Implementing an OCL 2.0 Compiler for Metamodeling Environments

Gergely Mezei, Tihamér Levendovszky, Hassan Charaf

Budapest University of Technology and Economics
{gmezei, tihamer, hassan}@aut.bme.hu

Abstract: The Unified Modeling Language (UML) has become a standard in modeling, but it cannot express all the necessary modeling information between the model items. Object Constraint Language (OCL) is used to extend the capabilities of UML diagrams, and define constraints for the model items. The combination of UML and OCL can be used to realize vision of OMG's Model Driven Architecture (MDA). OCL is based on, but not limited to UML modeling diagrams, therefore, it can be used also in generic metamodeling environments to validate the models. This paper presents the concepts of an OCL 2.0 compliant compiler for metamodeling environments. An illustrative case study is also provided.

Keywords: OCL, constraint, metamodeling, compiler

1 Introduction

Models and modeling-based software development is one of the most focused research fields. The growing importance of modeling made the customizable, flexible modeling languages popular. Domain Specific Modeling Languages (DSMLs) represent model elements with customized attributes in a customized editing environment. Although the domains are usually well documented and use effective solutions, domain specific modeling is rarely used by smaller developer groups, because they are very expensive. Metamodeling is a proven solution for this problem. The metamodel defines the constraints for the model: it can be used as a rule for the model level objects, the attributes of the objects and the connection between them. With the customization of the metamodel, the domain specific models can be created easily.

Metamodeling is an efficient technique for defining models by their metamodels. Metamodel collects the logical rules of the actual problem domain. Metamodels specify the modeling process, what kind of objects the modeler can use, what properties they have, what connections we can create between them. In the modeling phase, the modeling environment applies rules contained by the

metamodel. 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. Beside other issues, there is a need to describe additional constraints about the objects in the model. Such constraints are often described in a natural language. Practice has shown that this always results in ambiguities. For example a company uses a customized modeling language to model the hierarchies, and relations between the employers. The company wants to define a model that is only valid if all the employers are older than eighteen years. In order to write unambiguous constraints, formal languages have been developed. The disadvantage of traditional formal languages is that they require modelers with a strong mathematical background, but difficult for the average business or system modeler to use. Object Constraint Language (OCL) [1] is a formal language that remains easy to read and write. It is a subset of the industry standard Unified Modeling Language [2] that allows software developers to write constraints and queries over object models. A constraint is a restriction on one or more values of an object-oriented model or system. OCL 2.0 is perfect to extend the metamodel definitions in order to support model validation.

Visual Modeling and Transformation System (VMTS) [3] is an n-layer metamodeling environment that grants full transparency between the layers (each layer is handled with the same methods), and offers graphical metamodel editing features. VMTS is implemented using Microsoft .NET Framework [4]. VMTS consists of several subsystems. The complete structure can be seen in Fig. 1. Attributed Graph Architecture Supporting Interface (AGSI) is responsible for the graph database actions and it offers a high-level interface for the other components. The Rule Editor and the Rewriting Engine are used for graph transformations; Traversing Processors are used for traversing the models in order to generate program code or other artifacts. VMTS Presentation Framework is the graphical environment part of VMTS used for displaying and editing the models with their proprietary presentation. The Constraint Validator Modules consists of two components: the OCL Modul, and the Multiplicity Modul. OCL Modul contains the OCL Compiler, and the related functions, Multiplicity Modul is used to check if the model contains only those types which are defined in its metamodel, and validates the connections between the model elements.

This paper discusses new results, experiences and consequences evolved from the implementation of an OCL 2.0 compiler and further conceptual development of constraint checking and handling in a metamodel-based model transformation system. This work presents the constraint validation module of a metamodeling environment supporting creation, propagation and validation of constraints in metamodels.

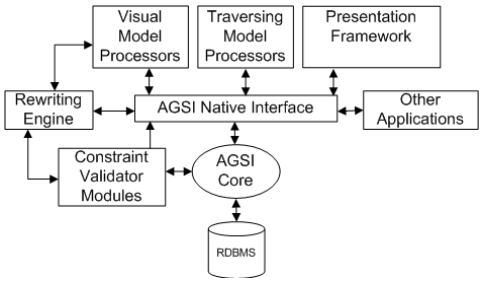

Figure 1
The structure of VMTS

2 Architecture

The OCL Module is responsible for all OCL constraint-based validation. It contains not only the OCL Compiler itself, but describes how to define and use these constraints. The OCL Module checks the metamodel containing OCL constraints for the instance models. The OCL Module is not a constraint interpreter, since VMTS generates validation code based on the metamodel containing OCL constraints (the constraints are attached to the metamodel items), then it compiles a binary from the generated code, and the instance models of the metamodel is checked by compiled binary against the constraints (Fig. 2). This method facilitates determining the complexity of the constraint validation method.

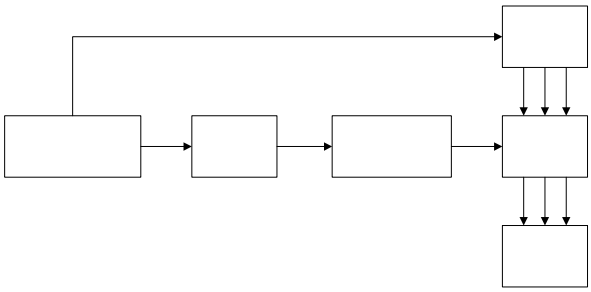

Figure 2
OCL Module

Implementing a compiler is in general a complex task consisting of several well-defined subparts. The input of a compiler is a textual file written in the source language, and the output is a textual file or a binary in the target language. The source language and the target language can be the same or different. The two main parts of the compilation are: (i) the analysis of the source language input,

and (ii) the generation (synthesis) of the target language output based on the retrieved semantic information. Accordingly, the OCL Compiler realized in VMTS consists of several parts (Fig. 3). First, the user defines the constraints in OCL, then the constraint definitions are parsed and syntactically analyzed. 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. As the next step, the constructed 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 Java). Using CodeDom, the generated source code is syntactically correct in all cases; our task is only to deal with the appropriate semantic content. Finally, the compiler transforms the CodeDom tree to C# source code, it compiles and builds it. The output of the OCL compiler is a binary executable (a .dll file) that implements the OCL constraint.

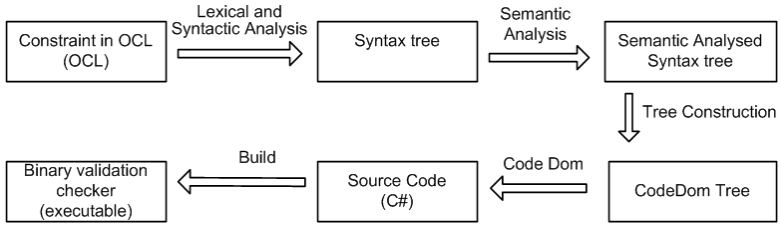

Figure 3
The structure of the OCL Compiler

3 The Compiler

3.1 Lexical Analysis

Lexical and syntactic analysis are realized using the tools Flex [5] and Bison [6]. These tools are optimized and well-tested solutions to syntax tree generation, both of them were used in thousands of software projects. Unfortunately Flex and Bison tools use ANSI C code, but they were relative easy to integrate in VMTS.

The first step of the lexical analysis is the tokenization, which parses the source code and reads lexical patterns (tokens) form it (e.g. identifiers (name) and the keywords of the language). Tokenization eliminates the whitespaces as well as

comments. Tokenization is achieved by a table, which contains the keywords. The result of this process is a sequence of tokens, which contains the meaning of the source program. Tokenization is made by Flex.

3.2 Syntactic Analysis

The next step is to create the syntax tree using the tokens. The OCL specification is used to construct the grammar rules for this step. The task of the syntactic analysis is to find the deduction which generates the source code of the program, starting from the sentence symbol (S). The analyzer reads the sequence of the tokens, and, using the production rules, it generates an Abstract Syntax Tree (AST), which is a model of the program that we want to compile. The AST is a direct association between the rules in the grammar and the nodes in the tree, and it is purely an abstract representation of the syntax, modeled as a tree [7] [8]. The inner nodes of the AST contain no terminal symbols, while the leaves contain the tokens. The generation of the AST is possible if and only if the program is syntactically correct [9].

The OCL specification [1] uses EBNF notation for the grammar specification. This EBNF description had to be modified, and simplified in certain places to be able to process it with Bison. The simplification does not reduce the expression power or the language capabilities, because the information avoided by the simplification is reconstructed in the semantic analysis step.

The first problem was with the EBNF specification that it uses the `?` (optional element), the `*` (`0..*` multiplicity) and the `+` (`1..*` multiplicity) notations. These symbols had to be modified in order to be able to process to Bison. Table 1. contains the original and the modified rules.

Original rules	Reworked rules
`A -> b c? d`	`A -> b d  |  b c d`

`A -> b c* d`	`A -> b optionalC d`
	`optionalC -> /* empty */`
	`|  optionalC c`

`A -> b c+ d`	`A -> b optionalC d`
	`optionalC -> c`
	`|  optionalC c`

Table 1
Reworked EBNF rules for Bison

The second problem is that several rules in the OCL specification cause shift/reduce or reduce/reduce conflicts. To solve this problem, the rules had to be transformed. There are rules that can be transformed, divided, or unified to support the specification, but there are rules that needed to process additional

(semantic) information. For example, both *VariableExpressions* and *AttributeCalls* can be defined by a single name (the name of the variable or the attribute). This two case can not be distinguished at the level of the tokens, because the compiler should check whether there is a variable defined with the name (and the current expression is in the scope of the variable), or the model items has an attribute with the name. There are five simplification groups that cannot be solved without semantic information:

- AttributeCall can conflict with VariableExpression
- NavigationCall can conflict with AttributeCall
- VariableExpressions can conflict with AttributeCalls
- EnumerationLiterals can conflict with AttributeCalls
- Iterators with an overridden operation can conflict with Attribute/NavigationCall

The complete OCL specification can be found on the VMTS homepage [3].

3.3 Semantic Analysis

The text of the constraint is tokenized, and the syntax tree is constructed. The next step is to analyze the syntax tree semantically, and add all information that is required by the further steps. In VMTS this step begins with a substep of interoperability communication between the unmanaged code generated by Bison, and the managed code of VMTS. Fortunately this substep is relative easy: the unmanaged data structure can be fit the managed structure using the .NET recommendations.

With the syntax tree constructed by Bison there are three problems. (1) OCL allows leaving out the *self* identifier if it does not cause ambiguity. These identifiers are required to generate code from the constraints. (2) The code generation and, generally, the type checking require the type information included in the nodes. (3) During the syntax tree construction there was a few simplifications that should be corrected.

The missing *self* identifier is the easiest to handle. If the currently processed node is a *NavigationCall*, *AttributeCall* or *OperationCall* (i.e. the node uses some kind of model information) then it is recursively examined. The *self* identifier should be inserted only if the innermost expression is not *self*. This algorithm ensures that every model-based operation use *self* as the start point. Model-independent operations (like definition of temporary variables) do not need *self* (neither in the code generation, nor according to the OCL specification).

In the OCL Compiler, we cannot use the traditional symbol table, because the symbols are not in the code to be compiled, but it must be obtained from another

place, namely, from the VMTS model database. Therefore instead of the static symbol table, we use a dynamic solution. The nodes in the syntax tree are decorated with the appropriate type information (the type information is added to the nodes in the tree) and a dynamic symbol table is also created (for faster access of the type information). The decoration of the nodes is combined with type checking. The analysis starts at the root of the tree. If the type checking fails then the compiler throws an error message, if it succeeds then the child nodes are processed recursively.

Type checking can be handled in many ways, but if the decorated type information not only defines the base type of the node (e.g. Integer), but also describes the available operations with the type of the result, then the type checking is simple. To support this, the compiler defines base classes for the available types, and all type information contains an instance of the appropriate type. The bases classes are classes supporting OCL base types (e.g. String, Set, Bag) and the model types (ModelItem, ModelAttribute). These classes contain a common function

GetTypeInfo (string methodName, object[] additionalParam)

This function returns with a type of the operation identified by *methodName* (or null if the operation is not defined for the current type). In general this information could be gained also by using .NET Reflection, but there are certain methods (e.g. *intersection* in *Bag*), where the type of the result depends on the parameters (here it is solved using the *additionalParam* parameter). In summary type information can fully describe the available operations and the result types.

The last problem is the reconstruction of the simplifications made in the syntactical analysis. Each of the five simplification groups should be handled separately, but the algorithm behind is the same for each group: (i) The node is processed as long as the required semantic is not known (this means typically the processing the first child node). (ii) We check the model information (whether the model has an attribute with the name) or the variable-scope information (whether there is a variable defined with that name in the scope) and check whether the type of the node should be changed. (iii) If the type of the node has changed, the compiler generates or modifies the new node structure, swaps the old and the new node in the tree and analyzes the new node (iv) If the type of the node was correct, then processing the node continues. Fortunately, the changes made in the type of the node do not affect the ancestors of the node.

3.4 Code Generation

The syntax tree was semantically analyzed; therefore it contains all the necessary information required by the code generation. Code generation is realized using the System.CodeDom namespace of .NET Framework [10]. This means that the code generation is a syntax tree composition, from which the framework generates the

source code. CodeDom has many advantages, for example, it supports the code generation for many languages, and syntax checking in the generated source code can be avoided, since CodeDom ensures that it is be syntactically correct in all cases.

OCL constraints can be of five base types: Initial Values, Derivation Rules, Invariants, Virtual Attribute or Operation Definitions and OperationConstraints. The current implementation of the OCL compiler handles only the invariant types in the code generation, because only this type of constraints can be used to validate a static model (a model that can not change its state). The non-invariant types are semantically analyzed, only the code generation steps does not compile them into source code.

The generated source code should support the operation of the OCL-based types defined in the OCL specification. This means that either the generated source code should contain the operation logic, or a class library is required with the base classes, and the source code instantiate the classes and use them. The first solution is neither elegant, nor efficient, thus, VMTS uses the second solution. The compiler already has classes for the base types (recall the type information checking part of the semantic step). These classes are extended with the necessary operations and they are used in the implementation. Using theses classes the operations contained by the constraints can easily be expressed in the C# language. While the version of C# used in development does not support class templates, the implementation of the collection types is more complex, than it would be with generics.

The transformation of the syntax tree to the CodeDom tree is easy, because each node type has an appropriate code sequence i.e. CodeDom tree branch. Packages are transformed to namespaces, contexts are realized with classes, and constraint expression as public methods. Contexts have a *self* attribute that is used the same way as in the OCL expressions; this attribute is initialized in the constructor of the class. To handle the invariants simply, the context class has always a method (*checkInvariants*) that calls the methods of the invariants one after the other.

There is only one decision left that deserves word in the Syntax tree to CodeDom transformation. The *OclExpression* can be transformed either to classes, methods, or simple code expressions. The simpler the generated CodeDom tree is, the faster the binary executer will be. If there is a separated class for every OclExpression, then the source code is well separated, easy-to-read, but it contains a lot of redundancy. If the *OclExpression*s are handled as simple code expressions, then source code is very small, but neither user-friendly, nor easy to debug. Therefore in VMTS the *OCLExpression*s are translated as methods. This decision facilitate the code generation for complex *OclExpressions* like iterators.

The source code generation based on a CodeDom tree is a simple function call, and the .NET technology offers also functions to compile and build the constructed source code.

4 Case Study

Using a case study, we introduce how VMTS generates source code from a simple OCL constraint. The case study is about a computer manufacturer company that makes CDs, flash memories, and pen drives. The models are used for statistical calculations. Although both the constraint example and the model itself are very simple, they are useful to see how the compiler works.

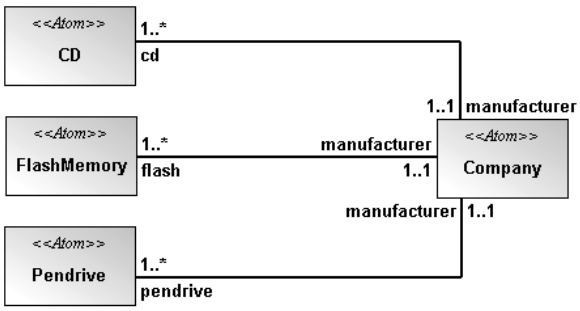

Figure 4
Metamodel of the Company

The metamodel of the case study is shown in Fig. 4. The attributes are not visible in the picture, *Company* has three attributes: Name (*string*), Location (*string*), Income (annual income in Euros, *integer*). All products have two attributes: *Capacity* (in Mbs, *integer*), and *serialNumber* (*string*).

Without constraints there are no validation facilities in the model, a *CD* with 1Mb capacity, or a *Company* with fictive location can be created without any problem. The metamodel definition should be extended.

The example constraint (Fig. 5) is used to validate the capacity of the CDs. The constraint supposes the capacity of the CDs is always 650Mb (to make the constraint simpler).

```
package CaseStudyPackage
context CD

inv capacityChecker:
      Capacity = 650

endpackage
```

Figure 5
Capacity constraint

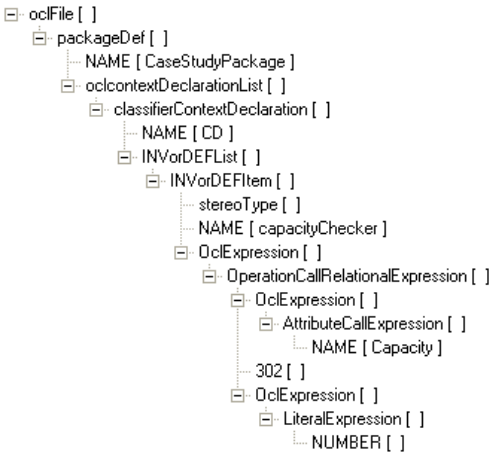

Figure 6

Capacity constraint – syntax tree

First, the OCL compiler constructs the syntax tree (Fig. 6) using the grammar rules. The root is the *OCLFile* node that is not the par of the OCL standard, but allows defining multiple packages in one validation source. The syntax tree shows the structure of the OCL source file in an expressive way. This expression contains an only constraint, namely, the *capacityChecker* invariant constraint that contains a simple *OclExpression* to compare the value of an attribute (*Capacity*) to a literal (650). Second, the semantic analysis processes the syntax tree. The nodes are decorated with type information, and the missing *self* reference is inserted (Fig. 7). This time each node in the syntax tree has its correct type, hence, the third step of semantic analysis can be skipped. At the end of the syntax tree analysis the nodes has correct type information (e.g. the type of the outermost *OclExpression* in Fig. 7 is *Integer*)

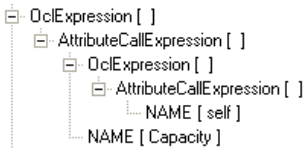

Figure 7

Compiler inserted *self* node

The last step is the code generation from the syntax tree. The source code (Fig. 8) uses the concepts introduced earlier: a package is transformed to a namespace, a

context to a class, an invariant to a method. The *self* value is provided to access the model attributes, and the *checkInvariants* method to validate the model. The two parts of the relation (*Capacity* = 650) are realized by two methods according to the tree representation.

```
namespace OclRuntime.Generated.CaseStudyPackage
{
 public class CD
 {
  ModelItem self;

  public CD()
  {self=new ModelItem(2282);}

  public OCLBoolean checkInvariants()
  {return Constraint_34_capacityChecker();}

  private OCLBoolean Constraint_34_capacityChecker()
  {return (AttributeCall_39() == Literal_45());}

  private OCLInteger AttributeCall_39()
  {return self.Capacity;}

  private OCLInteger Literal_45()
  {return 650; }
 }
}
```

Figure 8

The generated source code

Conclusions

This paper has presented the main concepts of an OCL Compiler in an n-layer metamodeling system. The paper has discussed the steps in depth from the lexical and syntactic and semantic analysis to the code generation. The OCL and metamodeling-based problems and its solutions were focused. Although the OCL Compiler was realized mainly using the .NET technology, the solutions and concepts presented in this work can be useful to another constraint compiler processes. The solutions were presented using an illustrative case study that has shown the underlying mechanisms in operation.

The presented OCL Compiler is useful to validate the models, but neither the generated code, nor the compiler contains optimization algorithms. Therefore the presented method can validate the models, but it can use more resources then necessary. For example the generated code above has a separated code for *Literal_45* in turn the expression "*650*" could be inserted directly into the code (because it is a primitive literal type).

Another weakness is that the OCL specification does not have a formalism to define custom error messages. If an invariant fails, the user does not receive any additional information about the reason of the failure. Since the OCL standard does not handle it but the users may need this feature, the compiler and the framework should offer a solution.

Future work focuses primary on these fields, and extends the OCL Compiler to support the aforementioned functions missing.

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] Thuan Thai and Hoang Lam, *.NET Framework Essentials* (O'Reilly, 2003)

[5] Flex, Official Homepage, http://www.gnu.org/software/flex/

[6] Bison, Official Homepage, http://www.gnu.org/software/bison/bison.html

[7] David Akehurst, Octavian Patrascoiu: OCL 2.0 - Implementing the Standard for Multiple Metamodels, Workshop Proceedings, 6th International Conference on the UML and its Applications,<<UML>> 2003, ENTCS, Oct. 2003

[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] Sten Loecher, Stefan Ocke: A Metamodel-Based OCL-Compiler for UML and MOF. In OCL 2.0 - Industry standard or scientific playground, Workshop Proceedings, 6th International Conference on the UML and its Applications,<<UML>> 2003, ENTCS, Oct. 2003

[10] Microsoft .NET Framework http://msdn.microsoft.com/netframework/