For example, the following constraint ensures that no template instance has an unused enumeration type:

```
context Enumeration inv:
  Class.allInstances()->collect
    (attribute.type)->includes(self)
```

Checking this constraint on the annotated class model took about 30 seconds. Obviously, this time depends on the size of the model.

8. Related Work

We are not aware of any existing work on (1) the verification of code or model templates whose input are feature models and (2) the verification of model templates in particular. Probably the closest work is that of Huang et al. [9] on verifying a particular flavor of Java templates, which uses a simple metalanguage equipped with selection and iteration. The input for these templates is also Java code, which must conform to a set of constraints expressed in predicate logic and specified by the template developer. Furthermore, the template developer specifies predicate logic constraints on the the template instances. The template verifier can then check that latter constraints will hold for any instance created from a correct input. The latter statement also applies to our approach; however, there are important differences. The focus of our approach is different since we want to support feature models created during domain analysis for a product line as the description of template input. Furthermore, we restricted the metalanguage for the purpose of the verification to selection only (i.e., presence conditions). While this may seem very restrictive, a great deal of model templates needed in the product-line context, such as templates for business process models, rarely require more than presence conditions. In particular, adding iteration directive would likely make the templates too programming-oriented for business modelers. Using feature models as the input structure coupled with the metalanguage being presence conditions in the form of propositional formulas over the features makes our approach computationally more tractable. In particular, the Java template verifier has to call a theorem prover, while we only need a SAT solver. Furthermore, the Java template verifier is not guaranteed to come up with a positive or negative answer for all templates and template inputs. This is not the case for our verification procedure, which is both sound and complete. Finally, the Java template verifier works on code templates, while our approach, through the use of MOF and OCL, is geared towards modeling. However, our approach can also be used for code templates whose input is described by feature models, in which case the abstract syntax of the programming language would need be expressed using MOF and OCL.

[2] Checking the constraint for dangling association is exponential with the arity of an association because of the complexity of the *size* operation. Fortunately, associations with an arity higher than three are extremely rare in practice. Also, checking for composition sharing is exponential with the number of compositions in which a class participates. Similarly, this number is usually small in practice.

The application of SAT solvers for creating configurations of and verifying feature models is not new. In particular, the formalization of feature models based on predicate logic presented in Section 2 was inspired by Batory's work [2], which also recommends the use of SAT solvers. Furthermore, the use of BDD-based constraint solvers has also been proposed by van der Storm [17].

9. Conclusions and Future Work

In this paper, we proposed a well-formedness verification procedure for feature-based model templates. The approach allows OCL constraints to be written against the metamodel of the target notation for the template instances. These constraints can be evaluated against a template using a novel template interpretation for OCL. The result of this evaluation can then be fed into a SAT solver in order to verify that no ill-formed template instances can be produced based on a correct configuration. In addition to giving a formal definition of our approach, we also validated its feasibility through a prototype implementation.

Based on our early positive experience with applying the prototype on another project within our research group, in which business model templates for an e-commerce platform are being developed, we believe that it greatly improves the usability of feature-based model templates by catching common mistakes in templates that otherwise would go unnoticed.

Ideas for future work include (1) extending the verification procedure to support additional metaprogramming facilities of the feature-based template approach, in particular, meta-expressions and flow closure [3]; (2) exploring optimization opportunities such as condition simplification in value-condition sets computed at intermediate steps of the template evaluation of an OCL expression; and (3) investigating the verification of semantic properties of model templates, such as the verification of activity model templates using model checking technology.

Acknowledgments

We would like to thank Andrzej Wasowski and the anonymous reviewers for their valuable comments on earlier drafts.

References

[1] M. Antkiewicz and K. Czarnecki. FeaturePlugin: Feature modeling plug-in for Eclipse. In *OOPSLA'04 Eclipse Technology eXchange (ETX) Workshop*, 2004. Paper at `http://swen.uwaterloo.ca/~kczarnec/etx04.pdf`; software at `gp.uwaterloo.ca/fmp`.

[2] D. S. Batory. Feature models, grammars, and propositional formulas. In *Software Product Lines Conference (SPLC)*, volume 3714 of *LNCS*, pages 7–20. Springer-Verlag, 2005.

[3] K. Czarnecki and M. Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In *Generative Programming and Component Enginering (GPCE)*, volume 3676 of *LNCS*, pages 422–437. Springer-Verlag, 2005.

[4] K. Czarnecki, S. Helsen, and U. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process Improvement and Practice*, 10(1):7–29, 2005.

[5] K. Czarnecki and C. H. P. Kim. Cardinality-based feature modeling and constraints: a progress report. In *International Workshop on Software Factories*, San Diego, California, Oct 2005. Paper available at `http://www.ece.uwaterloo.ca/~kczarnec/sf05.pdf`.

[6] M. de Jonge and J. Visser. Grammars as feature diagrams. In *ICSR7 Workshop on Generative Programming*, pages 23–24, 2002. `http://www.cwi.nl/events/2002/GP2002/GP2002.html`.

[7] É. Gagnon. Sablecc: An object-oriented compiler framework. Master's thesis, School of Computer Science, McGill University, Montreal, Mar. 1998. `http://sablecc.org`.

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

[9] S. S. Huang, D. Zook, and Y. Smaragdakis. Statically safe program generation with safegen. In *Generative Programming and Component Enginering (GPCE)*, volume 3676 of *LNCS*, pages 422–437. Springer-Verlag, 2005.

[10] S. Q. Lau. Domain analysis of e-commerce systems using feature-based model templates. Master's thesis, University of Waterloo, Ontario, Canada, Jan. 2006. `http://gp.uwaterloo.ca`.

[11] Object Management Group. *Meta-Object Facility*, 2002. `http://www.omg.org/technology/documents/formal/mof.htm`.

[12] Object Management Group. *Model-Driven Architecture*, 2004. `http://www.omg.org/mda`.

[13] Object Management Group. *Unified Modeling Language 2.0*, 2005. `http://www.omg.org/docs/formal/05-07-04.pdf`.

[14] OMG. *UML 2.0 OCL Specification*, 2003. `http://www.omg.org/docs/ptc/03-10-14.pdf`.

[15] M. Richters. *A Precise Approach to Validating UML Models and OCL Constraints*. PhD thesis, Universität Bremen, 2002. Logos Verlag, Berlin, BISS Monographs, No. 14.

[16] Technische Universität Dresden. *Dresden OCL Toolkit*, 2005. `http://dresden-ocl.sourceforge.net/`.

[17] T. van der Storm. Variability and component composition. In *International Conference on Software Reuse (ICSR8)*, volume 3107 of *LNCS*, pages 157–166. Springer-Verlag, 2004.

[18] J. Whaley. JavaBDD, 2003-2006. Library available at SourceForge, `http://javabdd.sourceforge.net/`.

A. Formalization of Feature-Based Model Templates

The following subsections give a formal definition of feature-based model templates, which includes feature models, annotated models, and the template instantiation process.

A.1 Feature Models

A feature model can be formally represented by the structure

$$FM = (F, E, G, \|\cdot\|, D) \qquad (7)$$

where (i) F is a finite set of features; (ii) $E \subset F^2$ is a set of edges; (iii) $G \subset (2^E \setminus \emptyset)$ is a set of feature groups represented as sets of edges; (iv) $\|\cdot\| : G \to \mathbb{Z}^2$ is a total function assigning cardinalities to groups; (v) and D is a set of additional constraints expressed as propositional formulas over F. The sets F and E form a tree with $r \in F$ as its root. All edges in a group must have the same source feature. For simplicity, we treat a solitary feature as a group of one. Consequently, every edge belongs to some group. Furthermore, all groups in G are pair-wise disjoint. Finally, for every $g \in G$, the corresponding cardinality $\|g\| = (m, n)$ has to be well-formed, i.e., $0 \leq m \leq n \leq |g|$. For example, the cardinality function for the feature model in Figure 1(a) is as follows:

$$\|g\| = \begin{cases} (1,1) & \text{if g=\{(\texttt{pi},\texttt{aa})\}} \\ (1,2) & \text{if g=\{(\texttt{aa},\texttt{i2}),(\texttt{aa},\texttt{i3})\}} \\ (0,1) & \text{otherwise} \end{cases}$$

The semantics of a feature model can be defined using an appropriate grammar or logic [2, 4, 6]. In this work, we use an interpretation of feature models using propositional logic in a similar style as defined by Batory [2]. In this interpretation, a configuration ϕ is defined as a particular assignment of *true* or *false* to every feature in F, i.e., $\phi \in \Phi$, where Φ is the set of all total functions with the signature $F \to \{\mathit{true}, \mathit{false}\}$. A configuration ϕ is *correct* iff $\phi \models q_{FM}$, i.e., q_{FM} is true for ϕ. Given a feature model

$FM = (F, E, G, \|\cdot\|, D)$, the propositional formula q_{FM} can be computed as follows:

$$q_{FM} = r \wedge q_H \wedge q_G \wedge q_D$$
$$q_H = \bigwedge_{(f_1,f_2)\in E} (f_1 \Leftarrow f_2)$$
$$q_G = \bigwedge_{\substack{g\in G\\ g=\{(f,f_1),\ldots,(f,f_k)\}\\ \|g\|=(m,n)}} (f \Rightarrow \mathit{choice}_{m,n}(f_1,\ldots,f_k))$$
$$q_D = \bigwedge_{d\in D} d$$

A.2 Model Templates

Our formalization of annotated models uses an existing formalization of class models that is given in the OCL 2.0 specification [14, Appendix A]. This formalization is based on Mark Richter's PhD thesis [15]. In that formalization, a class model[3] is represented by the structure:

$$\mathcal{M} = (\textsc{Class}, \textsc{Att}_c, \textsc{Op}_c, \textsc{Assoc}, \mathit{associates}, \mathit{roles}, \mathit{multiplicities}, \prec)$$

where (i) CLASS is a set of classes; (ii) $\textsc{Att}_c$ is a set of operation signatures $a_i : t_c \to t$ for functions mapping an object of class c to an associated attribute value with a_i being the attribute name, t_c being the OCL type corresponding to the class c, and t being the attribute type; (iii) $\textsc{Op}_c$ is a set of signatures for user-defined operations of a class c; (iv) ASSOC is a set of association names; (v) *associates* is a function mapping each association name to a list of participating classes; (vi) *roles* is a function assigning each end of an association a role name; (vii) *multiplicities* is a function assigning each end of an association a multiplicity specification; and (viii) $\prec$ is a partial order on CLASS reflecting the generalization hierarchy of classes. All these sets and functions are precisely defined in the specification [14, pp. 187–194]. Thus, a metamodel for a model template is given as $\mathcal{M}$ and a set $\mathcal{O}$ of additional well-formedness constraints expressed in OCL.

The semantics of $\mathcal{M}$ is a set of *system states* or instances of class model $\mathcal{M}$. model template because we use $\mathcal{M}$ as the metamodel for the template. A system state for a model $\mathcal{M}$ is the structure [14, p. 194]:

$$\sigma = (\sigma_{\textsc{Class}}, \sigma_{\textsc{Att}}, \sigma_{\textsc{Assoc}})$$

where (i) for every class $c \in \textsc{Class}$, the finite set $\sigma_{\textsc{Class}}(c)$ contains all objects of c existing in the system state; (ii) for each attribute $a : t_c \to t \in \textsc{Att}^*_c$, where $\textsc{Att}^*_c$ is the set of operation signatures for attribute functions for all attributes of c, including the inherited ones, the function $\sigma_{\textsc{Att}}(a) : \sigma_{\textsc{Class}}(c) \to I(t)$ assigns an attribute value (which may be $\perp$) to each object, where $I(t)$ is the interpretation of the type t; (iii) the finite sets $\sigma_{\textsc{Assoc}}$ contain *links* connecting objects. Links are instances of associations and are represented as sequences of objects. A link set must satisfy all multiplicity specifications defined for an association in $\mathcal{M}$. The complete definition of the object state structures (i–iii) is given in the specification [14, pp. 193–194].

Now we can give a precise definition of a model template, which is the structure:

$$\sigma^T = (\sigma, p)$$

where $p : \sigma^*_{\textsc{Class}} \to \mathcal{L}_F$ is a total function assigning presence conditions to template elements. In p's signature, $\sigma^*_{\textsc{Class}}$ is the set of all objects in the system state σ, i.e.,

$$\sigma^*_{\textsc{Class}} = \bigcup_{c\in\textsc{Class}} \sigma_{\textsc{Class}}(c)$$

[3] The OCL semantics specification [14, Appendix A 1] uses the term "object model" instead of "class model". However, we prefer the latter in order to be consistent with the terminology used at the UML model level.

and $\mathcal{L}_F$ is the set of all propositional formulas over F with the usual propositional operators: $\wedge$, $\vee$, $\neg$, $\Rightarrow$, and $\Leftrightarrow$, as well as $\mathit{choice}_{m,n}$.

A.3 Template Instantiation

Before we give a precise definition of template instantiation, we define $p^* : \sigma^*_{\textsc{Class}} \to \mathcal{L}_F$, which is a total function computing the accumulated presence condition for every template element:

$$p^*(\underline{c}) = p(\underline{c}) \wedge \bigwedge_{\underline{c_c}\in \mathit{wholes}(\underline{c})} p(\underline{c_c})$$

In the above definition, $\mathit{wholes}(\underline{c})$ denotes the set of all objects in $\sigma^*_{\textsc{Class}}$ that directly or indirectly contain $\underline{c}$ according to the containment associations in $\mathcal{M}$.

Figure 2(b) shows an instance of the template in Figure 2(a) for the configuration in Figure 1(b). The instance is obtained through *template instantiation* defined by the function T. Given a template σ^T and a configuration ϕ, T computes a new system state (i.e., a new instance of the metamodel) in which objects whose accumulated presence conditions are false with respect to the configuration ϕ are removed. The rationale for using the accumulated presence conditions rather than just presence conditions is that removing an element should also remove all the elements contained in it, independently of the presence condition of the contained elements. In addition to removing objects from $\sigma_{\textsc{Class}}$, T also needs to remove these objects from the links in $\sigma_{\textsc{Assoc}}$, and from the domains of the attribute functions in $\sigma_{\textsc{Att}}$. If the object removal from a link results in an object sequence with less than two objects, the entire link is also removed. More formally, T is defined as follows:

$$\begin{array}{lcl}
T((\sigma,p),\phi) & = & \sigma', \text{ where} \\
\\
\sigma & = & (\sigma_{\textsc{Class}}, \sigma_{\textsc{Att}}, \sigma_{\textsc{Assoc}}) \\
\sigma' & = & (\sigma'_{\textsc{Class}}, \sigma'_{\textsc{Att}}, \sigma'_{\textsc{Assoc}}) \\
\sigma'_{\textsc{Class}}(c) & = & \{\underline{c} \mid \underline{c} \in \sigma_{\textsc{Class}}(c) \wedge \phi \models p^*(\underline{c})\} \\
\sigma'_{\textsc{Att}}(a)(\underline{c}) & = & \sigma_{\textsc{Att}}(a)(\underline{c}) \text{ and } \sigma'_{\textsc{Att}}(a) : \sigma'_{\textsc{Class}} \to I(t) \\
\sigma'_{\textsc{Assoc}}(as) & = & \{l \mid \ \langle \underline{c_1}\ldots\underline{c_n}\rangle \in \sigma_{\textsc{Assoc}}(as) \wedge \\
 & & \quad l = \pi_{\phi\models p^*(\underline{c_i})}(\langle \underline{c_1}\ldots\underline{c_n}\rangle) \wedge \\
 & & \quad |\,l| > 1 \qquad\qquad\qquad\qquad \}
\end{array}$$

In the above definition, $\pi_{\phi\models p^*(\underline{c_i})}(\langle \underline{c_1}\ldots\underline{c_n}\rangle)$ projects all components $\underline{c_i}$, $i \in \{1,\ldots,n\}$, of the sequence $\langle \underline{c_1}\ldots\underline{c_n}\rangle$ for which $\phi \models p^*(\underline{c_i})$.

The result of template instantiation may not be well-formed with respect to the target notation. In particular, the result may violate the multiplicities of the metamodel and/or the additional OCL constraints in $\mathcal{O}$. Although reducing $\sigma_{\textsc{Class}}$ and the domain of $\sigma_{\textsc{Att}}$ will result in $\sigma'_{\textsc{Class}}$ and $\sigma'_{\textsc{Att}}$ that satisfy the class model of $\mathcal{M}$, removing objects from the links in $\sigma_{\textsc{Assoc}}$ may result in $\sigma'_{\textsc{Assoc}}$ that violates the multiplicities in $\mathcal{M}$.

B. Template Interpretation of OCL

Before giving the template interpretation of OCL, we need to briefly discuss how the standard interpretation of OCL expressions is formally defined in the OCL specification [14, Appendix A.3.1]: An evaluation context for an expression is given by an environment $\tau = (\sigma, \beta)$ consisting of a system state σ and a variable assignment $\beta : Var_t \to I(t)$, where Var_t is the set of variables of type t, and $I(t)$ denotes the interpretation of type t, e.g., $I(\mathit{Boolean}) = \{\mathit{true}, \mathit{false}, \perp\}$. The system state σ provides access to the set of currently existing objects, their attribute values, and association links between objects. The variable assignment β maps names of OCL variables to values. Given the set of all environments *Env*, $\tau \in \mathit{Env}$, the standard semantics of an OCL expression is given by the interpretation function $I[\![e]\!] : \mathit{Env} \to I(t)$, which is precisely defined in the OCL specification [14, Definition A.30]. Furthermore, the specification defines the interpretation function I for all OCL types and their operations.

Our proposed template interpretation of an OCL expression e is given by the new interpretation function $I^T[\![e]\!] : \mathit{Env}^T \to I^T(t)$. The evaluation context for an expression is now given by a *template environment* $\tau^T = (\sigma^T, \beta^T)$ consisting of a template $\sigma^T = (\sigma, pc)$, and a variable assignment $\beta^T : Var_t \to I^T(t)$, where the function I^T gives the template interpretation of types and their operations. We first discuss how I^T is defined before giving the definition for $I^T[\![e]\!]$.

The template interpretation $I^T(t)$ of a type t has to take into account that it needs to represent a collection of possible results for an expression evaluation rather than just a single result. For non-collection types, i.e., the basic types *Integer*, *Real*, *Boolean*, and *String*, as well as *Tuple* and object types, we need to represent a set of *alternative* values since for a given configuration ϕ one and only one alternative has to be selected. For the collection types *Set*(t) and *Bag*(t), we need to represent a collection of the same kind, i.e., a set or a bag, respectively, but containing values for *multiple selections* rather than alternatives since a subset of the values needs be selected for a given ϕ. Such a representation is more efficient than a set of alternative collections. Consequently, for a non-collection type t, we define

$$I^T(t) \quad = \quad \mathcal{F}(I(t) \times \mathcal{L}_F)$$

where $\mathcal{F}(S)$ denotes the set of all finite subsets of a given set S. For example, since $I(\mathit{Boolean}) = \{\mathit{true}, \mathit{false}, \perp\}$, $I^T(\mathit{Boolean}) = \mathcal{F}(\{\mathit{true}, \mathit{false}, \perp\} \times \mathcal{L}_F)$. Since $I^T(t)$ is a set of alternatives, we additionally require that for every $\{(v_1, p_1), \ldots, (v_n, p_n)\} \in I^T(t)$, where t is a non-collection type, the propositional formula $\mathit{choice}_{1,1}(p_1, \ldots, p_n)$ is valid. For the collection types *Set*(t) and *Bag*(t), we have the following definitions:[4]

$$\begin{aligned} I^T(\mathit{Set}(t)) &= \mathcal{F}(I(t) \times \mathcal{L}_F) \\ I^T(\mathit{Bag}(t)) &= \mathcal{B}(I(t) \times \mathcal{L}_F) \end{aligned}$$

where $\mathcal{B}(S)$ denotes the set of all finite multisets over S and $I(t)$ is the standard OCL interpretation of t.

The template interpretation of an operation $\omega : t_1, \ldots, t_n \to t$ is given by the function $I^T(\omega) : I^T(t_1), \ldots, I^T(t_n) \to I^T(t)$, with $n \geq 0$.

For a unary operation ω, such as $-$, *abs*, *floor*, *round*, *not*, and the string operations *size*, *toUpper*, and *toLower*, the template interpretation is defined according to the following schema:

$$\begin{gathered} I^T(\omega)(\{(x_1, p_1), \ldots, (x_n, p_n)\}) = \\ \{(I(\omega)(x_1), p_1), \ldots, (I(\omega)(x_n), p_n)\} \end{gathered}$$

For binary operations, such as $+$, $-$, $*$, $/$, *min*, *max*, $<$, $>$, $\leq$, $\geq$, *concat*, *and*, *or*, *xor*, and *implies*, the corresponding schema is as follows:

$$I^T(\omega)(\{(x_1, p_{x_1}), \ldots, (x_m, p_{x_m})\}, \{(y_1, p_{y_1}), \ldots, (y_n, p_{y_n})\}) =$$

$$\begin{array}{ccccc} \{ & (I(\omega)(x_1, y_1), p_{x_1} \wedge p_{y_1}), & \ldots & (I(\omega)(x_1, y_n), p_{x_1} \wedge p_{y_n}), & \\ & \vdots & \vdots & \vdots & \\ & (I(\omega)(x_m, y_1), p_{x_m} \wedge p_{y_1}), & \ldots & (I(\omega)(x_m, y_n), p_{x_m} \wedge p_{y_n}) & \} \end{array}$$

For nullary operations, such as *12* for *Integer* or *"foo"* for *String*, we have the following schema:

$$I^T(\omega)() = \{(I(\omega)(), \mathit{true})\}$$

Template semantics for operations on collection types are more complex and varied, but still relatively straightforward to define. Here we only show some examples. The operation *size* : *Set*$(t) \to$ *Integer* computes the size of a set. Its template interpretation is defined as follows:

$$\begin{aligned} & I^T(\mathit{size})(\{(x_1, p_1), \ldots, (x_n, p_n)\}) = \\ & \qquad \{(0, \mathit{choice}_{0,0}(p_1, \ldots, p_n)), \ldots, (n, \mathit{choice}_{n,n}(p_1, \ldots, p_n))\} \end{aligned}$$

[4] It is important to note that every OCL type t includes $\perp$ in the standard OCL interpretation, i.e., $\perp \in I(t)$. While the template interpretation of non-collection types takes this fact into account, for simplicity, we do not consider $\perp$ for template interpretation of collections. Also, in practice, $\perp$ is less useful for collections since an empty collection can be used instead.

Here is the template interpretation of *isEmpty* : *Set*$(t) \to$ *Boolean*:

$$I^T(\mathit{isEmpty})(s) =$$

$$\begin{cases} \{(\mathit{true}, \mathit{true})\} & \text{if } s = \emptyset \\ \{(\mathit{true}, \neg p_1 \wedge \ldots \wedge \neg p_n), & \\ \quad (\mathit{false}, p_1 \vee \ldots \vee p_n)\} & \text{if } s = \{(x_1, p_1), \ldots, (x_n, p_n)\} \wedge n \geq 1 \end{cases}$$

Let us take a look at two examples of operations with the signature ω : *Set*$(t) \times$ *Set*$(t) \to$ *Set*(t). The operation *union* has a straightforward definition:

$$I^T(\mathit{union})(s_1, s_2) = s_1 \cup s_2$$

However, *intersection* is a bit more complex:

$$\begin{gathered} I^T(\mathit{intersection})(s_1, s_2) = \\ \{(x, p) \mid \forall (x_1, p_1) \in s_1 : \forall (x_2, p_2) \in s_2 : \\ x_1 = x_2 \wedge x = x_1 \wedge p = (p_1 \wedge p_2)\} \end{gathered}$$

The template interpretation of OCL expressions is similar to the standard interpretation given in Definition A.30, items *i.–iv.*, in the OCL specification. The items *i.–vii.* from that definition are carried over to the template interpretation. For example, the template interpretation of variables (corresponds to item *i.*) is as follows:

$$I^T[\![v]\!](\tau^T) = \beta^T(v)$$

However, *if-then-else* (item *v.*) is defined differently. Here we only give the definition for the case when e_2 and e_3 are of non-collection types:

$$\begin{gathered} I^T[\![\mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3\ \mathit{endif}\,]\!](\tau^T) = \\ \{(x, p) \mid \forall (x, p_2) \in I^T[\![e_2]\!](\tau^T) : \forall p_t : (\mathit{true}, p_t) \in I^T[\![e_1]\!](\tau^T) : \\ p = (p_t \wedge p_2)\} \cup \\ \{(x, p) \mid \forall (x, p_3) \in I^T[\![e_3]\!](\tau^T) : \forall p_f : (\mathit{false}, p_f) \in I^T[\![e_1]\!](\tau^T) : \\ p = (p_f \wedge p_3)\} \cup \\ \{(x, p_\perp) \in I^T[\![e_1]\!](\tau^T) \mid x = \perp\} \end{gathered}$$

The item *vi.* remains unchanged except for replacing *true* and *false* by $I^T(\mathit{true})$ and $I^T(\mathit{false})$, respectively. Finally, the definition of *iterate* (item *vii.*) needs a small change in the environment definition, which we do not further explain here. Although iteration operations on collections such as *forAll* and *collect* can be expressed using *iterate*, more efficient, direct definitions can be given. For example, *forAll* can be defined as follows:

$$I^T[\![e_1\texttt{->forAll(}v\,|\,e_2\texttt{)}]\!](\tau^T) =$$

$$\begin{cases} \{(\mathit{true}, \mathit{true})\} & \text{if} \quad I^T[\![e_1]\!](\tau^T) = \emptyset \\ & \text{if} \quad I^T[\![e_1]\!](\tau^T) = \\ & \quad \{(x_1, p_1), \ldots, (x_n, p_n)\} \wedge \\ \{(\mathit{true}, p_1 \Rightarrow p_{e_2(x_1)=t} \wedge & \quad \forall i = 1, \ldots, x_n \\ \ldots \wedge p_n \Rightarrow p_{e_2(x_n)=t}), & \quad \mathit{Rdx}(I^T[\![e_2]\!]((\sigma^T, \\ (\mathit{false}, p_1 \Rightarrow p_{e_2(x_1)=f} \wedge & \quad \beta^T\{v/\{(x_i, true)\}\}))) = \\ \ldots \wedge p_n \Rightarrow p_{e_2(x_n)=f}), & \quad \{(\mathit{true}, p_{e_2(x_i)=t}), \\ (\perp, p_1 \Rightarrow p_{e_2(x_1)=\perp} \wedge & \quad (\mathit{false}, p_{e_2(x_i)=f}), \\ \ldots \wedge p_n \Rightarrow p_{e_2(x_n)=\perp})\} & \quad (\perp, p_{e_2(x_i)=\perp})\} \end{cases}$$
$$\}$$

where (i) $(\sigma^T, \beta^T\{v/\{(x_i, true)\}\})$ represents the environment τ^T updated with a variable binding from v to $\{(x_i, true)\}$ and (ii) $p_{e_2(x_i)=t}$ denotes the condition under which e_2 evaluates to true for x_i. Furthermore, (iii) the function *Rdx* : $I^T(t) \to I^T(t)$, where t is a non-collection type, takes a set of alternatives and returns an equivalent set of alternatives in which pairs with the same value were merged, e.g.:

$$\mathit{Rdx}(\{(1, p_1), (1, p_2), (2, p_3)\}) = \{(1, p_1 \vee p_2), (2, p_3)\}$$

Now we can give the precise definition of step one of our verification approach from Section 4.2. Given the template σ^T and object $\underline{c}$, the template semantics is given as

$$\mathit{Rdx}(I^T[\![e]\!])(\tau^T)$$

where $\tau^T = (\sigma^T, \{\mathit{self}/\{(\underline{c}, \mathit{true})\}\})$.