Electronic Communications of the EASST
Volume 5 (2006)


# Proceedings of the Sixth OCL Workshop OCL for (Meta-)Models in Multiple Application Domains (OCLApps 2006)

## Restrictions For OCL Constraint Optimization Algorithms

Gergely Mezei, Tihamér Levendovszky, Hassan Charaf

18 pages

Guest Editors: Dan Chiorean, Birgit Demuth, Martin Gogolla, Jos Warmer
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

# Restrictions For OCL Constraint Optimization Algorithms

**Gergely Mezei[1], Tihamér Levendovszky[2], Hassan Charaf[3]**

{gmezei[1], tihamer[2], hassan[3]}@aut.bme.hu
Budapest University of Technology and Economics
Goldmann György tér 3., 1111 Budapest, Hungary

**Abstract:** Efficient constraint handling is essential in UML, in metamodeling as well as 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 work 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.

**Keywords:** OCL, optimization, compiler, contraints

## 1 Introduction

Metamodeling techniques can describe the structural rules of Domain Specific Modeling Languages (DSMLs). 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 structural 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 the 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 also applies to graph rewriting-based model transformation [LLC04]. 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 [QVT]. Additionally, dealing with constraints means a solution to several unsolved model transformation issue [LLC04].

One of the most wide-spread approaches to constraint handling is the Object Constraint Language (OCL) [OCL]. OCL is a flexible formal language. It was originally created to extend the capabilities of UML [UML], but due to its flexibility, it can also be used in metamodeling environments with minor extensions [MLLC06]. Nowadays OCL is becoming essential both in metamodel-based model validation and model transformations.

Visual Modeling and Transformation Systems (VMTS) [VMTS] is an n-layer metamodeling and model transformation tool. VMTS uses OCL constraints in model validation as well as in the graph rewriting-based model transformation [LLC04]. VMTS contains an OCL 2.0 compliant constraint compiler that generates a binary executable for constraint validation [MLC06]. 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 [MLLCOPT06] and [MLCOPT06] 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, because (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 [MLLCOPT06] and in [MLCOPT06].

### 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 function *CalculateSteps* counts the number of steps, if the constraint would be relocated in a new context. The main *foreach* loop examines the navigation paths of the actual constraint and relocates the constraint to the node at the smallest navigation cost.

*UpdateNavigation* updates the navigation references, while *Relocate* applies the relocation. 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 ($age = 18$ *and* $name = 'Jay'$), 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 sub-terms 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.

The ANALYZECLAUSES algorithm (Algorithm 2) processes the syntax tree constructed from 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.

## 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

**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)
```

is essential. These guidelines, these two questions are taken into account when constructing limitations for the optimization algorithms.

### 3.1 Correctness

Primarily, the correctness of the relocation algorithm is examined. 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 we can apply *correct* relocations.

Since the original and the new context node are not always neighbors, the optimization stores a path between them. This path is called *RelocationPath*. *RelocationPath* does not allow circular references that could cause erroneous, infinite navigation sequences. Storing *RelocationPath* is necessary, because there can exist more then 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. 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). 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 (if they are* correct*), then the proposition states that the relocation from A to D is* correct *as well.*

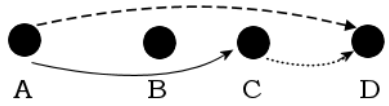


Figure 1: The steps and the whole RelocationPath

*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 as mentioned earlier. 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 path 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 (relocated) 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. 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 ($O_2$). Similarly, PrimaryMonitor is $N_1$, SecondaryMonitor is $N_2$, and finally, Monitor is $N_3$.*

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. A multiplicity allowing zero value means that there can be unconnected nodes in the relation. A multiplicity allowing a value greater than one means that navigation between the nodes must use a set operation. Since relations need different handling according their multiplicities, the following categories can be constructed:

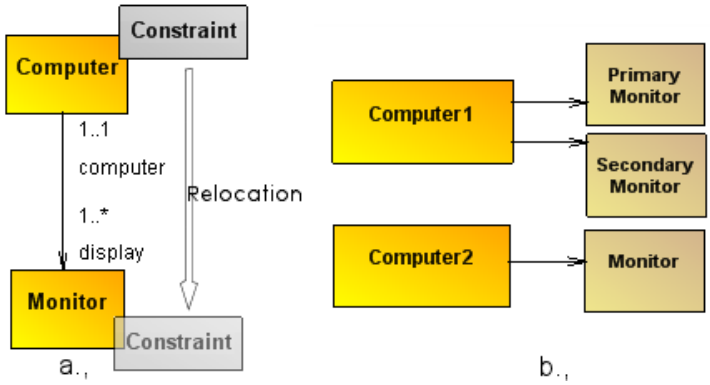


Figure 2: Example metamodel and model

- *ZeroOrOne* - the lower limit of the multiplicity is 0, the upper limit is 1
- *ZeroOrMore* - the lower limit of the multiplicity is 0, the upper limit is more than 1
- *ExactlyOne* - the lower and the upper limits are 1
- *OneOrMore* - the lower limit of the multiplicity is 1, the upper limit is more than 1

Multiplicities on the source and on the destination side have different meanings, thus, possible modeling structures can be described by a combination of the presented multiplicity categories. For example in the previous metamodell (Fig. 2) the multiplicity is *ExactlyOne* (source side) - *OneOrMore* (destination side).

The following propositions are based on the multiplicity combinations. This first case is the most simple, allowing only *ExactlyOne* multiplicity on both sides.

**Proposition 2** (Case A) *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) only. 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 presented above define that computers are able to handle exactly one monitor, and monitors are always connected to exactly one computer (Fig. 3).*

*Furthermore, let the constraint C state that the monitor is an LCD monitor* $(display.Type = 'LCD')$*. In this case relocating the constraint will result* $C'$*:* $Type = 'LCD'$*.*

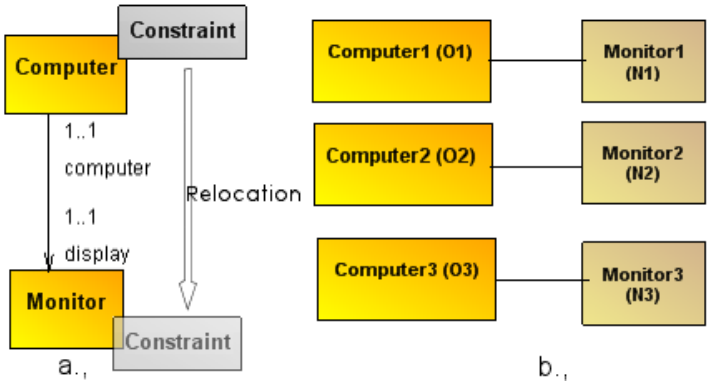


Figure 3: ExactlyOne multiplicity on both sides - metamodel and model

*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 remains 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 apply the expression there.

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

Handling multiplicity different than *ExactlyOne* on the source side can be applied by rewriting the constraints:

**Proposition 3** (Case B) *Navigation edges that allow zero multiplicity (ZeroOrOne, or ZeroOr−More) on the* source side *can be used in* RelocationPath *by encapsulating the original constraint by a* `notEmpty` *expression that checks whether the original context is accessible from the new one. If the original context cannot be reached from the new context, then the constraint results* true *showing that the expression cannot be evaluated (it does not violate the invariant).*

*Example* 4 *Fig. 4 shows an example metamodel and model for the proposition. Let C be defined as* `self.monitor.Price< 50`*. If this constraint is relocated, then it is transformed to*

```
if (self.computer->notEmpty()) self.Price < 50 else true
```

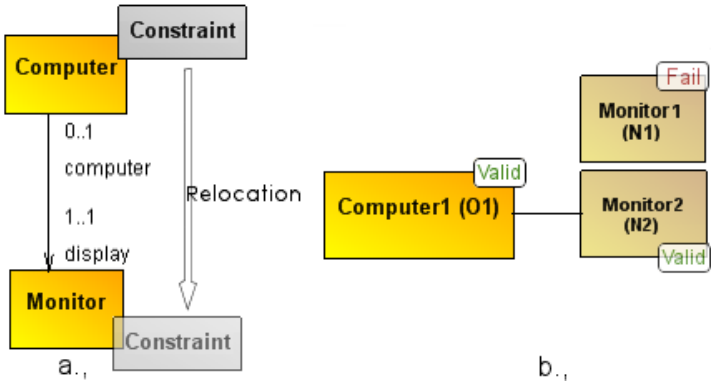


Figure 4: Zero multiplicity - metamodel and model

*expressing that monitors having at least one computer attached, has to have a price less then 50$. The invariant is automatically automatically evaluated to true in Monitor*1*.*

*Proof.* 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 $O_1$). If the source class is presented, then the evaluation can be applied the same way as in Case A. This is the case in $N_2$. If the source class is not present, then the original constraint does not check the destination class (as in $N_2$), thus the class is not required to check by the relocated constraints either. The encapsulating `notEmpty` expression ensures that only those nodes are checked, where the source class is presented. Thus, the relocation is always *correct*. □

**Proposition 4** (Case C) *If the multiplicity is ZeroOrMore, or OneOrMore on the* source side*, then the basic idea is to collect the original context nodes in a set, and iterate on this set to check the relocated constraint in each old contexts node. Therefore, the constraint expression is relocated by adding an encapsulating* `forall` *expression to the constraint (similarly to* `notEmpty` *in Case B). The* `forall` *expression iterates on all of the original context nodes, where the iterator values are the original context nodes (see example below). This also means that inside the* `forall` *expression, this case is similar to Case A. 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.*

*Example* 5 *Let O contain a simple constraint referring to one of its attributes, named* `Abstract`*. After the relocation, the constraint is located in N and the reference* `self.Abstract` *is transformed to*

```
self.O->forall(O | O.Abstract).
```

*This* `forall` *expression is true only if the condition holds for every elements in the set.*

*Example* 6 *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.*

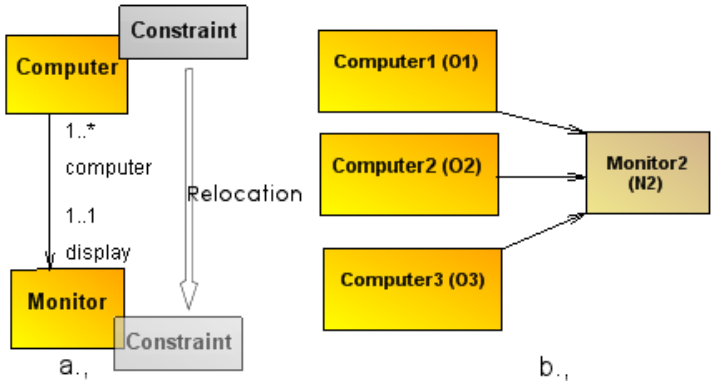


Figure 5: *MoreThanOne* → *ExactlyOne* multiplicity - metamodel and model

*Proof.* The presented method ensures that each model item on the original source side is processed, and the constraint is checked for all of them. If the multiplicity on the source side is *ZeroOrMore*, then it is necessary to check first whether a node with the original context is reachable (using the constructs of Case B), and apply the `forall` expression only if it is reachable. This method ensures that the navigation back to the original node is always possible. Therefore, inside the iteration, it is able to use the constructs of Case A, since in each iteration step there is exactly one node examined from the original context. Inside the `forall` loop, the name of the destination node is the iterator value to make rewriting simpler. Thus, the relocated and the original version are always equivalent. □

Multiplicity, other than *ExactlyOne* on the destination side is harder to handle, than on the source side. The following propositions describe when and how the relocation is possible in these cases:

**Proposition 5** (Case D) *Navigation edges that allow zero multiplicity (ZeroOrOne, or ZeroOrMore) on the* destination side *cannot be used in* RelocationPath.

*Proof.* Since the metamodell allows zero multiplicity on the destination side, therefore, it is possible to construct a model, where the constraint disappears during relocation: Let $M_0$ allow zero multiplicity between $O$ and $N$ (on the *destination side*). This means that a model $M$ — which contains the instantiations of $O$ only — is a valid model. If the the constraints of $O$ are relocated to $N$, then the constraint is completely eliminated in case of $M$. Since it is always possible to construct a counterexample, relocation cannot be used in this case. □

*Partial relocation* means that some of the expressions are executed in the new context, while others are executed in the original context. We use the term *semi-relocated* for expression executed in the original context, and *fully relocated* for expressions executed in the new context.

**Proposition 6** (Semi-relocation) *If the constraint contains more than one attribute reference expressions , then* partial relocation *is* always *feasible. The original context is reached using navigation. Partial relocation cannot be applied if zero multiplicity is allowed on the* destination side.

*Proof.* Since the proposition is true only for relations not allowing zero multiplicity on the *destination side*, the navigation between the original and the new context is always possible. This means that all relations can be traversed according to the constructs presented earlier. Therefore, it is always possible to navigate back to the original context and evaluate the constraint there. In this way, the relocated and the original functionality is the same. □

*Partial relocation* is useful, when the limitations of the relocation algorithm do not always allow executing the whole constraint in the new context as in the following cases.

**Proposition 7** (Case E) *If the multiplicity is OneOrMore on the* destination side*, then only those constraint expressions can be* fully relocated*, where the original expression uses* `forall`*, or* `not exists` *to obtain the referenced model items of the new context. This means that only those expressions can be used here, which selects all of the model items, or none of them (no partial selection, or another set operation is allowed).*

*Example* 7 *The expression* `self.N->count()` *or* `self.N->select(N.IsUnique)` *cannot be relocated, but the expression* `self.N->forall(N.IsUnique)` *can.*

*Example* 8 *The example model shows the requirements of the proposition (Fig. 6).*

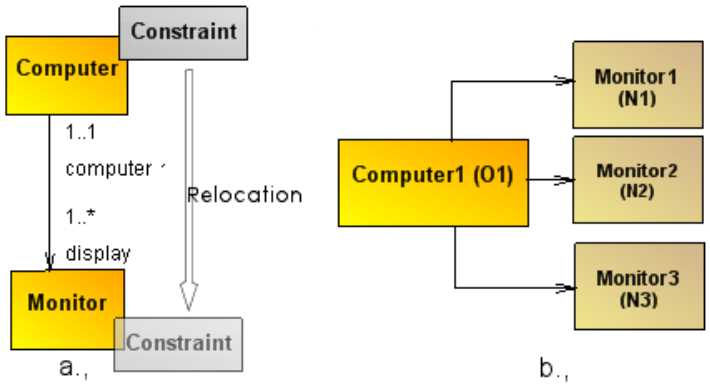


Figure 6: *ExactlyOne → OneOrMore* multiplicity - metamodel and model

Note that due to the preconditions of the proposition, the references to Monitor are always set

operations in Computer. For example, the expression `self.display.Price> 300` cannot be used, because `display` is a set, not a single value.

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 the constraints to use `forall` operation to query the attributes of the new context, or the navigation paths through the new context. The proposition does not restrict other type of operations, which do not navigate to the new context (for example a local attribute queries, such as in *c1*). In this case the rules of Case A can be used, thus, $C_1'$ becomes the following:

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

Complex set operations (e.g. $C_2$) cannot be fully relocated according to the proposition. 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` operations. 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 8** (Case F) *If the multiplicity is the combination of Cases A to E, then the combination of the previous propositions must be used:*

*Rule 1* *If the destination side allows zero multiplicity, then no relocation is possible, in any other cases relocation can be applied.*

*Rule 2* *If the destination side allows MoreThanOne multiplicity, but the expression uses not a* `forall` *or a* `not exists` *operation to navigate to the new context, then the expression is semi-relocatable only. Rule 2 can be used only to cases not violating Rule 1.*

*Rule 3* *In other cases full relocation is always possible:*

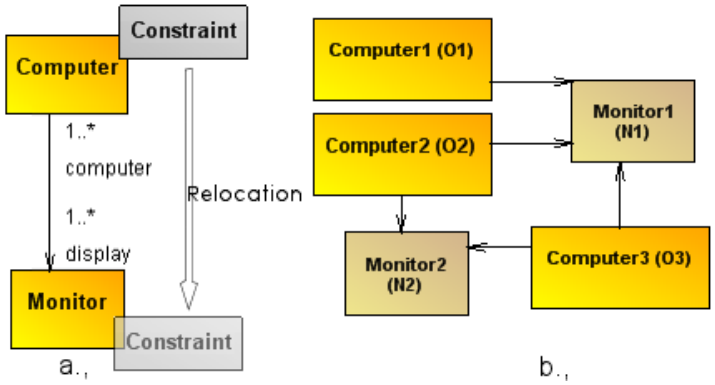


Figure 7: OneOrMore multiplicities - metamodel and model

*3/a* *If the source side allows zero multiplicity, then the expression must be encapsulated by a* `if (notEmpty)` *check.*

*3/b* *If the source side allows multiple multiplicity, then the expression must be encapsulated by a* `forAll` *operation.*

*3/c* *If the destination side allows multiple multiplicity and full relocation is possible, then the encapsulating* `forall` */* `(not exists)` *operation must be avoided.*

*3/d* *The rules of Case A must be used to rewrite the expression, modified by the previous steps.*

*Rule 3 can be used only to cases not violating Rule 1 or 2.*

*Example* 9 *The example (Fig. 7) shows a possible combination of the previous cases. Computers can have multiple monitors, and monitors can be attached to any number of computers. The proposition states that relocation is able in this concrete case by combining Case C and Case E and use Case A in rewriting indirectly.*

*Proof.* (i) If the multiplicity on the source side, or on the destination side is *ExactlyOne*, then one of the Case A - Case E can directly be used. Case F is useful if it is not this case.

(ii) Rule 1, which is based on Case D ensures that the constraint will not disappear during relocation. Since the multiplicity on the source side does not restrict the scope of relocation (according to Case A - Case C), thus, partial relocation is always possible and all expressions can be at least semi-relocated (Prop. of Semi-relocation).

(iii) Rule 2 separates the semi-relocatable and fully-relocatable expressions according to Case E. *ExactlyOne* multiplicity on the destination side cannot occur because of (i).

(iv) Rule 3 describes how the constraint rewriting must be applied. First the encapsulating expressions are used (Case B - Case C). These encapsulating expressions simulate *ExactlyOne* multiplicity on the source side, which means that the constraint expressions can be handled inside the encapsulating expressions as if the multiplicity where *ExactlyOne*. This also means that afterwards, the constructs of Case E can be used to rewrite the constraint. □

**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 on the* destination side *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. Note that *AnalyzeClauses* is useful mainly in a special case of partial relocation, where different parts of the constraints have different optimal context and the semi-relocatable and fully-relocatable are independent. Independent expressions in this context means that the result of either expression cannot affect the other. This independency between the two kinds of expressions allows to create clauses from the original constraint as described in Algorithm 2.

**Proposition 9** *The original version (Algorithm 2) of the constraint decomposition algorithm (*AnalyzeClauses*) is always* correct *if the external functions used in the algorithm are* correct.

*Proof.* The algorithm *AnalyzeClauses* consists of a multi level condition (Line 2, 8) and several external function calls in the condition branches (Line 2-5, 8, 11-13, 16). The conditions ensures that the correct execution branch is selected in all cases according to the type of the current expression. In the branches of the conditions clauses are created from the original expression (Line 3, 12), or the *RelocateConstrain* is used (Line 16). Creating clauses means to split the original expression in smaller or equal parts. *RelocateConstraint* is proved to be correct according to the previous propositions. This means that the only part of *AnalyzeClauses* that can cause erroneous results is the clause-handling (validating the expressions in the clauses and summarizing the results), 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 is allowed to used the metamodel only, without 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 performance gained from the optimization is predictable, but the cost of navigation is not predictable in any other cases. For example, if the number of model items on the destination side can vary, then the algorithm cannot decide between two paths different only in relation with *OneOrMore* 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 [MLCOPT06]. 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 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 6).

**Algorithm 3** The new RELOCATECONSTRAINT algorithm

```
1: RELOCATECONSTRAINT(Constraint, OriginalContext)
2: OptimalPath = SEARCHOPTIMALNODE(Constraint, OriginalContext, NULL)
3: if OptimalPath ≠ OriginalContext then
4:     UPDATE(Constraint,OriginalContext, OptimalPath)
```

The first part of the *RelocateConstraint* algorithm is based on the *SearchOptimalNode* function. This function checks the relocation requirements while searching (*StepIsValid*, Algorithm 5), 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*.

The algorithm *StepIsValid* is based on the propositions presented in the paper. It checks the multiplicity on the *destination side* and decides whether the relocation step is possible. The external function *CheckForAllExpr* checks whether the outermost expression is a `forall`, or a `not exists` expression.

The result of *SearchOptimalNode* is the *RelocationPath*. If the new context and the old context are not the same, then the constraint is relocated and updated by the algorithm *Update*. The relocation is based on path steps, thus, the algorithm updates the context declaration step-by-step. The the constraint updating mechanisms are based on the presented propositions, they are implemented in external functions to improve the readability of the algorithm.

**Algorithm 4** The SEARCHOPTIMALNODE algorithm

```
1: SEARCHOPTIMALNODE(Constraint C, Node N, PathP)
2: minSteps = CALCULATESTEPS(C,N)
3: optimumCandidate = APPEND(P, N)
4: for all CN in CONNECTIONS(N) do
5:    if STEPISVALID(C, CN) then
6:       LocalOptimum = SEARCHOPTIMALNODE(UPDATE(C, N, CN), APPEND(P, CN))
7:       LocalSteps = CALCULATESTEPS(C,LocalOptimum.LastElement)
8:       if LocalSteps < minSteps then
9:          minSteps = LocalSteps
10:         optimumCandidate = LocalOptimum
11: return optimumCandidate
```

**Algorithm 5** The STEPISVALID algorithm

```
1: STEPISVALID(Constraint C, Step Step)
2: if DESTMULTIPLICITY(Step)= ExactlyOne then
3:    return true
4: else
5:    if DESTMULTIPLICITY(Step)= ZeroOrOne or
       DESTMULTIPLICITY(Step)= ZeroOrMore then
6:       return false
7:    else
8:       return CHECKFORALLEXPR()
```

**Algorithm 6** The UPDATE algorithm

```
1: UPDATE(Constraint C, Node O, Path P)
2: for all Step in P do
3:    if SOURCEMULTIPLICITY(Step)= ExactlyOne) then
4:       CHANGECONTEXTSIMPLE(C,Step)
5:    else
6:       if SOURCEMULTIPLICITY(Step)= ZeroOrOne or
         SOURCEMULTIPLICITY(Step)= ZeroOrMore) then
7:          ADDISEMPTYCHECK(C,Step)
8:       if SOURCEMULTIPLICITY(Step)= OneOrMore or
         SOURCEMULTIPLICITY(Step)= ZeroOrMore) then
9:          ADDFORALL(C,Step)
10:   if DESTMULTIPLICITY(Step) ≠ ExactlyOne then
11:      REDUCEORIGINALFORALL(C,Step)
12: return C
```

In the case of *AnalyzeClauses* there is only one new limitation: *XOR* operations are excluded when creating the clauses. Thus, the algorithm — presented in Algorithm 7 — is similar to the original one.

**Algorithm 7** 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)
```

### 4.1 A 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. 8/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. 8/b shows an example instantiation of the metamodel.

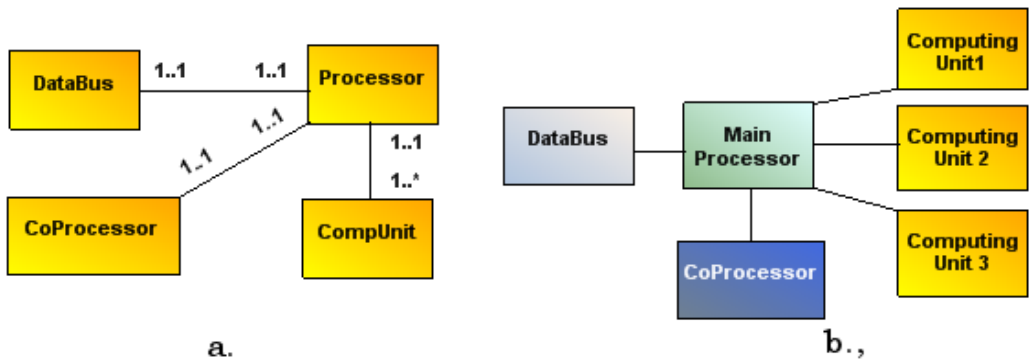


Figure 8: Case study Metamodel and Model

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 )
```

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, which means that the number of applied queries is reduced by 23%.

The optimizing compiler do not only modify the clauses, but adds the ability to cache the queries [MLCOPT06]. 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. The number of model queries is reduced in this case 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%.

## 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 improvements 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 [VMTS]. 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, our future work focuses 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*.

**Acknowledgements:** The found of "Mobile Innovation Centre" has supported, in part, the activities described in this paper. The authors would like to thank the valuable comments of the workshop participants, especially the helpful suggestions of Jordi Cabot.

## Bibliography

[LLC04] Lengyel L., Levendovszky, T., Charaf H. : Compiling and Validating OCL Constraints in Metamodeling Environments and Visual Model Compilers, IASTED, 2004, Innsbruck, Austria

[QVT] MOF QVT Specification, http://www.omg.org/docs/ptc/05-11-01.pdf

[OCL] Warmer, J. , Kleppe, A.: Object Constraint Language, The: Getting Your Models Ready for MDA, Second Edition, Addison Wesley, 2003

[UML] UML 2.0 Specification homepage, http://www.omg.org/uml/

[MLLC06] Mezei, G. , Lengyel, L. , Levendovszky, T., Charaf, H. : Extending an OCL Compiler for Metamodeling and Model Transformation Systems: Unifying the Twofold Functionality, INES, 2006, London, UK

[VMTS] VMTS Web Site, http://vmts.aut.bme.hu

[MLC06] Mezei, G. , Levendovszky, T., Charaf, H. : Implementing an OCL 2.0 Compiler for Metamodeling Environments, 4th Slovakian-Hungarian Joint Symposium on Applied Machine Intelligence

[MLLCOPT06] 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.

[MLCOPT06] Mezei, G. , Levendovszky, T., Charaf, H. : An Optimizing OCL Compiler for Metamodeling and Model Transformation Environments, Working Conference of Software Engineering, 2006, Warsaw, Poland, pp 61-73.