An optimizing OCL Compiler for Metamodeling and Model Transformation Environments

Gergely Mezei, Tihamér Levendovszky, Hassan Charaf

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

Abstract. Constraint specification and validation lie at the heart of modeling and model transformation. The Object Constraint Language (OCL) is a wide-spread formalism to express constraints in modeling environments. There are several interpreters and compilers that handle OCL constraints in modeling, but these tools do not support constraint optimization, therefore, the model validation can be slow. This paper presents algorithms to optimize OCL compilers to reduce the number of database queries during the validation process by eliminating the unnecessary traversing steps and caching the database queries. Proofs are also given to show that the optimized and the unoptimized code are functionally equivalent. The optimized compiler has been integrated into the Visual Modeling and Transformation System tool and applied to constraints appearing in both metamodels and graph rewriting-based model transformation rules.

1 Introduction

The information conveyed by a model created by a traditionally generic modeling language has a tendency to be imprecise [1]. For example, if a UML Class diagram [2] expresses a relation of type association between vehicles and the passengers traveling in the vehicle, the multiplicity between the two classes is 0..*, representing that several passengers can travel on a single vehicle. This multiplicity expresses that there is no upper limit to the number of passengers in general, because the limit depends on the type of the vehicle. Without additional techniques it is not possible to define that the maximum number of passengers equals the number of seats plus the number of standing rooms on the vehicle. Even if the generic modeling languages are extended with constraint handling, they cannot always describe the special attributes of the target domains. Thus, customizable models, modeling techniques, and model transformation algorithms are required by model-based software development. Domain Specific Modeling Languages (DSMLs) are a means to create customized models for domains where generic modeling languages would fail. Metamodeling is a proven solution for modeling DSMLs. The metamodel acts as a set of rules for the model level: it defines the available model elements, its attributes, and the possible connections between them. Metamodel definitions can usually define simple, topology-based rules, but they cannot express constraints for attribute values or other sophisticated requirements. Thus, sometimes the metamodeling rules are also incomplete. For example, if there is a resource editor domain for mobile phones, it is useful to define the valid range for slider controls. Specifying constraints in both generic and domain-specific models is

crucial to create precise and verifiable models. Constraint definitions are not only useful in modeling, but in model transformations as well. To define the transformation steps, beyond the topology of the visual models, additional constraints must be specified, which ensures, for example, value checking of the attributes. Dealing with constraints means a solution to several unsolved model transformation issue [3]. For example if the model transformation executes a search algorithm for non-abstract classes in a class diagram, then it is useful to express this condition. Constraint-based model transformation is very popular, it is used for example in QVT [4].

One of the most wide-spread approaches to constraint handling is the Object Constraint Language (OCL) [1]. OCL is a flexible, user-friendly yet formal language. Although it was created to extend the capabilities of UML [2], it can also be used in metamodeling environments to validate the models, or to define constraints in metamodel-based model transformations.

Visual Modeling and Transformation Systems (VMTS) [5] is an n-layer metamodeling and model transformation tool. VMTS uses OCL constraints in both model validation and in the graph rewriting-based model transformation [3]. VMTS contains an OCL 2.0 compliant constraint compiler to generate code for constraint validation. The constraints contained by both the rewriting rules and metamodel diagrams are attached to the metamodel, thus they can be handled with the same algorithms.

The primary aim of this paper is to give an overview on the optimizing algorithms used in the OCL compiler of VMTS. Previous work [6] has presented two efficient algorithms to reduce the navigation steps in the constraints by relocating the constraints and separating clauses based on Boolean operands. These algorithms are introduced in short, and they are extended with a third algorithm that can accelerate the database queries by an efficient caching technique. The paper also gives a concise description in which compilation step the optimization algorithms can be used and how the three algorithms can cooperate. Novel, detailed proofs are also discussed that the optimized and the unoptimized code are functionally equivalent.

The main advantage of the presented algorithms is that they do not rely on system-specific features, thus they can be easily implemented in any other modeling or model transformation framework. The algorithms do not require a specific implementation language, or database to store the models. The presented approach does not even need an environment based on a DBMS, it can be applied to all model repositories, such as MOF 1.4 repositories.

The paper is organized as follows: firstly, Section 2 elaborates some of the popular tools that support constraint checking. Section 3 introduces the previous work in short, while Section 4 presents the new results. Finally, Section 5 summarizes the presented work.

2 Related work

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

Object Constraint Language Environment (OCLE) [7] 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) [8][9] generates Java code from OCL expressions, and then instruments the system in five steps. (i) OCL expressions are parsed using a LALR(1) parser generated with SableCC [10]. The result of the step is an Abstract Syntax Tree (AST). (ii) A limited semantic analysis is performed on the AST to find errors. (iii) The AST is simplified in order to make the further processing simpler. (iv) The code generator traverses the simplified AST and builds Java expressions. (v) The generated code is inserted into the system that contains the constraint source code, thus, the contracts can be tested at runtime. DOT does not support metamodeling, or optimized constraint-checking.

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

Open Source Library for OCL (OSLO) [12] is a new tool and it is a further development of Kent OCL Library. OSLO is based on the Eclipse framework. OSLO supports OCL 2.0 functions for arbitrary metamodels based on EMF, and constraint checking for UML2 models (Eclipse UML2). OSLO supports therefore constraint checking for metamodeling system, but it cannot cooperate with model transformation systems. Since it is a recent project, only a few publications are available, and not all of the supported features are introduced in depth.

3 Backgrounds

3.1 VMTS OCL 2.0 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, more detailed information on the compiler can be found in [13] and [3].

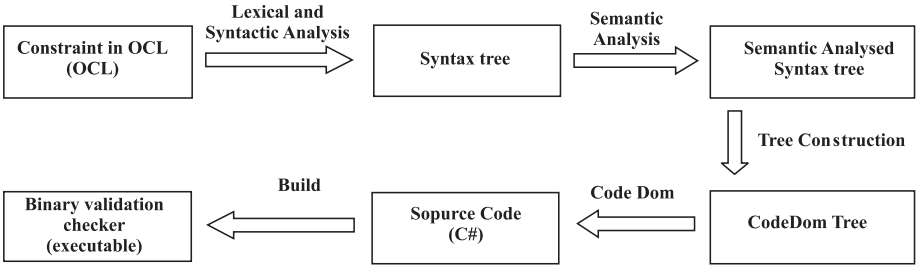

Fig. 1. VMTS OCL Compiler 2.0 Architecture

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. Syntactic analysis builds a syntax tree using the grammar rules of OCL specified in EBNF format [1]. To accommodate the ambiguities in the specification, the grammar rules are simplified. The information missing because of the simplification is reconstructed in the later compilation steps, where the analysis has more information (e.g. about available types and defined variables). Since the syntax tree does not contain all the 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 a semantically analyzed syntax tree. The semantic analysis also reconstructs the mentioned simplification made in the grammar. In the next step, the constructed and semantically analyzed tree is transformed to a CodeDOM tree. CodeDOM [14] is a .NET-based technology that describes 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#, or Visual Basic). The compiler transforms the CodeDOM tree to C# source code. To support the base types available in OCL, a class library has been developed. The constraint classes inherit from the base classes implemented in this class library. The output of the OCL compiler is a binary assembly (a .dll file) that implements the validation method.

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

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. Although the execution of the validation method can use several optimization methods, in this paper the presented algorithms focus on the first step. There are two reasons for this: (i) Since the efficiency of the validation depends on the realization of the OCL types and expressions, optimizing the validation process is usually more implementation-specific. (ii) In general, the first step has more serious computational complexity, because the model items are matched in the underlying model. If the model is stored in a DBMS, then each navigation step means a database query.

3.2 Normalization

If the constraint does not contain any unnecessary navigation steps, then it is in *Canonical Constraint Form*, or simply it is *normalized*. The normalization, namely reducing the navigation steps can accelerate constraint evaluation. The aim of the first introduced optimization algorithm, called RELOCATECONSTRAINT is to provide a method to normalize the OCL constraints if it is possible. The algorithm 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 and relocates the constraint to the node with the smallest navigation cost.

```
RELOCATECONSTRAINT (Model M)
foreach lnvariatConstraint C in M
  minNumberOfSteps = CALCULATESTEPS (CurrentNode in C)
  optimalNode = CurrentNode of the C
  foreach Node N in C
    numberOjSteps = CALCULATESTEPS(N)
    if(numberOJSteps < minNumberOfSteps) then
      minNumberOfSteps = numberOnSteps
      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. The Relocate Constraint algorithm

Using constraint relocation, the RELOCATECONSTRAINT algorithm eliminates all unnecessary navigation steps to produce non-decomposable (atomic) expressions. The proof of this statement, and the algorithm in more detail is discussed in [6].

3.3 Invariant decomposition

The goal of the constraint normalization is to achieve the pure canonical form, which does not contain navigation steps. Using the RELOCATECONSTRAINT 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 ($self.age = self.teacher.age$).

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.

The ANALYZECLAUSES algorithm (Fig. 3) is invoked for the outermost OCL expression of each invariant. The algorithm recursively searches the constraint for possible clause expressions and creates the clauses.

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) [6].

```
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 ExpressionInParantheses) 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. The Analyze Constraint algorithm

4 Contribution

4.1 Caching algorithm

Since the relocation and constraint decomposition algorithms can eliminate the *unnecessary* navigation steps only, the compiler cannot reach the pure canonical form in all cases. The clauses can also contain navigation steps, the validation still requires queries to obtain the model elements, and their attributes.

In compiler optimization, an occurrence of the expression E is called a *common subexpression* if the value of E has previously been computed, and it has not changed since then [15]. In these cases recomputing this expression can be avoided, because the value of the expression is already known.

Proposition 1. *In OCL constraints navigation steps and attribute references are always* common subexpression*s if they are used more than once* .

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

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

Collecting the statistical information set from the whole constraint expression is not straightforward, because sometimes only partial validation is required on a model. Thus, the caching algorithms are used at the context level, the statistical information of the different contexts are separated.

The GETCOMMONREFERENCES algorithm is shown in Fig. 4. The algorithm uses a breadth-first search to traverse the syntax tree recursively. It processes the attributes, the navigations, and the *control flow expressions*. The attributes and navigation expressions increment the statistic of their path reference (*IncReferencePath* method). To minimize the number of queries, the algorithm increments not only the reference of the full path, but also the references of the path steps. For example the expression $self.employee.wife.Name$ will increase the statistics with four entries: $self$, $self.employee$, $self.employee.wife$ and $self.employee.wife.Name$. The statistics contains even the $self$ element, because it is not cached always, if there is only one reference to it. This solution is useful if two expression have a common subset in the navigation steps, for example, in the expression $self.employee.wife.Name = {'Mrs.'} + self.employee.Name$, the path self.employee is used twice.

```
GET COMMON REFERENCES (CurrentNode)
switch(CurrentNode.Type)
    case AttributeDefinition:
        if(CurrentMode.HasOneChild) then
          IncReferencePath(SelfExpression, null)
        else
          IncReferencePath(Attribute,
              GETCOMMONREFERENCES(CurrentNode.Children))
        endif
        return
    case NavigationStep:
        IncReferencePath(ModelItem,
              GETCOMMONREFERENCES(CurrentNode.Children))
        return
    case ControlFlowExpression:
        GetMinimumReferencesForEveryExecutionPath()
        UpdateCurrentGlobalReferences()
        return
endswitch
if(CurrentNode.HasChildren) then
  GETCOMMONRENCES(CurrentNode.Children)
endif
```

Fig. 4. The Get Common References algorithm

The *control flow expressions* are complex expressions that have several execution paths, thus, they can affect the number of the references, for example conditional expression, or loops. In this case the algorithm should obtain the minimum number of the references for each referenced objects for each execution paths. For example in case of the conditional expressions this means that both branches are processed, statistical information is collected for both branches, and then the results are compared. For each

model reference path (attribute, or navigation reference), the minimum number is obtained and placed into the global statistical information set.

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

Proposition 2. *Using the REFERENCECACHING algorithm to evaluate the constraint the number of the applied queries is equal or less than that without optimization.*

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

Proposition 3. *Each attribute or navigation cached by the algorithm reduce the number of the database queries, namely no unnecessary caching is applied.*

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

4.2 An optimizing OCL compiler

In order to create the optimizing OCL compiler, the presented algorithms (i) have to be placed in the compiler control flow, (ii) a proper order of execution should be set, and (iii) proofs should show that the results of the optimized and unoptimized compiler are always the same.

The optimization algorithms require a semantically analyzed syntax tree, since, for example, the caching algorithms would not work without proper type-information.

Thus, the optimization algorithms are used after the semantic analysis. The constraint decomposition, relocation, and the statistical information retrieval algorithms are executed before the code generation phase, because they affect the syntax tree from which the code is generated. The CACHINGMANAGEMENT algorithm affects the code generation directly, it is used during the generation phase.

The next step is to set the order of execution of the optimization algorithms. Since the CACHINGMANAGEMENT algorithm is used in code generation compilation step it is executed as the last of the optimization algorithms. 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, and obviously, the processing order cannot be changed [6]. The GETCOMMONREFERENCES algorithm uses the syntax tree only, thus, it can be used both for processing clauses and normal constraints. At the same time the caching algorithm handles the contexts separated from each other. Since the constraint decomposition can change the contexts, for example it can divide them into several clauses, the GETCOMMONREFERENCES algorithm should be used after the decomposition. The relocation algorithm can also affect the context definitions by relocating the expressions into other contexts, thus, the execution order of the optimization algorithms is the following: (1) ANALYZECLAUSES, (2) RELOCATECONSTRAINT, (3) GETCOMMONREFERENCES, (4) CACHINGMANAGEMENT.

The last step to create the optimizing compiler is to prove that the optimization does not change the result of the validation.

Proposition 4. *Applying the optimization algorithms for an optional input model does not modify the result of the constraint evaluation.*

Proof. Let *H* be an optional input model, and let H' be the result model of the optimization executed by the ANALYZECLAUSES, RELOCATECONSTRAINT and REFERENCECACHING (GETCOMMONREFERENCES and CACHINGMANAGEMENT) algorithms. We prove that evaluating the constraints contained by *H'* produces always the evaluation in *H*.

Suppose that a constraint processed by the algorithm conflicts with the original constraint definition, because the cached references created and used by the REFERENCECACHING algorithm are not up-to-date. This contradicts Proposition 1.

In the RELOCATECONSTRAINT algorithm *UpdateNavigation*, and the *Relocate* function calls can modify the result, because other steps examine the existing constraints only. *UpdateNavigation* step replaces the existing context references with the new ones. The function *Relocate* does not modify the constraint but relocates it to a new model item. The functions together do not affect the result of the constraint according their definition.

The algorithm ANALYZECLAUSES can be divided into three main cases: (i) the examined expression is a complex (non-atomic) expression with Boolean operators; (ii) the examined expression is an expression between parentheses; (iii) or the expression is an atomic expression. The simplest case to examine is (ii), where the inner expression (the expression between the parentheses) is recursively processed. The evaluation order of the subexpressions is the same as that of the original expression, and since no further modification is made, therefore case (ii) does not affect the result of the constraints. Case (iii) has two subcases. If the examined expression is the only expression in the

constraint, then a special clause is created, and the relocated constraint is placed into it. The special clause type is required only because of the uniformity. The inner expression (the normalized constraint) is processed when it is validated as if it were not contained in any clauses. The second subcase applies when the examined expression is a part of the constraint. In this case the relocated expression is returned. In both subcases the result of the constraint is not modified. Case (i) is used only if the constraint consists of two subparts linked with Boolean operators. A clause is created that preserves the Boolean operator, and the subexpressions are recursively processed. The subexpression is processed individually when validating the constraint, and their results are connected using the operator (the order of the subexpressions are the same as in the original constraint). Therefore the result of the validation is modified only if the subexpressions cannot be processed independently. The independency is not true only if the first subexpression has an effect on the second subexpression, this means that the first expression modifies one or more value used in the second expression. These modified values can be either model attributes, or variables defined in the current scope. The constraints used in validation cannot modify the model according to the specification of OCL [1]. Local variables can be defined for example in *Iterate*, and *Let* expressions, but using any variable definition expression would mean that the outermost expression cannot be an expression linked with Boolean expressions. This means that the subexpressions of the clauses are independent, thus the result of the validation cannot be modified.

To summarize, the algorithms - if they are executed separately - relocate the constraint without changing its meaning, thus, the only way in which *H'* and *H* can have different results is that the algorithms affect each other, and thus their composition changes the meaning of the constraint. The algorithm REFERENCECACHING is executed independently from the other algorithms, and the proven correct output of the ANALYZECLAUSES is the input of the RELOCATECONSTRAINT algorithm. Thus, the result created by the composition of the algorithms is always correct.

5 Conclusions

This paper has presented the main concepts of an optimizing OCL Compiler in an n-layer metamodeling and model transformation system. The primary aim of the optimization was to reduce the number of database queries by normalizing and caching the constraints. Constraint relocating, constraint decomposition and caching techniques have been proposed. The correctness and the efficiency of the algorithms have been proven.

Optimizing OCL constraints is a rather new idea; none of the existing tools supports constraint optimization. This means that these tools can only use external optimization algorithms offered by the underlying applications, such as the query optimization in the underlying database system, or the code optimization of the executing environment. Although these external optimization algorithms are optimal in general, they (i) require system-specific (tool-specific) solutions and (ii) cannot use particular OCL-specific algorithms. For example, the executing environment that executes the validation code cannot recognize automatically that attributes are always common subexpressions. In contrast, the presented optimizing OCL compiler can use OCL-specific, but implementation- independent optimization algorithms. These

algorithms can be based on the characteristics of the OCL, which means a higher level of optimization. Furthermore optimizing compilers can also take the advantages of the underlying tools. We have accomplished several simplified performance tests, and we have found that the optimization can speed up the validation by 10-15% according to the circumstances. Since only basic tests were applied, further testing is required to give a detailed overview about the efficiency of the algorithms against the optimization supported by the external tools.

Although three efficient optimization algorithms have been 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, 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 constraint and avoiding time consuming expressions, such as *AllInstances*.

6 Acknowledgements

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

References

1. Jos Warmer, Anneke Kleppe, Object Constraint Language, The: Getting Your Models Ready for MDA, Second Edition, Addison Wesley, 2003
2. UML 2.0 Specification homepage, http://www.omg.org/uml/
3. László Lengyel, Tihamér Levendovszky, Hassan Charaf, Compiling and Validating OCL Constraints in Metamodeling Environments and Visual Model Compilers, IASTED 2004, Innsbruck
4. MOF QVT Specification, http://www.omg.org/docs/ptc/05-11-01.pdf
5. VMTS Web Site, http://avalon.aut.bme.hu/∼tihamer/research/vmts
6. G. Mezei, L. Lengyel, T. Levendovszky, H. Charaf, Minimizing the Traversing Steps in the Code Generated by OCL 2.0 Compilers, Issue 4, Volume 3, February 2006, ISSN 1109-0832, pp. 818-824.
7. Object Constraint Language Environment, http://lci.cs.ubbcluj.ro/ocle/
8. Ali Hamie, John Howse, Stuart Kent, Interpreting the Object Constraint Language, Proceedings 5th Asia Pacific Software Engineering Conference (APSEC ’98), December 2-4, 1998, Taipei, Taiwan, 1998
9. Dresden OCL Toolkit, http://dresden-ocl.sourceforge.net/index.html
10. SableCC, http://sablecc.org/
11. David Akehurst, Peter Linington, and Octavian Patrascoiu, OCL 2.0: Implementing the Standard, Technical report, Computer Laboratory, University of Kent, November 2003.
12. Open Source Library for OCL,http://oslo-project.berlios.de/
13. Gergely Mezei, Tihamér Levendovszky, Hassan Charaf, Implementing an OCL 2.0 Compiler for Metamodeling Environments, 4th Slovakian-Hungarian Joint Symposium on Applied Machine Intelligence
14. Thuan, T.,Hoang, L.: .NET Framework Essential”, O’Reilly,2003.
15. Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, Compilers Principles, Techniques, and Tools, Addison - Wesley, 1988