Identification of Crosscutting Concerns in Constraint-Driven Validated Model Transformations

László Lengyel, Tihamér Levendovszky, and Hassan Charaf

Budapest University of Technology and Economics, 1111 Budapest, Goldmann György tér 3., Hungary

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

Abstract

Domain-specific model processors facilitate the efficient synthesis of application programs from software models. Often, model compilers are realized by graph rewriting-based model transformation. In Visual Modeling and Transformation System (VMTS), metamodel-based rewriting rules facilitate to assign Object Constraint Language (OCL) constraints to model transformation rules. This approach supports validated model transformation. Unfortunately, the validation introduces a new concern that often crosscuts the functional concern of the transformation rules. To separate these concerns, an aspect-oriented solution is applied for constraint management. This paper introduces the identification method of the crosscutting constraints in metamodel-based model transformation rules. The presented algorithms make both the constraints and the rewriting rules reusable, furthermore, supports the better understanding of model transformations.

Keywords Aspect-Oriented Constraints, Constraint Weaving, Identifying Crosscutting Constraints, Model Transformation

1. Introduction

Model-driven development approaches (for example Model-Integrated Computing (MIC) [22] and OMG's Model-Driven Architecture (MDA) [14]) emphasize the use of models at all stages of system development. They have placed model-based approaches to software development into focus.

MIC advocates the use of domain-specific concepts to represent the system design. Domain-specific models are then used to synthesize executable systems, perform analysis or drive simulations. Using domain concepts to represent system design helps increase productivity, makes systems easier to maintain and evolves and shortens the development cycle.

MDA 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), one or more platform-specific models (PSM) including 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 generate the platform-dependent artifacts automatically by model compilers.

Model transformation lies at the heart of the model-driven approaches [14] [23]. With Model-Driven Software Development (MDSD), a modeling environment operates according to a modeling paradigm, which is a set of requirements that define how a system within a domain is modeled. The modeling paradigm is captured in the form of formal modeling language specifications referred to as metamodel. Once a metamodel is created for a particular domain, a modeling environment allows a modeler to create domain-specific models that can be synthesized into various artifacts.

Transformations appear in many, different situations in a model-based development process. A few representative examples are as follows. (i) Refining the design to implementation; this is a basic case of PIM/PSM mapping. (ii) Aspect weaving; the integration of aspect models/code into functional artifacts is a transformation on the design [1]. (iii) Analysis and verification; analysis algorithms can be expressed as transformations on the design [2].

One can conclude that transformations in general play an essential role in model-based development, thus, there is a need for highly reusable model transformation tools that support validated model transformation.

At the implementation level, system validation can be achieved by testing. Various tools and methodologies have been developed to assist in testing the implementation of a system (for example, unit testing, mutation testing, and white/black box testing). However, in the case of model transformation environments, it is not enough to validate that the transformation engine itself works as it is expected. The transformation specification should also be validated. There are only few and not complete facilities provided for testing offline transformation specifications in an executable style. However, online validated model transformation can guarantee that if the transformation finishes successfully, the generated artifact is valid, and it is in accordance with the required output [9] [10].

For example, require a transformation that transforms class model to relational database management system (RDBMS) model (transformation *Class2RDBMS*) to guarantee the followings: a class that is marked as non-abstract in the source model is transformed into a single table of the same name in the target model, each table has primary key, each class attribute is part of a table, each many-to-many association has a distinct table, and so on.

These types of requirements can be specified by Object Constraint Language (OCL) [17] constraints assigned to the transformation rules. Unfortunately, often, the same constraint is repetitiously applied in many different places in a transformation, therefore the constraints crosscut the transformation rules and their management becomes hard.

In [8], a solution of the case study *Class2RDBMS* is provided where a transformation is presented with 9 transformation rules and two constraints are emphasized, from which the first one appears 30 times in 9 transformation rules and the second one 16 times in 6 transformation rules. This is very difficult to manually manage crosscutting and scattering constraints, because all of the modifications have to be done on all occurrences of the constraints. Constraints appearing several times in a transformation increase the time of constraint handling and the possibility of making a mistake during the modification. Using aspect-oriented constraints, a method has been given to solve the problem of the crosscutting con-

straint in model transformations [8] [10]. The main idea is to handle constraints similarly to Aspect-Oriented Programming (AOP) aspects, to provide the transformation constraints with the properties of the AOP aspects. AO constraints are created separately from transformation rules and, using a weaver method, they are woven back to the transformation rules before the execution of the transformation. The result of this method is a consistent constraint management (modification, deletion and propagation) with crosscutting constraint separation and weaving.

The current work proposes a method to identify the crosscutting constraints in model transformations. Our motivation is driven by the fact that transformation designers prefer defining transformation rules directly with constraints. Therefore, a solution should be provided to extract constraints from existing transformation rules. The proposed approach includes several algorithms, from which many can be applied independently of the current context as well, for example: constraint relocation and decomposition. The method makes both the constraints and the rewriting rules reusable, furthermore, facilitates the better understanding and maintainability of the metamodel-based model transformations.

2. Backgrounds

This section as a background information introduces the Visual Modeling and Transformation System (VMTS), the problem of the crosscutting constraints in metamodel-based model transformation rules, and the methods provided by VMTS to define and apply transformation constraints as aspects.

Graph rewriting [20] is a powerful technique for graph transformation with a strong mathematical background. The atoms of graph transformations are rewriting rules, each 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 LHS in the graph to which the rule is applied (host graph), and replacing this subgraph with RHS.

2.1 Visual Modeling and Transformation System

Visual Modeling and Transformation System (VMTS) [13] [24] supports editing models according to their metamodels, and allows specifying constraints written in Object Constraint Language (OCL) [17]. Models are formalized as directed, labeled graphs. VMTS uses a simplified class diagram for its root metamodel ("visual vocabulary"). Also, VMTS is a model transformation system, which transforms models using graph rewriting techniques. Moreover, the tool facilitates the verification of the constraints specified in the transformation rule during the model transformation process.

In VMTS, LHS and RHS of the transformation rules are built from metamodel elements. This means that an instantiation of LHS must be found in the input graph instead of the isomorphic subgraph of LHS.

Rewriting rules can be made more relevant to software engineering models if the metamodel-based specification of the transformations allows assigning OCL constraints to the individual transformation rules. This technique facilitates a natural representation for multiplicities, multi-objects and assignments of OCL constraints to the rules with a syntax close to the UML notation.

An example metamodel-based transformation rule that generates database tables from UML classes is depicted in Figure 1. Constraints propagated to the transformation rule nodes are also presented: *Cons_C1*, *Cons_C2*, *Cons_H1*, *Cons_T1*, and *Cons_T2*. With the help of these constraints we can require certain properties from the transformation rule, and we can make them validated [10].

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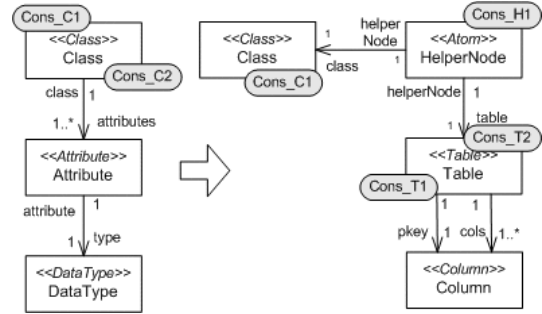

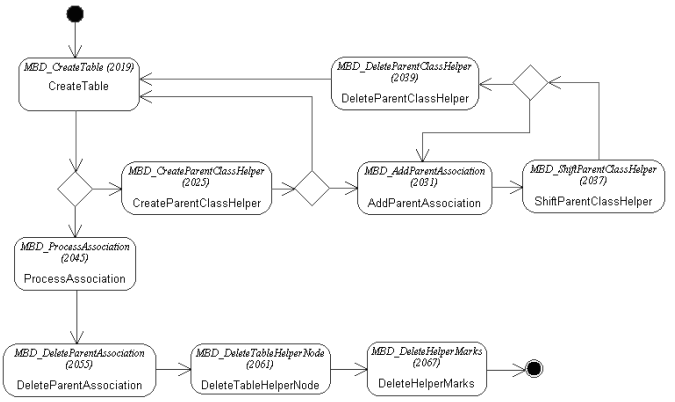**Figure 1.** Example transformation rule: *ClassToTable*

Figure 2. Example transformation (VCFL model): *ClassToRDBMS*

The constraint *NonAbstract* (*Cons_C1*) is assigned to the pattern rule node *Class* in LHS of the rule *CreateTable*. This link forms a precondition for the rule, it requires the rule to process only non-abstract classes.

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

The constraint *PrimaryKey* (*Cons_T1*) is a postcondition of the rule *CreateTable*, it is assigned to the rule node *Table*. This constraint requires the rule that all created table has a primary key of *int* type.

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

The constraint *ClassAttrsAndTableCols* (*Cons_H1*) is linked to the rule node *TableHelperNode*, it requires that each class attribute should have a created column with the same name in the resultant table.

The constraints assigned to the transformation rule guarantee our requirements. After a successful rule execution, the conditions hold and the output is valid, this cannot be achieved without constraints.

VMTS applies the presented metamodel-based model transformation rule definition methods. When the transformation is performed, the changes are specified by the RHS and *internal causality* relationships defined between the LHS and the RHS elements
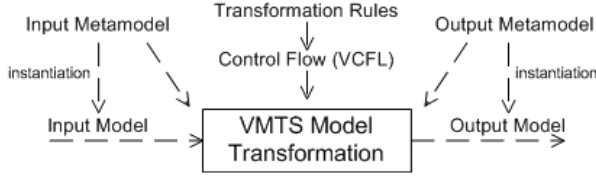
Figure 3. Principles of VMTS metamodel-based model transformation

of a transformation rule. Internal causalities can express the modification or removal of an LHS element, and the creation of an RHS element. Imperative OCL [18] or XSLT scripts can access the attributes of the objects matched to the LHS elements, and produce a set of attributes for the RHS element to which the causality points.

Classical graph grammars apply any production that is feasible. This technique is appropriate for generating and matching languages but model transformations often need to follow an algorithm that requires a stricter control over the execution sequence of the rules, with the additional benefit of making the implementation more efficient.

VMTS Visual Control Flow Language (VCFL) supports the following constructs: sequencing transformation rules, branching with OCL constraints, hierarchical rules, parallel execution of the rules, and iteration. An example VMTS transformation, a VCFL model, is presented in Figure 2.

VMTS transformation rules have two specific properties: *Exhaustive* and *MultipleMatch*. An exhaustive transformation rule is executed repeatedly, as long as LHS of the rule can be matched to the input model. The *MultipleMatch* property of a rule allows that the matching process finds not only one but all occurrences of LHS in the input model, and the replacement is executed on all the found places.

The interface of the transformation rules allows the output of one rule to be the input of another rule (parameter passing), in a dataflow-like manner. In VCFL, this construction is referred to as *external causality*. An external causality creates a linkage between a node contained by RHS of the rule i and a node contained by LHS of the rule $i + 1$. Since rule i provides partial match to rule $i + 1$, this feature accelerates the matching process and reduces the complexity.

VMTS has state-of-the-art mechanisms for validated model transformation, constraint management and control flow definition. The environment has several standalone algorithms and other solutions that make them efficient. Moreover, VMTS has a unique, aspect-oriented technique-based constraint management [10]. The constraint-driven branching mechanism of the VMTS is unique in the sense that the decision is made not only based on the actual state of the input model but using system variables ($SystemLastRuleSucceed$) as well.

2.2 Crosscutting Constraints

In model transformation, the dominant decomposition is the functional behavior of the transformation rules. The constraints ensure the correctness of the transformation only if they are well-defined by the designer. Although they are responsible for the correctness, the constraints are usually specified after the first draft of the transformation, and treated with secondary importance. They crosscut the transformation, and it is almost impossible for the designer to perform the intuitive activity of verifying the transformation.

Our motivating example (transformation *ClassToRDBMS*) is presented in the previous section. The control flow model (stereotyped activity diagram, where each activity represents a rule) of the

example is depicted in Figure 2. The control flow model defines the execution order of the rules. The model can be divided into three parts according to the goal of the units: (i) The large loop on the top is responsible for the table creation and inheritance-related issues. (ii) The rule *ProcessAssociation* processes the associations. (iii) Finally, the last group of rules remove the helper nodes and temporary associations.

Constraints such as *NonAbstract*, *PrimaryKey*, and *ClassAttrsAndTableColls* (Figure 1) cannot be encapsulated in any of the rules or rule nodes. However, they express the same constraint concerns on the rules: therefore they should be defined separately from the transformation rules and woven automatically to the appropriate rule nodes later.

Since there is not enough space to present all transformation rules and all occurrences of the constraints appearing in rule *CreateTable*, we provide statistical data only, and the details can be found in [10] and [24]. The transformation *Class2RDBMS* contains nine transformation rules. In these rules the constraint *NonAbstract* appears 30 times, constraint *Abstract*, which requires the presence an abstract class, appears 16 times. Furthermore, the constraints *PrimaryKey* and *PrimaryAndForeignKey* are linked 6 times, and constraints *OneToOneOrOneToMany* and *ManyToMany*, which are related to processing the associations between classes, appear 4 times [8] [11]. This means that one of the open issues with respect to the transformation is that the same constraints appear several times.
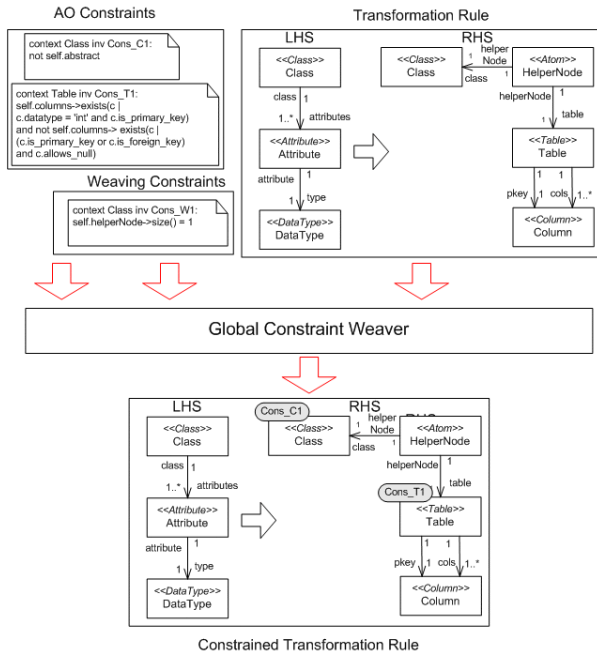
2.3 Aspect-Oriented Constraint Management in VMTS

As it was presented in the previous section, in model transformation, some constraints assigned to transformation rules represent the crosscutting concerns.

In VMTS, aspect-oriented constraints are OCL constrains defined separately from the transformations and transformation rules, and are woven to the rules later using a weaver method. Recall that in VMTS, transformation rules are built from metamodel elements, where a metamodel element can appear arbitrary times in a transformation rule. A rule is not an instance of the metamodel, both of them are on the same meta level. The input and output models are the instances of the metamodels. Each transformation rule node and edge has a metatype that corresponds to a metamodel type. The context information of the aspect-oriented constraints can be used as a type-based pointcut that selects rule nodes based on their metatype. The weaving process driven by the type-based pointcuts is referred to as *type-based weaving* [12]. In Figure 1, *Class* is the context of the constraint *NonAbstract* and the target rule nodes (rule node *Class* in LHS and rule node *Class* in RHS) of the propagation are selected based on this metatype.

To refine the weaving procedure, *weaving constraints* are applied. A weaving constraint is similar to a property-based pointcut, it is also an OCL constraint, which specifies the weaving, but it is not woven to transformation rules, and thus, it is not used during the transformation process. Weaving constraints facilitate optional conditions during the weaving process. Therefore, it is referred to as *constraint-based weaving* in VMTS [12]. A weaving constraint can be used to represent one or many characters as a means of specifying more than one attribute during a search procedure. This enables to select multiple rule nodes with a single specification. For example, the propagation of constraint *PrimaryKey* (Figure 1) can be refined with weaving constraints. The constraint *PrimaryKey* can be separated from the transformation rules, and using the weaving constraint *table.name = 'Table*'* it can be woven to rule nodes, whose names start with the *'Table'* character sequence.

Having separated the constraints from rule nodes, we also need a weaver which facilitates the propagation (linking) of the constraints to the rule nodes. Our approach solves the aspect-oriented

AO Constraints

Transformation Rule

context Class inv Cons_C1:
not self.abstract

context Table inv Cons_T1:
self.columns->exists(c |
c.datatype = 'int' and c.is_primary_key)
and not self.columns-> exists(c |
(c.is_primary_key or c.is_foreign_key)
and c.allows_null)

Weaving Constraints

context Class inv Cons_W1:
self.helperNode->size() = 1

LHS
RHS

<<Class>>
Class

<<Class>>
Class

<<Atom>>
HelperNode

class 1

1..* attributes

helperNode 1

table

<<Attribute>>
Attribute

<<Table>>
Table

attribute 1

1 type

pkey 1 cols 1..*

<<DataType>>
DataType

<<Column>>
Column

Global Constraint Weaver

LHS
RHS

Cons_C1

<<Class>>
Class

<<Class>>
Class

<<Atom>>
HelperNode

class 1

1..* attributes

helperNode 1

table

<<Attribute>>
Attribute

Cons_T1

<<Table>>
Table

attribute 1

1 type

pkey 1 cols 1..*

<<DataType>>
DataType

<<Column>>
Column

Constrained Transformation Rule

Figure 4. The weaving process and the input and output of the GCW

constraint propagation with the Global Constraint Weaver (GCW) algorithm [10] (Figure 4).

This mechanism facilitates our approach towards managing constraints using aspect-oriented techniques. Similarly to aspects, the constraints are specified and stored independently of any model transformation rule or rule node and are linked to rule nodes by the GCW.

The output of the weaver is not stored as a new transformation rule. The result is handled as a link between the constraints and a transformation rule. This link is referred to as *weaving configuration* [12]. A weaving configuration can be executed similarly to a transformation. The difference is that it contains the links between the transformation rule nodes and the constraints, therefore, during the execution the transformation engine applies the constraints woven to the transformation rules. Weaving configurations are created once, they are stored in the database, but they require significantly less space than transformation rules, because weaving constraints represent only the rule-constraint relations. Furthermore, this representation makes the transformation rule management transparent: it is not required to modify the rules in each weaving configuration, but only on their original place.

2.4 Constraint Normalization in VMTS

OCL constraints often contain complex expressions with several navigation steps. The constraint evaluation consists of two parts. (i) Selecting the object and its properties that the constraint needs to be checked on, and (ii) executing the checking. In general, the larger part of the evaluation is the first step, because of its computational complexity. Each navigation step in a constraint means several queries on the model database. Therefore the original motivation of the normalization method was to reduce the navigation steps contained by the constraints, because the eliminated navigation steps accelerate the first part of the constraint evaluation. In [7] and [10] a method is provided with algorithms to normalize OCL constraints in metamodel-based transformation rules. This normalization method with the results of the constraint relocation and de-

composition algorithms support the aspectification of the crosscutting constraints in model transformations (Section 3).

3. Identification of Aspect-Oriented Constraints

This section provides a method with several algorithms to support the detection of the crosscutting constraints in metamodel-based model transformations.

The input of the method is a transformation (transformation rules and a control flow model), and the expected output is the list of the crosscutting constraints separated as aspects. A simple but promising idea is the following:

1. Collect the constraints appearing in the transformation.

2. Identify the repetitive constraints.

3. In fact, not all of the repetitive constraints are crosscutting constraints. Therefore, for each repetitively appearing constraints decide if the actual constraint is crosscutting for the transformation or not.

4. Extract crosscutting constraints as aspects.

The separation of concerns principle states that a given problem involves different kinds of concerns that should be identified and separated to cope with complexity, and to achieve the required engineering quality factors such as robustness, adaptability, maintainability, and reusability. The principle declares that each concern of a given software design problem should be mapped to one module in the system. Otherwise, the problem should be decomposed into modules such that each module has one concern. The advantage of this is that concerns are localized and as such can be easier understood, extended, reused, and adapted.

Many concerns can indeed be mapped to single modules. Some concerns, however, cannot be localized and separated easily, and given the design language we are forced to map such concerns over many modules. This is called *crosscutting concern* or *aspect*. Aspects are not the result of a bad design but have more inherent reasons. A bad design including mixed concerns over the modules could be refactored to a neat design in which each module only addresses a single concern. However, if we deal with these crosscutting concerns, this is not possible in principle that is, each refactoring attempt will fail and the crosscutting will remain. A crosscutting concern is a serious problem, since it is harder to understand, reuse, extend, adapt and maintain a concern because it is spread over many rule nodes. Finding the places where the crosscutting occurs is the first problem, adapting the concern appropriately is another problem.

Our crosscutting constraint identification method can be divided into two main parts: coloring (Section 3.1) and extracting (Section 3.2) constraints. Based on the user defined or automatically identified concerns the coloring algorithm assigns colors to the constraints of the processed model transformation. Each color represents a concern that should be modularized. Ideally each constraint will have exactly one assigned color, which means there is no crosscutting. After the coloring, when we realized that there are constraints with more than one color, the extracting is applied to realize crosscutting constraints as aspects (aspect-oriented constraints). Extracting is supported by constraint decomposition [10].

3.1 Coloring Algorithm

The input of the coloring algorithm is the model transformation with the propagated constraints. The output is the coloring, where each of the colors represent a concern. Of course the relevant concerns are also specified by the transformation designer.

A concern, which represents a color can be related to an optional property that is expressed by a constraint: e.g. an attribute value or the existence of specific type adjacent nodes. For example:

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

The constraint *NonAbstract* represents the concern, which states that the processed class should be non-abstract.

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

The constraint *SourceClass* represents the concern, which predicates that a generated table has a source class with the same name.

If we have only these two constraints, then the coloring is simple: the algorithm assigns to each of the constraint concerns a different color. But if a constraint comprises more concerns, then the coloring will be compound:

```
context Class inv NonAbstractAndProcessed:
not self.abstract and not self.isProcessed
```

The constraint *NonAbstractAndProcessed* incorporates two concerns: (i) the matched class should be non-abstract, and (ii) the matched class should be non-processed. In this case two colors are assigned by the coloring algorithm to the constraint.

The concerns that should be taken into account by the coloring algorithm are defined by meta OCL constraints. These meta OCL constraints form an OCL Set. Each element of this Set represents a color (concern ID). These constraints are evaluated for the model transformation constraints. The result is the coloring, which is refactored by the extracting algorithm (Section 3.2).

Meta OCL constraints are defined as {*context*, *constraint expression*, *color*} triplets. Example simple meta OCL constraints for boolean type attributes: {*Class*, *abstract*, *Color 1*} {*Class*, *isProcessed*, *Color 2*}. Example meta OCL constraint for existing adjacent nodes {*Class*, *self.helperNode.table->size() > 0*, *Color N*}.

An important requirement that the whole method should provide is that each concept can have only one color, e.g. it is not allowed that *abstract* has two or more colors. This means that the elements of the meta OCL Set should be unique.

Algorithm 1 presents the pseudo code of the COLORING algorithm. The model transformation T and the meta OCL Set $metaOCLs$ are passed to the algorithm, which iterates on the constraints propagated to the rule nodes of the transformation T (line 3). In an embedded loop, for each constraint the algorithm iterates on the meta OCL constraints (line 4), and evaluates the relevance of the actual meta constraint ($metaConstraint$) for the actual constraint (C) (line 5). If the constraint C contains the concern represented by the actual meta constraint $metaConstraint$ then the color of the meta constraint $metaConstraint$ is assigned to the constraint C (line 6). Finally the coloring is returned by the algorithm.

Algorithm 1 Pseudo code of the COLORING algorithm

1: COLORING (Transformation T, OCLSet $metaOCLs$): ColoringTable
2: ColoringTable $coloringTable$ = new ColoringTable();
3: **for all** Constraint C in T **do**
4:    **for all** Constraint $metaConstraint$ in $metaOCLs$ **do**
5:       **if**   CHECKCONSTRAINTRELEVANCE(C, $metaConstraint$) **then**
6:          UPDATECOLORINGTABLE($coloringTable$, C, $metaConstraint.Color$)
7:       **end if**
8:    **end for**
9: **end for**
10: **return** $coloringTable$

This is obvious that meta OCL constraints contain the understanding of the concepts, and this is provided by the developer. Theoretically, this method can provide a 100% solution for our problem. At this point the main question is the quality of the meta OCL constraints, because meta OCL constraints should cover all concerns and should take into account everything from the point of transformations view. To obtain a useful result we should ensure the completeness of the defined meta constraints. (Currently this is the developers responsibility.) Otherwise, the result is relevant only for the covered part of the concerns.

3.2 Extracting Algorithm

The inputs of the constraint extracting algorithm are the model transformation and the result of the coloring algorithm. The outputs are the constraints extracted into aspects.

If the coloring is unambiguous, each constraint has maximum one color, then aspects can be created based on the colors. Constraints without color are not extracted as aspects. But complex constraints may have several colors at the same time. On the level of the source code, crosscutting resulted by the bad design can be solved with refactoring. In the domain of the metamodel-based model transformation, we can apply constraint relocation and decomposition in order to eliminate the annoying consequences of the bad design (crosscutting constraints).

Algorithm 2 presents the pseudo code of the EXTRACTING algorithm, which uses the constraint relocation and constraint decomposition provided by our constraint normalization method [10].

Algorithm 2 Pseudo code of the EXTRACTING algorithm

1: EXTRACTING   (Transformation T,   ColoringTable $coloringTable$): AspectList
2: AspectList $aspectList$ =   new AspectList($coloringTable.Colors.Size$);
3: Transformation $decomposed$ =   DECOMPOSECONSTRAINT(T)
4: **for all** ColoringItem $coloringItem$ in $coloringTable$ **do**
5:    **for all** Color $color$ in $coloringItem$ **do**
6:       UPDATEASPECTLIST($aspectList$, $color$, $decomposed$.GETDECOMPOSEDCONSTRAINT($coloringItem$.GETCONSTRAINTBYCOLOR($color$)))
7:    **end for**
8: **end for**
9: **return** $aspectList$

The transformation T and the coloring $coloringTable$ is passed to the EXTRACTING algorithm. The algorithm decomposes the constraints of the transformation (line 3) [10]. The algorithm iterates on the coloring items provided by the coloring (line 4), and for each item iterates on the colors assigned to the actual coloring concern (line 5). Based on the actual color the algorithm retrieves the relevant constraint from the actual coloring item. Using this constraint the correspondent constraint is queried from the decomposed version of the transformation. Based on the decomposed constraint the algorithm updates the already prepared aspect list (line 6). Finally the aspect list is returned.

4. Related Work

An aspect-oriented approach is introduced in [5] for software models containing constraints, where the dominant decomposition is based upon the functional hierarchy of a physical system. This approach provides a separate module for specifying constraints and their propagation. A new type of aspect is used to provide the weaver with the necessary information to perform the propagation: the strategy aspect. A strategy aspect provides a hook that

the weaver may call in order to process the node-specific constraint propagations.

At the time of the writing we have no knowledge about that any other approach supports aspect-oriented constraint management in model transformation rules, therefore, there is no other method for identifying crosscutting constraints in model transformations. But there are other software development fields, where identifying crosscutting concerns is also crucial.

In [4] an evaluation of clone detection techniques for identifying crosscutting concerns is presented. [6] introduces a tool that finds clones and displays them to the programmer. The approach is based on program dependence graphs (PDGs) and program slicing. In [21] a method is provided for identifying crosscutting concerns in requirements specifications. [3] provides support for developers to identify aspects early in the software lifecycle. A method is presented for aspect identification and analysis in requirements documentation. The Prism project [25] develops tools and techniques for discovering non-localized units of modularity in large software systems. [15] proposes a model to identify and specify quality attributes that crosscut requirements including their systematic integration into the functional description at an early stage of the software development process.

5. Conclusions

This paper has introduced an aspect-oriented solution for the problem of crosscutting constraints in metamodel-based model transformations. We have presented the aspect-oriented constraint management of Visual Modeling and Transformation System. So far VMTS is the only environment that provides aspect-oriented methods for constraint management. The main contribution of the paper is the identification of crosscutting constraints in model transformations. We have presented an approach with algorithms that semi-automatically identifies the crosscutting constraints and separates them into aspects.

Of course we would like to automate the largest possible part of the meta constraint definition. Therefore, the next question is: which concerns can be identified automatically, and, of course, in which way. The method can be supported by providing concern suggestions for the developer. This method will not provide a 100% solution, but the suggested concerns can be refined by the developer. Our first suggestion (*Suggestion 1*) is to identify the constraints of the transformation: simple constraints and the subterms of complex constraints based on the boolean separators (and, or, xor). The suggested constraints will be the ones, which are propagated at least twice to any of the rule nodes of the transformation.

A heuristic-based solution could improve the flexibility and usability of the current coloring method: suggestions, mentioned above, should be defined on a higher level. A language should be provided that facilitates to define what should be checked, and the source code that performs the checking is generated automatically based on it. For example: *Suggestion 1* is defined as a heuristic on higher abstraction level, and the checker that performs the control is automatically generated. The research related to the heuristic-based solution is the subject of our future work.

Acknowledgments

References

[1] U. Assmann and A. Ludwig, Aspect Weaving by Graph Rewriting, Generative Component-based Software Engineering, Springer, 2000.

[2] U. Assmann, How to Uniformly specify Program Analysis and Transformation, Int. Conference on Compiler Construction (CC) 96, LNCS 1060, Springer, 1996.

[3] e. Baniassad, S. Clarke, Finding Aspects in Requirements with Theme/Doc, Early Aspects 2004, 2004.

[4] M. Bruntink, A. van Deursen, R. van Engelen, T. Tourw, On the Use of Clone Detection for Identifying Crosscutting Concern Code, IEEE Trans. on Software Engineering, 2005, Vol. 31, No. 10, pp. 804-818.

[5] J. Gray, T. Bapty, S. Neema and J. Tuck, Handling Crosscutting Constraints in Domain-Specific Modeling, Communications of the ACM, October 2001, pp. 87-93.

[6] R. Komondoor, S. Horwitz, Using slicing to identify duplication in source code, 8th Int. Symposium on Static Analysis, pp. 4056, 2001.

[7] L. Lengyel, T. Levendovszky, H. Charaf, Normalizing OCL Constraints in UML Class Diagram-Based Metamodels - AND/OR Clauses, EUROCON 2005 Int. Conference on Computer as a tool, IEEE, Belgrade, 2005, pp. 579-582.

[8] L. Lengyel, T. Levendovszky, H. Charaf, Eliminating Crosscutting Constraints from Visual Model Transformation Rules, ACM/IEEE 7th Int. Workshop on AOM, Montego Bay, Jamaica, October 2, 2005.

[9] L. Lengyel, T. Levendovszky, H. Charaf, Constraint Validation Support in Visual Model Transformation Systems, Acta Cybernetica, ISSN 0324-721X, Vol. 17(2), pp. 339-357, 2005.

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

[11] L. Lengyel, T. Levendovszky, G. Mezei, H. Charaf, Model-Based Development with Strictly Controlled Model Transformation, 2nd Int. Workshop on Model-Driven Enterprise Information Systems, MDEIS 2006, Cyprus, 2006, pp. 3948.

[12] L. Lengyel, T. Levendovszky, H. Charaf, Optimizing Constraint Weaving in Model Transformation with Structural Constraint Specification, 8th Int. Workshop on AOM, March 21, 2006, Bonn, Germany.

[13] T. Levendovszky, L. Lengyel, G. Mezei, H. Charaf, A Systematic Approach to Metamodeling Environments and Model Transformation Systems in VMTS, ENTCS, 2004.

[14] A. Metzger, A systematic look at model transformations, In Model-driven Software Development, Vol. II of Research and Practice in Software Engineering. Springer, 2005.

[15] A. Moreira, J. Araujo, I. Brito, Crosscutting Quality Attributes for Requirements Engineering, 14th Int. Conf. on Software Engineering and Knowledge Engineering, pp. 167174. ACM Press, 2002.

[16] OMG MDA Guide Version 1.0.1, 2003. Document number: omg/2003-06-01, www.omg.org/docs/omg/03-06-01.pdf

[17] OMG OCL Spec., Version 2.0, 2006. http://www.omg.org/

[18] OMG QVT, MOF 2.0 Query/Views/Transformation Specification, http://www.omg.org/cgi-bin/apps/doc?ad/05-03-02.pdf

[19] OMG UML Spec., Version 2.1.1, 2007. http://www.uml.org/

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

[21] L. Rosenhainer, Identifying Crosscutting Concerns in Requirements Specifications, 2004.

[22] J. Sztipanovits and G. Karsai, Model-Integrated Computing, IEEE Computer, Apr. 1997, pp. 110-112.

[23] J. Sztipanovits, Generative programming for embedded systems, In Generative Programming and Component Engineering (GPCE), vol. 2487 of LNCS, pp. 32-49, Pittsburgh, October 2002.

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

[25] C. Zhang, H.-A. Jacobsen, A Prism for Research in Software Modularization through Aspect Mining, Technical report, Middleware Systems Research Group, University of Toronto, 2003.