Constraint handling in Feature Models

László Lengyel, Tihamér Levendovszky, Hassan Charaf

Budapest University of Technology and Economics, Magyar tudósok körútja. 2.
H-1111 Budapest, Hungary

lengyel@aut.bme.hu, tihamer@aut.bme.hu, hassan@aut.bme.hu

Abstract: Model transformation means converting an input model available at the beginning of the transformation process to an output model. The paper addresses the expressiveness issues of the graph rewriting-based topological model transformation. The problem and its solution are discussed in this work, illustrated by a case study from the field of Generative Programming. It is shown how metamodel-based graph rewriting method extended with constraints can be applied to transform software models. These constraints are specified on the meta-layer and impose restrictions on the models on the instance layer of the rules. Dealing with these constraints facilitates a solution for the unsolved issues, because topological and attribute transformation methods cannot perform and express constraint validation. Handling constraints can support properties to guarantee, preserve or validate visual model processors, and the presented approach is a practical application of these mechanisms. The work presents the relation between the constraints and the pre- and postconditions as well.

Keywords: Constraints, Constraint Validation, Feature Modeling, OCL, VMTS

1 Introduction

OMG's Model Driven Architecture [1] offers a standardized framework to separate the essential, platform independent information from the platform dependent constructs and assumptions. A complete MDA application consists of a definitive platform-independent model (PIM), and one or more platform-specific models (PSM) and complete implementations, one on each platform that the application developer decides to support. The platform independent artifacts are mainly UML and other software models containing enough specification to automatically generate the platform dependent artifacts by so-called model compilers. Hence software model transformation providing a basis for model compilers in general lies at the heart of the MDA architecture.

To illustrate the techniques of constraint validation a case study is provided from the field of generative programming. It is applied in the fields where the target

domain of the software development can be treated, conceived and captured as one model set with optional, exclusive and inclusive construct. The first step of capturing the whole domain is the method called feature modeling [2]. Instead of addressing only the part of the domain which is directly relevant to the application to be developed, we model the whole domain for a forthcoming set of application which are generated on demand. This model contains all the possible result of the engineering process thus including the reusability design of the application set. The next steps are to create the generators and the software components for the generators. On the one hand we need large effort to create generators, but the software creation with generators is flexible and safe method [3] on the other hand.

Software modeling is a synonym for producing diagrams. Most models consist of a number of "nodes and edges" pictures and some accompanying text. The information conveyed by such a model has a tendency to be incomplete, imprecise, and sometimes even inconsistent. A feature diagram or a UML diagram, such as a class diagram, is typically not refined enough to provide all the relevant aspects of a specification. There is, among other things, a need to describe additional constraints about the objects in the model. Such constraints are often described in natural language. Practice has shown that this will always result in ambiguities. In order to write unambiguous constraints, so-called formal languages have been developed. The disadvantage of traditional formal languages is that they are usable to persons with a strong mathematical background, but difficult for the average business or system modeler to use. Object Constraint Language (OCL) [4] is a formal language that remains easy to read and write.

To fully specify models and generators we assign constraints to model elements and to the steps accomplished by generators. The help of these constraints we get precise and consistent feature models and generator steps.
The rest of this paper is organized as follows: in the next section the backgrounds and the motivation are described, which is followed by the contributions: (i) we shortly introduce our implementation, the VMTS [5] [6], then (ii) the concepts of constraint validation is presented, and (iii) the feature model normalization with the help of constraints is discussed. Finally conclusions and future work are delineated.

2 Backgrounds and Motivation

Feature modeling is particularly important if one engineers for reuse. The reason is that reusable software contains inherently more variability than concrete applications and feature modeling is the key technique for identifying and capturing variability. Feature modeling helps us to avoid two serious problems: (i) relevant features and variation points are not included in the reusable software,

(ii) many features and variation points are included but never used and thus cause unnecessary complexity, development cost, and maintenance cost.

Besides this, the feature models produced during the feature modeling activities provide us with an abstract (since implementation independent), concise, and explicit representation of the variability present in the software.

Following the conceptual modeling perspective, a feature is an important property of a concept. Features allow us to express the commonalities and differences between concept instances. They are fundamental to formulating concise descriptions of concepts with large degrees of variation among their instances. Organized in feature diagrams, they express the configurability aspect of concepts. Features are primarily used in order to discriminate between instances (and thus between choices). In this context, the quality of a feature is related to properties such as its primitiveness, generality, and independency. Feature modeling is the activity of modeling the common and the variable properties of concepts and their interdependencies and organizing them into a coherent model referred to as a feature model. It is important to note that feature modeling is a creative activity. It is much more than just a simple rehash of the features of existing systems and the available domain knowledge. For example, one technique used in feature modeling is the analysis of combinations of variable features, which may lead to the discovery of innovative feature combinations and new features. The systematic organization of existing knowledge allows us to invent new, useful features and feature combinations more easily.

Often we need to specify a model more precisely than a modeling language enables it. It is a common case that we want to define expressions and constraints on our model. The Object Constraint Language (OCL) [4] is a formal language for analysis and design of software systems. It is a subset of the industry standard Unified Modeling Language (UML) [7] that allows software developers to write constraints and queries over object models. A constraint is a restriction on one or more values of an object-oriented model or system. There are four types of constraints: (i) An invariant is a constraint that states a condition that must always be met by all instances of the class, type, or interface. (ii) A precondition to an operation is a restriction that must be true at the moment that the operation is going to be executed. The obligations are specified by postconditions. (iii) A postcondition to an operation is a restriction that must be true at the moment that the operation has just ended its execution. (iv) A guard is a constraint that must be true before a state transition fires. Besides these, OCL can be used as a navigation language as well.

Graph rewriting [8] is a powerful tool for graph transformations with strong mathematical background. The atoms of graph transformation are rewriting rules, each rewriting rule consists of a left hand side graph (LHS) and right hand side graph (RHS). Applying a graph rewriting rule means finding an isomorphic occurrence (match) of the LHS in the graph the rule being applied to (host graph),

and replacing this subgraph with RHS. Replacing means removing elements which are in the LHS but not in the RHS, and gluing elements which are in the RHS but not in the LHS.

The motivation of this research was to work out a method that facilitates to assign constraints to software models (i) which have an affect on the model transformation, and (ii) specifies more precisely the source code and application generation based on these models achieved by generators. For constraint specification we choused the OCL. The paper introduces the constraint handling via an illustrative case study.

3 Contribution

Our implementation is called Visual Modeling and Transformation System (VMTS) [5] [6], which is an n-layer multipurpose modeling and metamodel-based transformation system. The system architecture of the VMTS is depicted in Figure 1. The user interfaces (Adaptive Modeler, Rule Editor) are functionally separated from the model storage unit (AGSI Core - Attributed Graph Architecture Supporting Inheritance) which uses an RDBMS to store the model information. The model transformation can be accomplished by Traversing Model Processors [9], Rewriting Engine and Other Applications. The AGSI Core exposes its interface to any other applications which may use other technique to process AGSI data. Using this environment it is easy to edit metamodels, design models according to their metamodels, transform models using graph rewriting [3] [6] [9], and validate metamodel and rewriting rule containing constraints. The main contribution of this section is to present the constraint validation process.

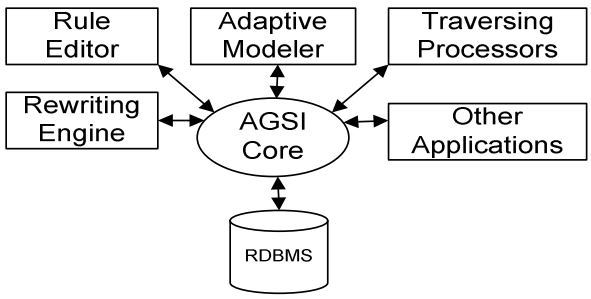

Figure 1. VMTS System Architecture

3.1 Constraint validation

In this section we introduce the relation between the pre- and postconditions as well as the OCL constraints assigned to the rewriting rules. Besides this the type of the constraints we enlist in the LHS and RHS graphs of the rewriting rule is also presented.

A precondition (postcondition) assigned to a rewriting rule is a boolean expression that must be true at the moment when the rewriting rule is fired (after the completion of a rewriting rule). If a precondition of a rewriting rule is not true then the rewriting rule fails without being fired. If a postcondition of a rewriting rule is not true after the execution of the rewriting rule then the rewriting rule fails. A direct corollary of this is that an OCL expression in LHS is a precondition to the rewriting rule, and an OCL expression in RHS is a postcondition to the rewriting rule. A rewriting rule can be fired if and only if all conditions enlisted in LHS are true. Also, if a rewriting rule finished successfully then all conditions enlisted in RHS must be true.

There are three properties: *validation*, *preservation*, and *guarantee*, which are checked during the rewriting process. A transformation step *S validates* a property *P*, when the following condition always holds: if a property *P* was true before the step *S* it remains true after the execution of the step *S*, and if *P* is false, the step *S* fails. A step *S preserves* a property *P*, when the following condition always holds: if a property *P* was false (true) before the step *S* it remains false (true) after the execution of the step *S*. A transformation step *S guarantees* a property *P*, when the following condition always holds: if a property *P* was true before the step *S* it remains true after the execution of the step *S*, and if *P* is false, the step *S* changes property *P* to true. Table 1 summarizes the meaning of these properties.

	property *P* before the step *S*	**property *P* after the step *S***
Validation	true	true
	false	step *S* fails
Preservation	true	true
	false	false
Guarantee	true	true
	false	true

Table 1. Truth table of the validation, preservation and guarantee properties

Based on these properties we can introduce the concepts of the general validation, general preservation and general guarantee. (i) A step *S* validates a property *P* for an input model *M* if the property *P* is enlisted in both pre- and postconditions of the step *S*, and the step *S* has been executed successfully for the model *M*. (ii) A step *S* preserves a property *P* for an input model *M* if the expression *(NOT P or P@pre) and (P or NOT P@pre)* is enlisted as an OCL expression in postconditions of the step *S*, and the step *S* has been executed successfully for the model *M*. (Where *x@pre* means the value of *x* immediately before the step was fired even if the value of *x* has not changed.) (iii) A step *S* guarantees a property *P* for an input model *M* if the property *P* is enlisted in postconditions of the step *S*, and the step *S* has been executed successfully for the model *M*.

3.2 Feature model normalization with the help of constraints

Our case study is a part of the feature model driven scenario, namely, to transform feature models in a canonical and application specific form, such that it can be fed to the generator directly. Here we want to illustrate how VMTS validates the constraints during the transformation.

The metamodel of feature diagram is shown in Figure 2. This metamodel enforces the followings: a node can be a root node (so-called concept) or a child. A node can have either XOR (alternative) or OR connector (denoted by arcs). Either via these connectors or directly a child can connect to a node of any type (Root or Child).

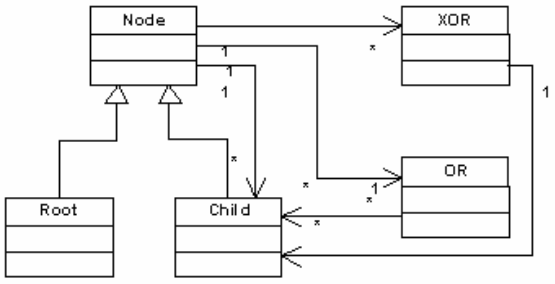

Figure 2. Feature diagram metamodel

A feature model represents the common and the variable features of concept instances and the dependencies between the variable features. We distinguish between *mandatory*, *alternative (xor)*, and *optional features* [2]. In addition to these feature types, we also introduce *or-features*. Furthermore, optionality can be combined with alternative features and with or-features resulting in the two additional feature types *optional alternative features* and *optional or-features*. However, as it is illustrated later, the optional or-feature type is equivalent to the

optional feature type and thus it is redundant. As an example Figure 3 presents a feature model of a car.

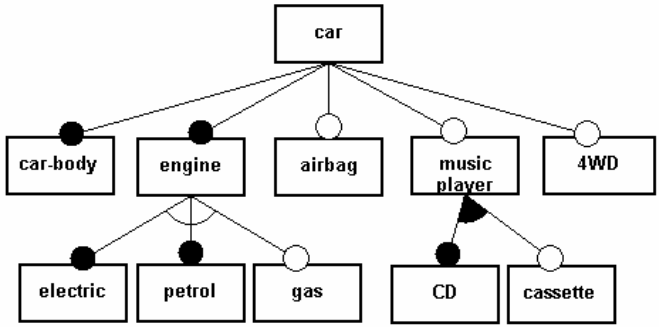

Figure 3. Feature model of a car

A *mandatory feature* is included in the description of a concept instance if and only if its parent is included in the description of the concept. In Figure 3 the *car-body* and *engine* are mandatory features – every car has car-body and engine. An *optional feature* may be included in the description of a concept instance if and only if its parent is included in the description. *Music player, airbag* and *4WD (four wheel drive)* are optional features – in a car there is a music player or not. A concept may have one or more sets of direct *alternative features* (*xor-features*). *Electric, petrol* and *gas* are xor-features. If the parent of a set of alternative features is included in the description of a concept instance, then exactly one feature from this set of xor-features is included in the description. A concept may have one or more sets of direct *or-features*. *CD* and *cassette* are or-features. If the parent of a set of or-features is included in the description of a concept instance, then any non-empty subset from the set of or-features is included in the description. Figure 3 represents that a car has a CD player or a cassette player or both of them.

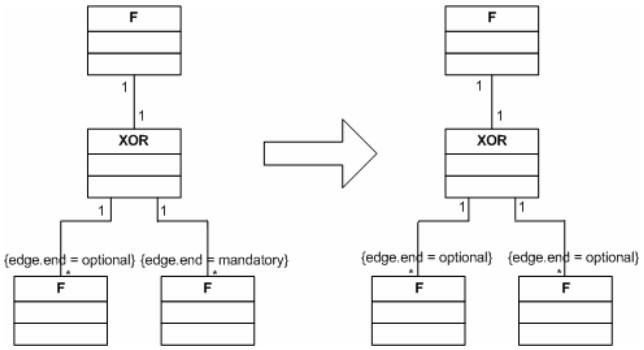

Figure 4. Normalization step 1: If one or more of the features in a set of xor-features are optional, this has the same effect as if all the alternative features in this set were optional.

A node in a feature diagram can have mandatory feature subnodes, optional feature subnodes, xor-feature subnodes, optional xor-feature subnodes, or-feature subnodes, and optional or-feature subnodes. During the feature model normalization we examine the *optional xor-feature nodes* and *optional or-feature nodes*.

The Figure 4 reveals the first normalization steps, it describes that a feature diagram with one optional xor-feature can be normalized into a diagram with any optional xor-features. This rewriting rule does not change the topology, but updates the feature attributes.

The following constraint-pair facilitates that optional xor-features do not alter during the normalization. This constraint-pair describes a *validation* property. The first part of the constraint-pair is the precondition, it is enlisted in the LHS graph, and the second part of the constraint-pair is the postcondition, and it is enlisted in the RHS graph:

```
context edge inv norm_const_1_LHS:
self.end = optional and self.sourceFeature.type = 'xor-feature'

context edge inv norm_const_1_RHS:
self.end = optional and self.sourceFeature.type = 'xor-feature'
```

Mandatory xor-features are normalized into optional xor-features, this is a *guarantee* property:

```
context edge inv norm_const_2_LHS:
self.end = mandatory and self.sourceFeature.type = 'xor-feature'

context edge inv norm_const_2_RHS:
self.end = optional and self.sourceFeature.type = 'xor-feature'
```

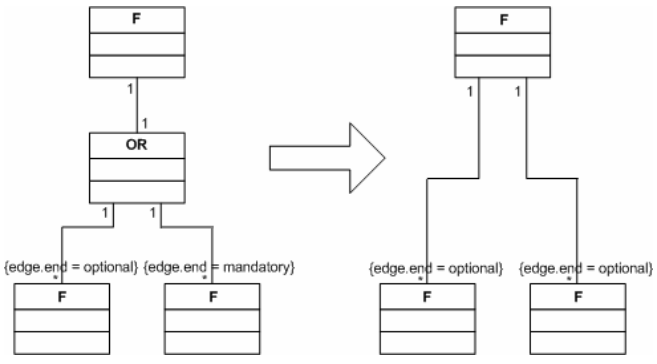

Figure 5. Normalization step 2: A feature with at least one optional child or-feature can be normalized into optional child features.

Figure 5 presents the second normalization step: if one or more of the features in a set of or-features is optional, it has the same effect as if all the features in this set were optional or-features. Therefore, if one or more features in a set of or-features is optional, we can replace all these features by optional features. In conclusion the category of optional or-features is redundant since it is equivalent to optional features.

Optional or-features are normalized into optional features, this is a *guarantee* property:

```
context edge inv norm_const_3_LHS:
self.end = optional and self.sourceFeature.type = 'or-feature'

context edge inv norm_const_3_RHS:
self.end = optional and self.sourceFeature.type = 'feature'
```

Mandatory or-features normalizes into optional features, this is a *guarantee* property:

```
context edge inv norm_const_4_LHS:
self.end = mandatory and self.sourceFeature.type = 'or-feature'

context edge inv norm_const_4_RHS:
self.end = optional and self.sourceFeature.type = 'feature'
```

Any feature diagram can be transformed into a feature diagram which does not have any optional or-features and whose sets of alternative features may contain either only alternative features or only alternative optional features. The transformation can be accomplished by the presented normalization steps. The resulting feature diagram is a *normalized feature diagram,* which is equivalent to the original feature diagram. Figure 6 presents the normalized feature model of the car depicted in Figure 3.

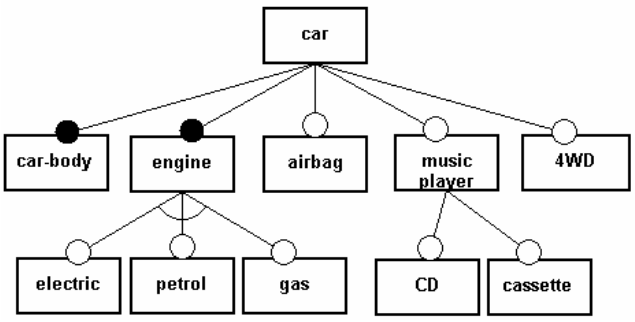

Figure 6. Normalized feature model of a car

In Figure 4 and Figure 5 the normalization steps are rewriting rules built from the metamodel elements presented in Figure 2.

3.3 Constraint handling

Feature models contain not only mandatory and variable features, but also dependencies between variable features. These dependencies are expressed in the form of constraints and default dependency rules. Constraints specify valid and invalid feature combinations. Default dependency rules suggest default values for unspecified parameters based on other parameters.

Constraints and default dependency rules allow us to implement automatic configuration. For example, in addition to our feature diagram of a car (Figure 3), we could also have an extra feature diagram defining the three high-level alternative features of a car: *classic*, *sport*, and *overland*. Furthermore we could have the following vertical default dependency rules relating the three high level features and the variable detail features from: (i) classic implies electric engine and cassette music player; (ii) sport implies petrol engine, CD music player and airbag; (iii) 4WD implies petrol engine, CD music player and 4WD. An example constraint that implies that if a car is *overland* then contains 4WD (otherwise it does not) is the following:

```
context car inv car_type_overland:
if car.type = 'overland' then
        car.is4WD = true
else
        car.is4WD = false
endif
```

Given these default dependency rules, we can specify a car with all extras as follows: *overland and airbag*.

Our approach, VMTS does not interpret the constraints, but it automatically generates source code based on the constraints and compiles it to a binary which validates the metamodel and rewriting rule containing constraints. The LHS graph of the first normalization step (rewriting rule) in Figure 4 contains the following constraint:

```
context XOR_F3_Edge inv endCheckLHS:
self.end = mandatory
```

The constraint in the RHS graph:

```
context XOR_F3_Edge inv endCheckRHS:
self.end = optional
```

Based on the constraints the aim of the rewriting rule (normalization step) is to change the value of end properties. VMTS Compiler generates checking code based on these OCL constraints and compiles it to a binary. The VMTS Validation

Module checks the matched subgraph against the LHS graph containing constraints, and checks the result of the rewriting by the RHS graph containing constraints. Code list 1 contains a part of the generated C# code which validates the above presented, RHS graph containing constraint.

```
public class XOR_F3_Edge : OclRuntime.UML_OCL.OCLOclAny,
OclRuntime.ConstraintChecker {
        public virtual string TypeName {
            get { return "XOR_F3_Edge";}
        }
        public virtual OclRuntime.OclResult
CheckConstraint_INV_endCheckRHS(OclRuntime.OCLModelType self) {
            OclRuntime.OclResult _res = new
OclRuntime.OclResult("XOR_F3_Edge", "endCheckRHS", self.InstanceId);
            _res.Result =
((bool)(((OclRuntime.UML_OCL.OCLBoolean)(new
OclExpression_2(self).Value))));
            return _res;
        }
        public class OclExpression_2 :
OclRuntime.UML_OCL.OCLOclExpression {
            private OclRuntime.OCLModelType self;
            public OclExpression_2(OclRuntime.OCLModelType self) {
                this.self = self;
            }
            protected override OclRuntime.UML_OCL.OCLOclAny
GetEvaluatedValue() {
                return
((OclRuntime.UML_OCL.OCLString)(self.GetPropertyValue("end"))).opGT(n
ew OclRuntime.UML_OCL.OCLString("optional"));
            }
        }
 }
```

Code list 1. Example generated validation code

Conclusions and Future work

In this paper a summary of the constraint handling in feature models is provided. It has been shown that metamodel-based graph rewriting method can be applied to normalize feature diagrams. Without dealing with OCL constraints this problem could not have been solved, because topological and attribute transformation methods cannot perform and express the type of the constraint validation demanded by the normalization process. Besides this the paper discussed the relation between the constraints and the pre- and postconditions, and the validation of rewriting rule containing constraints during the graph transformation process.

One of the most important part of the method that our constraint checking approach does not interpret the constraints; we generate source code and compile it to a binary which validates the metamodel and rewriting rule containing constraints.

Future work includes the optimization of the current implementation, and the design and implementation of branch conditions. With the help of branch conditions VMTS will support that a rewriting rules can have not only one but optional number of RHS graphs. The result of a LHS graph containing constraint checking will decide which RHS graph will be used. It is similar to a *switch-case* structure where a constraint plays the role of the *switch* and RHS graphs are the *case* branches. Furthermore we will implement a module which handles constraint-pairs. The module will be able to determine the property type (*validation*, *preservation*, and *guarantee*) of a constraint-pair, and it will help if one would like to write a constraint-pair with arbitrary property type.

References

[1] MDA Guide Version 1.0.1, OMG, doc. number: omg/2003-06-01, 12th June 2003 www.omg.org/docs/omg/03-06-01.pdf

[2] Czarnecki et al.: Generative programming: methods, tools, and applications (Addison-Wesley, 2000)

[3] Levendovszky T., Lengyel L., Charaf H., Software Composition with a Multipurpose Modeling and Model Transformation Framework, IASTED 2004, Innsbruck, 2004, pp.590-594

[4] Object Constraint Language Spec. (OCL), www.omg.org

[5] Visual Modeling and Transformation System Web Site http://avalon.aut.bme.hu/~tihamer/research/vmts/.

[6] Levendovszky T., Lengyel L., Mezei G., Charaf H., A Systematic Approach to Metamodeling Environments and Model Transformation Systems in VMTS, International Workshop on Graph-Based Tools (GraBaTs) Electronic Notes in Theoretical Computer Science, Rome, 2004

[7] UML 2.0 Specifications, http://www.omg.org/uml/

[8] G. Rozenberg (ed.), Handbook on Graph Grammars and Computing by Graph Transformation: Foundations, Vol.1 World Scientific, Singapore, 1997.

[9] Levendovszky T., Lengyel L., Charaf H., Implementing a Metamodel-Based Model Transformation System, Buletinul Stiintific al Universitatii "Politehnica" din Timisoara, ROMANIA Seria AUTOMATICA si CALCULATOARE PERIODICA POLITECHNICA, Transactions on AUTOMATIC CONTROL and COMPUTER SCIENCE Vol.49 (63), 2004, ISSN 1224-600X