Modularized Constraint Management in Model Transformation Frameworks

László Lengyel

Department of Automation and Applied Informatics
Budapest University of Technology and Economics
Magyar tudósok körútja 2, H-1117 Budapest, Hungary
lengyel@aut.bme.com

Abstract: Model-based development methods are increasingly being applied in the production of software artifacts. The processing of visual models, within these frameworks, is an essential issue that can be addressed using graph rewriting techniques. The precise definition of graph rewriting-based model transformation requires that beyond the topology of the rules, further textual constraints be added. These constraints often appear repetitively in a transformation; therefore, constraint concerns crosscut the transformation. It is useful to define often applied constraints as physically separated modules and indicate the places where to use them. This effort provides solutions to structuring, modularizing and propagating repetitively occurring and crosscutting constraints. We propose an aspect-oriented approach that allows for consistent constraint management, in which repetitive and crosscutting constraints can be semi-automatically identified.

Keywords: Aspect-oriented constraints; Constraint aspects; Constraint modularization; Graph rewriting

1 Introduction

Model-based software development [13] [18] applies different software models during system development. Model-based approaches highlight the relevance of model-driven methods in the software industry. They facilitate defining the applications with software models and automatically transform them into executable artifacts.

Model transformations appear in various situations in application development [2]. Graph rewriting is a widely utilized technique for model transformation [8] [9] [19]. Model transformations, like all software, must be validated to ensure their usefulness for each intended application. In [10] [11], an approach has been introduced for validating model transformation that applies Object Constraint Language (OCL) [14]. Constraints are the pre- and post-conditions of transformation rules. OCL as a constraint and query language in software modeling is an effective way to define textual constraints [3] [5]. We have already demonstrated that it can also be utilized in model transformation definitions [15].

Often we require the validation of several rules or whole transformations, which may cause the same constraint concerns to appear numerous times in a transformation. Regarding this recurrence of constraint concerns, it is beneficial to distinguish between the classical constraint repetition and the crosscutting constraints. According to [17], the definition for the term *concern* is "any matter of interest in a software system".

The classical constraint repetition is similar to the frequently appearing lines of program code in a source file (also known as code clones). In the source code domain, this problem is handled with program segmentation. In most cases, it is implemented with functions; the recurring lines of source code are placed into a function and the function is then called from the appropriate position. This method can be applied to model transformation constraints as well. This can be achieved by extracting the repetitive constraints into separated components and, similarly to function calls, manually designating the points in the model transformation in which they will be applied.

Regarding crosscutting concerns, the situation is significantly different. As opposed to repetitions, crosscutting concerns of a design cannot be modularly separated. If a concern attempts to decompose, according to a specified design principle, other concerns will crosscut this decomposition. This implies that crosscutting is relative to each particular decomposition.

To summarize crosscutting concerns, there is no way to achieve a modular design. In the case of repetitive constraints, consistent constraint management is difficult. In order to mitigate these issues, our aim is to physically separate the different concerns, namely the structure of the transformation rules and constraints, and design them separately. Next, using a weaving mechanism, we generate the executable artifact that combines the two concerns. This generated representation, containing both repetitive and crosscutting constraints, is similar to a binary file compiled from source code and is not edited by the transformation engineer. Therefore, no problems arise, despite the generated artifact concerns not being separated.

The approach presented in this paper provides solutions for both repetitive and crosscutting constraints. Our previous works [10] [12] have already introduced the problem of crosscutting constraints in model transformations. In order for this paper to be self-contained, we briefly summarize the constructs and methods we have developed for crosscutting constraint management in model transformations. The novel results provided by this paper are: (i) the distinction of repetitive and crosscutting constraints in model transformations, (ii) the mechanism that handles the repetitive constraints and (iii) a generalized, semi-automatic identification of repetitive and crosscutting constraints.

With the help of a case study, the next section introduces the problem of repetitive and crosscutting constraints in model transformations. Section 3 gives background information about our model transformation framework and introduces the approach developed for managing crosscutting constraints. In Section 4, we identify the difference between repetitive and crosscutting constraints and discuss the handling of repetitive constraints in model transformations. Section 5 provides a generalized method for semi-automatic detection of repetitive and crosscutting constraints. Finally, concluding remarks are elaborated.

2 Constraint Management Problems in Model Transformations

Graph rewriting [16] is a widely applied technique for graph transformation. The basic elements of graph transformations are graph rewriting rules. Each rule consists of a left-hand side graph (LHS) and right-hand side graph (RHS). Initially, performing a rule requires locating an occurrence (match) in which the rule is applied on the LHS of a graph and replacing this pattern with the RHS. In most model transformation tools, the LHS and RHS of the rules are defined via pattern language [1] [8] [9]. In this case, the structure defined by the pattern language must be found, not an isomorphic occurrence.

A *precondition* assigned to a transformation rule is a Boolean expression that must hold at the moment the rule is fired. A *postcondition* assigned to a transformation rule is a Boolean expression that must hold after the completion of the rule. If a *precondition* of a transformation rule is invalid, then the rule fails without being fired. If a *postcondition* of a transformation rule is invalid after the execution of the rule, then the transformation rule fails. OCL expressions in model transformation rules correlate with it: in the LHS of a transformation rule they represent preconditions, and in the RHS, OCL expressions are postconditions [10].

The dominant decomposition of model transformations provides the functional behavior. The additional constraints ensure the correctness of certain transformation properties. These constraints are responsible for correctness, but often they are treated with secondary importance. They are applied repetitively and in several cases crosscut the transformation. Therefore, it is difficult for the designer to perform the intuitive activities required to verify the transformation.

In order to illustrate the issue of repetitive and crosscutting constraints, a case study is introduced. In [12], a variation of the "class model to relational database management system (RDBMS)” model transformation (also referred to as object-relational mapping) [19] is presented. In Figure 1, using the concrete syntax of our model transformation environment (VMTS, Section 3.1), the control flow model of the transformation is presented.

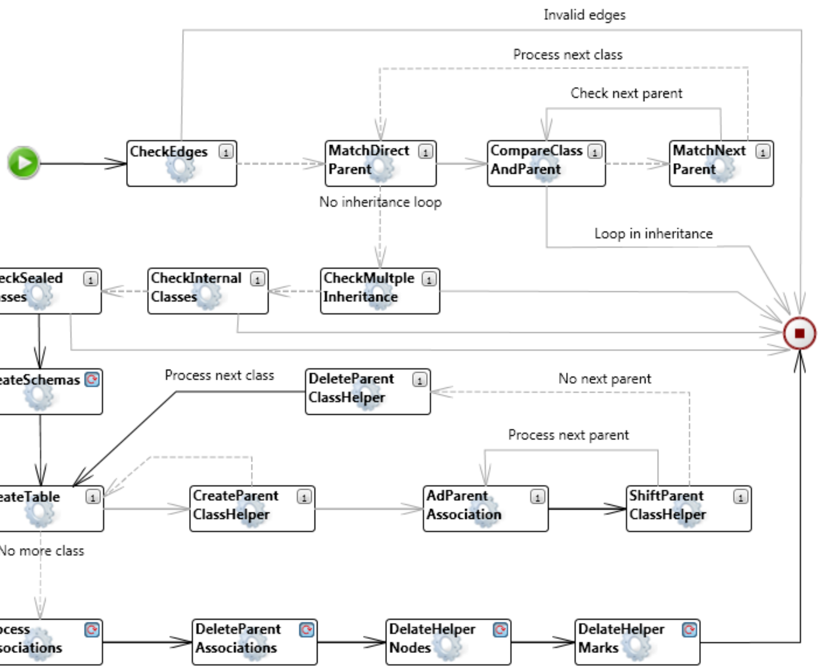


Figure 1
The control flow model of the transformation *ClassToRDBMS*

This model is a stereotypical activity diagram, in which each activity represents a rule. According to the goal of the units, the model can be divided into four parts: (i) The rules *CheckEdges*, *MatchDirectParent*, *CompareClassAndParent*, *MatchNextParent*, *CheckMultipleInheritance*, *CheckInternalClasses*, *CheckSealedClasses* verify the input model. (ii) The rule *CreateSchemas* and the substantial loop in the middle (*CreateTable*, *CreateParentClassHelper*, *AddParentAssociation*, *ShiftParentClassHelper*, *DeleteParentClassHelper*) are responsible for the schema and table creation as well as inheritance-related issues. (iii) The rule *ProcessAssociations* processes the associations. (iv) Finally, the last three rules remove the helper nodes and temporary associations.

In the control flow model, some rules have two outgoing edges. If a rule is successful, then the control is passed via the solid line; otherwise, the dashed line is used. For example, the first rule (*CheckEdges*) is successful if there is at least one dangling edge in the input model. Therefore, the solid outgoing line goes to the end node, because dangling edges are not permitted. If the rule *CheckEdges* was unsuccessful, then the control is passed to rule *MatchDirectParent*.

The first seven transformation rules verify five class diagram-related conditions. We differentiate between class diagram-related conditions that are general language-independent conditions (Conditions 1 and 2), and specific programming language (Conditions 3, 4 and 5). These condition groups form our well-formedness concerns.

Condition 1. Each association and inheritance edge should connect two nodes because no dangling edges are allowed in class diagrams. This condition is checked by rule *CheckEdges*. The constraints related to this condition are as follows:

```
context Association inv DanglingEdges1:
self.LeftNodeID is NULL or self.RightNodeID is NULL


context Inheritance inv DanglingEdges2:
self.LeftNodeID is NULL or self.RightNodeID is NULL
```

Condition 2. The *'no directed inheritance loop is allowed'* condition is checked by rules *MatchDirectParent*, *CompareClassAndParent*, and *MatchNextParent*. The rule *MatchDirectParent* selects a class yet to be processed, marks it, then matches its direct parent class. Rule *CompareClassAndParent* verifies that the class marked by a previous rule and actual parent class are not the same. The rule *MatchNextParent* matches the direct parent of the actual class. If the rule has successfully found the next parent, the control is passed to the rule *CompareClassAndParent*, where the originally marked class, the recently found parent, and the actual parent are compared. Otherwise, if there is no next parent, then the transformation continues with rule *MatchDirectParent* in conjunction with the next unprocessed class. If all of the classes have been checked, then the control is passed to rule *CheckMultipleInheritance*. If rule *CompareClassAndParent* finds that a class and its parent (direct indirect) are the same, then the transformation ends with error. The related constraint:

```
context Class inv ClassAndItsParentAreTheSame:
self = self.parentHelper.parent
```

Condition 3. No multiple direct parents are allowed. The condition is checked by rule *CheckMultipleInheritance*. If the rule finds a match where a class has more than one direct parent, then the transformation terminates with error.

Condition 4. The building blocks of software applications are components. They form the fundamental unit of deployment, version control, reuse, activation scoping and security permissions. A component is a collection of types and resources that are built to work in unison to form a logical unit of functionality. If the visibility of the class is set to *'internal'*, the type it defines is accessible only to types within the same component. The condition is checked by the rule *CheckInternalClasses*. The constraint related to this rule is:

```
context Class inv CheckInternalCondition:
self.Internal = true and self.neighborClasses->
exists(neighborClass | neighborClass.package <> self.package)
```

Condition 5. If the *'sealed'* attribute of a class is set to true, then other classes cannot be inherited from it. The condition is checked by rule *CheckSealedClasses*. The constraint related to this rule is:

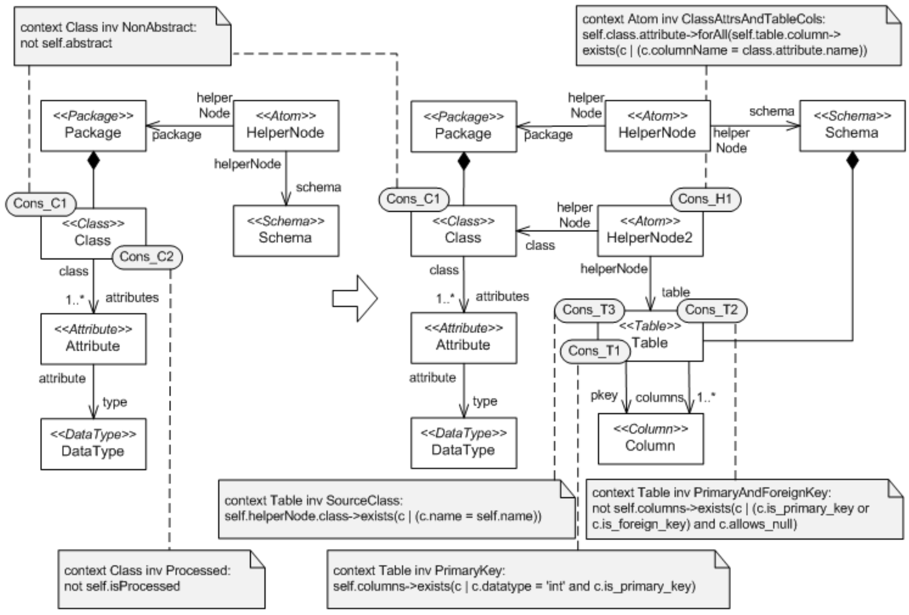
```
context Class inv CheckSealedCondition:
self.Sealed = true and self.childClasses->size() > 0
```

As was mentioned earlier, these conditions are aggregated into condition groups. The groups representing the well-formedness concerns are the *syntactic well-formedness* and the *semantic well-formedness* groups. Syntactic well-formedness conditions represent the general class diagram-related conditions. However, semantic well-formedness conditions are related to a specific programming language. Unfortunately, these concerns are logically scattered across several transformation rules. The syntactic well-formedness concern affects the rules *CheckEdges* and *CompareClassAndParent*. Furthermore, the semantic well-formedness concern affects the rules *CheckMultipleInheritance*, *CheckInternalClasses* and *CheckSealedClasses*. In the current case, these rules are developed based on their functional requirement, meaning they are designed around the functional concern. We could have designed the transformation around the well-formedness concerns, but in that case the rules would have crosscut the well-formedness conditions. In order to achieve the same functionality, transformation rules within a loop should be combined (e.g., with Concurrency Theorem [6]), and other rules should be designed in an unintuitive way.

In conclusion, the transformation cannot be refactored into a modular design in which both transformation rules and well-formedness conditions are elegantly expressed. Therefore, within these rules we can observe valid crosscutting.

The transformation rule *CreateTable* is shown in Figure 2. *CreateTable* works on the non-abstract classes and, based on them, defines tables for the resulting software model. The created table gets the same name based on the class. The table has an additional primary key column and a separate column for each class attribute. The rule matches the package of the class and the schema created for that package. Thus, the rule ensures that the table is created and inserted into the corresponding schema. In addition to these, *CreateTable* creates an edge between the class and its table. With the help of this edge, the subsequent rules can reach the right table from the class.

In order to ensure certain properties and provide validation for the rule *CreateTable*, six different constraints are propagated to it. Because we cannot discuss all transformation rules, we provide statistical data. The transformation *ClassToRDBMS* contains seventeen rules. The constraint *NonAbstract* appears 30 times and the constraint Abstract appears 16 times. Furthermore, the constraints *PrimaryKey* and *PrimaryAndForeignKey* are utilized 6 times. The constraints responsible for processing the associations between classes (*OneToOneOrOneToMany* and *ManyToMany*) are used 4 times. Gathering from this, the actual open issue is the repetitively appearing constraints. Further details of the transformation can be found in [12].



```
context Class inv NonAbstract:
not self.abstract

context Atom inv ClassAttrsAndTableCols:
self.class.attribute->forAll(self.table.column->
exists(c | (c.columnName = class.attribute.name))

context Table inv SourceClass:
self.helperNode.class->exists(c | (c.name = self.name))

context Table inv PrimaryAndForeignKey:
not self.columns->exists(c | (c.is_primary_key or
c.is_foreign_key) and c.allows_null)

context Class inv Processed:
not self.isProcessed

context Table inv PrimaryKey:
self.columns->exists(c | c.datatype = 'int' and c.is_primary_key)
```

Figure 2
Transformation rule *CreateTable*

The problems of crosscutting and repetitive constraints make understanding both the constraints and model transformation more difficult. Therefore, our goal is to achieve a consistent constraint management by separating constraints and weaving them automatically.

3 Backgrounds

This section introduces the Visual Modeling and Transformation System (VMTS) [20], which is our modeling and model transformation framework. The aspect-oriented constructs provided by the VMTS are also discussed. These aspect-oriented constructs are used in later sections, during the discussion of the novel constraint identification and weaving algorithms.

3.1 The Visual Modeling and Transformation System

Visual Modeling and Transformation System (VMTS) supports domain-specific modeling via metamodeling. Visual metamodel definitions can be extended through textual constraints, defined in OCL.

Furthermore, VMTS is a model transformation system which applies template-based text generation and graph rewriting-based [16] model transformation. Templates are used to produce textual output from model definitions in an efficient way, while graph transformation describes transformations in a visual way. A set of rewriting rules define a graph transformation system. The applications of these rules are the elementary operations of graphs. In our framework, a *model transformation* defines the algorithm of a model processing. We use graph rewriting rules and a control flow graph, which specifies the execution order of the rules. Furthermore, VMTS makes possible the verification/validation of the constraints of the transformation rules.

The results discussed in this paper, handling repetitive constraints (Section 4) and semi-automatic modularization of transformation constraints (Section 5), have been validated in VMTS as a proof-of-concept implementation.

3.2 Managing Constraints in an Aspect-oriented Way

This section provides an overview of aspect-oriented constraint management that was developed to address the problem of the crosscutting constraints in graph rewriting-based model transformations. Depending on the parameterization settings, VMTS provides certain aspect-oriented constraint notions: *aspect-oriented constraints* and *constraint aspects*. In order to turn crosscutting constraints into a coherent module, they are separated from the transformation rules. If a separated constraint can be parameterized by types only in the constraint expression, it is called an aspect-oriented constraint. If a separated constraint is parameterized by a model structure, it is referred to as a constraint aspect. Subsequent sections introduce the concept of aspect-oriented constraints and discuss the advantages of their use in visual model transformations.

The approach presented highlights the different role of the transformation rule constraints and the model constraints. Model constraints, defined in metamodels, should always hold for each instance of a certain metatype. However, in model transformation, preconditions should hold only at the beginning of the rule execution and postconditions at the end of the rule execution. Of course, metamodel constraints hold because the input and output models should be valid instances of the input and output metamodels; this is ensured by the tool during the modeling and can also be checked by the transformation.

3.2.1 Aspect-oriented Constraints

In VMTS, aspect-oriented constraints are OCL constraints; we separate them physically from transformation rules. Weaver algorithms weave them into the rules. The context information of the aspect-oriented constraints is used as a type-based pointcut. This pointcut, based on the metatype information, selects the appropriate rule nodes. This weaving process is referred to as *type-based weaving* [12].

In order to further develop the weaving procedure, we apply weaving constraints. A weaving constraint is similar to a property-based pointcut [7]. This is also an OCL constraint, which restricts the type-based weaving. Obviously, weaving constraint is not added to. Weaving constraints allow for the verification of optional conditions during the weaving process. We refer to it as *constraint-based weaving* [12].

The physically separated constraints require a weaver that applies type-based and constraint-based weaving mechanisms and facilitates the assignment of constraints to transformation rules. Our approach addresses the challenge of aspect-oriented constraint propagation with the Global Constraint Weaver (GCW) algorithm. The GCW algorithm is presented in Section 3.2.3.

3.2.2 Constraint Aspects

In order to make both the constraint weaving process and the constraint evaluation more efficient, we have developed the concept of *Constraint Aspects*. A constraint aspect is a model structure (pattern) to which we assign textual OCL constraints. This means that a constraint aspect, besides the textual conditions, also contains structure information, metatype, and multiplicity conditions, as well as weaving constraints. The structure, metatype conditions and weaving constraints are checked at propagation time, while the OCL constraints are validated during the model transformation.

During the constraint aspect propagation, we search for topological matches throughout transformation rules. These matches must satisfy metatype requirements. Next, the weaving constraints are verified.

In comparison, constraint aspects and aspect-oriented constraints can express the same conditions, but the structure of the constraint aspects makes their propagation to transformation rules more efficient.

3.2.3 Constraint Weaving

The constraint weaving is an offline method that is performed once for a constraint set and once for a transformation. Because of the two different notations of the aspectified constraints, there are also two weaver algorithms in VMTS: the Global Constraint Weaver (GCW) and the Constraint Aspect Weaver (CAW). The GCW algorithm receives the transformation rule, the aspect-oriented constraints and the weaving constraints as input parameters. The CAW receives the transformation rule and the constraint aspects as input parameters. The output of both weavers is the transformation rule with the propagated constraints.

The GCW algorithm, using type-based weaving and applying weaving constraints, weaves the aspect-oriented constraints to the appropriate rule nodes of the transformation rules. The CAW algorithm, using similar methods to GCW, weaves constraint aspects into model transformations.

4 Managing Repetitive Constraints

In our approach, model transformation-related problems concerning validation constraint management are separated into two groups: namely, the management of repetitively appearing constraints and the management of crosscutting constraints. This section clarifies the differences between these two types of constraints and discusses the methods applied for the handling of repetitive constraints.

In software engineering, it is advisable to follow the separation of concerns [4] (SoC) principle. In essence, this indicates that, in dealing with complex problems, the only possible solution is to divide the problem into sub-problems, and then to solve them separately. Next, combine the partial solutions to create a complete solution. One type of concerns, such as rewriting rules, may smoothly be encapsulated within building blocks by means of conventional techniques of modularization and decomposition, whereas the same is not possible for other types. More specifically, these types crosscut the design and are therefore called crosscutting concerns. Because of their specialty, crosscutting concerns raise two significant problems:

- The *scattering problem*: the design of certain concerns is scattered over many building blocks.
- The *tangling problem*: a building block can include the design of more than one concern.

Recall that in the validation of model transformations there are two concerns: the functionality of the transformation and the constraints ensuring the validation. Sometimes modularizing one of the two concerns implies that the other concern will crosscut the transformation, and vice versa.

Both scattering and tangling have several negative consequences for the transformations they affect. However, the aim of aspect-oriented methods is to alleviate these problems by modularizing crosscutting concerns. Therefore, in the case of crosscutting constraints, aspect-oriented methods should be applied in order to achieve consistent constraint management. Both logically coherent constraints (crosscutting constraints) and repetitively appearing constraints should be physically maintained in a modularized manner.

For the problem of crosscutting constraint management, a solution has been provided in [10] and this solution has been summarized in Section 3. Current section provides a novel approach for handling repetitive constraints in model transformations.

4.1 The Constraint Management Process

As we previously stated, consistent constraint management requires a mechanism that supports the handling of repetitive constraints. Our approach provides the following methods for their management:

- Constraints are defined independently from transformation rules. This allows us to maintain the constraints in a physically separated place.
- Constraint calls are defined, along with the designation where the constraints should be applied. Using the generalized version of the Global Constraint Weaver, the approach automatically assigns the constraints to the indicated points of the transformation.

In this approach, the selection of the rules, where the aspect should be propagated (constraint calls), is performed manually by the transformation designer. This method is supported by the weaver tool: the potential transformation rule nodes are offered for the transformation designer, who can manually select those which are required.

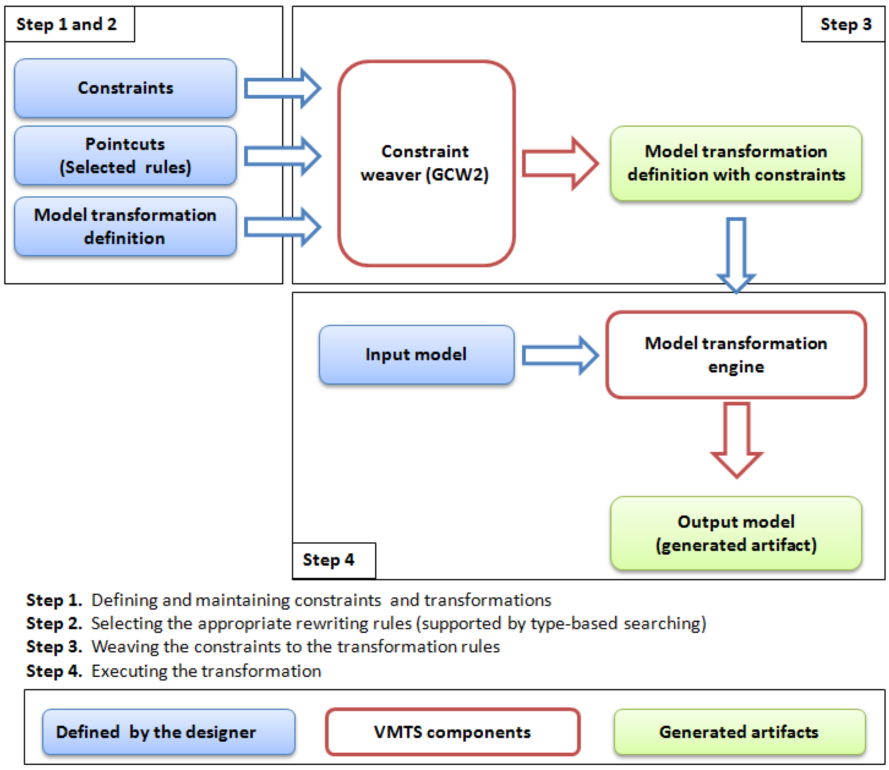

Step 1 and 2
Constraints
Pointcuts (Selected rules)
Model transformation definition
Step 3
Constraint weaver (GCW2)
Model transformation definition with constraints
Input model
Model transformation engine
Output model (generated artifact)
Step 4
Step 1. Defining and maintaining constraints and transformations
Step 2. Selecting the appropriate rewriting rules (supported by type-based searching)
Step 3. Weaving the constraints to the transformation rules
Step 4. Executing the transformation
Defined by the designer
VMTS components
Generated artifacts

Figure 3

The process of repetitive constraint handling

The whole process of repetitive constraint handling, and its role in the model transformation, is illustrated in Figure 3. Related to this process, we have identified four steps:

1 *Defining and maintaining constraints and transformations.* This step is performed by the transformation designer.

2 *Selecting the appropriate rewriting rules.* This step is also completed by the transformation designer. The result of this step is the constraint calls that designate the rewriting rules where to propagate the constraints (from where the constraints should be called during the transformation).

3 *Propagating the constraints to the rules.* This step is executed by the weaver component. The weaving method receives the transformation, the constraints, and the constraint calls. The result of the weaving process is the transformation definition with the assigned constraints.

4 *Executing the transformation.* This step is performed by the model transformation engine. The inputs are the transformation definition that contains the constraints and the input model. The output of the model transformation is the generated artifact that can also be a model or optional text, e.g. source code.

4.2 Generalizing the Constraint Weaving

Based on the Global Constraint Weaver (GCW), presented in Section 3, a generalized constraint weaving mechanism has been developed. This Generalized GCW (GCW2) method supports the weaving of the following constraint constructs:

- Aspect-oriented constraints driven by weaving constraints (introduced in Section 3).
- Repetitive constraints driven by their constraint calls.

In this approach, aspect-oriented constraints and repetitive constraints both represent constraints which are defined separately from model transformations. They are handled separately, because their weaving is driven by different constructs. The weaving of aspect-oriented constraints is supported by the weaving constraints, and the weaving of repetitive constraints is driven by constraint calls. Therefore, these two types of constraints are not mixed.

The inputs of the GCW2 algorithm include the transformation definition, the aspect-oriented constraints with their weaving constraints, and the repetitive constraints with their constraint calls. The output of the weaver is the constrained transformation. Algorithm 1 depicts the pseudo code of the GCW2 algorithm.

Algorithm 1. Pseudo code of the GLOBALCONSTRAINTWEAVER2 algorithm

```
1:  GLOBALCONSTRAINTWEAVER2 (Transformation T, ConstraintList AOCs, ConstraintList
    weavingCs, ConstraintList repetitiveCs, ConstraintCallList constraintCalls)
2:  for all Constraint AOC in AOCs do
3:    for all TransformationRule R in T do
4:      nodesWithProperMetaT ype = GETNODESBYMETATYPE (context type of AOC, R)
5:      nodesWithProperStructure = CHECKSTRUCTURE (nodesWithProperMetaT ype, R,
        AOC)
6:      checkedNodes = CHECKWEAVINGCONSTRAINTS (nodesWithProperStructure,
        weavingCs)
7:      WEAVECONSTRAINT (AOC, checkedNodes)
8:    end for
9:  end for
10: for all Constraint RC in repetitiveCs do
11:   for all ConstraintCall CC in constraintCalls do
12:     nodesToWeave = EVALUATECONSTRAINTCALL (CC, RC)
13:     WEAVECONSTRAINT (RC, nodesToWeave)
14:   end for
15: end for
```

The GCW2 algorithm is passed through a model transformation, a list of aspect-oriented constraints, a list of weaving constraints, a list of repetitive constraints and a list of constraint calls. The algorithm, using type-based weaving and applying weaving constraints, weaves the aspect-oriented constraints to the appropriate nodes of the rules. Furthermore, the algorithm weaves the repetitive constraints to the rules designated by the constraint calls.

The GCW2 algorithm uses a different block to manage the aspect-oriented constraint weaving (line 1-8) and the repetitive constraint weaving (line 9-14). In the first block, for each aspect-oriented constraint and transformation rule pair, the algorithm identifies the possible places where the constraint can be woven. It then checks the surrounding structures of these locations and evaluates the weaving constraint for the appropriate places. Finally, the constraint is woven to the correct rules. In the second block, for each repetitive constraint and constraint call pair, the algorithm decides where to weave the actual repetitive constraint, then performs the weaving.

An example of a constraint that repetitively occurs in transformation *ClassToRDBMS* is the *PrimaryKey* constraint:

```
context Table inv PrimaryKey:
self.columns->exists(c | c.datatype = 'int' and c.is_primary_key)
```

The constraint call used to propagate the constraint *PrimaryKey* is the following:

ConstraintCall_PrimaryKey {constraint: *PrimaryKey*, rules:

CreateTable (Table), CreateParentClassHelper (Table), AddParentAssociation (Table),

ProcessAssociations (Table1, Table2)}

The constraint call definition is named and contains a constraint reference (*PrimaryKey*) and an optional number of rule references. The enlisted rule names indicate from where the repetitive constraint should be called. The node names, following the rule names, are the parameters of the constraint calls. They designate where the exact rule nodes call the constraints.

The proposed method for handling repetitive constraints facilitates the definition of constraints independent of transformation rules and designates the rewriting rules, i.e., where to apply them. The approach automatically weaves the constraints to the designated points in the transformation. The benefit of this approach is that the constraints are maintained in one place and in one copy. Furthermore, our method supports a better understanding of both the transformations and constraints.

This section introduced the GCW2 algorithm, which facilitates the constraint weaving driven by both weaving constraints and manually defined constraint calls. The next section discusses the method to modularize transformation constraints if they already exist in model transformations.

5 Semi-Automatic Modularization of Transformation Constraints

In [12], a mechanism is introduced for systematically identifying crosscutting constraints. This section provides a generalized, semi-automatic method for modularizing both repetitive and crosscutting constraints.

In model transformations, some validation or other concerns can be expressed by several constraints. These concerns (expressed by more than one constraint) are the source of the crosscutting. In our approach, transformation designers can aggregate constraints into groups, in which each group represents a concern. The examples provided are the *syntactic well-formedness* and the *semantic well-formedness* groups.

Group_SyntacticWellFormedness {*DanglingEdges1, DanglingEdges2,*

ClassAndItsParentAreTheSame}

Group_SemanticWellFormedness {*MultipleInheritance, CheckInternalCondition,*

CheckSealedCondition}

The inputs of the modularization method are the transformation itself and the grouping definitions. The expected outputs are the modularized constraints and the constraint calls that support the weaving process. The tasks required by the modularization method are as follows:

1. Collect the constraints from the transformation.
2. Identify the crosscutting constraints.
3. Identify the repetitive constraints.
4. Extract the crosscutting constraints as aspects, and generate the constraint calls to support their weaving.
5. Extract the repetitive constraints as aspects, and generate the constraint calls to support their weaving.

In Step 2, the identification of crosscutting concerns is supported by the grouping definition. The algorithm checks whether the semantically coherent concerns are, physically, in the same rule or scattered across several rules. Concerns represented by single constraints cannot crosscut the transformations, but if they appear several times they are classified as repetitive constraints.

The crosscutting constraint identification method, presented in [12], provides the coloring and extracting algorithms. These algorithms have been updated to support both the repetitive and crosscutting constraint modularization in a general way. Based on the groups and the identified concerns, the reworked coloring algorithm assigns different colors to the constraints of the transformation. The automatic concern identification also accounts for the constraints not appearing in any of the user defined groups. In the output of the coloring algorithm, each color represents a concern. These concerns should be modularized. After the coloring, the extracting algorithm creates aspects from crosscutting and repetitive constraints, as well as generates the constraint call definitions.

The subsequent sections elaborate upon the algorithms, and their operation is illustrated with the help of our case study.

5.1 Generalized Coloring Algorithm

The algorithm receives the transformation with its constraints and the grouping definitions. The expected result is a concern list and a coloring table which provides the transformation rule and affected concern relations.

A concern is represented by a color and can be an optional condition or property expressed by one (simple) or several constraints. Examples of this include: the well-formedness concerns of our case study, as well as more simplified versions, namely, those including an attribute value or the existence of adjacent nodes of a specific type.

Algorithm 2 shows the pseudo code of the COLORING algorithm. The model transformation *T* and its corresponding groups are passed to the algorithm. The algorithm creates a list of rule-constraint pairs. These contain each transformation rule-constraint pair assignment, defined in transformation *T*. Based on their rule constraint pair assignment, the algorithm identifies the crosscutting concerns for each group (line 4). Next, the coloring table is updated with the actual group information, even if there exists no crosscutting related to the actual group. Then, the algorithm creates a concern (constraint) list (line 7) in which each member of the list represents a separated concern. This means that, if a constraint in the transformation contains more than one concern, the constraint is decomposed into several constraints. Therefore, more than one list member is created from such constraints. Groups are also added to the constraint list. Based on the list of constraint, the algorithm identifies repetitive constraints (line 9) and updates the coloring table accordingly.

Algorithm 2. Pseudo code of the COLORING algorithm

```
1:  COLORING (Transformation T, GroupList groups)
2:  ColoringTable coloringTable = new ColoringTable();
3:  RuleConstraintPairList ruleConstraintPairs = COLLECTRULECONSTRAINTPAIRS (T)
4:  for all Group G in groups do
5:    CrosscuttingList crosscuttings = IDENTIFYCROSSCUTTING (G, ruleConstraintPairs)
6:    UPDATECOLORINGTABLE (G, ruleConstraintPairs, crosscuttings)
7:  end for
8:  ConcernList concerns = COLLECTSEPARATEDCONCERNCONSTRAINTS (T)
9:  for all Constraint C in constraints do
10:   ConstraintList repetitives = IDENTIFYREPETITIVECONSTRAINTS (C, concerns)
11:   UPDATECOLORINGTABLE (C, constraints, repetitives)
12: end for
```

5.2 Generalized Constraint Extracting Algorithm

The algorithm receives the model transformation and the results of the coloring algorithm. The results of the algorithm are the modularized constraints and the constraint calls supporting the weaving.

The algorithm creates the modularized constraints based on the provided concern list. The group concerns are handled in a different way from simple constraints or constraint part concerns: each member of the group concern is modularized into a different constraint. The second part of the extracting algorithm creates the constraint calls both for crosscutting and repetitive constraints. These constraint calls contain the exact list of the rules from which they should be called. In general, for modularized crosscutting constraints (aspects) we prefer to use weaving constraints instead of the constraint calls. This is because with weaving constraints more complex conditions can be defined, and this type of weaving definition is used when defining these artifacts manually. In the current case, the artifacts are created by the extracting algorithm. Our aim is to provide a simple method that modularizes the concerns and creates such weaving artifacts that can reproduce the original transformation, exactly. Therefore, creating constraint calls for crosscutting constraints is the correct decision.

Algorithm 3. Pseudo code of the EXTRACTING algorithm

```
1:  EXTRACTING (Transformation T, ConcernList concerns, ColoringTable coloringTable)
2:  ConstraintList modularizedConstraints = new ConstraintList ()
3:  for all Concern groupConcern in concerns.GroupConcerns do
4:      for all Constraint C in groupConcern do
5:          modularizedConstraints.Add (C)
6:      end for
7:  end for
8:  for all Concern nonGroupConcern in concerns.NonGroupConcerns do
9:      modularizedConstraints.Add (nonGroupConcern.Constraint)
10: end for
11: ConstraintCallList constraintCalls = new ConstraintCallList()
12: for all ColoringItem coloringItem in coloringTable do
13:     ConstraintCall constraintCall = CreateConstraintCall(coloringItem, T)
14:     constraintCalls.Add (constraintCall)
15: end for
```

The EXTRACTING algorithm receives transformation *T*, the concerns identified by the COLORING algorithm and the *coloringTable*. The algorithm processes the concerns in two blocks. In the first block, the group concerns (*SyntacticWellFormedness* and *SemanticWellFormedness*) are processed; each constraint, although related to the group, is independent and is added to the modularized constraint list (line 2-6). In the second block, the constraints of the non-group concerns are processed: simple constraints and constraint parts (line 7-9). Next, using the coloring table (transformation rule - constraint mappings), the algorithm creates the constraint calls for each constraint.

Conclusions

We have discussed that in graph rewriting-based model transformations, the two main concerns are functionality, defined by the transformation rules, and the validation properties, expressed through constraints. Concerning model transformations, we have introduced the problem of repetitive and crosscutting constraints. We have identified the difference between repetitive and crosscutting constraints. We have shown that, in certain cases, crosscutting cannot be eliminated, but it can be solved by applying aspect-oriented mechanisms. We have briefly summarized our previous results related to aspect-oriented constraint management in model transformations. As a novel contribution, we have provided a mechanism for handling repetitive constraints. Unifying their treatment, we have developed a generalized method with its algorithms for semi-automatic modularization of repetitive and crosscutting constraints in model transformations.

The disadvantage of the repetitive constraint management approach regard is its being based on manual decisions: the transformation designer should designate the points where a constraint call should be applied. Therefore, the designer can miss some constraint call definitions, which would result in unexpected behavior of the transformation execution, especially in the case of complex transformations. The introduced approach has an UML-compliant notation that is easy to use and simple to understand.

Acknowledgement

The author would like to thank Tihamér Levendovszky for his valuable comments and support. This work was partially supported by the European Union and the European Social Fund through project FuturICT.hu (grant no.: TÁMOP-4.2.2.C-11/1/KONV-2012-0013).

References

[1] AGG, The Attributed Graph Grammar System Website, http://tfs.cs.tu-berlin.de/agg

[2] Assmann U., Ludwig, A.: Aspect Weaving by Graph Rewriting, Generative Component-based Software Engineering, Springer (2000)

[3] Bottoni, P., Koch, M., Parisi-Presicce, F., Taentzer, G.: Consistency Checking and Visualization of OCL Constraints, 294-308 (2000)

[4] Dijkstra, E. W.: A Discipline of Programming. Prentice Hall, Englewood Cliffs, NJ (1976)

[5] Dresden OCL Toolkit Website, http://dresden-ocl.sourceforge.net

[6] Ehrig, H., Ehrig, K., Prange, U., Taenzer, G.: Fundamentals of Algebraic Graph Transformation, Monographs in Theo. Comp. Sci., Springer (2006)

[7] Filman, R. E., Elrad, T., Clarke, S., Aksit, M.: Aspect-Oriented Software Development, Addison-Wesley (2004)

[8] Karsai, G., Agrawal, A., Shi, F., Sprinkle, J.: On the Use of Graph Transformation in the Formal Specification of Model Interpreters, Journal of Universal Comp. Science, Special issue on Formal Spec. of CBS (2003)

[9] Lara, J., Vangheluwe, H., Alfonseca, M.: Meta-Modelling and Graph Grammars for Multi-Paradigm Modelling in AToM, Software and Systems Modeling (SoSyM), Vol. 3(3), 194-209 (2004)

[10] Lengyel, L.: Online Validation of Visual Model Transformations, PhD thesis, Budapest University of Technology and Economics, Department of Automation and Applied Informatics (2006)

[11] Lengyel, L., Levendovszky, T., Charaf, H.: Validated Model Transformation-Driven Software Development, International Journal of Computer Applications in Technology, Vol. 31(1), 106-119 (2008)

[12] Lengyel, L., Levendovszky, T., Angyal, L.: Identification of Crosscutting Constraints in Metamodel-Based Model Transformations, IEEE Eurocon 2009, St. Petersburg, Russia, 359-364 (2009)

[13] OMG MDA Specification, MOMG document ormsc/01-07-01, 2001, http://www.omg.org/

[14] OMG OCL Specification, Version 2.2, OMG document formal/2010-02-01, 2010, http://www.omg.org/

[15] Pollet, D., Vojtisek, D., Jezequel, J. M.: OCL as a Core UML Transformation Language, WITUML: Workshop on Integration and Transformation of UML models, ECOOP 2002, Malaga, Spain (2002)

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

[17] Sutton, S. M., Rouvellou, I.: Modeling of Software Concerns in Cosmos. In Proceedings of the 1st International Conference on Aspect-Oriented Software Development, ACM Press, 127-133 (2002)

[18] Sztipanovits, J., Karsai, G.: Generative Programming for Embedded Systems, In GPCE '02: ACM SIGPLAN/SIGSOFT Conf. on Generative Programming and Component Eng., Springer, London, UK, 32-49 (2002)

[19] Taentzer, G., Ehrig, K., Guerra, E., de Lara, J., Lengyel, L., Levendovszky, T., Prange, U., Varro D., Varro-Gyapay, Sz.: Model Transformation by Graph Transformation: A Comparative Study, ACM/IEEE 8th Int. Conf. on Model Driven Engineering Languages and Systems, Jamaica (2005)

[20] VMTS Website, http://www.aut.bme.hu/vmts