Restrictions for OCL constraint optimization algorithms

Gergely Mezei, Tihamér Levendovszky, Hassan Charaf

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

Abstract. Efficient constraint handling is essential in UML, in metamodeling, and also in model transformation. OCL is a popular, textual formal language that is used in most of the modeling frameworks to express constraints. Our research focuses on the optimization of OCL handling. Previous works have presented algorithms that can accelerate the constraint validation by rewriting and decomposing the constraints and caching the model queries. Although these algorithms can be used in general, there are special cases, where additional restrictions apply. The aim of this paper is to present these refined restrictions and the extended optimization algorithms.

1 Introduction

Metamodeling techniques can describe the rules of Domain Specific Modeling Languages (DSMLs), but these descriptions mainly consist of topological rules only. The available model items, their attributes and the possible relations between the items can be defined, but these definitions have a tendency to be incomplete, or imprecise. For example, there is a resource editor domain for mobile phones. Here, it is useful to define the valid range for slider controls that cannot be accomplished using metamodeling techniques. Another example is a metamodel that defines a DSML with computer networks. A single computer can have input and output connections, but these connections use the same cable with maximum n channels. Thus, the number of maximum available output connections equals the total number of channels minus the current number of input channels. Such constraints cannot be expressed by metamodel rules.

The real need for constraints applies also to graph rewriting-based model transformation [1]. Here the Left Hand-Side (LHS) of the rewriting rules define the pattern to find in the host graph. Beyond the topology of the visual models, additional constraints must be specified. Model transformations constraining the pattern matching are very popular, they are used for example in QVT [2]. Additionally, dealing with constraints means a solution to several unsolved model transformation issue [1].

One of the most wide-spread approaches to constraint handling is the Object Constraint Language (OCL) [3]. OCL is a flexible formal language. It was originally created to extend the capabilities of UML [4], but due to its flexibility, it can also be used in metamodeling environments with minor extensions [5].

Nowadays OCL is becoming essential both in metamodel-based model validation and model transformations.

Visual Modeling and Transformation Systems (VMTS) [6] is an n-layer metamodeling and model transformation tool. VMTS uses OCL constraints in model validation and also in the graph rewriting-based model transformation [1]. VMTS contains an OCL 2.0 compliant constraint compiler that generates a binary executable for constraint validation [7]. The constraints contained both by the rewriting rules and by metamodel diagrams are attached to the metamodel, thus they can be handled with the same algorithms.

Previous papers [8] and [9] have presented three optimization algorithms. These algorithms can reduce the navigation steps in the constraints (i) by relocating the constraints, (ii) separating clauses based on Boolean operands and (iii) caching the result of the model queries applied during validation. The main advantage of the algorithms is that they do not rely on system-specific features, thus, they can easily be implemented in any modeling or model transformation framework. The general correctness of the algorithms has also been proved.

While implementing and by further examining these algorithms, we have refined their application conditions. We have found that the scope of usability of the first algorithm is limited. Furthermore, the second algorithm can accelerate the validation in certain cases only, according to the type of the Boolean operand. The cases where the decomposition to clauses are meaningless, thus, the advantage of the optimization that equals zero have to be excluded from the algorithm. The primary aim of the paper is to present these restrictions and the extended algorithms.

The paper is organized as follows: firstly, Section 2 elaborates the original version of the two optimization algorithms. Secondly, Section 3 introduces the limitations of the algorithms, while Section 4 presents the new, extended algorithms. Finally, Section 5 summarizes the presented work.

2 Backgrounds and Related Work

In general, the evaluation of OCL constraints consists of two steps: (i) selecting the object and its properties that we need to check against the constraint and (ii) executing the validation method. Although the second step can use several OCL-related optimization methods, our optimization algorithms focus on the first step, bacause: (i) The efficiency of the validation depends on the realization of the OCL library (types and expressions), thus, optimizing the validation process is usually more 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 algorithms were published in [8] and in [9].

2.1 Relocation

One of the most efficient way to accelerate the constraint evaluation is to reduce the navigation steps in a constraint. 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. The main *foreach* loop examines the navigation paths of the actual constraint and relocates the constraint to the node at the smallest navigation cost. Here, relocation means changing the context of the constraint without changing the result of the evaluation.

Algorithm 1 RELOCATECONSTRAINT algorithm

```
1: RelocateConstraint(Model M)
2: for all InvariantConstraint C in M do
3:    minNumberOfSteps = CalculateSteps(CurrentNode in C)
4:    optimalNode = CurrentNode of the C
5:    for all Node N in C do
6:       numberOfSteps = CalculateSteps(N)
7:       if numberOfSteps < minNumberOfSteps then
8:          minNumberOfSteps = numberOfSteps
9:          optimalNode = N
10:   if optimalNode ≠ CurrentNode of C then
11:      UpdateNavigations of C
12:      Relocate C to optimalNode
```

2.2 Decomposition

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 ($self.age = self.teacher.age$). Thus, using the *RelocateConstraint* algorithm, it is not always possible to eliminate all navigation steps. Although these 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/OR/XOR/IMPLIES) between them. By separating the clauses, we can reduce the number of the navigation steps contained by the OCL expressions and the complexity of the constraint evaluation during the constraint validation process. It is simpler to evaluate the logical operations between the members of a clause than to traverse the navigation paths contained by the constraints.

The ANALYZECLAUSES algorithm (Algorithm 2) works on the syntax tree of the constraint. The algorithm is invoked for the outermost OCL expression of each invariant, recursively searches the constraint for possible clause expressions and creates the clauses. The algorithm uses the following rules: (i) A clause is created for every logical expression, the two sides of the expression are added to the clause as children. The children are recursively checked to decompose nested Boolean relations. (ii) Parentheses are eliminated, the inner expressions are checked. (iii) In other cases, if there is only one expression in the whole

constraint, then a special clause is created, otherwise the *RelocateConstraint* algorithm is used on the expression.

Algorithm 2 AnalyzeClauses algorithm

```
1: AnalyzeClauses(Model Exp)
2: if Exp is LogicalExpresssion 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)
```

3 Contributions

In general, there are two key questions in connection with optimization algorithms: (i) whether they result in the same output as the original algorithm for every possible input and (ii) whether they are more efficient. The first question is crucial, because having proper evaluation results is essential. These guidelines, these two questions are taken into examination when constructing the limitations for the optimization algorithms.

3.1 Correctness

Primarily, the correctness of the relocation algorithm is taken into examination. An algorithm or a relocation is *correct* only if the output of the optimized and original constraint is the same for every possible input. 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 often 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. 1). 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*.

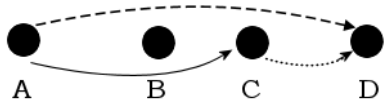

Fig. 1. The steps and the whole RelocationPath

Proof. 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 paths are allowed. 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 $O_1,\ O_2 \dots O_n$, and $N_1,\ N_2 \dots N_n$.

Example 2. Fig. 2 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. 2/a, M is its instantiation (Fig. 2/b). O is Computer, N is Monitor in M_0. O has two instantiations, Computer1 (O_1) and Computer2 (O_2).

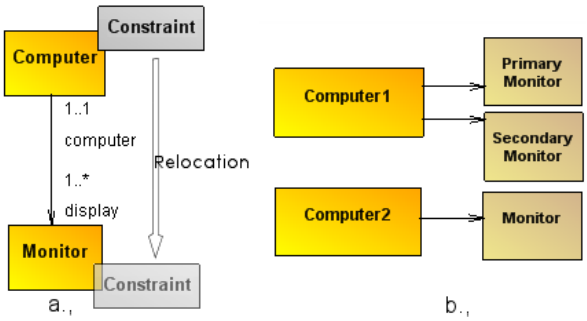

Fig. 2. Example metamodel and model

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*.*

Proof. Let M be a model with O_1, N_1 *and* N_2 defined (Fig. 3). Let N_1 be isolated (or at least not connected with O_1).

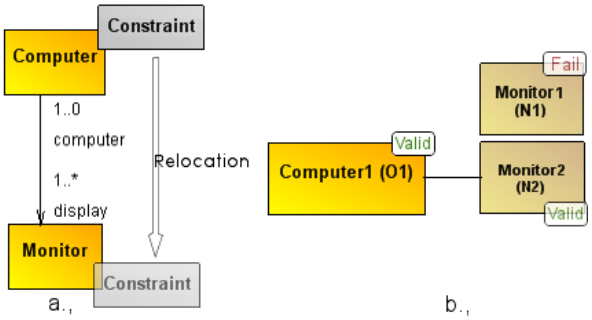

Fig. 3. Null multiplicity - metamodel and model

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. 4). 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': $Type = \ 'LCD'$.

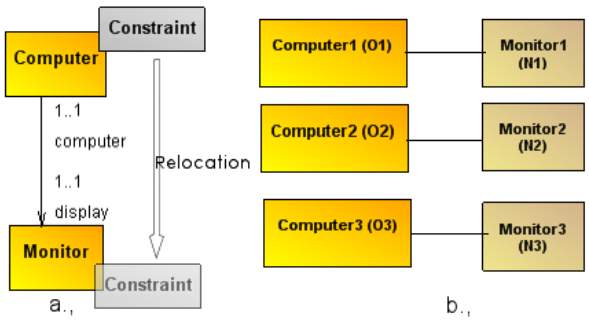

Fig. 4. ExactlyOne multiplicity on both sides - metamodel and model

Proof. 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. covers 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. 5).

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. 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.

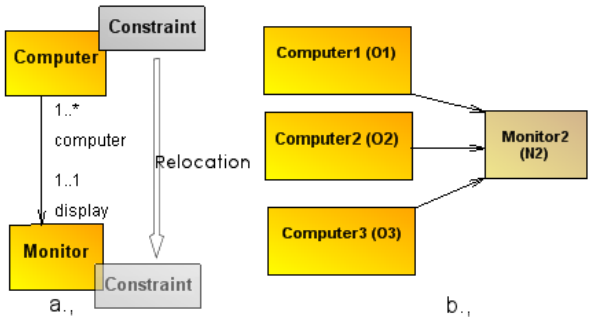

Fig. 5. MoreThanOne → ExactlyOne multiplicity - metamodel and model

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. 6). 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.

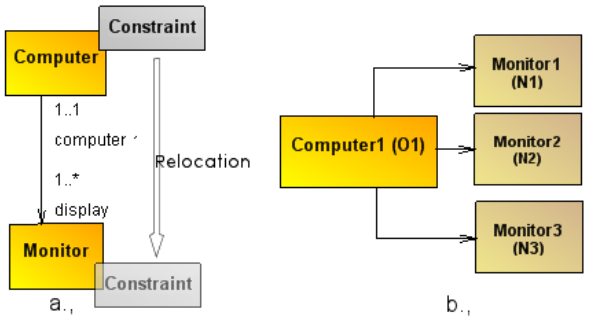

Fig. 6. ExactlyOne → MoreThanOne multiplicity - metamodel and model

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. 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, an 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 side (not allowing zero multiplicity) (Fig. 7), 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.*

Proof. 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

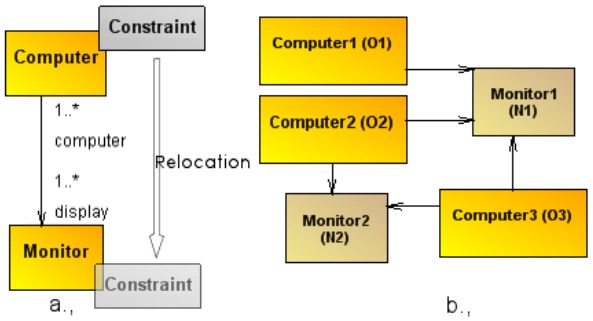

Fig. 7. MoreThanOne multiplicities - metamodel and model

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. 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.*

One of the main difference between the *RelocateConstraint* and the *AnalyzeClauses* algorithm is that the first one modifies the constraint only by relocating it, but the algorithm does not need special support from the validation framework. In contrast, the second algorithm does not really modify the text of the constraint, but it requires support for clause-handling during validation. Therefore, *AnalyzeClauses* relies more on the framework, but depends less from the constraint text.

Proposition 8. *The original version (Algorithm 2) of the constraint decomposition algorithm (*AnalyzeClauses*) is always* correct *if the supporting functions are* correct*.*

Proof. The algorithm *AnalyzeClauses* consists of a condition and several method calls in the condition branches. The conditions ensures that the correct execution branch is selected in all cases. Thus, the only part of *AnalyzeClauses* that can cause erroneous results is the clause-handling, which is an external function. If this external function is implemented well, the decomposition is always correct.

3.2 Efficiency

RelocateConstraint can reduce the number of navigation steps in the constraint, but since the optimization uses only the metamodel, not the models, it does not know exactly how many model items are affected by a single navigation step. If the model uses ExactlyOne multiplicities only, then the optimization is *correct*, but the cost of navigation is not predictable if the model contains MoreThanOne multiplicities. In this case the number of model items on the destination side can vary, thus, for example, the algorithm cannot decide between two paths different only in relation with MoreThanOne multiplicities. This problem can be handled using heuristics, but a globally optimal method cannot be constructed.

The situation is completely different in case of *AnalyzeClauses*. Here the performance gained from the optimization depends on how efficient the construction of the clauses is. The basic idea behind the algorithm 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.Size>50 and self.display.Size>80` it is enough to check the value of the first operand if it evaluates to *false*. This is why the boolean operators are special, why the *AnalyzeClauses* algorithm is based on them instead of other types of operations.

Since the operands cannot affect each other, they can be evaluated separately according to [9]. In case of AND, OR and IMPLIES operations the value of one operand can affect the results of the whole operation:

- 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 is evaluated.

Similar simplification is not available for XOR operations, because in this case both operands need to be evaluated.

4 The new algorithms

The restrictions to the optimization algorithms were presented in the previous section, now the last step is to construct new, extended algorithms according to these limitations.

The new *RelocateConstraint* is shown in Algorithm 3. It consists of two major parts: (i) searching for the optimal node (and RelocationPath) (Algorithm 4) and (ii) relocating the constraint if necessary (Algorithm 5).

Algorithm 3 The new RelocateConstraint algorithm

```
1: RelocateConstraint(Constraint, OriginalContext)
2: OptimalPath = SearchOptimalNode(OriginalContext, NULL)
3: if OptimalPath.LastElement ≠ OriginalContext then
4:    UpdateAndRelocate(Constraint,OptimalPath)
```

The first part of the *RelocateConstraint* algorithm is based on the *SearchOptimalNode* function. This function checks the relocation requirements while searching (*StepIsValid*), thus invalid*RelocationPath* candidates are dropped as soon as possible. *SearchOptimalNode* uses a recursive breadth-first-search strategy to find every possible candidates. The external function *CalculateSteps* calculates the number of model queries in the case when the new context is located in N.

Algorithm 4 The SearchOptimalNode algorithm

```
1: SearchOptimalNode(Node N, PathP)
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
```

The result of *SearchOptimalNode* is the *RelocationPath*. The last element of the path is the new context itself. If the new context and the old context are not the same, then the constraint is relocated and updated by the function *UpdateAndRelocate*. The relocation is based on path steps, thus, the algorithm updates the context declaration step-by-step. The multiplicity checking and the constraint updating mechanisms are implemented in external functions to improve the readability of the algorithm.

Algorithm 5 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) ≠ MoreThanZero then
6:       AddForeach C)
7:    if DestMultiplicity(Step) ≠ MoreThanZero then
8:       RemoveForeach(C)
9: return optimumCandidate
```

In the case of *AnalyzeClauses* there is only one new limitation: *XOR* operations are excluded when creating the clauses. The algorithm is presented in Algorithm 6.

Algorithm 6 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)
```

5 Conclusions

Due to the importance of constraints in modeling and model transformation, efficient validation methods are required. Previous work has presented three algorithms, which can accelerate the validation. This paper has examined the algorithms, especially the relocation algorithm *RelocateConstraint*. Based on the results, several necessary limitations and modifications have been introduced to the original algorithms. The statements have been illustrated by small examples

and their correctness has also been proved. More complex examples — focusing on the acceleration gained from the optimization — can be downloaded from [6]. According to the novel results, the algorithms have been updated.

As this paper has shown, proving the correctness of the algorithms precisely is hard to manage. A mathematical formalism could help, but the current formalism of OCL is based on set theory, which is hard to use in examination of dynamic behavior. Abstract State Machines offer a technique that has successfully been used in many similar domains as formalism. Such a formalism could prove the correctness of the algorithms applying formal semantics. Therefore, we are currently working on the formalism of the algorithms either using and extending the old formalism, or creating a new, ASM-based formalism.

Although the steps of the three optimization algorithms have been made more rigorous, processing the OCL constraints is not optimal. The decomposition and the normalization of atomic expressions have reduced the navigation steps to the minimum, and the caching algorithm has reduced the number of queries, but further research is required to extend the scope of the optimization algorithms and accelerate the process. The validation process can be optimized by rewriting the constraints and avoiding time consuming expressions, such as *AllInstances*.

6 Aknowledgements

The paper is established by the support of the National Office for Research and Technology (Hungary).

References

1. Lengyel L., Levendovszky, T., Charaf H. : Compiling and Validating OCL Constraints in Metamodeling Environments and Visual Model Compilers, IASTED 2004
2. MOF QVT Specification, http://www.omg.org/docs/ptc/05-11-01.pdf
3. Warmer, J. , Kleppe, A.: Object Constraint Language, The: Getting Your Models Ready for MDA, Second Edition, Addison Wesley, 2003
4. UML 2.0 Specification homepage, http://www.omg.org/uml/
5. Mezei, G. , Lengyel, L. , Levendovszky, T., Charaf, H. : Extending an OCL Compiler for Metamodeling and Model Transformation Systems: Unifying the Twofold Functionality, INES, 2006
6. VMTS Web Site, http://vmts.aut.bme.hu
7. Mezei, G. , Levendovszky, T., Charaf, H. : Implementing an OCL 2.0 Compiler for Metamodeling Environments, 4th Slovakian-Hungarian Joint Symposium on Applied Machine Intelligence
8. Mezei, G. , Lengyel, L. , Levendovszky, T., Charaf, H. : Minimizing the Traversing Steps in the Code Generated by OCL 2.0 Compilers, WSEAS Transactions on Information Science and Applications, Issue 4, Volume 3, February 2006, ISSN 1109-0832, pp. 818-824.
9. Mezei, G. , Levendovszky, T., Charaf, H. : An Optimizing OCL Compiler for Metamodeling and Model Transformation Environments, Working Conference of Software Engineering, 2006 (accepted)