ꟼ periodica polytechnica

Electrical Engineering
51/3-4 (2007) 99–110
doi: 10.3311/pp.ee.2007-3-4.04
web: http://www.pp.bme.hu/ee
© Periodica Polytechnica 2007

RESEARCH ARTICLE

Optimization algorithms for OCL constraint evaluation in visual models

Gergely Mezei / *Tihamér* Levendovszky / *Hassan* Charaf

Received 2007-10-03

Abstract

The growing importance of modeling and model transformation has attracted attention to create precise models and transformations. Visual model definitions have a tendency to be incomplete, or imprecise, thus, the definitions are often extended by textual constraints attached to the model items. Textual constraints can eliminate the incompleteness stemming from the limitations of the visual definitions. Object Constraint Language (OCL) is one of the most popular constraint languages in the field of UML, Domain Specific Modeling Languages, and model transformations. Efficient constraint handling needs the optimization of the constraints. Our research focuses on creating optimization algorithms for OCL constraint handling. This paper presents three algorithms that can accelerate the validation process, and thus, make the modeling more efficient. Proofs are also provided to show that the optimized and the unoptimized code are functionally equivalent, and the paper contains a simple case study to show the practical relevance of the algorithms.

Keywords
Modeling · OCL · Formalism

Acknowledgement
The found of Mobile Innovation Centre supported, in part, the activities described in this paper.

Gergely Mezei
Department of Automation and Applied Informatics, BME, H-1111 Budapest, Goldmann György tér 3., Hungary
e-mail: gmezei@aut.bme.hu

Tihamér Levendovszky
Department of Automation and Applied Informatics, BME, H-1111 Budapest, Goldmann György tér 3., Hungary
e-mail: tihamer@aut.bme.hu

Hassan Charaf
Department of Automation and Applied Informatics, BME, H-1111 Budapest, Goldmann György tér 3., Hungary
e-mail: hassan@aut.bme.hu

1 Introduction

Language engineering is the basis of several well-known techniques, such as Domain Specific Modeling Languages (DSMLs). On the one hand visual language definitions have many advantages, since they allow creating models using a high level of abstraction, the customization of model rules and notation. On the other hand these definitions have the tendency to be imprecise, incomplete, and sometimes even inconsistent. For example, assume a domain describing computer networks. A computer can have input and output connections, but these connections use the same cable with maximum *n* channels. Thus, the number of the maximum available output connections equals the total number of channels minus the current number of input channels. It is hard, or even impossible to express this relation in a visual way in a UML class diagram, for instance. Another example is a resource editor domain for mobile phones, where it is useful to define the valid range for slider controls.

The lack of completeness applies to model transformations as well. Beyond the topology of the visual models in the transformation, additional constraints must be specified ensuring, for example, the validation of attribute values. Assume a transformation defining a breadth-first searching algorithm. Here, it is useful to distinguish the visited and not visited nodes. It is often tedious to describe this information using topological transformation rules only.

The solution of both problems is to extend the visual definitions by textual constraints. There exist several textual constraint languages, the Object Constraint Language (OCL) is possibly the most popular among them. OCL was originally developed to create precise UML diagrams [1] only, but the flexibility of the language made possible to reuse OCL in language engineering, such as in metamodeling [2]. Nowadays, OCL is one of the most wide-spread approaches in metamodeling and model transformations. The textual constraint definitions of OCL are unambiguous and still easy to use.

Using OCL, precise models and transformations can be created, but the efficiency, the performance of the validation is essential, especially when the size of the models (the number of the model items) is large. There are several academic and industrial tools and environments that use OCL to extend incomplete model definitions in visual languages, or in model transformations, but none of these tools supports constraint optimization.

Our tool, named Visual Modeling and Transformation Systems (VMTS) [3] is an n-layer metamodeling and model transformation tool. VMTS uses OCL constraints in model validation and also in graph rewriting-based model transformation [4]. VMTS contains an OCL 2.0 compliant constraint compiler that generates a binary executable for constraint validation [5].

Our work focuses on creating a complete optimized constraint handling solution based on the experiences gained from the implementation of an OCL compiler in VMTS. The primary aim of this paper is to give an overview on this method. The presented solution consists of three algorithms, which have been implemented in VMTS to increase the efficiency of the constraint validation. The algorithms do not rely on system-specific features, thus, they can be easily implemented in any other modeling or model transformation framework. The first two algorithms reduce the number of navigation steps by relocating and decomposing the constraints. The first version of these algorithms was presented in [6]. Since then, the algorithms have been improved and more appropriate usability conditions have been created. The paper presents these conditions and the improved version of the algorithms as well. The third algorithm is used to reduce the number of model queries by caching the referenced values. The paper also gives a concise description about placing the algorithms in the compiler control flow, and it describes how the three algorithms can cooperate. Proofs of correctness for the algorithms and a short case study are also provided.

The paper is organized as follows: Section 2.1 elaborates on some of the most popular tools that support constraint checking based on OCL. Section 2.2 shows a basic OCL compiler implemented in VMTS. The introduction of the non-optimizing compiler is useful to place the optimizing algorithms in the compiler control flow, and make the analysis of the mechanism of the algorithms easier. Section 3.1 and 3.2 present the constraint relocation algorithm, Section 3.3 describes the constraint decomposition, and Section 3.4 elaborates on a caching algorithm. The details of the optimizing OCL compiler are presented in Section 3.5. Section 3.6 contains a case study, where the algorithms are shown in practice. Finally, Section 4 summarizes the presented work.

2 Background

2.1 Related work

There are several tools supporting OCL constraint handling. This section deals with the most typical validation tools and compilers only.

Object Constraint Language Environment (OCLE) [7] is a UML CASE Tool. OCLE supports both static and dynamic checking at the user model level. The tool has a user-friendly graphical interface. Although the tool supports model checking, it does not use compiling techniques.

The Dresden OCL Toolkit (DOT) [8][9] generates Java code from OCL expressions, and then instruments the system in five steps: (i) OCL expressions are parsed using a LALR(1) parser generated with SableCC [10]. The result of the step is an Abstract Syntax Tree (AST). (ii) A limited semantic analysis is performed on the AST to find errors. (iii) The AST is simplified in order to make the further processing simpler. (iv) The code generator traverses the simplified AST and builds Java expressions. (v) The generated code is inserted into the system that contains the constraint source code, thus, the contracts can be tested at runtime. DOT does not support metamodeling, or constraint optimizing techniques.

Kent Modeling Framework [11] is a set of projects that supports model driven software development. One of these projects is KMFStudio that can generate modeling tools from metamodels. KMFStudio supports the dynamic evaluation of OCL constraints. The tool has been integrated into Eclipse. It enables the language to be bridged to other Eclipse-based modeling frameworks. The Kent Modeling Framework does not use optimizing algorithms to improve the efficiency of the constraint validation.

Open Source Library for OCL (OSLO) [12] is a further development of Kent OCL Library. OSLO is based on the Eclipse framework. OSLO supports OCL 2.0 functions for arbitrary metamodels based on EMF, and constraint checking for UML2 models (Eclipse UML2). OSLO supports constraint checking in metamodeling, but not in model transformations. Since it is a recent project, not all of the supported features are introduced in depth.

2.2 VMTS OCL 2.0 Compiler

VMTS OCL Compiler consists of several parts (Fig. 1). This section gives a short description of the architecture of the compiler.

The user defines the constraints in OCL, then the textual constraint definitions are tokenized and syntactically analysed. The lexical analysis creates a sequence of token from the constraints. Tokenization is accomplished by Flex [13]. Syntactic analysis uses Bison [14] to build a syntax tree from the tokens according to the grammar rules of OCL specified in EBNF format [1]. To accommodate the ambiguities in the specification, the grammar rules are simplified. The syntax tree does not contain all the necessary information, thus, it is extended e.g. with type information, and implicit *self* references. This amendment is performed in the semantic analysis phase, and it produces a semantically analysed syntax tree. Using the semantic information, the simplifications made during the tree building can be corrected. In the next step, the constructed and semantically analysed tree is transformed to a CodeDOM tree. CodeDOM [15] is a .NET-based technology that describes programs using abstract trees. Using the abstract trees, it can generate code to any languages that is supported by the .NET CLR (like C#, or Visual Basic). The compiler transforms the CodeDOM tree to C# source code. To support the base types available in OCL, a class library has

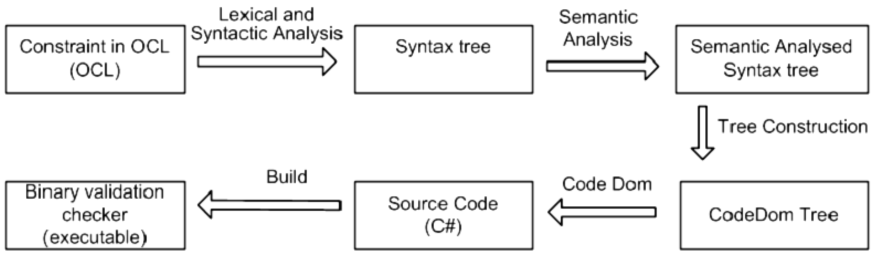

Fig. 1. VMTS OCL Compiler 2.0 Architecture

been developed. The constraint classes inherit from the base classes implemented in this class library. The output of the OCL compiler is a binary assembly (a .dll file) that implements the validation methods.

Since the constraints are compiled only once, not each time when the constraints are evaluated, the validation process is fast and efficient. The compiled OCL validation assembly can be used either in model validation, or in graph transformation. There are no differences between the two cases in handling the constraints: the editing framework (VMTS Presentation Framework, [3]) collects the appropriate model items and invokes the validation method for them.

3 Optimizing Algorithms

In general, the evaluation of OCL constraints consists of two steps: (i) selecting the model items and their attributes that are used in the constraint, and (ii) executing the validation method. Our optimization algorithms focus on the first step, because of two reasons: (i) The efficiency of the validation is heavily affected by the implementation of the OCL library (types and expressions), thus, the optimization is usually implementation-specific. (ii) In general, the first step has more serious computational complexity, since each navigation step means a query in the underlying model. The original version of the first two algorithms were published in [6].

It is essential not to change the result of the constraint evaluation by the optimization algorithms. A constraint modification is *correct* if, and only if the output of the optimized and original constraint is the same for every possible input. In general, *correctness* is even more important, than efficiency. Thus, it is rigidly checked whether the presented algorithms are *correct*.

3.1 Constraint Relocation

One of the most efficient way to accelerate the constraint evaluation is to reduce the navigation steps in a constraint without changing the result of it. This is the aim of the first algorithm, called *RelocateConstraint* (Alg. 1). The algorithm processes the propagated OCL constraints, and tries to find the optimal context for the constraint. Therefore, the algorithm consists of two major parts: (i) searching for the optimal node (and *RelocationPath*) (Alg. 2) and (ii) relocating the constraint if necessary (Algorithm 3).

The first part of the *RelocateConstraint* algorithm is based on

Algorithm 1 The new RelocateConstraint algorithm

```
1: RELOCATECONSTRAINT($Constraint$, $OriginalContext$)
2: $OptimalPath$ = SEARCHOPTIMALNODE($OriginalContext$, $NULL$)
3: if $OptimalPath.LastElement \neq OriginalContext$ then
4:     UPDATEANDRELOCATE($Constraint$,$OptimalPath$)
```

the *SearchOptimalNode* function (Alg. 2). Since the original and the optimal node are not always neighbours, the optimization stores a path between the original and the new context. This path is called *RelocationPath*. Storing this additional information is necessary, because there can exist more than one paths between the two nodes in the host graph. The differences between the paths can mean that one path is acceptable, while the other is not. Where an acceptable *RelocationPath* means a path that results a *correct* relocation of the constraint. The result of the *SearchOptimalNode* function is the *RelocationPath*.

Since the relocation is not always possible, the function checks the relocation requirements during the search (*StepIsValid*). Thus, invalid *RelocationPath* candidates are dropped as soon as possible. *SearchOptimalNode* uses a recursive breadth-first-search strategy to find all possible candidates. *RelocationPath* is handled by the external funcion *Append*. *CalculateSteps* is another external function that calculates the number of model queries in the case when the new context is located in *N* using the current *RelocationPath*.

CalculateSteps examines the OCL expressions in the constraints one by one and counts the number of navigations and attribute references, used during evaluation of the constraint. *CalculateSteps* simulates executing the constraint in order to be able to apply this computation. Since only the metamodel, not the model is available at the moment of optimization, the function uses worst case approximation, where the multiplicity of model items is a range, not a number. Therefore, the complexity of *CalculateSteps* can be expressed as $O(n^k)$, where n is the number of model references in the constraint expression, while k is the size of the largest interval of possible multiplicities. Note that the optimization is applied offline, thus, the execution of *CalculateSteps* does not increase the time of evaluation

If the new and the old context found by the *SearchOptimalNode* function are not the same, then the constraint is relocated and updated by the function UpdateAndRelocate (Alg. 3).

The updating mechanism is based on path steps of the *Relo-*

Algorithm 2 The SearchOptimalNode algorithm

```
1: SearchOptimalNode(Node $N$, Path $P$)
2: $minSteps$ = CalculateSteps($N$)
3: $optimumCandidate$ = Append($P$, $N$)
4: for all $CN$ in ConnectedNodes($N$) do
5:     if StepIsValid($CN$) then
6:         $LocalOptimum$ = SearchOptimalNode($CN$, Append($P$, $N$))
7:         $LocalSteps$ = CalculateSteps($LocalOptimum.LastElement$)
8:         if $LocalSteps < minSteps$ then
9:             $minSteps = LocalSteps$
10:            $optimumCandidate = LocalOptimum$
11: return $optimumCandidate$
```

Algorithm 3 The UpdateAndRelocate algorithm

```
1: UpdateAndRelocate(Constraint $C$, Node $O$, Path $P$)
2: for all $Step$ in $P$ do
3:     if SourceMultiplicity($Step$)= $ExactlyOne$ and
       DestMultiplicity($Step$)= $ExactlyOne$ then
4:         ExactlyOneRewrite($C$)
5:     if SourceMultiplicity($Step$) $\neq MoreThanZero$ then
6:         AddForeach ($C$)
7:     if DestMultiplicity($Step$) $\neq MoreThanZero$ then
8:         RemoveForeach($C$)
9: return $optimumCandidate$
```

cationPath: the algorithm updates the context declaration step-by-step. Multiplicities on the source and destination side of the path step under execution can affect, thus, the function handles the different subcases distinctly. The multiplicity checking and the constraint updating mechanisms are implemented in external functions to improve the readability of the algorithm.

3.2 Restrictions to Constraint Relocation

The aim of the limitations is to eliminate the cases where the result of the original and the optimized algorithms would differ. To achieve this, it is necessary to examine when and how *correct* relocations can be applied. In the following propositions, we say — for the sake of simplicity — that a *RelocationPath* is *correct*, although we mean that the relocation using the *RelocationPath* is correct.

Proposition 1 *If the steps of* RelocationPath *are separately correct, then their composition, the* RelocationPath *is also correct.*

Example 1 The original constraint is located in node A, the optimal node is D (Fig. 2). Thus, the *RelocationPath* is drawn from A to D (dashed line). If neither the relocation from node A to C (solid line), nor the relocation from node C to D (dotted line) change the result of the constraint, namely they are *correct*, then the proposition states that the relocation from A to D is also *correct*.

Proof 1 Let C be the original constraint and P a complex *RelocationPath* found by the search steps. P contains finite number of steps, since the host model contains finite number of model items and no circular navigation operations are allowed in the

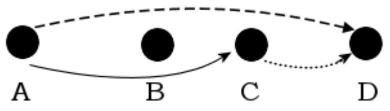

Fig. 2. The steps and the whole RelocationPath

path. When creating the *RelocationPath* we store visited model items of the metamodel, if a certain step would like to add a model item, which is already in the path, then we remove the loop from the path. For example there is a metamodel with there model items: A, B, C, D. Furthermore, there is a navigation from A to B, a navigation from B to C, a navigation to C to B and a navigation from C to D as well. When we try to create *RelocatePath* from A to D we do not add the loop between B and C infinite times, but only once to the path.

Furthermore, let O be the original context; S the first step of P and *O'* the destination node of S in P. According to the premise of the proposition the correctness of S is proven, thus, relocating the constraint from O to *O'* can be accomplished. After applying this relocation, a new constraint, *C'* can be constructed. Applying the relocation algorithm on *C'* results a new *RelocationPath*, *P'* containing one less step, than the original one. Since P has a finite number of steps, the algorithm always terminates.

Corollary 1 *The steps in a path can be examined separately. If in a certain case the correctness of the algorithm is proven to be* correct *for each single navigation step in the* RelocationPath*, then it is also proven for the whole* RelocationPath*. Thus, in general, if the correctness of each possible single navigation step is proven, then the correctness of the whole relocation is proven. Therefore, it is enough to examine the correctness of single relocation steps.*

In the next propositions, the following abbreviations are used: C denotes the original constraint, C' the new constraint, M_0 is metamodel, M is model, O is the original context, N is the new context. O and N are metamodel elements, and their instantiations are $O_1,\ O_2 \ldots O_n$, and $N_1,\ N_2 \ldots N_n$.

Example 2 Fig. 3 shows an example metamodel, its instantiation, and the constraint relocation. The metamodel represents a domain that can model computers, and display devices (here monitors only). A single computer can use multiple monitors. The model defines a simple constraint attached to the node *Computer*, this constraint is relocated by the optimization to the node *Monitor*. Using the abbreviations, we can say the following: M_0 is the metamodel shown in Fig. 3/a, M is its instantiation (Fig. 3/b). O is Computer, N is Monitor in M_0. O has two instantiations, Computer1 (O_1) and Computer2 (O_2). Similarly, PrimaryMonitor is N_1, SecondaryMonitor is N_2, and finally, Monitor is N_3.

Proposition 2 *Navigation edges that allow zero multiplicity (on either or both sides) cannot be used in* RelocationPath.

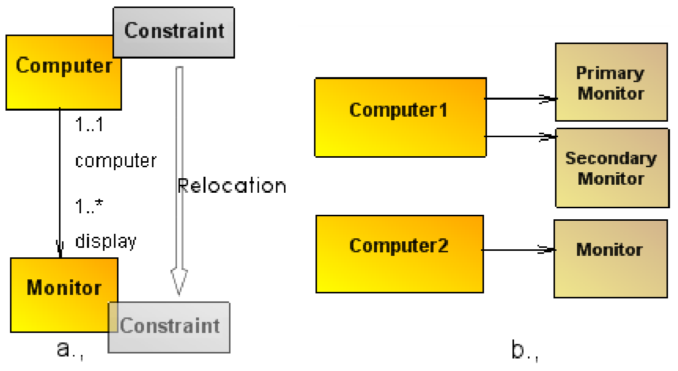

Fig. 3. Example metamodel and model

Proof 2 Let M be a model with O_1, N_1 *and* N_2 defined (Fig. 4). Let N_1 be isolated (or at least not connected with O_1).

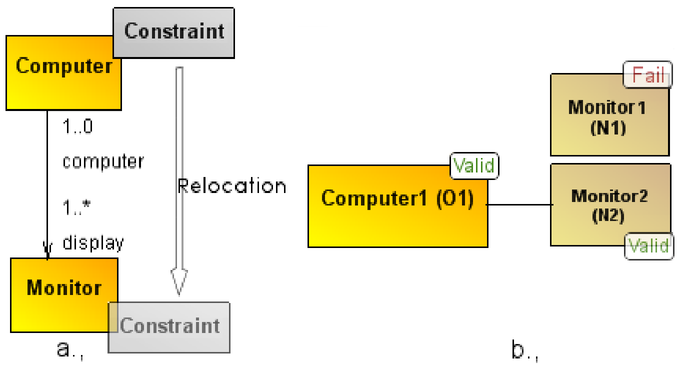

Fig. 4. Null multiplicity - metamodel and model

Let C and thus C' contain an expression that is not valid in N_1, but valid in N_2. The evaluation of C results true, since N_1 is not checked, because it is not connected with O_1. However C' fails, thus, the relocation is not *correct*.

The multiplicity of relations in metamodels is defined by a lower, and an upper limit. The limits can contain an integer representing the number of participants exactly, or * allowing any number of objects. In the following propositions, we categorize the multiplicities:

- *ZeroOrMore* - the lower limit of the multiplicity is 0 (the upper limit is not important)
- *ExactlyOne* - the lower and the upper limit is also 1
- *MoreThanOne* - the lower limit is not 0, while the upper limit is more, than 1

Proposition 3 *A relation with multiplicity ExactlyOne on both sides can be used for relocation. In this case the relocated expression differs from the original version in the navigation steps (or navigation step sequences). The new constraint expression is transformed from the original definition using the following rules:*

Rule 1. *If the expression is a navigation to the new context (N), then the expression is transformed into* `self`*.*

Rule 2. *If the expression is an attribute query in the old context (O), then the new expression is a navigation from N to O and an attribute query applied there (e.g.* `self.Manufacturer` *is transformed to* `self.computer.Manufacturer`*).*

Rule 3. *If the expression is a navigation from the old context (O), then the new expression is a navigation from N to O.*

Rule 4. *Other expressions in the constraint are not altered.*

Example 3 Let the example metamodel cited above define that computers are able to handle exactly one monitor, and monitors are always connected to exactly one computer (Fig. 5). Furthermore, let the constraint C state that the monitor is an LCD monitor ($display.Type = 'LCD'$). In this case relocating the constraint will result C': $Type = 'LCD'$.

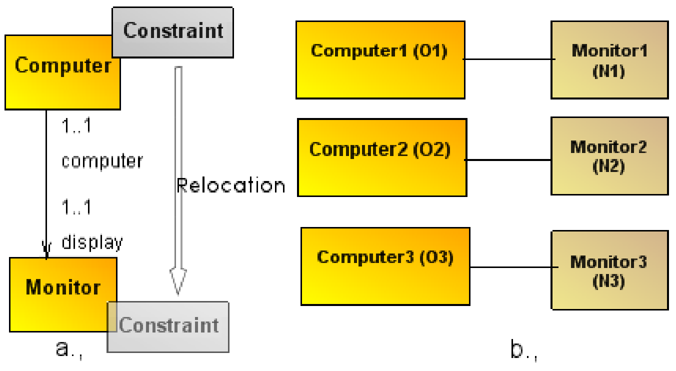

Fig. 5. ExactlyOne multiplicity on both sides - metamodel and model

Proof 3 An ExactlyOne multiplicity on both sides means that O and N objects can refer to each other the same way (using the role name of the destination node). The result of the navigation reference is always a single model item, not a set of model items and not an undefined value. This means that changing the navigation steps can be accomplished.

The transformation rules are also correct if the rules above are satisfied:

Rule 1. The relocation has changed the context, thus, the navigation step in the original context is not necessary any more.

Rule 2. and **Rule 3.** Since the original attribute reference, or the destination node of the navigation is invalid in the new context, thus, the constraint has to navigate back to the original context first, and applying the expression there.

Rule 4. Rule 1-3 cover all possible valid attribute and navigation expressions, thus, no additional rules are required.

Proposition 4 *If the multiplicity is ExactlyOne on the destination side, but MoreThanOne on the source side (not allowing zero multiplicity), then the constraint expression can be always relocated. In this case the constraint is encapsulated by a new constructed* `forall` *expression. If the relocated constraint does not contain any attribute reference to the original context node, or navigation through it, then the* `forall` *expression can be avoided.*

The original expression cannot be used after relocation, because of the multiplicity MoreThanOne, which retrieves a set of model items. The basic idea is to create an iteration on the elements of the set; the iteration is not contained in the original constraint.

Example 4 Let O contain a simple constraint referring to one of its attributes, named `IsAbstract`. After the relocation, the constraint is located in N and the reference `self.IsAbstract` is transformed to `self.O->forall(O | O.IsAbstract)`. This `forall` expression is true only if the condition holds for every elements in the set.

Example 5 The example model has been changed to meet the requirements of the proposition (Fig. 6).

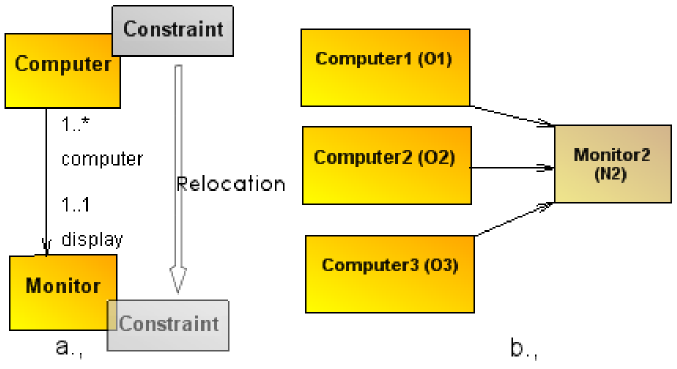

Fig. 6. MoreThanOne → ExactlyOne multiplicity - metamodel and model

Let C be defined as `self.Price` < `display.Price`. If this constraint is relocated, then it is transformed to

```
self.computer->forall(computer| computer.Price > self.Price)
```

expressing that *each* computer attached to the monitor has to accomplish the condition. Note that the navigation from O to N in `display.Price` was reduced to a single `self` reference similarly to the ExactlyOne-ExactlyOne case.

Proof 4 The presented method ensures that each model item on the original source side is processed, and the constraint is checked for each model item. Since the ZeroOrMore multiplicity is not allowed, the navigation is always possible. Inside the `forall` loop, the name of the destination node is the iterator value. Thus, this solution simulates ExactlyOne multiplicity on both sides. The relocated and the original version are equivalent.

Proposition 5 *If the multiplicity is ExactlyOne on the source side, but MoreThanOne on the destination side (not allowing optional multiplicity), then the constraint expression can be relocated if and only if the original expression uses* `forall`*, or* `not exists` *expression to obtain the referenced model items of the new context. This means that only those relations can be used where the original navigation selects all of the model items, or none of them (no partial selection, or another operation is allowed).*

Example 6 The constraint `self.N->count()` or `self.N->select(N.IsUnique)` cannot be relocated, but the constraint `self.N->forall(N.IsUnique)` can.

Example 7 The example model shows the requirements of the proposition (Fig. 7). Note that due to the preconditions of the proposition, the references to Monitor are always set operations in Computer. This means that, for example, the expression `self.display.Price>300` cannot be used, because `display` is a set, not a single value.

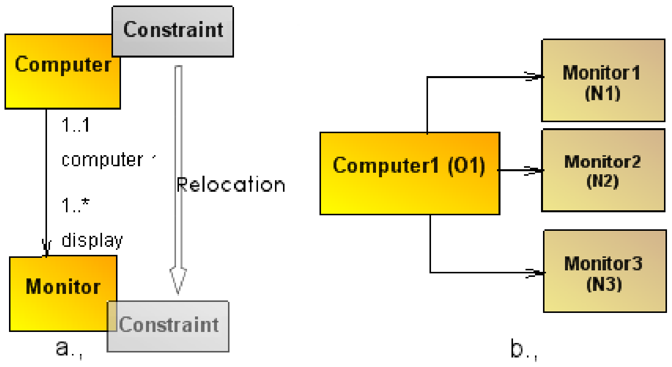

Fig. 7. ExactlyOne → MoreThanOne multiplicity - metamodel and model

Let M_0 contain three constraints: C_1, C_2 *and* C_3 using the following definitions:

```
inv c1: self.Price > 650
inv c2: self.display->count() > 5
inv c3: self.display->forall(m:Monitor| m.Price<300)
```

The proposition requires constraints to use `forall` expressions to query the attributes of the new context, or the navigation paths through the new context. But this also means that any other expression can be applied (for example a local attribute query, such as in *c1*). In this case the method of ExactlyOne-ExactlyOne multiplication can be used, thus, C_1' becomes the following:

```
inv c1: self.computer.Price > 650.
```

Complex set operations cannot be relocated according to the proposition, thus, C_2 cannot be relocated either. This limitation does not apply to C_3:

```
inv c3: self.Price<300.
```

Although the original and the relocated version of the constraint seems to differ, they have the same meaning: all monitors must be cheaper than 300 USD.

Proof 5 Firstly, the limitation to set operations is proven. In case of the general selection operations, such as `exists`, the selection criterion is *true* for some of the items and *false* for the others. This can lead to two problems with the constraint rewriting: (i) the constraint validation can generate false results where the selection criteria in the original expression is *true*/*false*, and (ii) the partial results arising in N cannot be processed (for example summarized) in O. Neither of these problems can be solved, thus, a universal relocation in this case is not possible.

Secondly, it needs to be proven that relocation is possible along `forall`, or `not exists` expressions. Note that `not exists` can be expressed using `forall` by negating the condition. The main difference between the previous (erroneous) subcase and this one is that here — if the model is valid — the condition in the select operation is *true* (or *false*) for *each* model item. Thus, the relocated constraint fails only, when the original constraint also fails. The relocation algorithm transforms `forall` expressions to single references. The relocated constraint is checked for each node of the new context, thus, the constraints are functionally equivalent.

Proposition 6 *If the multiplicity is MoreThanOne on both sides (not allowing zero multiplicity) (Fig. 8), then the constraint expression can be relocated if and only if the original expression uses* `forall`*, or* `not exists` *expressions to query the referenced model items of the new context node.*

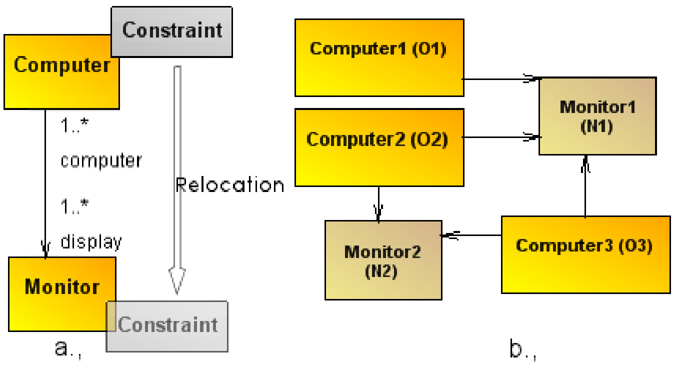

Fig. 8. MoreThanOne multiplicities - metamodel and model

Proof 6 This case is a combination of the previous cases. A new `forall` expression is constructed such that it contains the whole relocated constraint, then, inside this newly constructed `forall`, the original `forall` and `not exists` expressions are transformed to single navigation steps. The outer `forall` ensures that each O object is checked for each N, while the inner expression holds the transformed original constraint.

Proposition 7 *If the constraint contains more than one attribute reference expressions and these expressions do not depend on each other, then* partial relocation *is feasible.* Partial relocation *means that some of the expressions are executed in the new context, while others are executed in the original context. The original context is reached using navigation. Partial relocation does not apply to edges with zero multiplicity.*

Proof 7 Since the proposition is true only for relations not allowing zero multiplicity, the navigation between the original and the new context is always possible. Both ExactlyOne and MoreThanOne relations can be traversed according to the constructs presented earlier (either by single navigation steps, or `forall` expressions). Thus, when the constraint is evaluated, navigating back to the original context is always possible. In this way, the relocated and the original functionality is the same.

Corollary 2 *The task of finding possible destinations of relocation can be reduced to a simple path-finding problem from the original context to the new one, where relations allowing zero multiplicity cannot be the part of the path. Note that this path, if exists, is the* RelocationPath *mentioned earlier.*

Proposition 8 *The* RelocateConstraint *algorithm is* correct*.*

Proof 8 The steps of the SearchOptimalNode function do not modify the constraint expression, they are used for information gathering only. Thus, only the *UpdateAndRelocate* function needs to be examined. This function applies the relocation according to the presented restrictions. The relocation path is decomposable according to Prop. 1, and all possible multiplicity variations are covered for a single path step. This means that the function *UpdateAndRelocate* does not modify the result of the evaluation, thus, the *RelocateConstraint* algorithm is *correct*.

3.3 Decomposing Constraints

Constraints are often built from sub-terms and linked with operators (*self.age=18 and self.name='Jay'*), or require property values from different nodes (*self.age=self.teacher.age*). In these cases, using the *RelocateConstraint* algorithm, it is not possible to eliminate all navigation steps from the query. Although the subterms are not decomposable in general, they can be partitioned to clauses if they are linked with Boolean operators. A clause can contain two expressions (OCL expression, or other clauses) and one operation (AND/XOR/ IMPLIES) between them. The basic idea behind is that the result of the Boolean operations sometimes requires the evaluation of one of the operands only. For example in an AND expression, such as `self.Price>500 and self.display.Price>150` it is enough to check the value of the first operand if it evaluates to *false*.

Proposition 9 *The operands of a Boolean operations cannot affect each other, if the Boolean operation is the outermost expression in the constraint.*

Proof 9 The only case, in which the independency is not true between the operands is when the first subexpression has an effect on the second subexpression, thus, the first operand modifies one or more values used in the second operand. These modified values can be either model attributes, or variables defined in the current scope. The constraints used in validation cannot modify the model according to the specification of OCL [1]. Local variables can be defined for example in *Iterate*, and *Let* expressions, but using any variable definition would mean that the outermost expression cannot be an expression linked with Boolean operators. This means that the subexpressions of the clauses are independent.

The independence of the operands is important, because this means that their order of execution is not important. In case of AND, OR and IMPLIES operations the value of one operand can affect the results of the whole operation. In case of XOR operations no such simplification is possible, thus, the optimization does not use XOR in decomposing the constraint.

- If either operand is *false*, then the AND operation is always *false*.
- If either operand is *true*, then the OR operation is always *true*.
- If the first operand is *false*, then the IMPLIES operation is always *true*.
- If the presented condition for the given operand is not satisfied, then both operands are evaluated.

The constraint decomposition is made by *AnalyzeClauses* algorithm (Alg. 4) works on the syntax tree of the constraint. The algorithm is invoked for the outermost OCL expression of each invariant, and recursively searches the constraint for possible clause expressions and creates the clauses.

Algorithm 4 AnalyzeClauses algorithm

```
1: AnalyzeClauses(Model $Exp$)
2: if ($Exp$ is AndExpression) or ($Exp$ is OrExpression) or
   ($Exp$ is ImpliesExpression) then
3:     $Clause$ = CreateClause($Exp.RelationType$)
4:     $Clause$.AddExpression(AnalyzeClauses($Exp.Operand1$))
5:     $Clause$.AddExpression(AnalyzeClauses($Exp.Operand2$))
6:     return $Clause$
7: else
8:     if $Exp$ is ExpressionInParentheses then
9:         return AnalyzeClauses($Exp.InnerExpression$)
10:    else
11:        if $Exp$ is OnlyExpressionInConstraint then
12:            $Clause$ = CreateClause($SpecialClause$)
13:            $Clause$.AddExpression(RelocateConstraint($Exp$))
14:            return $Clause$
15:        else
16:            return RelocateConstraint($Exp$)
```

The steps of the algorithms are as follows: (i) If the current expression is a logical expression, then a new clause is created with the appropriate relation type (AND/OR/IMPLIES), and the two sides of the expressions are added to the clause as children. The children are recursively checked, because they can also be OCL expressions connected with logical operators (clauses can contain other clauses as children). The result clause is retrieved to handle the recursive calls. (ii) If the expression is between parentheses, then the function returns the inner expression. This substep is necessary, because the parentheses can modify the order of the constraint processing. (iii) In other cases the OCL expression cannot be decomposed. If it is the only expression in the constraint then a special clause is created, the *RelocateConstraint* algorithm is processed on the expression, and the clause is retrieved. If the expression is not the only expression in the constraint, then the expression itself is atomic. In this case the expression is relocated and then retrieved.

Example 8 There is a model for computers and monitors, where the metamodel contains ExactlyOne multiplicities only (Fig. 5). Furthermore, there is a constraint defined in Computer, which ensures that the system can display images with 1024*768 pixels:

```
inv ComputerMonitorCompatibility:
    self.Videocard.MaxResolution> 1024*768 and
    self.Monitor.MaxWidth> 1024 and
    self.Monitor.MaxHeight> 768
```

Note that Videocard is an attribute of Computer and MaxResolution is the maximum resolution supported by the videocard of the computer. Monitors manage this data by storing maximum width and height (they are attributes of the Monitor item).

When using the *AnalyzeClauses* algorithm the following steps are applied: (i) the original constraint C is divided into the clauses C_1 and C_2, where C_1 is `self.Videocard.MaxResolution>1024*768`, while C_2 is `self.Monitor.MaxWidth> 1024 and self.Monitor.MaxHeight>768` . The clauses are *AND* clauses, what means that the model is valid only if both of the clauses result in true. (ii) C_1 and C_2 are analysed again, C_2 contains an outermost Boolean expression, it is divided into C_{21} and C_{22}. C_{21} and C_{22} are *AND* clauses, they are parts of C_2. (iii) No further decomposition is possible, thus, the compiler tries to find the optimal context for the clauses. As result, C_{21} and C_{22} is relocated into Monitor.

The final, hierarchical clause structure is as follows:

```
- AND Clause
  |- C1
  |- AND Clause (C2)
      |- C21
      |- C22
```

The overall navigation cost of the constraint (9) is reduced by 2, because $MaxWidth$ and $MaxHeight$ attributes can now be accessed directly.

Proposition 10 *The algorithm* AnalyzeClauses *is* correct.

Proof 10 The algorithm *AnalyzeClauses* can be divided into three main cases according to the type of the examined expression: (i) the expression is a complex (non-atomic) expression with Boolean operators; (ii) the expression is an expression between parentheses; (iii) or the expression is an atomic expression.

In case (i) the result of the validation is modified only if the subexpressions cannot be processed independently. That contradicts Prop. 9.

In case (ii), where the inner expression (the expression between the parentheses) is recursively processed. The evaluation order of the subexpressions is the same as that of the original expression, and since no further modification is made, therefore case (ii) does not affect the result of the constraints.

Case (iii) has two subcases. If the examined expression is the only expression in the constraint, then a special clause is created, and the relocated constraint is placed into it. The special clause type is required only because of the uniformity. The inner expression (the normalized constraint) is processed when it is validated as if it were not contained in any clauses. The second subcase applies when the examined expression is a part of the constraint. In this case the relocated expression is returned. In both subcases the result of the constraint is not modified. Case (i) is used only if the constraint consists of two subparts linked with Boolean operators. A clause is created that preserves the Boolean operator, and the subexpressions are recursively processed. The subexpressions are processed individually when validating the constraint, and their results are connected using the operator (the order of the subexpressions are the same as in the original constraint).

Therefore the *AnalyzeClauses* algorithm is always *correct*.

3.4 Caching

Relocation and constraint decomposition algorithms can reduce the number of navigation steps, but cannot eliminate all of them. Therefore, the validation still requires queries to obtain the model elements, and their attributes. Thus, the number of model queries is not optimal.

In compiler optimization, an occurrence of the expression E is called a *common subexpression* if the value of E has previously been computed, and it has not changed since then [16]. In these cases recomputing the expression can be avoided, because the value of the expression is already known.

Proposition 11 *In OCL constraints navigation steps and attribute references are always* common subexpression*s, if they are used more than once.*

Proof 11 OCL specification defines the constraints as restrictions on one or more values, but these restrictions cannot have any side-effects. This means that the constraint cannot change the model, thus, the computed values can always be reused.

The presented idea is the basis of the third optimization algorithm. On the one hand, caching the model items can eliminate the redundant model queries in the constraint expressions. On the other hand, the more attribute or navigation is cached, the more memory the cache requires. Thus, only those expressions are cached that are referenced more than once. The optimization algorithm (the *ReferenceCaching* algorithm) has two main steps: (i) obtaining statistical information about the model references (*GetCommonReferences* algorithm), and (ii) caching the evaluation expressions (*CachingManagement* algorithm).

Collecting the statistical information set from the whole constraint expression is not straightforward, because sometimes only partial validation is required on a model. Thus, the caching algorithms are used at the context level, the statistical information of the different contexts are separated. Since the constraint decomposition can change the contexts, for example it can divide them into several clauses, the *GetCommonReferences* algorithm is used after the decomposition.

The *GetCommonReferences* algorithm is shown in Alg. 5. The algorithm uses a breadth-first search to traverse the syntax tree recursively. It processes the attributes, the navigations and the *control flow expressions*.

Algorithm 5 GetCommonReferences algorithm
```
1: GetCommonReferences($CurrentExpression$)
2: if ExpressionType($CurrentExpression$) is AttributeDefinition then
3:     IncreaseReferencePath($CurrentExpression$)
4:     return
5: if ExpressionType($CurrentExpression$) is NavigationStep then
6:     IncreaseReferencePath($CurrentExpression$)
7:     for all $CurrentExpression.Children$ as $navStep$ do
8:         GetCommonReferences($navStep$)
9:     return
10: if ExpressionType($CurrentExpression$) is ControlFlowExpression
    then
11:     $minReferences$ = GetMinimumReferencesForExecutionPath()
12:     for all $minReferences$ as $modelItem$ do
13:         IncreaseReferencePath($modelItem$)
14:     return
15: for all $CurrentExpression.Children$ as $child$ do
16:     GetCommonReferences($child$)
```

The attribute calls and navigation expressions increment the statistic of their path reference (using the *IncreaseReferencePath* function). To minimize the number of queries, the algorithm increments not only the reference of the full path, but also the references of the path steps. For example, the expression $self.employee.wife.Name$ will increase the statistics with four entries: $self$, $self.employee$, $self.employee.wife$ and $self.employee.wife.Name$. The statistics contain even the $self$ element, because it is not cached if it is referred to only once. In the algorithm this is why the child expressions, namely, the steps of the path are recursively checked in the case of *NavigationSteps*. Increasing the reference counter of the path steps is useful if two expressions have a common subset in the navigation steps, for example, in the expression $self.employee.wife.Name = 'Mrs.' + self.employee.Name$, the path $self.employee$ is used twice.

The *control flow expressions* are complex expressions that have several execution paths, for example conditional expression, or loops. These expressions can affect the number of the references according to their execution parameters. The problem is that these execution parameters are usually obtained at run-time only. Therefore, the algorithm obtains the minimum number of the references for each referenced objects for each execution paths. For example in case of the conditional expressions this means that both branches are processed, statistical information is collected for both branches, and then the results are compared. For each model reference path (attribute, or navigation reference), the minimum number of references is obtained and placed into the global statistical information set.

As the result of *GetCommonReferences* algorithm, the compiler has reliable statistical information. *CachingManagement* algorithm uses this information to handle caching. *CachingManagement* algorithm differs from the previously presented algorithms, because it affects the generated source code directly instead of affecting the syntax tree. Each time the compiler generates a navigation step or an attribute query, the statistics are checked, and a cache (a local variable) is created if required. This variable obtains the appropriate value from the database if it has not been read before, or returns the value from the cache if it is not the first query. If the model reference is not cached, the code generator will create a conventional source code for it.

Proposition 12 *The* ReferenceCaching *algorithm is* correct.

Proof 12 The first step of *ReferenceCaching* algorithm (GetCommonReferences) obtains statistical information only, it does not modify the evaluation. Therefore the only way *ReferenceCaching* algorithm can conflict with the original constraint definition is, if the cached references are not up-to-date. That would contradict Prop. 11, thus the *ReferenceCaching* algorithm is always *correct*.

Proposition 13 *Using the* ReferenceCaching *algorithm the number of the applied queries is less than or equal to that without optimization. Additionally, each attribute or navigation cached by the algorithm reduce the number of the database queries, thus no unnecessary caching is applied.*

Proof 13 The *GetCommonReferences* algorithm is applied at design-time, it does not raise the number of the queries during the evaluation. The *CachingManagement* algorithm handles two types of model references: the cached, and the uncached references. The source code and thus, the number of database queries of uncached model references is the same as in the unoptimized code. The cached references execute the appropriate database query only if the required value is not in the cache, i.e. it has not been read before. Therefore, neither the uncached nor the cached references increase the number of the database queries.

The *GetCommonReferences* algorithm is executed for each referenced context. If the context contains an expression that has several possible execution paths, then every path is examined, and for each model attribute and navigation the smallest number of references is stored. The sequential execution paths are examined step-by step, and the statistics are increased if required. As result the statistics contain the minimum number of the references in the context for every model item (attribute, or navigation). The *CachingManagement* algorithm creates caching code only for the model references that have greater statistical index, than one. Since the statistics contain the minimum number of the references of the current item, thus, no unnecessary caching is performed.

3.5 An Optimizing Compiler

When constructing the optimizing compiler it is important to place the algorithms in the compiler control flow. The optimization algorithms require a semantically analysed syntax tree, since, for example, the caching algorithms would not work without proper type-information. Thus, the optimization algorithms are used after the semantic analysis. The constraint decomposition, relocation, and the statistical information retrieval algorithms are executed before the code generation phase, because they affect the syntax tree from which the code is generated. The *CachingManagement* algorithm affects the code generation directly, it is used during the code generation phase.

Proposition 14 *The optimizing compiler consisting of the presented algorithms is* correct.

Proof 14 Let *H* be an optional input model, and let H' be the result model of the optimization executed by the *AnalyzeClauses*, RelocateConstraint and *ReferenceCaching* (GetCommonReferences and CachingManagement) algorithms. We prove that evaluating the constraints contained by *H'* produces always the evaluation in *H*.

The *correctness* of each algorithm has been proven in Props 8, 10 and 12, thus, only the composition of the algorithms, namely the optimizing compiler is to be examined. The only way in which *H'* and *H* can have different results is that the algorithms affect each other, and thus their composition changes the result of the constraint. The algorithm *ReferenceCaching* is executed independently from the other algorithms, and the proven correct output of the *AnalyzeClauses* is the input of the *RelocateConstraint* algorithm. Thus, the result created by the composition of the algorithms is always correct.

3.6 Case study

To show the applicability and the practical relevance of the results, a case study is provided. The case study contains a metamodel (Fig. 9/a) defining a DSL about processors. There are three main types defined besides processors: data buses, coprocessors and computing units. Each helper unit can be connected with the processor, additionally, the processor can communicate with optional number of computing units. Fig. 9/b shows an example instantiation of the metamodel.

In the metamodel, there is a constraint defined in DataBus model item:

```
context DataBus::CheckCacheSize() : Boolean
  self.processor.coprocessor.Cache>1024 or
 (self.processor.compunit->forall(CU | CU.PrimaryCache
 CU.SecondaryCache > 512 )
 and self.processor.compunit->count()>2 )
```

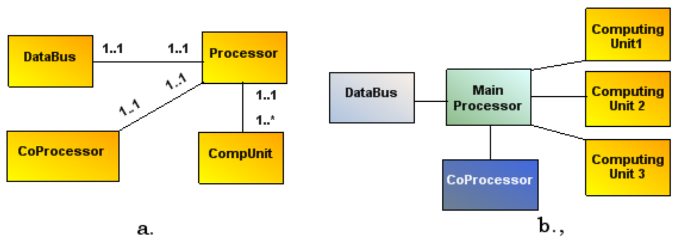

Fig. 9. Case study Metamodel and Model

The constraint evaluates to *true*, if there is at least 1024 byte cache available. The constraint is useful to check for example before memory operations. The original version of the constraint uses 22 model queries: (i) four queries to obtain the *Cache* attribute of the *CoProcessor*, (ii) two queries to navigate to *Processor* and another four queries for every *ComputingUnit* attached to the *Processor*, (iii) four queries to get the number of *ComputingUnits*. If the *RelocateConstraint* algorithm is used as optimization, then the constraint is relocated to context *Processor*, thus, the number of queries is reduced to 3+1+3*4+3 = 19. If both the *AnalyzeConstraint* and the *RelocateConstraint* algorithms are used, then two clauses are created from the constraint along the two boolean operands. The first part of the first clause `coprocessor.Cache>1024` is then relocated to *CoProcessor*, the first part of the second clause `CU.PrimaryCache+ CU.SecondaryCache > 512` to *ComputingUnit* items, but the second part of the second clause (`compunit->count()>2`) cannot be relocated from *Processor*, because of the `count` function. This optimized version requires 2+3*4+3= 17 queries. The optimizing compiler, including all the three algorithms does not only modify the clauses, but adds the ability to cache the queries. In this case it is efficient in the second clause only, where each *ComputingUnit* is retrieved twice. Another clauses do not reuse the values retrieved from the model. The number of queries in this case is 2+3*2+3=11. This means that the number of model queries is reduced by 50%. This ratio is rather high, because the primary aim of the case study was to show how the optimization works. We have found that in general, real life examples the optimization can accelerate the validation process by approximately 10-15%.

4 Conclusions

Constraint specification and validation lie at the heart of modeling and model transformation. The Object Constraint Language (OCL) is a wide-spread formalism to express constraints in modeling and transformation environments. There are several interpreters and compilers that handle OCL constraints, but OCL constraint optimization is a rather new idea; none of the existing tools supports it. This paper has presented three efficient and platform-independent optimization algorithms. The *RelocateConstraint* algorithm tries to find the optimal context for the constraint, and relocates it, if it is necessary. The relocation is applied along a path between the original and the optimal context, this path is called *RelocationPath*. Several limitations exist to the algorithm based on the multiplicity between the nodes of the path steps. The second algorithm, *AnalyzeClauses* can decompose the constraints to clauses if the outermost expression is a boolean operation (AND/OR/IMPLIES, but not XOR). This decomposition is useful, because the result of the operation often depends only on one of the clauses. The third algorithm, *ReferenceCaching* is slightly different, instead of modifying the constraints, it accelerates the validation by caching the model queries. The presented algorithms together can form the base of an optimizing OCL compiler. The correctness and the efficiency of the algorithms have been proven. The paper has also discussed a simple case study to show the optimization in practice.

The optimization used by the presented algorithms is based on the characteristics of OCL, thus it produces a better result than general optimization strategies. The weaknesses of the general optimization strategies are that they (i) usually require system-specific (tool-specific) solutions and (ii) cannot use particular OCL-specific algorithms. For example, the executing environment that executes the validation code cannot recognize automatically that attributes are always common subexpressions. The different optimization algorithms, such as the algorithms presented in the paper, and the query optimization of the underlying databases can be combined, to provide the optimal solution.

We have accomplished several simplified performance tests, and we have found that the optimization presented in the paper can accelerate the validation by 10-15% according to the circumstances. Since only basic tests were applied, further testing is required to give a detailed overview about the efficiency of the algorithms against the optimization supported by the external tools. Also, further research is required in extending the scope of the optimization algorithms and to accelerate the validation process by focusing the execution of the OCL statements avoiding time consuming expressions, such as *AllInstances*.

References

1 **Warmer J, Kleppe A**, *The Object Constraint Language: Getting Your Models Ready for MDA*, Addison Wesley, 2003. Second Edition.
2 **Mezei G, Lengyel L, Levendovszky T, Charaf H**, *Extending an OCL Compiler for Metamodeling and Model Transformation Systems: Unifying the Twofold Functionality*, INES (2006).
3 available at `http://avalon.aut.bme.hu/tihamer/research/vmts`. VMTS Web Site.
4 **Lengyel L, Levendovszky T, Charaf H**, *Compiling and Validating OCL Constraints in Metamodeling Environments and Visual Model Compilers*, IASTED (2004).
5 **Mezei G, Lengyel L, Levendovszky T**, *Implementing an OCL 2.0 Compiler for Metamodeling Environments*, 4th Slovakian-Hungarian Joint Symposium on Applied Machine Intelligence, SAM, 2006.
6 **Mezei G, Lengyel L, Levendovszky T, Charaf H**, *Minimizing the Traversing Steps in the Code Generated by OCL 2.0 Compilers*, WSEAS Transac-

tions on Information Science and Applications, Vol. 3, February 2006. Issue 4, pp. 818-824.

7 *Object Constraint Language Environment*, available at `http://lci.cs.ubbcluj.ro/ocle/`.

8 **Hamie A, Howse J, Kent S**, *Interpreting the Object Constraint Language*, Proceedings 5th Asia Pacific Software Engineering Conference (APSEC '98), Taipei, Taiwan, 1998.

9 *Dresden OCL Toolkit*, available at `http://dresden-ocl.sourceforge.net/index.html`.

10 *SableCC*, available at `http://sablecc.org/`.

11 **Akehurst D, Linington P, Patrascoiu O**, *OCL 2.0: Implementing the Standard*, Technical report, Computer Laboratory (November 2003). University of Kent.

12 *Open Source Library for OCL*, available at `http://oslo-project.berlios.de/`.

13 *Flex, Official Homepage*, available at `http://www.gnu.org/software/flex/`.

14 *Bison, Official Homepage*, available at `http://www.gnu.org/software/bison/bison.html`.

15 **Thuan T, Hoang L**, *"NET Framework Essential", O'Reilly*, 2003.

16 **Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman**, *Compilers Principles, Techniques, and Tools*, Addison – Wesley, 1988.