Proceedings of the 5th WSEAS Int. Conf. on Software Engineering, Parallel and Distributed Systems, Madrid, Spain, February 15-17, 2006 (pp55-60)

Optimization Algorithms For OCL Compilers

GERGELY MEZEI, LÁSZLÓ LENGYEL, TIHAMÉR LEVENDOVSZKY, HASSAN CHARAF
Automation and Applied Informatics
Budapest University of Technology and Economics
1111 Goldman György tér 3.
HUNGARY

Abstract: Constraint handling is one of the most focused research field in both model validation and model transformation. Constraints are often simple topological conditions such as multiplicity checks, but the main strength of the constraint validation lies in the textual constraints defined in high-level languages. Object Constraint Language (OCL) is a wide-spread formalism to express model constraints. We have found that OCL is also useful in graph transformation-based model transformation rules. There exist several interpreters and compilers that handle OCL constraints in modeling, but these tools do not support constraint optimization, therefore, the model validation is not always efficient. This paper presents algorithms to optimize OCL compilers, and accelerate the validation process. The presented algorithms were implemented in the OCL Compiler of Visual Modeling and Transformation System, and they were tested in both metamodels and transformation rules.

Key-Words: OCL, Compiler, Navigation Step, Metamodeling, Constraints, Model validation

1 Introduction

Models and model-based software development is one of the most focused research fields. The growing importance of modeling made customizable, flexible modeling languages popular. Domain Specific Modeling Languages (DSMLs) represent model elements with customized attributes in an editing environment. Domain specific modeling is rarely used by smaller developer groups, because they are very expensive. Metamodeling is a proven solution for this problem. Metamodels specify the modeling language, what kind of objects the modeler can use, what properties they have, and what connections one can create between them. The information represented by a model has a tendency to be incomplete, informal, imprecise, and sometimes even inconsistent. For example, a UML diagram, such as a class diagram, is typically not refined enough to provide all the relevant aspects of a specification. Besides other issues, there is a need to describe additional constraints about the objects in the model.

One of the most wide-spread approaches to constraint handling is the Object Constraint Language (OCL) [1]. OCL is a formal language that remains easy to read and write. Although OCL was created to extend the capabilities of UML [2], and define constraints for the model items, OCL can be used also in generic metamodeling environments to validate the models, or the define constraints in the model transformations.

Visual Modeling and Transformation Systems (VMTS) [3] is an n-layer metamodeling and model transformation tool, that grants full transparency between the layers (each layer is handled with the same methods). VMTS uses OCL constraints in model validation and in the graph rewriting-based model transformation steps. This paper presents the optimizing algorithms of an OCL compiler, which can be used to refine both metamodel instantiation and the applications of model transformation rules. Proofs are also provided to show that the optimized and the unoptimized code are functionally equivalent, i.e. for all input, the output of the constraint validation is the same.

2 Related work

There exist several modeling frameworks, and extension tools for frameworks that support OCL constraints in a more or less efficient way. This section mentions the most typical compilers only.

Oclarity [4] is an AddIn for one of the most popular modeling tools, Rational Rose. Oclarity supports syntactic, semantic validation of the constraints, and can correct minor mistakes (e.g. typos) in the constraint definitions. Oclarity is a

well-designed helper add-in, but the model validation is interpreting without optimization.

Object Constraint Language Environment (OCLE) [5] is a UML CASE Tool. OCLE helps the users to realize both static and dynamic checking at the user model level. The tool also has a user-friendly graphical GUI. Although the tool supports model checking, it does not use compiling techniques.

The Dresden OCL Toolkit (DOT) [6] 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 the SableCC. 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. Although DOT implements a real OCL compiler, it does not support metamodeling, or optimizing of the constraints.

3 The Compiler

The OCL Compiler realized in VMTS consists of several parts (Fig. 1). This section gives a short description of the architecture of the compiler, and introduces the main steps of the compilation. The presented information is required to understand the mechanisms of the optimization algorithms.

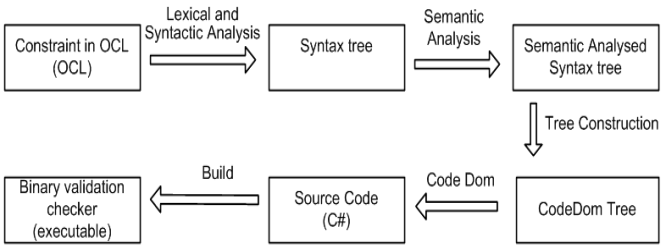

Fig. 1. The structure of the OCL Compiler

Firstly, the user defines the constraints in OCL, then the constraint definitions are tokenized and syntactically analyzed. The lexical analysis reads the constraint definition as a text, and creates a sequence of token such as the keywords of the language. Tokenization is performed by Flex [7]. Syntactic analysis build a Syntax tree using the grammar rules specified in OCL. Unfortunately, the grammar cannot be mapped directly into the OCL specification because of its ambiguities. The grammar rules are simplified to solve this problem (this information is reconstructed in the later compilation steps). Syntactic analysis uses Bison [8].

The Syntax tree does not contain every necessary information, it should be extended e.g. with type information, and implicit self references. This amendment is performed in the semantic analysis phase, and it produces the semantic analyzed syntax tree. The semantic analysis also reconstructs the simplification made in the grammar in the syntactical analysis phase.

In the next step, the constructed and semantically analyzed tree is transformed to a CodeDOM tree. CodeDOM is a .NET-based technology that can describe programs using abstract trees, and it can use this tree representation to generate code to any languages that is supported by the .NET CLR (like C#, Visual Basic or J#). The transformation to the CodeDOM tree is simple, because each node type in the syntax tree has an appropriate code sequence i.e. CodeDOM tree branch.

The compiler transforms the CodeDOM tree to C# source code. Packages are transformed to namespaces; contexts are realized with classes, and constraint expressions as public methods. To support the base types available in OCL, a class library was developed. The constraint classes are inherited from base classes implemented in this class library. To handle the invariants simply, the context class has always a method (*checkInvariants*) that calls the methods of the invariants one after the other.

Finally, the compiler compiles the source code and builds it. The output of the OCL compiler is an assembly (a .dll file) that implements the validation on the OCL constraint. The steps of the unoptimized OCL compiler are discussed in detail in [3], and in [9].

4 Optimization algorithms

The evaluation of the OCL constraint consists of two parts. (i) Selecting the object and its properties that we need to check against the constraint and (ii) executing the validation method. In general, the first

step has more serious computational complexity, thus, our optimization algorithms focus on minimizing the number of navigation steps required to check the constraint. Since each navigation step means a query on the database, the process can be optimized by reducing these steps. If the constraint does not contain any unnecessary navigation steps, then it is in *Canonical Constraint Form*, or simply it is *normalized*. The normalization (reducing the navigation steps) can accelerate the first part of the constraint evaluation. The primary aim of the introduced optimization algorithms is to provide a method to normalize the OCL constraints if it is possible.

4.1 Constraint relocation

The first normalization algorithm called RELOCATECONSTRAINT is shown in Fig. 2. The algorithm processes the OCL constraints propagated to the transformation step. The main foreach loop examines the navigation paths of the actual constraint. The algorithm calculates the number of all navigation steps for each node contained by the model. This calculation sums the number of the navigation steps which is necessary to reach all destination nodes of the original constraint *C* from the new place. The algorithm stores the most appropriate node (*optimalNode*) and finally updates the navigation paths and relocates the constraint to the optimal node.

```
RELOCATECONSTRAINT (Model M)
foreach InvariantConstraint C in M
  minNumberOfSteps = CALCULATESTEPS (CurrentNode in C)
  optimalNode = CurrentNode of the C
  foreach Node N in C
    numberOfSteps = CALCULATESTEPS(N)
    if (numberOfSteps < minNumberOfSteps) then
      minNumberOfSteps = numberOfSteps
      optimalNode = N
    endif
  end foreach
  if (optimalNode != CurrentNode of the C) then
    UPDATENAVIGATIONS of the C
    RELOCATE C to optimalNode
  endif
end foreach
```

Fig. 2. RELOCATECONSTRAINT algorithm

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

Proposition 1. Applying the RELOCATE-CONSTRAINT algorithm, each constraint is relocated in the node that implements the constraint using minimal number of navigation step if only constraint relocation is allowed. Using constraint relocation, the RELOCATECONSTRAINT algorithm eliminates all unnecessary navigation steps in non-decomposable (atomic) expressions.

Proof. Let *H* be an optional input model, and let *C* be an atomic OCL constraint which is propagated to *H*. Running the RELOCATECONSTRAINT algorithm results in the optimal node *A* to which we should assign the constraint. Assume that there exists another node (*B*) for which the following holds: if one links the constraint *C* to the node *B* and updates the navigation paths of *C*, then the constraint *C* contains fewer navigation steps, than if it had been propagated to the *A* node. The RELOCATECONSTRAINT algorithm visits all the nodes in the input model *H*, and it calculates for all nodes what the number of the navigation steps would be if the constraint were relocated to the actual node and the navigation paths were updated. Therefore, if the node *B* were better in the case of the constraint *C*, then it would be found by the RELOCATECONSTRAINT algorithm. That contradicts the assumption.

The goal of the constraint normalization is to achieve the pure canonical form, which does not contain navigation steps. Using RELOCATE-CONSTRAINT algorithm, it is not possible in all cases, because 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.secureId = self.manufacturer.secureId`).

4.2. Constraint decomposition

Although 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. 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.

It is useful to handle invariants uniformly, hence we have extended the clause definitions by a special type that can contain exactly one OCL expression (without any logical operation). This special clause is used only if the invariant does not contain any Boolean operators that can be used to decompose the constraint.

```
ANALYZECLAUSES (Expression Exp)
 if (Exp is LogicalRelationExpression) then
   Clause=CreateClause(Exp.RelationType);
   Clause.ADDEXPRESSION(ANALYZECLAUSES (Exp.Operand1));
   Clause.ADDEXPRESSION(ANALYZECLAUSES (Exp.Operand2));
   return Clause;
 else
  if (Exp is ExpressionInParentheses) then
    return ANALYZECLAUSES (Exp.InnerExpression);
  else
    if (Exp is OnlyExpressionInConstraint) then
      Clause=CreateClause(SpecialClause);
      Clause.ADDEXPRESSION(RELOCATECONSTRAINT(Exp));
      return Clause;
    else
      return RELOCATECONSTRAINT(Exp);
    endif
  endif
endif
```

Fig. 3. ANALYZECLAUSES algorithm

The ANALYSECLAUSES algorithm (Fig. 3) is invoked for the outermost OCL expression of each invariant. The decomposition works as follows: (i) If the expression is a logical expression, then a new clause is created with the appropriate relation type (and/or/xor/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 (recall that clauses can contain other clauses as children). The result clause is retrieved to handle the recursive calls. (ii) If the expression is an expression in 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 passed to the RELOCATECONSTRAINT algorithm, and then it is retrieved.

The constraint relocation algorithm is optimal only in case of non-decomposable constraints, hence the constraint decomposition should be processed firstly, and then the relocation (the processing order can not be changed). This statement can be proved with a simple example: `self.age = 18 and self.manager.name = 'Jay'`.

Proposition 2. Applying the ANALYZECLAUSES algorithm, the number of the navigation steps in the constraints contained by the output model is minimal (supposing that only the logical relations can be decomposed).

Proof. Let *H* be an optional input, and let *C* be an OCL constraint which is propagated to the *H*. Running the ANALYSECLAUSES algorithm results in the model *H'* with normalized constraints. Assume that there exists a normalized model of *H* (*H''*) which contains less navigation steps than *H'*. The ANALYSECLAUSES algorithm partition the constraint according to the logical expressions. The algorithm produces either expressions with at most one navigation step or complex expressions that cannot be processed further. The algorithm produces atomic expressions in both cases. In the next step, the decomposed parts are processed by the RELOCATECONSTRAINT algorithm to reduce the number of the navigation steps. Using the RELOCATECONSTRAINT algorithm, the number of the navigation steps in the constraints contained by the output model is minimal, because the expressions were atomic. That contradicts our assumption.

Proposition 3. Applying the RELOCATE-CONSTRAINT and ANALYSECLAUSES

algorithms for an optional input model does not modify the result of the constraint evaluation.

Proof. Let *H* be an optional input model, let *H'* be the result model of the ANALYZECLAUSES algorithm and let *H''* be the result model of the RELOCATECONSTRAINT after the constraint normalization. Assume that evaluating constraints contained by *H'* or *H''* produces different value than evaluating constraints contained by *H*. Both ANALYZECLAUSES and RELOCATE-CONSTRAINT algorithms update the navigation paths and the context information of the constraints. The algorithms do not modify the constraint conditions (cf. above the pseudo code and the descriptions of the algorithms), and they are processed sequentially (they do not affect the result of each other). That contradicts the assumption.

The optimization algorithms require an appropriate syntax tree, since, for example, the constraint relocation algorithm would not work if navigation calls were recognized as attribute calls because of the simplification made in the syntax analyzing step. Furthermore the constraint relocation step requires proper type information to query the available navigation destinations. Therefore the optimization steps are used between the semantic analysis and the code generation. The optimization algorithms must be executed only once for the specified constraints, and they accelerate the constraint validation for an arbitrary model.
The CodeDOM tree is constructed using the optimized syntax tree. Clauses are transformed to methods, and the parts of the clauses are compiled to method calls. All four Boolean operators are expressed by *or*, *and*, and *not* operations (e.g. *a xor b* is expressed as *a and !b or !a and b*).

5 Case study

Using a case study, we introduce how the optimization algorithms work. The case study is about a computer manufacturer company that produces CDs, flash memories, and pen drives. Although both the model and the constraint are very simple, it can show the algorithms in working.

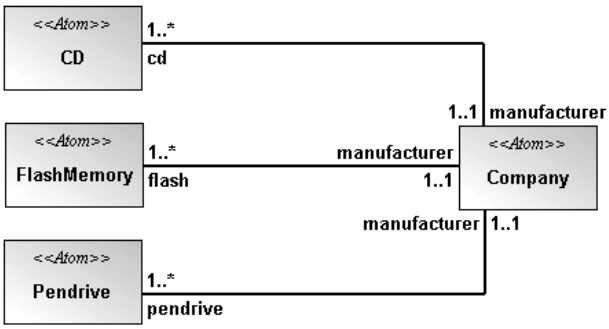

Fig. 4. Company – products case-study

The metamodel of the case study is shown in Fig 4. The attributes are not visible in the picture, *Company* has two attributes: *Name* (*string*), and *LastSerialIndex* (*integer*), *SerialBase* (*integer*). The products (all three types) have only one attribute: *SerialNumber* (*integer*).

The constraint (Fig. 5) checks the *SerialNumber* of the product (in this case the serial number of the CDs). The last valid *SerialNumber* for the products of the *Company* is computed by adding up *SerialBase* and *LastSerialIndex*. Another restriction is that the *SerialNumber* is only valid if it is greater than 0 independently from the manufacturer.

```
package CaseStudyPackage
context CD

inv serialChecker:
 self.SerialNumber>0
 and
  self.manufacturer.LastSerialIndex +
  self.manufacturer.SerialBase >
  self.SerialNumber
endpackage
```

Optimization executes the ANALYZECLAUSES algorithm on the invariant. The algorithm creates two clauses that are connected with an *and* operator. The first contains the simple condition (*SerialNumber* > 0), while the second checks the number according to the data stored in the manufacturer. The clauses are processed further using the RELOCATECONSTRAINT algorithm. The algorithm does not change the first clause, but the second clause is relocated to the manufacturer node. The relocation is necessary, because the constraint in the original node (*CD*) contains two navigation steps, while in the new node (*Company*) only one step is required.

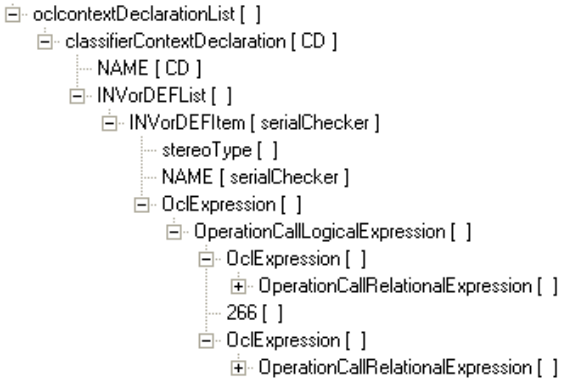

Fig. 6. Syntax Tree – before optimization

Fig. 6 shows the syntax tree of the constraint after the semantic analysis, but before the optimization. *ClassifierContextDeclaration* represents the invariant. The context of the invariant (*CD*) is shown in brackets. The *OperationCallLogical-Expression* expresses the subexpressions connected with *and* relation (node *266* in the picture).

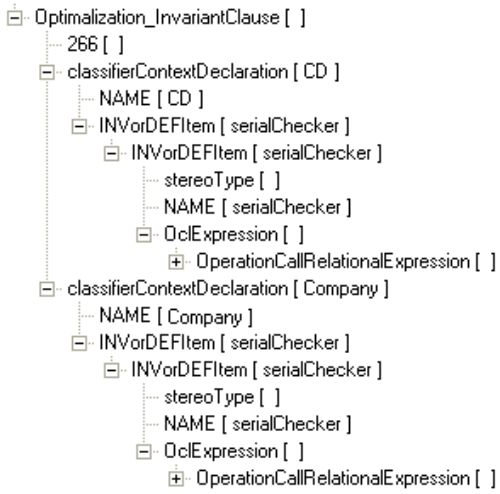

Fig. 7. Syntax Tree – after optimization

Fig. 7 shows the syntax tree after the optimization. A clause (*Optimalization_InvariantClause*) is created instead of the deleted context definition by the ANALYSECLAUSES algorithm, and RELOCATECONSTRAINT has changed the context of the second subclause to *Company*.

6 Conclusions

This paper has presented the main concepts of an optimizing OCL Compiler in an n-layer metamodeling and model transformation system. The paper has discussed the steps of the compiler construction in short from the lexical and syntactic analysis to the code generation. The compiler was extended by optimizing algorithms. The primary aim of the optimization was to normalize the constraints, therefore, constraint relocating and constraint decomposition techniques have been proposed. The correctness and the efficiency of the algorithms have been proven. Finally, a simple but illustrative case study has been shown the underlying mechanisms in operation.

Although two effective optimization algorithms were presented, processing the OCL constraints is not optimal. The decomposition and the normalization of the atomic expressions have reduced the navigation steps to the minimum, but further research is required to extend the scope of the optimization algorithms and accelerate the processing, focusing on constraint transformations besides the navigation steps.

Acknowledgements

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

References:

[1] Object Constraint Language Specification (OCL), www.omg.org
[2] UML 2.0 Specification http://www.omg.org/uml/
[3] VMTS Web Site http://avalon.aut.bme.hu/~tihamer/research/vmts
[4] Oclarity, http://www.empowertec.de/products/rational-rose-ocl.htm
[5] Object Constraint Language Environment, http://lci.cs.ubbcluj.ro/ocle/
[6] Dresden OCL Toolkit, http://dresden-ocl.sourceforge.net/index.html
[7] Flex, Official Homepage, http://www.gnu.org/software/flex/
[8] Bison, Official Homepage, http://www.gnu.org/software/bison/bison.html
[9] Gergely Mezei, Tihamér Levendovszky, Hassan Charaf, Implementing an OCL 2.0 Compiler for Metamodeling Environments, 4th Slovakian-Hungarian Joint Symposium on Applied Machine Intelligence