PERIODICA POLYTECHNICA SER. EL. ENG. VOL. 49, NO. 1–2, PP. 59–86 (2005)

ASPECT-ORIENTED CONSTRAINT MANAGEMENT IN METAMODEL-BASED MODEL TRANSFORMATION STEPS

László LENGYEL and Hassan CHARAF

Department of Automation and Applied Informatics
Budapest University of Technology and Economics
H–1521 Budapest, Hungary
e-mail: lengyel,hassan@aut.bme.hu

Received: Sept. 14, 2004

Abstract

A widely applied approach to model transformation uses graph rewriting as the underlying transformation technique. In case of diagrammatic languages, such as the Unified Modeling Language (UML), the exclusive topological matching is found to be not enough. To define the transformation steps precisely beyond the structure of the visual models, additional constraints must be specified which ensures the correctness of the attributes, or other properties to be enforced. Dealing with OCL constraints provides a solution for these unsolved issues. The use of OCL as a constraint and query language in modelling is essential. We have shown that it can be applied to model transformations as well. Often, the same constraint is repetitiously applied in many different places in a transformation. It would be beneficial to describe a common constraint in a modular manner, and to mark the places where it is to be applied. This paper discusses (i) the problem of crosscutting constraints in visual model transformation steps, and provides an aspect-oriented solution for a consistent constraint management. It introduces the concepts of aspect-oriented constraints and a new type of aspect, the constraint aspects. (ii) In general, it is difficult to require a whole transformation to validate, preserve or guarantee certain properties because transformations are built form isolated transformation steps ordered by a control structure. This problem is solved by the provided constraint weaver methods, which weave the constraints into the model transformation steps prior to the execution. (iii) Furthermore, the work presents offline constraint optimization (normalization) algorithms, which are part of the presented weaving process.

Keywords: aspect-oriented constraints, constraint normalization, constraint weaving, crosscutting constraints, metamodel-based model transformation, OCL.

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 platform-independent model (PIM), one or more platform-specific models (PSM) and complete implementations, one on each platform that the application developer intends to support. The platform-independent artefacts are mainly UML and other software models containing enough specification to generate the platform-dependent artefacts automatically by model compilers. Hence, software model transformation provides a basis for model compilers, which plays a central role in the MDA architecture.

The increasing demand for visual languages (VL) in software engineering (e.g., Unified Modeling Language - UML; Domain-Specific Languages - DSLs) requires more sophisticated transformation mechanisms for diagrammatic languages. Although these VLs can often be modelled with labelled, directed graphs, the complex attribute dependencies peculiar to the individual software engineering models cannot be treated with this general model. Consequently, often it is not enough to transform graphs based on the structural information only, we want to restrict the desired match by other properties, e.g. we want to match a node with a special integer type property whose value is between 2 and 12.

Previous work [2] has shown that the steps can be made more relevant to software engineering models if the metamodel-based specification of the transformations allows assigning OCL [3] constraints to the individual transformation steps. Because these constraints are bound to the transformation steps, they are able to express the constraints local to the host model area affected by the steps. This approach is inherently a local construct, because the elements not appearing in graph transformations cannot be directly included in the OCL statements. Although the specification has this local nature, this does not mean that validating them does not involve checking other graph elements in the input graph: constraint propagation needs to be taken into account by both the algorithms and the user of the transformation.

Often, the same constraint is repetitiously applied in many different places in a transformation; therefore it crosscuts the transformation steps. Aspect-Oriented Software Development (AOSD) [4] [5] provides a technique to address an emerging separation of concerns (SoC) that is focused on crosscutting. The methods of AOSD facilitate the modularization of crosscutting concerns within a system. Aspects may appear in any stage of the software development lifecycle (e.g. requirements, specification, design and implementation). Crosscutting concerns can range from high-level notions of security to low-level notions, like caching. Furthermore, functional requirements such as business rules and non-functional requirements, like transactions can also be expressed by aspects. AOSD has started on the programming level of the software development life-cycle, and over the last decade several aspect-oriented programming languages were introduced (e.g. AspectJ [6]). Aspect-oriented programming eliminates the crosscutting concerns on the programming language level, but the aspect-oriented techniques must also be applicable on a higher abstraction level. It would be beneficial to describe a common constraint in a modular manner, and propagate it automatically to the adequate places [7] [8].

The goal of this work is to use aspect-oriented methods in order to solve the problem of crosscutting constraints in metamodel-based transformation steps. The paper introduces the aspect-oriented (AO) constraints, provides an algorithm to create constraint aspects from them. A constraint aspect also contains a structure besides the textual conditions. Weaver algorithms are provided to propagate AO constraints and constraint aspects to model transformation steps. In addition, an algorithm to normalize constraint aspects and to obtain their (pure) canonical form is also discussed, which is an optimization part of the weaving method.

The approach, presented here, has made it possible to define constraints separately from the transformation steps, and facilitates specifying their propagation assignment to model transformation steps.

2. Backgrounds

Graph rewriting [9] is a powerful technique for graph transformation with a strong mathematical background. The atoms of the graph transformation are rewriting rules, each rewriting rule consists of a left-hand side graph (LHS) and a 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. Replacing means removing the elements that are in LHS but not in RHS, and gluing the elements that are in RHS but not in LHS.

Algebraic graph rewriting [9] provides a way to manipulate objects in a graph category, where the objects are labelled directed graphs, and the arrows are graph homomorphisms. There are two main branches of algebraic graph rewriting, namely, the double pushout (DPO) and the single pushout (SPO) approaches.

The DPO approach achieves the rule firing in two steps: after finding a redex (the part of the host graph matched by LHS of the rewriting rule), the first step removes the elements (vertices and edges) from the redex which are in the redex but not in the right-hand side graph. Then, as a second step, the elements of RHS graph not in LHS graph but in RHS graph are glued to the host graph. Related to the DPO approach a rather tutorial like description can be found in [10, 11, 12, 13], and a more complete summary in [9, 14].

The single pushout approach [9, 15, 16, 17, 18, 19] uses partial graph homomorphisms to form a single pushout as a rule firing condition, and if a conflict occurs when violating the gluing condition, deletion has priority over preservation.

Models can be considered special graphs, which contain nodes and edges between them. This mathematical background facilitates to treat models as labelled graphs and to apply graph transformation algorithms to models using graph rewriting. Previous work [2] has introduced an approach, where LHS and RHS of the rules are built from metamodel elements. This means that an instantiation of LHS must be found in the host graph instead of the isomorphic subgraph of LHS.

The Object Constraint Language (OCL) [3, 20] is a formal language for the analysis and design of software systems. It is the subset of the UML standard [21] 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. A precondition to an operation is a restriction that must hold at the moment that the operation is going to be executed. Similarly, a postcondition to an operation is a restriction that must hold at the moment that the operation has just ended its execution.

Aspect-oriented programming (AOP) [5] is based on the idea that computer systems are programmed in a better way by separately specifying the various crosscutting concerns of a system. Then the program relies on the mechanisms of the underlying AOP environment, which weaves or composes the separated pieces into a coherent program. AOP regards scattered concerns as first-class elements, and eject them horizontally from the object structure. A concern whose code becomes tangled into other structural elements becomes scattered. To ameliorate this problem, AOP offers the notion of aspects: mechanisms beyond subroutines and inheritance for localizing the expression of a crosscutting concern. AOP systems also provide some mechanism for weaving the aspects and the base code into a coherent system.

An aspect-oriented approach is introduced in [22] 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. Strategy aspect provides a hook that the weaver may call in order to process the node-specific constraint propagations.

The Visual Modeling and Transformation System (VMTS) [2] [23] is an implemented n-layer multipurpose modelling and metamodel-based transformation system. Using this environment, it is simple to edit metamodels, design models according to their metamodels, transform models using graph rewriting [2, 24]. Furthermore, it facilitates checking the constraints specified in the metamodel during the metamodel instantiation, and the transformation step constraints during the model transformation process.

VMTS has a Visual Control Flow Language (VCFL) which uses stereotyped activity diagrams to specify control flow structures and OCL constraints to choose between different control flow branches. In VCFL, if a transformation step fails and the next element in the control flow is a decision object then it could provide the next branch based on the OCL statements and the value of the *SystemLastRuleSucceed* system variable. If no decisions can be found, the control is transferred to the parent step, if there is no parent step, the transformation terminates with error.

In VMTS, applying a transformation step twice for an input model produces the same matches and the same transformation result in both cases. In practice, executing the same query on a database twice result the same data rows in both cases. The matching and transformation algorithms do not contain any random branch. Therefore we assume a deterministic matching and transformation process, constructing the theoretical background.

The results discussed in this paper have been validated in VMTS as a proof-of concept implementation.

The constraint validation method of the VMTS benefits from the results of the mathematical background of formal languages, graph rewriting, aspect-oriented software development and research related to the metamodel-based software model transformation. It also incorporates several ideas from other environments.

The GReAT framework [25] is a transformation system for domain specific languages (DSL) built on metamodeling and graph rewriting concepts. The LHS of the GReAT rules can contain OCL constraints to refine the pattern. PROGRES [26] is a visual programming language in the sense that it has a graph-oriented data model and a graphical syntax for its most important language constructs. In PROGRES the precondition and the postcondition of a transaction are queries. VIATRA [27] is a model transformation framework developed mainly for the formal dependability analysis of UML models. In VIATRA model constraints are graph patterns with arbitrary levels of negation. The Attributed Graph Grammar (AGG) [28] system is a Java-based visual programming environment. Constraints can be specified either visually or textually. The textual constraint expressions are provided in Java.

3. Contributions

This section, using aspect-oriented methods, presents how we can avoid repetitive constraints in model transformation steps, which are frequently scattered along the entire transformation. Furthermore, it provides algorithms for an aspect-oriented constraint management.

A *precondition* assigned to a transformation step is a Boolean expression that must hold at the moment when the step is fired, and a *postcondition* assigned to a step is a Boolean expression that must hold after the completion of a transformation step. If a *precondition* of a step is not true then the step fails without being fired. If a *postcondition* of a transformation step is not true after the execution of the step, then the transformation step fails. A direct corollary of this is that an OCL expression in LHS is a precondition to the transformation step, and an OCL expression in RHS is a postcondition to the transformation step.

There are three properties: *validation, preservation,* and *guarantee,* which are checked and required during the transformation 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.

In the VMTS approach the *pattern* is a model structure built from metamodel elements [2] which allows multiplicities on edges. During the matching process the type compatibility is required which takes into consideration the inherited types as well. The nodes contained by LHS and RHS graphs of the transformation steps are called *Pattern Rule Nodes* (*PRN*).

3.1. Crosscutting Constraints

A transformation consists of several steps, often not only a transformation step but a whole transformation is required to validate, preserve or guarantee a certain property. To meet this expectation all the transformation steps have to be taken into consideration. If one defines a constraint for more transformation steps or for a whole transformation, then the same constraint appears in the transformation numerous times.

Fore instance, we have a transformation which modifies the properties of the *Person* type objects and we would like the transformation to validate that the *age* of a *Person* is always under 200 (*Person.age* < *200*). It is certain that the transformation preserves this property if the constraint is defined for all PRN of type *Person*. This means that the constraint can appear in each transformation step several times. Therefore the constraint crosscuts the whole transformation. Its modification and deletion is not consistent, because such an operation must be performed on all occurrences of the constraint. Moreover, it is often difficult to estimate the effects of a complex constraint when it is scattered across the numerous PRNs of the transformation steps [7].

Fig. 1 introduces a metamodel and a transformation with 2 transformation steps and crosscutting constraints. The *const_age* constraint appears at several places in this transformation. *Fig. 1* shows a concrete case where a constraint is present in many places of the two transformation steps. Another example constraint could be that we require from the transformation to preserve that the number of employees of a *Company* is always between 50 and qnewline 300 ($50 \leq$ *Company.NumbeOfEmployees* ≤ 300).

3.2. Aspect-Oriented Constraints

A disadvantage of our earlier metamodel-based model transformation approach [29] can be seen in many tangling constraints throughout of our transformation steps. We need a mechanism to separate this concern. Having separated the constraints from the PRNs, we need a weaver method which facilitates the propagation (linking) of constraints to PRNs. The Global Constraint Weaver algorithm (presented in Section 3.3) is supplied with a transformation with optional number of transformation steps and a constraint list, and it links the constraints to the PRNs contained by the transformation steps.

This method means that our approach manages constraints using AO techniques [8]. Similarly to aspects, the constraints are specified and stored independently of any model transformation step or PRN and are linked to the PRNs by the Global Constraint Weaver (GCW). *Fig. 2* introduces the weaving process.

The output of the weaver is not stored as a new transformation step; the result is handled as a linking between the constraints and a transformation step. This linking is referred to as *Weaving Configuration*.

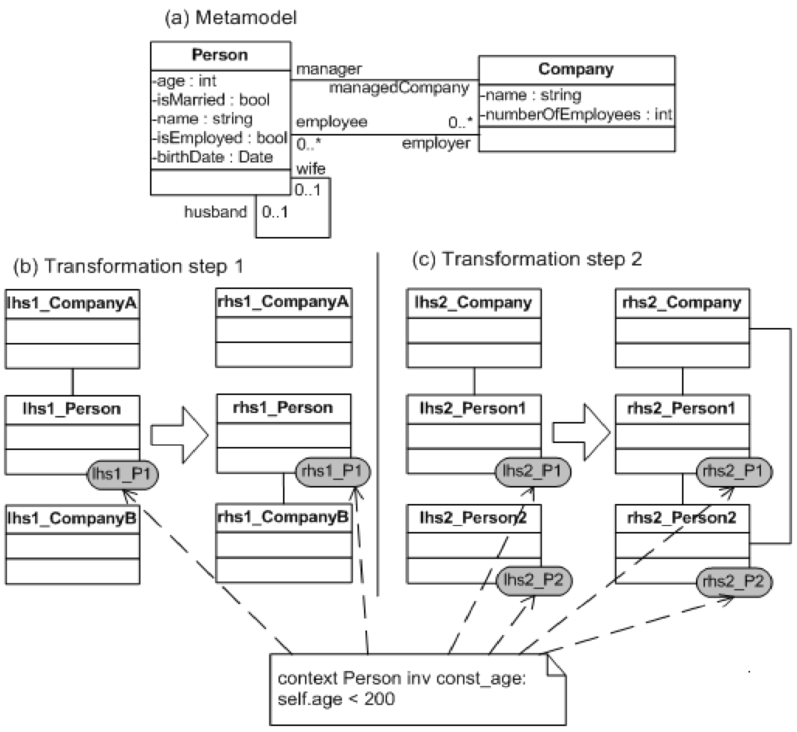

Fig. 1. (a) Example metamodel, (b-c) Two transformation steps built from metamodel types with crosscutting constraints

To summarize the main idea of the AO constraints, we can say that one can create the constraints and the transformation steps separately. Then, with the help of a weaver, constraints can be propagated to the PRNs contained by the transformation steps.

3.3. Constraint Aspects

Before we introduce the concepts and algorithms that are used to manage AO constraints for the unified treatment, we give our definitions.

Definition 1 *(Constraint Aspect):*

A Constraint Aspect is a pattern (structure) built from metamodel elements to which OCL constraints are propagated.

A Constraint Aspect contains not only textual conditions described by the OCL constraints but structure, type and multiplicity conditions as well. The structure

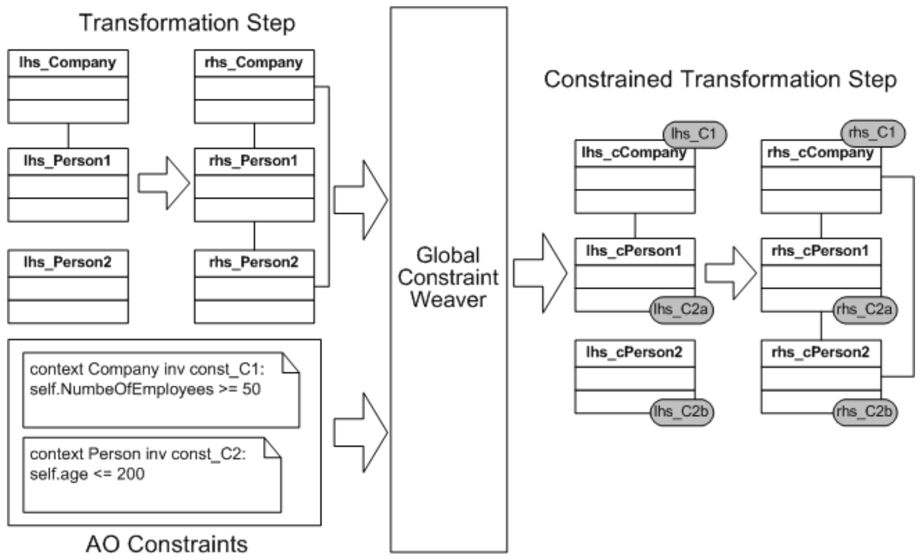

Fig. 2. The Weaving process / a concrete weaving example

and type conditions are checked at propagation time, while the OCL constraints are validated during the transformation.

Definition 2 *(Propagated Constraint, Propagated Constraint Aspect):*

A Propagated Constraint is an OCL constraint linked to a transformation step. It forms a Weaving Configuration that contains the OCL constraint and the transformation step.

A Propagated Constraint Aspect is a Constraint Aspect linked to a transformation step. It forms a Weaving Configuration that contains the Constraint Aspect and the transformation step.

Definition 3 *(Normalized Constraint Aspect - Canonical Constraint Form and Pure Canonical Constraint Form):*

The Canonical Constraint Form of a Constraint Aspect is the form which contains the fewest possible navigation steps.

The Pure Canonical Constraint Form of a Constraint Aspect is the form which does not contain navigation steps.

A constraint aspect is graphical, therefore it fits better the visual transformation system as the OCL constraint. *Fig. 3* introduces the concepts of the formulated definitions.

In *Fig. 3a* a constraint aspect is depicted: *Person1*, *Person2*, *Company* and the associations between them represents the structure of the constraint aspect. *C_P1* and *C_C1* are the OCL constraints propagated to the pattern of the constraint aspect.

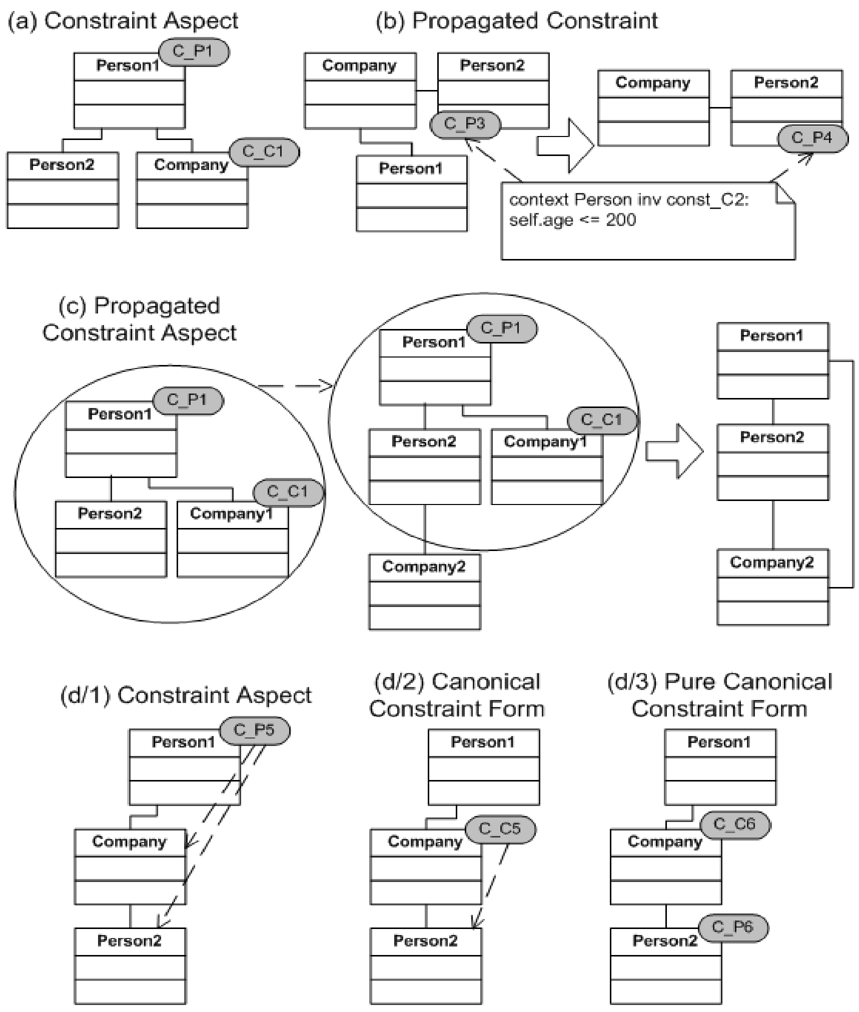

Fig. 3. (a) Constraint Aspect, (b) Propagated Constraint, (c) Propagated Constraint Aspect, (d/1) Constraint Aspect, (d/2) Canonical Constraint Form, (d/3) Pure Canonical Constraint Form

Fig. 3b shows a transformation step with a propagated OCL constraint (*const_C2*), which appears both in LHS and RHS graphs (*C_P3, C_P4*). *Fig. 3c* represents a transformation step, a constraint aspect and the dashed line shows the propagation of the constraint aspect to LHS of the transformation step.

In *Fig. 3d*/1, a constraint aspect is depicted with a propagated OCL constraint (*C_P5*). The dashed lines denote that *C_P5* refers to the *Company* and *Person2* nodes. *Fig. 3d*/2 shows the Canonical Constraint Form of the constraint aspect depicted in *Fig. 3d*/1. The propagated OCL constraint is relocated to the *Company* node and the navigation steps of the constraint are updated. Therefore the *C_C5* constraint refers to the *Person2* node only. The *C_C5* constraint contains the fewest possible navigation steps. *Fig. 3d*/3 depicts the Pure Canonical Constraint Form of the constraint aspect (*Fig. 3d*/1). The *C_P5* OCL constraint is divided into two simpler constraints (*C_C6* and *C_P6*), which are assigned to the *Company* and *Person2* nodes. *C_C6* and *C_P6* constraints do not contain navigation steps.

3.4. The Global Constraint Weaver Algorithm

The Global Constraint Weaver algorithm weaves AO constraints into the model transformation steps. It has three main steps to select the PRNs to which the constraint must be linked. (i) It obtains the PRNs from the transformation steps with metatype corresponding to the context information of the constraint. (ii) If the constraint contains navigation steps, the algorithm checks the structure of the transformation step for each previously selected PRN whether it satisfies the pattern required by the navigation paths. The pattern of the transformation step is suitable for a constraint if starting from the PRN to which the constraint is linked, one can walk through the paths described by the navigation paths of the constraint. In other words, the pattern of the transformation step contains nodes with appropriate metatype, and the relation between them which facilitates to traverse the navigation paths contained by the constraint. (iii) In the third step, examining the transformation steps, the algorithm decides if it is necessary to assign a constraint to each transformation step, or it is sufficient to assign it only to the first step as a precondition and to the last step as a postcondition. If an intermediate state modifies one of the properties contained by the constraint, the algorithm assigns the constraint to this intermediate state to prevent a violated condition not being revealed until the end of the transformation. The pseudo code of the GCW algorithm is as follows.

The following proposition states that the Global Constraint Weaver propagates constraints only to the necessary places and optimizes the constraint validation for the whole transformation process.

Proposition 1 *T_1 and T_2 are two identical transformations which contain optional number of transformation steps, andC is an OCL constraint. (i) C is propagated to the T_1 using Global Constraint Weaver algorithm, and (ii) based on the required transformation type (validation, preservation or guarantee), C is enlisted in all*

```
GlobalConstraintWeaver(Constraint[]Cs)Transformation T)
1 foreach Constraint C in Cs
2  foreach Transformation step S in T
3    nodesWithProperMetaType = MetaTypeBasedSearching(ContextInfo of C, S)
4    nodesToCheck = CheckStructure (nodesWithProperMetaType, S, C)
5    if(IsRequiredToWeave(C, nodesToCheck, out nodesToWeave, true)) then
6      WeaveConstraint(C, nodesToWeave )
7    endif
8  end foreach
9 end foreach
```

```
IsRequiredToWeave(Constraint C, VMTSRuleNode[] nodesToCheck,
out VMTSRuleNode[] nodesToWeave): bool
1 requiredToWeave = false
2 foreach VMTSRuleNode ruleNode in nodesToCheck
3    if (the step of the ruleNode is first step in the transformation or
the step of the ruleNode is last step in the transformation or
the step of the ruleNode can modify a condition contained by the C) then
4        ruleNode add to nodesToWeave
5        requiredToWeave = true
6    endif
7 end foreach
8 return requiredToWeave
```

transformation steps of T_2. Then T_1 and T_2 transformations produce the same model as a result step by step.

Proof *Appendix*

Corollary 1 *T_1 and T_2 transformations step by step produce the same result model, therefore the wholeT_1 and T_2 transformations have the same effect in all cases during the rewriting process.*

Remark 1 *VMTS executes the transformations in transactions. This means that if a transformation is unsuccessful, VMTS rolls back the changes made on the input model, hence, we have the original state of the model before the transformation. Consequently, if T_1 transformation fails in the step 1 and T_2 transformation fails in the step n, then because of the rollback at the end the T_1 and T_2 transformations have the same effect.*

3.4.1. Computational Complexity Considerations

In VMTS, the data is stored in datasets, which are in-memory cache of the data retrieved from the database. There is a dataset for the currently used metamodel, one for the currently used model, and one for the currently used transformation. These datasets contain only two data tables: one for nodes and one for edges. We denote the number of the actual transformation (metamodel, model) nodes with v_r (v_m, v) and the edges with e_r (e_m, e). This means that the data tables containing the actual transformation nodes and edges have v_r and e_r data rows. The complexity of finding a row in a data table is O(lg n), where n is the number of the rows. Therefore the complexity of querying a *VMTSRuleNode* or a *VMTSRuleEdge* from the dataset of the current transformation is O(lg v_r) and O(lg e_r) steps. Similarly to get a metamodel (model) node or edge means O(lg v_m) and O(lg e_m) (O(lg v) and O(lg e)) steps.

The complexity of constraint validation and navigation.

The complexity of the constraint validation is O(1) [29]. During the navigation the cost of each navigation step is O(lg ev) if the navigation path contains only multiplicities 0..1 or 1. It is a query on the table which contains the model edges to obtain the node IDs of the appropriate adjacent nodes (O(lg e)) and we have to select the adjacent nodes by their IDs (O(lg v) for each node): O(lg $e + m$*lg v) = O(lg e+lg v^m) = O(lg ev^m), where m is the number of the selected adjacent nodes. If m is 1 then it is O(lg ev).

If the navigation path contains multiplicities like 0..*, the result of a query can be a collection of nodes (maximum v nodes) instead of a simple node, thus, the next step must work on the result of the previous step. Traversing the navigation paths, the metatype of the nodes need to be taken into consideration, because it reduces the complexity of the whole evaluation process.

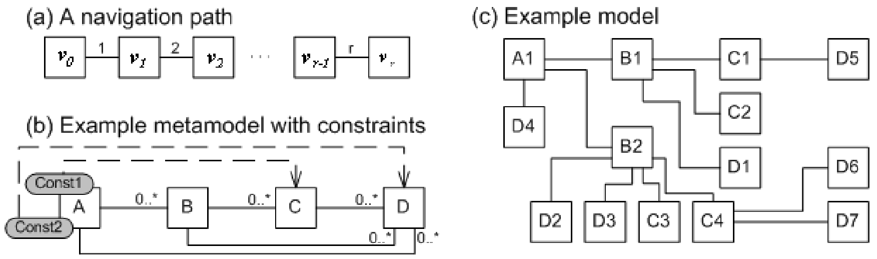

Fig. 4. (a) A navigation path, (b) Example metamodel with constraints, (c) Example model

An OCL constraint C contains optional number of navigation paths. A navigation path P is depicted in *Fig. 4a*, P contains r navigation steps and traverses

$r+1$ nodes (v_0, $v_1 ... v_r$). Let $metatype(v_i)$ denote the collection of the model nodes in the input model with metatype v_i, where $0 \leq i \leq n$, and n is the number of the nodes in the input model. Let R_i be a node collection resulted by the navigation step$i$. Since R_0 is the start node collection of the constraint C, this means that the number of the nodes in collection R_0 is 1 ($\#R_0 = 1, R_0 = \{v_0\}$). Using *SelectNodes* method we can obtain the result node collection of the navigation step$i$, based on the result of the navigation step$i-1$ and the nodes with metatype of the path node$i$: $R_i = SelectNodes(R_{i-1}, metatype(v_i))$. The maximum number of the nodes in the collection $\#R_i$ is $\#R_i = \#R_{i-1} * \#metatype(v_i)$, and the cost of the navigation step i is $Cost_{R_i} = \#R_{i-1} * \lg e + \#R_i * \lg v = \lg e^{\#R_{i-1}} v^{\#R_i}$. In summary, the cost of the whole navigation is maximum

$$Cost_R = \lg ev + \sum_{i=1}^{r} \#R_{i-1} * (\lg e + \#metatype(v_i) * \lg v). \qquad (1)$$

The first part of the recursively given formula ($\lg ev$) is the cost of the step 0(to select the start node), and the second part is the cost of the steps from 1 up to r. There are r navigation steps, in the navigation step i there is $\#R_{i-1}$ queries on the table of edges and $\#R_i = \#R_{i-1} * \#metatype(v_i)$selects on the table of nodes.

The formula (1) is only an approximation, it is the maximum value of the real cost of the navigation. This approach utilizes the different metatypes of the nodes, therefore the worst case is if all the model nodes have the same metatype.

To better approximate the real cost of the navigation we define the $metatype$ $Neighbour(v_{ij})$ operation which retrieves not all of the model nodes with the metatype v_i, but only the adjacent nodes of the collection node j with metatype v_i. Based on it and on $\#R_i^{\cdot} = \sum_{j=1}^{\#R_{i-1}} \#metatypeNeighbour(v_{ij})$, the modified formula is

$$Cost_{R^{\cdot}} = \lg ev + \sum_{i=1}^{r} \#R_{i-1}^{\cdot} * \lg e + \sum_{j=1}^{\#R_{i-1}^{\cdot}} \#metatypeNeighbour(v_{ij}) * \lg v = \lg ev + \sum_{i=1}^{r} \#R_{i-1}^{\cdot} * \lg e + \#R_i^{\cdot} * \lg v. \qquad (2)$$

The complexity of the GLOBALCONSTRAINTWEAVER *algorithm.*

The complexity of the METATYPEBASEDSEARCHING method is O(lg v_r) [30], where v_r is the number of the PRNs contained by the current transformation. The computational complexity of the CHECKSTRUCTURE method is the number of the PRNs with the meta type which corresponds to the context information of the actual constraint multiplied by the complexity of finding the subgraph in the actual transformation step: O(#*nodesWithProperMetaType* * v_r^h), where h is the number of subgraph nodes the algorithm has to match (this subgraph is specified by the navigation steps of the actual constraint). The computational complexity of the ISREQUIREDTOWEAVE method is the number of the PRNs that the constraint must be checked on: O(#*nodesToCheck*). The computational complexity of the WEAVECONSTRAINT method is the number of the PRNs the constraint is propagated to: O(#*nodesToWeave*). The cost of the GLOBALCONSTRAINTWEAVER algorithm is at mostO($\sum_{i=1}^{c}\sum_{j=1}^{s}(\lg v_r + n_{ij} * v_r^h + n_{ij})$), where c denotes the number of the constraints to propagate, s is the number of the transformation steps, n_{ij} is the number of the PRNs in the transformation step$j$ with the same metatype as the context of the constraint$i$. In this case the v_r^h is the worst case for finding the subgraph [9, 31].

Evaluating a constraint that contains navigations requires more computational complexity than a constraint without any navigation steps. To resolve the problem of the navigation we can create a constraint aspect from an OCL constraint and normalize (optimize) it in order to obtain the Canonical Constraint Form. The complexity of creating the constraint aspect from an OCL constraint and its normalization equals to the evaluation of the OCL constraint. If we take the original OCL constraint without normalization, we must traverse the navigation paths at each evaluation of the OCL constraint. A normalized constraint aspect incorporates the constraints on their adequate places; therefore it includes the fewest possible navigation steps. Thus, we need not to execute additional query operations during the evaluation of the normalized constraint aspects.

3.5. The Constraint Aspect Weaver Algorithm

The constraint aspect weaver method consists of a constraint aspect creation from OCL constraint, an optimization part and the actual constraint assignment. Section 3.4.1 presents how to build a structure from an OCL constraint and propagate the constraint to it. This results a constraint aspect. Section 3.4.2 discusses the constraint aspect optimization to achieve that the cost of the constraint evaluation during the transformation is as minimal as it is possible. Section 3.4.3 provides the algorithms that VMTS uses for constraint aspect assignment. Furthermore, Section 3.5 gives an overview on the presented aspect-oriented constraint notions and algorithms.

3.5.1. Creating Constraint Aspect from OCL Constraint

Fig. 5 introduces the creation process of the constraint aspects from an OCL constraint (AO constraint), and the normalization of the created constraint aspect. The lines with numbers from 1 to 4 show the steps of the constraint aspect creation: (i) the algorithm identifies the context type (*Person*), (ii) and the referred types by the association ends (*managedCompany – Company*, *wife – Person*), based on this information it builds the pattern, and finally, (iii) the algorithm assigns the OCL constraint to the root PRN of the created constraint aspect. In *Fig. 5b*, the dashed lines denote that the constraint *C_P1* contains path expressions. Lines 5 and 6 show the constraint decomposition and replacement.

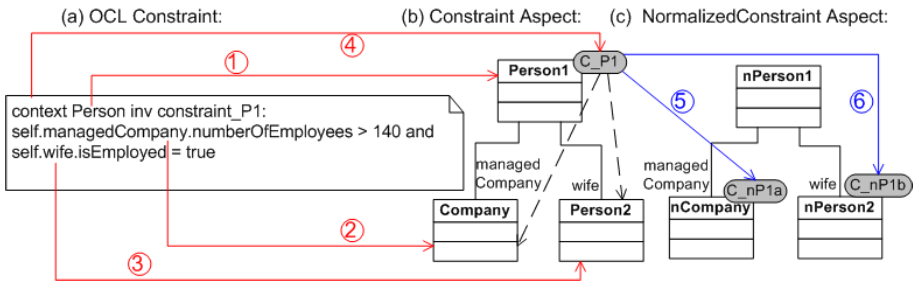

Fig. 5. (a) OCL Constraint, (b) Generated Constraint Aspect, (c) Normalized Constraint Aspect – Canonical Constraint Form

The pseudo code of the CREATECONSTRAINTASPECT algorithm is as follows.

```
CREATECONSTRAINTASPECT (Constraint C): VMTSConstraintAspect
 1 CREATE CA by Context of C
 2 foreach Navigation Step N in C
 3     CREATE PRN Pattern Rule Node with type of the DestinationNode of the N
 4     LINK PRN to CA
 5 end foreach
 6 PROPAGATE Cto the root node of the CA
 7 return CA
```

The computational complexity of the CREATECONSTRAINTASPECT method is O(n), where n is the number of the navigation steps contained by the constraint.

3.5.2. *Constraint Aspect Normalization*

The normalization process eliminates navigation steps from the constraint aspect using constraint decomposition and relocation, therefore it decreases the evaluation time of the constraint during the transformation. The constraint evaluation consists of two parts. (i) Selecting the object and its properties that VMTS needs to check against the constraint, and (ii) executing the validation method. The eliminated navigation steps accelerate the first part of the constraint evaluation. *Fig. 6* presents two constraint aspects and their Pure Canonical Constraint Form (normalized constraint aspects). The dashed lines denote that the constraints *C_P1* and *C_C1* contain path expressions, and numbered line in *Fig. 6a* shows the update process of the navigation paths and the constraint relocation. In *Fig. 6c* presents the result of the normalized constraint *C_C1*, which is achieved with constraint decomposition.

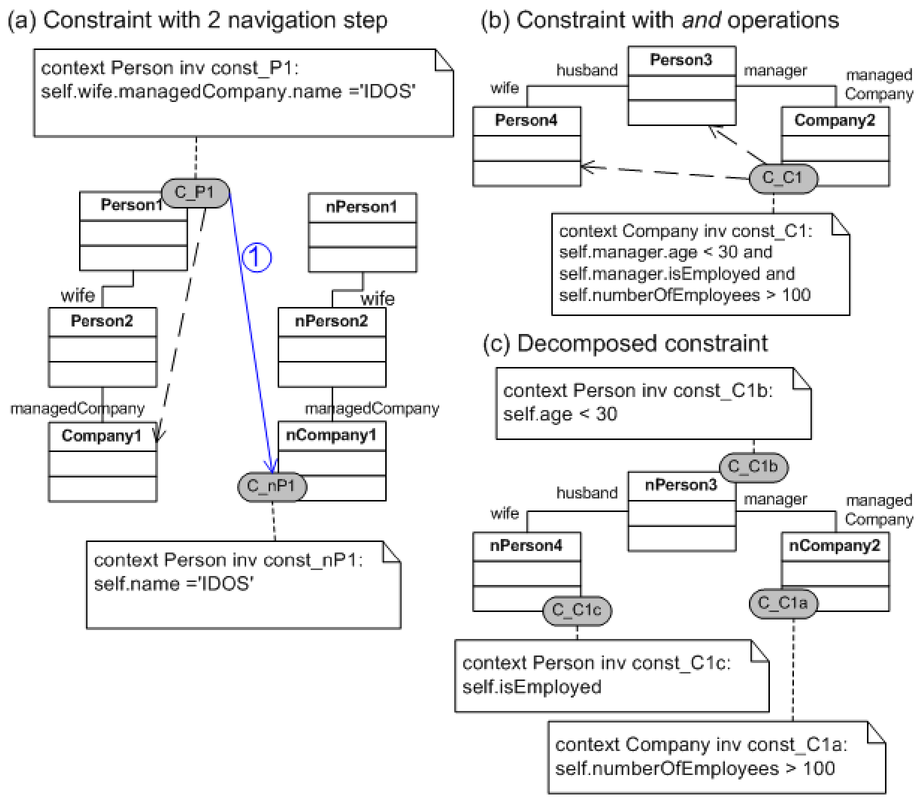
(a) Constraint with 2 navigation step
context Person inv const_P1:
self.wife.managedCompany.name ='IDOS'
Person1
C_P1
wife
Person2
managedCompany
Company1
1
nPerson1
wife
nPerson2
managedCompany
nCompany1
C_nP1
context Person inv const_nP1:
self.name ='IDOS'
(b) Constraint with and operations
husband
Person3
manager
managed Company
wife
Person4
Company2
C_C1
context Company inv const_C1:
self.manager.age < 30 and
self.manager.isEmployed and
self.numberOfEmployees > 100
(c) Decomposed constraint
context Person inv const_C1b:
self.age < 30
C_C1b
husband
nPerson3
manager
managed Company
wife
nPerson4
nCompany2
C_C1c
C_C1a
context Person inv const_C1c:
self.isEmployed
context Company inv const_C1a:
self.numberOfEmployees > 100

Fig. 6. Constraint Aspects and Normalized Constraint Aspects: (a) Propagated constraint with 2 navigation step, (b) Propagated constraint with *and* operations, (c) Decomposed constraint

The pseudo code of the NORMALIZECONSTRAINTASPECT algorithm, which generates the (pure) canonical form of the constraint aspect, is as follows.

During constraint normalization, the NORMALIZECONSTRAINTASPECT algorithm modifies the navigation paths. It calculates the paths between the original place of the constraint as well as the referred node (path $P1$), and between the new place of the constraint and the referred node (path $P2$). Finally, the algorithm replaces the path $P1$ with the path $P2$. In *Fig. 6a* the path *self.wife.managedCompany* is replaced with path *self*.

```
NormalizeConstraintAspect (ConstraintAspect CA)
1  foreach PropagatedConstraint C in CA which contains navigation
2    if (C.DestinationNodes.Count == 1) then
3      Relocate C to DestinationNode of the C
4    else if (C contains only and operation) then
         DecomposeAndRelocateConstraint(C)
5    else
6      minNumberOfSteps = CalculateNavigationSteps
         (DestinationNodes of the C, CurrentNode of the C)
7      optimalNode = CurrentNode of the C
8      foreach Pattern Rule Node PRN
         in PatternRuleNodes of the CA
9        numberOfSteps = CalculateNavigationSteps
           (DestinationNodes of the C, PRN)
10       if (numberOfSteps < minNumberOfSteps) then
11         minNumberOfSteps = numberOfSteps
12         optimalNode = PRN
13       endif
14     end foreach
15     if (optimalNode ! = CurrentNode of the
         C and there is only multiplicity 1 on the path between optimalNode and CurrentNode) then
16       UpdateNavigationPats of the C
17       Relocate C to optimalNode
18     endif
19   endif
20 end foreach
```

If the multiplicity 0 is allowed on the path P that is selected to be replaced in constraint C then the NormalizeConstraintAspect algorithm does not replace the constraint C. This means that the constraints are not moved along edges which allow the multiplicity 0.

In *Fig. 7a*, a sample LHS graph is depicted with multiplicity 0..*. *Fig. 7b* shows an example for a matched host model. The constraint *C_P1* is assigned to the node *Person1* and refers to the node *Person2*. If the constraint *C_P1* is relocated to *Person2* then it is validated on the node *P2_a*. But if the constraint *C_P1* is on its original place, then it does not need to be validated on *P2_a* because there is no edge between nodes *P1_a* and *C1_a*. Consequently, moving a constraint along an edge which allows multiplicity 0 may cause that the constraint validation becomes incorrect. It is possible that (i) a constraint C does not need to be validated on its original place but after the replacement it must be checked or (ii) the constraint C needs to be validated on its original place but because of the replacement it is not checked.

The normalization is an offline algorithm: it works without any input model.

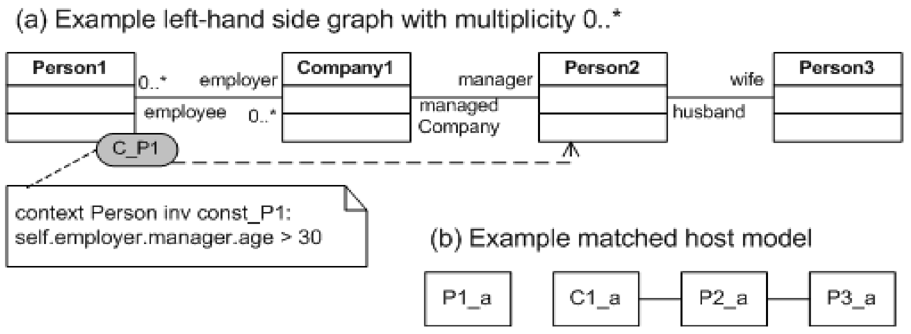
(a) Example left-hand side graph with multiplicity 0..*
Person1
0..*
employer
Company1
manager
Person2
wife
Person3
employee
0..*
managed Company
husband
C_P1
context Person inv const_P1:
self.employer.manager.age > 30
(b) Example matched host model
P1_a
C1_a
P2_a
P3_a

Fig. 7. Example left-hand sides (a) with 1..* multiplicity and a matched host model, (b) with 0..* multiplicity and a matched host model

In the case of complex multiplicities, like 1..*, it is not possible to predict how many instances are in the host model and in the matched sub-model of the certain types. In the presented approach it is not necessary to traverse the navigation paths for each matched instance element. Each navigation path is parsed once, similarly to the case with multiplicity 1, and the constraint validation is performed on the collections of the nodes. In each navigation step, the result can be a node collection not only a simple node. This collection is the input of the next navigation step, and the nodes of this collection are validated without any extra navigation. In this case, if not only the multiplicity 1 is allowed, it is possible that after normalization the cost of the constraint evaluation increases. The reason for this is that a constraint navigating along 1..* edges can be more complex, than another OCL constraint that uses navigations along edges with 1 multiplicities. An example for it is depicted in Fig. 8. The number of the navigation steps is 3 in the example LHS (Fig. 8a), and 2 in the normalized LHS (*Fig. 8b*). But the cost of the constraint validation on the example matched host graph is higher in case of the normalized version, because of the allowed 1..* multiplicities.

As far as complex multiplicities are concerned, the optimization depends on the actual input model. As it has been mentioned, the normalization is an offline algorithm, it is not possible to state further normalization solution for the normalized models which contain complex multiplicities.

The computational complexity of the NORMALIZECONSTRAINTASPECT algorithm is $O(\sum_{i=1}^{c} n_i + v^3)$, where c denotes the number of the propagated constraints contained by the constraint aspect, n_i is the number of the navigation steps contained by the constraint i (it denotes the complexity of finding the destination nodes), and$v$ denotes the number of the PRNs in the constraint aspect. The complexity of finding the shortest path in the constraint aspect *is* v^2. We have to execute it v times (for each PRN in constraint aspect).

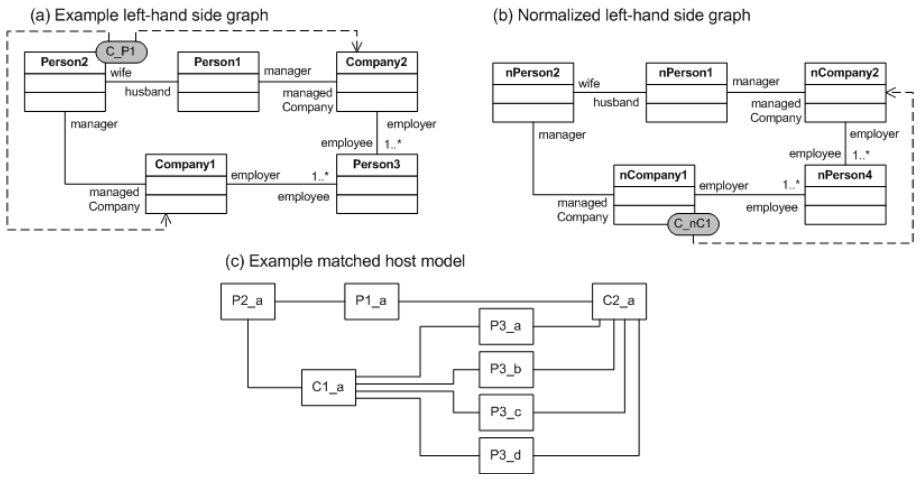

Fig. 8. (a) Example left-hand side graph, (b) Normalized left-hand side graph, (c) Example matched host model

The NORMALIZECONSTRAINTASPECT algorithm is applicable for models without multiplicities 0 and results that the number of the navigation steps in normalized constraints is as few as possible.

Proposition 2 *Applying the* NORMALIZECONSTRAINTASPECT *algorithm for an optional input model H without multiplicities 0, the number of the navigation steps in the constraints contained by the output modelH' is minimal.*

Proof: **Appendix.**

Informally, if the constraint aspect *CA* is created from an OCL constraint *C* using CREATECONSTRAINTASPECT algorithm, and the normalized constraint aspect *CA'* is created from the *CA* with NORMALIZECONSTRAINTASPECT algorithm, then *C*, *CA* and *CA'* are equivalent in the sense of the contained conditions. Furthermore, after their propagation to the transformations the constrained transformations are also equivalent during the transformation.

Proposition 3 *i The constraint aspect CA is created with the* CREATECONSTRAINTASPECT *algorithm from the OCL constraintC. S_1 and S_2 are two identical transformation steps, CA is propagated to S_1, and C is propagated to S_2. Then no input model H exists for which transformation steps S_1 and S_2 produce different result models.*

ii The normalized constraint aspect CA' is created with the NORMALIZECONSTRAINTASPECT *algorithm from the CA constraint aspect. S_1 and S_2 are two identical transformation steps, CA' is propagated to S_1, and CA is prop-*

agated to S_2. Then no input model H exists for which transformation steps S_1 and S_2 produce different result models.

iii *The normalized constraint aspect CA' is created from the OCL constraint C with the* CREATECONSTRAINTASPECT *and* NORMALIZECONSTRAINTASPECT *algorithms. S_1 and S_2 are two identical transformation steps, CA' is propagated to S_1, and C is propagated to S_2. Based on (i) and (ii), no input model H exists for which the transformation steps S_1 and S_2 produce different result models.*

Proof: **Appendix.**

3.5.3. The Constraint Aspect Weaving

The Constraint Aspect Weaver (CAW) algorithm weaves constraint aspects into model transformations. CAW uses similar methods as GCW which weaves AO constraints to transformations. The pseudo code of the CAW algorithm is as follows.

```
CONSTRAINTASPECTWEAVER(ConstraintAspect[] CAs, Transformation T)
 1 foreach ConstraintAspect CA in CAs
 2 foreach Transformation Step S in T
 3 matches = METATYPEBASEDMATCHING(pattern of the CA, S)
 4 foreach Constraint C in CA
 5 nodesToCheck = GETNODESBYTYPE(ContextType of C, matches)
 6 if (ISREQUIREDTOWEAVE(C, nodesToCheck, out nodesToWeave)) then
 7 WEAVECONSTRAINTASPECT(CA, nodesToWeave)
 8 break
 9 endif
 10 end foreach
 11 end foreach
 12 end foreach
```

The computational complexity of the CONSTRAINTASPECTWEAVER method is at most $O(\sum_{i=1}^{ca}\sum_{j=1}^{s}(n_j^{k_i}+\sum_{p=1}^{c_i}m_{jp}))$, where *ca* denotes the number of the constraint aspects, s is the number of the transformation steps, c_i is the number of the constraints contained by the constraint aspect i, $n_j^{k_i}$ is the complexity of the metatype-based search (worst case) [30], n_j is the number of PRNs contained by the transformation step j and k_i is the number of PRNs contained by the constraint aspect i. Variable m_{jp} is the number of the PRNs selected by the GETNODESBYTYPE method.

In summary, it is more efficient to work with constraint aspects than with OCL constraints, because during the propagation of the constraint aspects we can use the metatype-based search for pattern matching to reduce the possible places of the constraint assignment. Then we run the ISREQUIREDTOWEAVE algorithm on the selected places only to decide if it is necessary to assign the constraint aspect. In case of simple OCL constraints, the GCW algorithm uses the CHECKSTRUCTURE method to find the structurally appropriate places in the transformation steps. Based on [30] the complexity of the metatype-based searching in the worst case is equal to the complexity of the subgraph search.

Informally, assume that an AO constraint *C* and a constraint aspect *CA* express the same conditions, if *C* is propagated to a transformation with the GCW and *CA* with the CAW to the same transformation then the results of the two constraint propagation (the constrained transformations) are equivalent.

Proposition 4. The normalized constraint aspect *CA'* is created from the OCL constraint *C* with the CREATECONSTRAINTASPECT and NORMALIZECONSTRAINT-ASPECT algorithms. T_1 and T_2 are two identical transformations, which contain arbitrary number of transformation steps. *C* is propagated to T_1 using the Global Constraint Weaver algorithm and *CA'* is propagated to T_2 using the Constraint Aspect Weaver algorithm. Then the transformations T_1 and T_2 produce the same result models step by step.

Proof: **Appendix.**

3.6. Overview

An overview of different aspect-oriented constraint notions and presented algorithms is depicted in *Fig. 9*.

The constraint aspect creation and normalization need to be accomplished once for a constraint and the weaving needs to be performed once for a transformation. They obviously take time, but they are all offline algorithms, prior to execution. After they are accomplished once, their results, the constrained transformations, can be reused optional time during the transformation process.

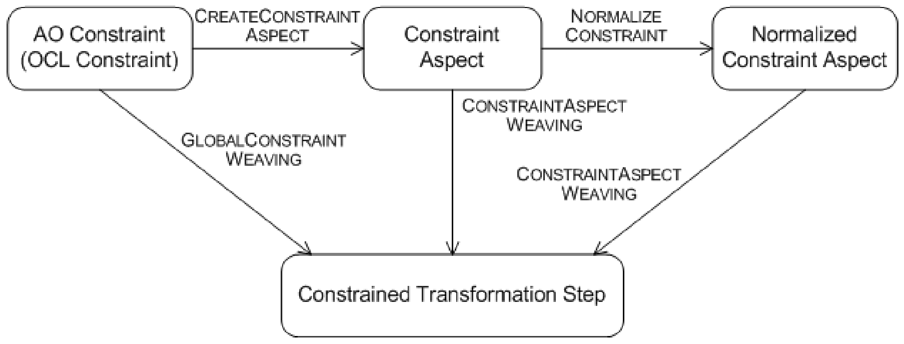

Fig. 9. An overview of different aspect-oriented constraint notions and presented algorithms

Normalized constraint aspects contain the constraints on their adequate places, with as few navigation steps as it is possible. Therefore, for the transformation steps which allow only the multiplicity 1 we can state the following. As far as the execution time is concerned, the evaluation of the constraints contained by the propagated normalized constraint aspect is more efficient, than the evaluation of the simple propagated OCL constraint, because the normalized constraint aspect contains as many navigation steps as the OCL constraint at most.

The most important benefits of the aspect-oriented constraint management in metamodel-based model transformations among others are the following: the same constraint does not appear repetitiously in many different places, consistent constraint modification and simple constraint removal.

The aspect-oriented constraint management does not replace the classical constraint assignment; it extends the possibilities of constraint handling in metamodel-based model transformation frameworks.

4. Conclusions

In metamodel-based model transformation methods, the two main advantages of the AO constraint management are the following. (i) It eliminates the crosscutting constraints from model transformations. (ii) Using AO methods, constraints become aspects. This means that the transformation steps can be executed with or without the propagated constraints as well. Moreover, the optimized transformation steps and constraints can be reused. Hence, the transformation can be executed with different propagated constraint set based on the required conditions. Furthermore, constraints are defined and stored independently from the transformations. Therefore they can be propagated to different transformations, thus, the constraint themselves can also be reused.

The problem of crosscutting constraints in metamodel-based model transformation steps, and an aspect-oriented solution has been provided for the open issue. The concept of AO constraints and a new type of aspect, constraint aspect, has been introduced. In addition, the algorithms creating and normalizing the constraint aspects have been discussed, and two weaver methods are also presented for AO constraint and constraint aspect propagation.

We have found that the source of our rewriting problems is often related to the lack of support for modularizing crosscutting concerns. As we have adopted an aspect-oriented approach to our metamodel-based transformation process, it was observed that the maintainability and understandability of our transformation steps have been increased along with the attached constraints. With the help of this technique we achieved several benefits. Consistent constraint modification and simple constraint removal has become possible. The same constraint does not appear repetitiously in many different places. Moreover, it is not necessary for the transformation steps to be aware of the constraints, or for the modeler who creates the transformation steps. The provided weaver algorithms work on whole transformation, they handle all transformation steps contained by the transformation. Therefore, the result of the weaving, the constrained transformation as a whole satisfies the conditions required by the constraints. The discussed methods have successfully been applied in industrial developments like generating user interface from resource model and user interface handler code from statechart model for Symbian [7] and .NET Compact Framework mobile platform [8].

The introduced approach can be generalized to other transformation languages which facilitate to assign constraints to transformation steps. The presented concepts and algorithms can be reused with minor, approach related modifications.

The main limitation of the aspect-oriented constraint management is that it requires more preprocessing steps than the general approach. We have to define the constraints and transformation steps separately, and then the propagation of the constraints to the transformation steps must be performed automatically.

Future work includes the completion of the normalization algorithm with the concept of AND/OR Clauses [32] to eliminate all navigation steps (except if a constraint requires property values from different nodes (e.g. *self.age + self.husband.age > 60*)). AND/OR Clauses are linked to the appropriate PRNs without any navigation steps, therefore they solve the cost problem of navigating across the complex multiplicities.

5. Appendix

Proof for Proposition 1. Assuming two identical host graphs (H_1 and H_2), one applies T_1 to H_1 andT_2 to H_2. After each transformation step one compares the success of the actual transformation steps and in the step$n$ stepsT_{1n} and T_{2n} produce different results - one of them is successful but the other fails because of a constraint failure.

We have to differentiate between two cases: (i) If the step n modifies the property checked by the constraint C, then the constraint C is propagated to the transformation T_{1n} by the GCW algorithm. Hence both transformations (T_1 and T_2) contain the constraint$C$ in the step$n$, therefore the result of the T_{1n} and T_{2n} steps can not be different. (ii) If the step n does not modify the checked property, (the T_{1n} does not contain the constraint C) the constraint evaluation in T_{2n} step can not be unsuccessful. This contradicts the assumption.

Proof for Proposition 2. H is an optional input model (UML class diagram), let C be an OCL constraint which is propagated to the H. Running the NORMALIZECONSTRAINTASPECT algorithm, it results that the A is the optimal node to which the OCL constraint Cshould be assigned. But, assuming that there exists another node (B) for which the following holds: if one links the C constraint to the node B and updates the navigation paths of C, then the C constraint contains less navigation steps than if it had been propagated to the A node.

The NORMALIZECONSTRAINTASPECT algorithm visits all the nodes in the input model H, it calculates for all nodes what would be the number of the navigation steps if the constraint were relocated to the actual node and the navigation paths would be updated. Therefore, if the node B was better in the case of the constraint C, then it would be found by the NORMALIZECONSTRAINTASPECT algorithm. This contradicts to the assumption.

Proof for Proposition 3.

Assume two identical transformation steps S_1 and S_2 and two identical host graphs H_1 and H_2 as well as an OCL constraint C. A constraint aspect *CA* is created from the OCL constraint *C* with the CREATECONSTRAINTASPECT algorithm. Furthermore we assume that we propagate *C* to S_1 and *CA* to S_2 and apply S_1 toH_1 and S_2 toH_2. Then during the execution of S_1 and S_2 the evaluations of the propagated constraints (*C* and the constraint contained by *CA*) are different, namely, one of them returns true and the other returns false.

During the propagation of the constraint$C$, the algorithm checks all the possible places of the transformation step S_1 where the constraint can be assigned. The algorithm selects the PRNs with the metatype which corresponds to the context information of the constraint C, it checks those places if the structure of the transformation step S_1 satisfies the pattern required by the navigation paths of the constraint$C$, and assigns the constraint C only to the appropriate places.

The CREATECONSTRAINTASPECT algorithm identifies the root node type of the constraint aspect pattern by the context of the constraint. It creates the root node (line 1), walks through the navigation paths of the C and creates the pattern of the *CA* constraint aspect (lines 2-5). Furthermore, the algorithm propagates the OCL constraint to the root node of the constraint aspect (line 6). Therefore the pattern of *CA* is equal to the pattern which is required during the propagation of the C constraint.

As a conclusion: (a) In the case of the constraint C the navigation paths are checked by the propagation algorithm during the constraint linking. (b) The pattern of the *CA* includes the structure information as well, and the propagation of the *CA* constraint aspect is achieved by this structure information. This means that the same pattern is checked during the propagation processes, therefore the constraints are propagated to the same places.

This contradicts the assumption. Therefore the transformation steps with the propagated OCL constraint *C* and the constraint aspect *CA* can not produce different result models.

Assume two identical transformation steps S_1 and S_2, two identical host graphs H_1 and H_2 and a constraint aspect *CA*. A normalized constraint aspect *CA'* created from the constraint aspect *CA* with the NORMALIZECONSTRAINTASPECT algorithm. Assume that we propagate *CA* to S_1 and *CA'* to S_2 and apply S_1 toH_1 and S_2 toH_2. Then during the execution of S_1 and S_2 the evaluations of the propagated constraints (the constraint contained by the *CA* and the constraint contained by the *CA'*) are different i.e. one of them returns true and the other returns false.

The NORMALIZECONSTRAINTASPECT algorithm processes the OCL constraints propagated to the constraint aspect individually. The main *foreach* loop (lines 1-19) examines the navigation paths of the actual constraint. (a) If all the paths have the same destination node, then the algorithm removes the navigations from the constraint and relocates it to the destination node (lines 2-3). (b) Otherwise the constraint has more than one navigation path, and the destination nodes of the paths are different. In this case the algorithm checks what would happen if the constraint were relocated to another PRN (lines 6-13). The check is achieved with a calculation which takes all the PRNs contained by the constraint aspect into consideration. This calculation sums the number of the navigation steps which is necessary in aggregate to reach all destination nodes of the original constraint (C) from the new place. The algorithm stores the most appropriate node (*optimalNode*) (lines 8-14) and, finally, it updates the navigation paths and relocates the constraint to the optional node (lines 15-17). Updating the navigation paths means that the navigation paths of a constraint are modified such that the constraint refers to the same destination nodes from its new place as well. The relocation of a constraint means that the constraint is removed from its original place and linked to its new PRN.

The NormalizeConstraintAspect algorithm does not change the structure of the constraint aspect, the propagation of the constraint aspect *CA* and the normalized constraint aspect *CA'* is accomplished based on their structure information, thus, the constraints are propagated to the same places. Furthermore, the NormalizeConstraintAspect algorithm updates the navigation paths and relocates the constraints, but it does not modify the conditions of the constraints, which results that the same conditions are required both in the case of the constraint aspect *CA* and the normalized constraint aspect *CA'*.

This contradicts the assumption. Therefore, the transformation steps with the propagated constraint aspect *CA* and the normalized constraint aspect *CA'* can not produce different result models.

It follows from (i) and (ii) that the OCL constraint C and the normalized constraint aspect *CA'* created with the CreateConstraintAspect and NormalizeConstraintAspect algorithms from C are equivalent: The path information from the OCL expression has been built into the pattern of the constraint aspect. The OCL constraint contains the path information in its text, the constraint aspect contains the same path information in its pattern. Thus, both of them contain the same conditions for the same destination nodes. The equivalence relation is transitive, therefore this statement follows from (i) and (ii).

Proof for Proposition 4. We assume two identical host graphs H_1 and H_2 and one applies T_1 to H_1 andT_2 to H_2. After each transformation step, the results of the actual transformation steps are compared. In step n T_{1n} and T_{2n} produce different results - one of them is successful, but the other fails because of a constraint failure.

Based on the *Proposition 3* the OCL constraint C and the normalized constraint aspect *CA'* are equivalent.

(a) The inputs of the GCW algorithm are OCL constraint(s) and a transformation (an ordered sequence of the transformation steps), let it be constraint C and transformationT_1. The GCW checks all the possible places of the passed rewriting rules ($T_{11}...T_{1n}$). These are places to which the constraint can be assigned. The algorithm selects the PRNs with the metatype that corresponds to the context information of the constraint C (line 3), it checks those places if the structure of the actual transformation step satisfies the pattern required by the navigation paths of the constraint$C$ (line 4). For the PRNs selected by their structure, the algorithm checks if their step requires the constraint C to be propagated to any of them. It decides whether it is the first or the last step in the transformation or whether it can modify the property contained by the constraint C (ISREQUIREDTOWEAVE method line 3). Finally, the GCW algorithm propagates the constraint C to the required places (line 6).

(b) The inputs of the CAW algorithm are constraint aspect(s) and a transformation. They are denoted with the constraint aspect *CA* and the transformation T_2. The CAW algorithm checks the transformation steps individually. In each step, it searches for matches by the pattern in *CA* (line 3), and for each constraint ($CA\_C_1 \ldots CA\_C_n$) contained by *CA*, the algorithm selects the PRNs from the matches with the metatype which corresponds to the context information of the actual constraint (line 5). Similarly to the GCW algorithm, CAW also uses the ISREQUIREDTOWEAVE method to decide if a transformation step requires *CA* to be propagated to the actual PRN (line 6). If at least one of the constraints ($CA\_C_1 \ldots CA\_C_n$) contained by *CA* requires to be propagated, the whole constraint aspect is linked (line 7).

As it is presented the difference between the GCW and CAW algorithms is that the GCW checks the pattern of the transformation step according to the text of the OCL constraint, while the CAW utilizes that the constraint aspect includes the pattern in its structure. Since the two approaches differ in their data representation only, the algorithms cannot give different results. This contradicts the initial assumption.

Acknowledgement

The fund of 'Mobile Innovation Centre' has supported in part, the activities described in this paper. The authors are grateful to Tihamér Levendovszky for continual help and support in research work and would like to thank to an anonymous reviewer who pointed out an error in the previous version of this paper.

References

[1] OMG 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] LEVENDOVSZKY, T. – LENGYEL, L. – MEZEI, G. – CHARAF, H. , A Systematic Approach to Metamodeling Environments and Model Transformation Systems in VMTS, Electronic Notes in Theoretical Computer Science, *International Workshop on Graph-Based Tools (GraBaTs) Rome*, 2004.

[3] OMG Object Constraint Language Specification (OCL), www.omg.org

[4] AOSD Homepage. http://www.aosd.net/

[5] ROBERT E. FILMAN, – TZILLA ELRAD,– SIOBHAN CLARKE,– MEHMET AKSIT, Aspect-Oriented Software Developement, Addison-Wesley, 2004.

[6] The AspectJ Programming Guide, ttp://www.aspectj.org

[7] LENGYEL, L. – T. LEVENDOVSZKY, – MEZEI, G. – FORSTNER, B.– CHARAF, H. Metamodel-Based Model Transformation with Aspect-Oriented Constraints, Accepted to International Workshop on Graph and Model Transformation, GraMoT, Tallinn, Estonia, September 28, 2005.

[8] LENGYEL, L.– LEVENDOVSZKY, T.– CHARAF, H. Eliminating Crosscutting Constraints from Visual Model Transformation Rules, Accepted to ACM/IEEE 7th International Workshop on Aspect-Oriented Modeling, Montego Bay, Jamaica, October 2, 2005.

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

[10] CORRADINI, A. – EHRIG, H. – LÖWE, M. – MONTANARI, U.–ROSSI, F. Abstract Graph Derivations in the Double-Pushout Approach. In H.-J. Schneider and H. Ehrig, editors, Proceedings of the Dagstuhl Seminar 9301 on Graph Transformations in Computer Science, vol.776 of Lecture Notes in Computer Science, pp. 86-103. Springer Verlag, 1994.

[11] EHRIG, H.Introduction to the Algebraic Theory of Graph Grammars, In: Graph Grammars and Their Applications to Computer Science and Biology, Springer, Ed. V. Claus, Ehrig, H. G. Rozemberg, Berlin, 1979.

[12] EHRIG, H. Tutorial Introduction to the Algebraic Approach of Graph- grammars. In H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors, *Proceedings of the 3rd International Workshop on Graph-Grammars and Their Application to Computer Science*, **291** *Lecture Notes in Computer Science*, pp. 3–14. Springer Verlag, 1987.

[13] EHRIG, H. – KORFF, M.– LÖWE, M., Tutorial Introduction to the Algebraic approach of graph grammars based on double and single pushouts. In Ehrig, H. H.-J. Kreowski, and G. Rozenberg, editors, Proceedings of the 4th International Workshop on Graph-Grammars and Their Application to Computer Science, vol. 532 of Lecture Notes in Computer Science, pp 24–37. Springer Verlag, 1991.

[14] EHRIG, H. – ENGELS, G. – KREOWSKI, H-J.– ROZEMBERG, G. (ed.), Handbook on Graph Grammars and Computing by Graph Transformation: Application, Languages and Tools, Vol.2. World Scientific, Singapore, 1999.

[15] LÖWE, M. – DINGEL, J., Parallelism in Single-pushout Graph Rewriting. Lecture Notes in Computer Science 776, pp. 234–247, 1994.

[16] LÖWE, M., Algebraic approach to single-pushout graph transformation. Theoret. Comput. Sci., 109:181–224, 1993.

[17] EHRIG, H. – LÖWE, M., Parallel and distributed derivations in the single pushout approach. TCS, 109:123 -143, 1993.

[18] KORFF, M. – RIBEIRO, L., Concurrent Derivations as Single Pushout Graph Grammar Processes. In A. Corradini and U. Montanari, editors, Proceedings SEGRAGRA'95 Workshop on Graph Rewriting and Computation, volume 2 of Electronic Notes in Theoretical Computer Science. Elsevier Sciences, 1995. http://www.elsevier.nl/locate/entcs/volume2.html.

[19] KORFF, M. True Concurrency Semantics for Single Pushout Graph Transformations with Applications to Actor Systems. In Proceedings International Workshop on Information Systems – Corretness and Reusability, IS-CORE'94, pages 244–258. Vrije Universiteit Press, 1994. Tech. Report IR-357.

[20] LOECHER, S. – OCKE, S., A Metamodel-Based OCL-Compiler for UML and MOF. In OCL 2.0 - Industry standard or scientific playground, Workshop Proceedings, 6th International Conference on the UML and its Applications,«UML»2003, ENTCS, Oct. 2003.

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

[22] GRAY, J. – BAPTY, T. – NEEMA,S. – TUCK, J., Handling Crosscutting Constraints in Domain-Specific Modeling, Communications of the ACM, October 2001, pp. 87–93.

[23] The VMTS Homepage. http://avalon.aut.bme.hu/~tihamer/research/vmts/

[24] LEVENDOVSZKY, T. – LENGYEL, L. – CHARAF, H., Software Composition with a Multipurpose Modeling and Model Transformation Framework, IASTED 2004, Innsbruck, 2004, pp. 590–594.

[25] KARSAI, G. – AGRAWAL, A. – FENG SHI – SPRINKLE, J., On the Use of Graph Transformation in the Formal Specification of Model Interpreters, Journal of Universal Computer Science, Special issue on Formal Specification of CBS, 2003.

[26] PROGRES system can be downloaded from http://www-i3.informatik.rwth-aachen.de/research/projects/progres/main.html

[27] VARRÓ, D. – PATARICZA, A. , VPM: A Visual, Precise and Multilevel Metamodeling Framework for Describing mAthematical Domains and UML, *Journal of Software and Systems Modeling*, 2003.

[28] TAENTZER, G. AGG: A Graph Transformation Environment for Modeling and Validation of Software, In J. Pfaltz, M. Nagl, and B. Boehlen (eds.), Application of Graph Transformations with Industrial Relevance (AGTIVE'03), volume 3062. Springer LNCS, 2004.

[29] LENGYEL, L. – LEVENDOVSZKY, T.– KOZMA, P.– CHARAF, H., Compiling and Validating OCL Constraints in Metamodeling Environments and Visual Model Compilers, IASTED on SE, February 15-17, 2005, Innsbruck, Austria, pp. 48–54.

[30] LEVENDOVSZKY, T. – CHARAF, H., Pattern Matching in Metamodel-Based Model Transformation Systems, submitted to Periodica Polytechnica Electrical Engineering, ISSN 0324-6000.

[31] ZÜNDORF, A., Graph Pattern Matching in PROGRES, In: Graph Grammars and Their Applications in Computer Science, LNCS 1073, Cuny J et al. editors, Springer-Verlag, 1996, pp. 454–468.

[32] LENGYEL, L. – LEVENDOVSZKY, T. – CHARAF, H., Normalizing OCL Constraints in UML Class Diagram-Based Metamodels - AND/OR Clauses, Accepted to EUROCON 2005 International Conference on 'Computer as a tool', *Proceedings of the IEEE, Serbia & Montenegro,* November 21–24, 2005.