Imperative OCL Compiler Support for Model Transformations

Tamás Vajk, Tihamér Levendovszky

Department of Automation and Applied Informatics, Budapest University of Technology and Economics
{tamas.vajk, tihamer}@aut.bme.hu

Abstract: Model-Driven Architecture (MDA) is a widely known software design approach, which is intended to support model-driven engineering of software systems with specifications expressed as models. Using the MDA methodology, system functionality may first be defined as a platform-independent model (PIM) through an appropriate modeling language. Then the PIM may be translated into one or more platform-specific models (PSMs) for the actual implementation. These translations between the PIM and PSMs are normally performed using automated tools, such as model transformation systems, for example, tools compliant to the new OMG standard, named QVT (Queries/Views/Transformations). To facilitate flexible and powerful model transformations, OMG has specified the Imperative OCL language in QVT; this language allows using the most common programming constructs. This paper descibes the necessary steps of developing a compiler from lexical analysis to code generation through syntactic and semantic analysis. Our implementation of the Imperative OCL compiler is attached to the Visual Modeling and Transformation System (VMTS), which is an n-layered modeling environment.

Keywords: MDA, QVT, Imperative OCL, compiler

1 Introduction

One of the most focused fields of software engineering is modeling and a flexible tool support for improving the quality of the development process. Model-Driven Architecture (MDA) [1] is a software design approach that supports model-driven engineering of software systems with specifications expressed as models. Several UML-based modeling tools exist; the problem with these applications is that they do not allow the modification of their model definition. Metamodeling tools give the ability to edit a metamodel, which defines the rules of a model. The metamodel determines which types of objects are allowed during the modeling process, what kind of attributes or relations they can have.

After having customizable models based on metamodels, the need to transform a model into another type appeared. Generally, model transformation is the process of converting a model conforming to a metamodel to another model, which conforms to another metamodel. The first metamodel is called source metamodel, the second is referred to as target metamodel. The way of translating a model into another is not trivial, a standard named QVT [2] has been proposed by the Object Management Group (OMG) to handle this task.

This paper only deals with a small part of model transformation; input model matching and validation of transformations are not discussed. During a transformation, new model elements should be created, and their attributes as well as their relations to each other must be set. In QVT, these tasks are handled by the Imperative OCL language. This paper presents a way of creating a compiler that transforms Imperative OCL sources into C# codes.

The implemented compiler has been installed into the Visual Modeling and Transformation System (VMTS) [3], which is an n-layer metamodeling environment that supports editing models according to their metamodels. Moreover, VMTS is a UML-based model transformation system, which transforms models using graph-rewriting techniques.

2 Model Transformation

The aim of Model-Driven Architecture is to support the model based software design by the use of models on every stage of the system development. MDA offers a framework to separate the platform-independent and the platform-specific information used in models. The general modeling information is collected into a platform-independent model (PIM), which is usually expressed in UML. The actual implementation of an application is defined in a platform-specific model (PSM). A complete MDA application may contain several PSMs, according to the number of supported platforms. The idea of generating PSMs from PIM is obvious, because the platform specific information is well-known, and can be added to the base model automatically. This generation process is realized by model transformation. More generally, converting a model to another is the model transformation process. If the metamodels of the models equal, then the transformation is endogenous, otherwise it is exogenous [4].

2.1 QVT

Model transformation is a critical component of MDA. Therefore, a request for proposal has been issued by the OMG on MOF Query/Views/Transformations [2] to seek a standard that is compatible with the MDA recommendation suite (UML,

MOF, OCL, etc.). QVT defines a standard way of transformations between models. The QVT specification has a hybrid declarative/imperative nature, with the declarative part being split into a two-level architecture.

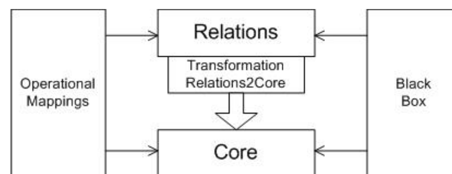

Figure 1
Structure of QVT

The two layers of the declarative part are the Relations and the Core metamodels and languages. The user-friendly Relations metamodel and language supports complex object pattern matching and object template creation, and implicitly creates trace classes and their instances to record what occurred during the execution of the transformation. The Core metamodel and language is defined using minimal extensions to the EMOF and OCL. The Core language supports pattern matching over a flat set of variables by checking conditions over those variables against a set of models. It is equally powerful to the Relations language, and because of its relative simplicity, its semantics can be defined more simply, although transformation descriptions described using the Core are therefore more verbose [2, 5].

In addition to the declarative Relations and Core languages that embody the same semantics at two different levels of abstraction, there are two mechanisms for invoking imperative implementations of transformations from Relations or Core: one standard language, Operational Mappings, as well as non-standard Black-box MOF Operation implementations. The Operational Mappings language provides OCL extensions with side-effects that allow a more procedural style of programming. The Black Box implementation for invoking transformation facilities expressed in other languages (XSLT, XQuery) is also an important part of the specification. It is especially useful for integrating existing non-QVT libraries.

2.2 Imperative OCL

The QVT Operational Mappings is an imperative language that supports the creation of powerful model transformations. It extends the Object Constraint Language (OCL) [6] with all the necessary programming constructs that are needed to write complex transformations in a comfortable way. It also extends the type hierarchy of OCL, for instance with dictionaries (hashtables).

OCL is a declarative language for defining rules that have to apply to UML models. With the use of OCL, UML has been extended, because it allows the creation of rules that cannot be expressed by UML structures. OCL is textual language that provides constraint and object query expressions. As OCL is a query language, it cannot modify the models, and therefore OCL is a purely side-effect-free language.

An assignment expression represents a value assignment to a variable or to a property of a model element. In case of multiple-value variables (i.e. set, ordered set), there are two types of assignments. The first type of assignment resets the value of the variable, while the second one adds the new values to the collection. Only the assignment signs differentiate these cases (:=, +=). An instantiation expression creates an instance of a class.

The importance of these expressions is arisen by the fact that these are the constructs that can modify the models; thus, these expressions outrage the side-effect-freeness of the OCL and these are the most powerful innovations in the Imperative OCL.

3 Compiler Theory

A compiler is a program that translates a program code written in the source language into another equivalent program code written in the target or object language. Typically, the source language is a programming language, such as C++, and the target language is the machine code for the computer being used.

3.1 Structure of a Compiler

The compilation process can be divided into a number of logical phases. Some of these phases can run simultaneously, but generally, these are executed consecutively. The translation of a programming language is divided into two main blocks: the front end and the back end. The front end analyses the source code to build an internal representation of the program, called the intermediate representation (IR). It also manages the symbol table, a data structure mapping each symbol in the source code to associated information such as location, type and scope. The back end is responsible for generating the target language code. These parts may be further subdivided into logical blocks. The main phases of the front end are lexical, syntax and semantic analyser. The back end breaks down into a code optimizer and a code generator [7].

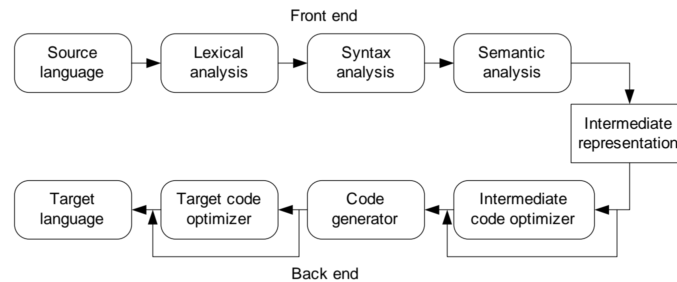

Figure 2
Logical structure of a compiler

3.2 Analysis

3.2.1 Lexical Analysis

Lexical analysis is the first of two stages in the analysis of the structure of a source code. The smallest fragment of a source code is a character, but this symbol is too small to bear with enough information. The smallest logical entities in programming codes are words; more precisely these can be keywords (*while*, *for*), literals (*'example string'*, *1000*), operators (==, +=) and identifiers (variable names). These strings of characters are named tokens. The grammar of a programming language can be divided into two parts: the first one can express the structure of the tokens and the second part can define the main logic of the grammar, in which we use tokens. The prime reasons of separating the structural analysis into two stages are efficiency and clarity [7].

The pattern describing each token can always be expressed as a conventional regular grammar [8, 9]. Therefore, this part of the analysis can be made by finite state machines, which are much simpler and faster than those parsers that have to be used for the complete syntactical analysis. Lexical analysis can be the most time-consuming part of a compilation process [7]. This is primarily caused by the fact that it handles the whole input, therefore, the previously mentioned separation is essential.

3.2.2 Syntactical Analysis

The task of the syntactical analysis is to find a derivation for the input sentence from the sentence symbol. A parser uses the input stream or code and the production rules to creates a syntax tree, which shows the applying order of the production rules to generate the given source code. This parse tree is an ordered, directed tree. Ordered, because the sequence of the outgoing edges in a vertex is determined. Each inner vertex of this tree contains a non-terminal symbol (the root

vertex is the sentence symbol), while each leaf holds a terminal character (more accurately a token, because the lexical and syntax analysis is separated). If there is no syntax tree for the given input, then the given input was not a syntactically correct program, as the parse tree defines a concrete derivation for the input.

There are two obvious ways of building up a parse tree. The first is to start with the sentence symbol and build down towards the terminals; the second is to start from the terminals and build up towards the start symbol [7, 8]. These are known as top-down and bottom-up parsing methods. Algorithms exist to parse any Type 2 languages, but only a subset of grammars can be parsed efficiently. Fortunately, most programming languages can be analysed simply.

A well-known analyser is the LL(k) parser. It parses the input from **L**eft to right, constructs the **L**eftmost derivation of the sentence and looks ahead k tokens. Another parser is the LR(k) parser, which also analyses the input from **L**eft to right, but constructs the **R**ightmost derivation of the sentence. LL parsers always start with the sentence symbol, therefore these are top-down parsers, on the other hand, LR parsers are bottom-up analysers, because these start from the terminals and go upward in the tree. It has been proved that every LL(k) language can be parsed with LR(k) parsers, but there exist languages which cannot be analysed by LL(k) parsers. This means that LR parsing method is more powerful than the LL. The complexity of a parser grows with the look-ahead number, thus k should be minimized. Most programming languages can be parsed with only one token look-ahead, therefore LR(1) parsers are used in practice [4].

3.2.3 Semantic Analysis

The semantic analysis is the phase of the compilation process in which semantic information is added to the parse tree and certain checks based on this information is performed. Typical examples of semantic information that should be added and checked is type information (type checking) and the binding of variables and function names to their definitions (object binding).

Symbol Tables

A symbol table is a mechanism that associates values, or attributes with the names. A symbol table is a necessary component of a compiler because the definition of a name appears only one place in a program, while the name may be used in any number of places within the program code [7, 9, 10]. Each time a name is used, the symbol table provides access to the information collected about the name when its declaration was processed. There are programming languages in which the declaration of a variable may not precede the use of the variable, in these cases the creation of the symbol table cannot easily be created in parallel with the semantic processing. Fortunately, in most languages the declaration comes first, this means that the symbol table can be filled with information during the semantic analysis, therefore the parse tree does not have to be processed several times.

Most programming languages allow name scopes to be nested; a name scope is usually defined by program units such as a package or a block. Name scopes can be current (actual, innermost), open and closed. Obviously, these are not fixed attributes of scopes; they are defined relative to a particular point in the program. Two well-known solutions exist to solve the problem of symbol tables in block-oriented programming languages: an individual table for each scope or a single symbol table. If an individual symbol table is created for each scope, some mechanism must be used to ensure that a search produces the name defined by the visibility rules. As the name scopes are opened and closed in a LIFO manner, a stack is appropriate for this organization. Using a single table for all symbols, the scopes have to be differentiated. Each name scope should be given a unique number. In this case, a name can appear in the table several times if the scope number is different [9].

Attribute Grammars

Parse trees are used to drive the semantic analysis of the source code in a compiler. A semantic analysis approach is to augment our conventional production rules with information to control the analysis. Such grammars are called attribute grammars.

We augment our grammar by associating attributes with each grammar symbol to describe its properties; such an attribute can be the type of a variable or the integer value of an integer node. After defining the attributes, the production rules have to be extended with semantic actions, which describe how to compute the associated attribute value [7].

Two types of attributes exist: synthesized attribute and inherited attributes. Synthesized attribute means that the attributes of the symbols on the left-hand side of the production rules have been created from those at the right-hand side. This can be imagined as the attribute values being passed up in the parse tree. It is also useful to be able to pass semantic information down in the parse tree. In this case, the right-hand side attribute values are generated from the left-hand side ones and it is also possible to use other right-hand side attribute values. These attributes are called inherited attributes.

3.3 Code Generation

Having completed syntax and semantic analysis of the source program, all the necessary information is available to generate the target language code from the parse tree. Typical ways of creating the target code is traversing the parse tree and processing every node in it. In this case processing means that the target code for the actual node should be created. Generally, in the code generation process, the previously created parse tree is not modified just read.

4 Implemented Compiler

This compiler is not a general-purpose Imperative OCL translator application; it is specifically created for the use in the Visual Modeling and Transformation System (VMTS) [3]. The semantic analysis uses a VMTS interface to check the necessary restrictions; also the generated C# code is specially created for the VMTS and it utilize huge amount of VMTS built-in functions.

4.1 Architecture

Figure 3 depicts the simplified architecture of the created compiler. The class diagram only displays the classes that are connected with the compiler, thus the Adaptive Modeler is not shown, however, without that, the creation of models and transformations are difficult tasks as that is a widely configurable user interface for the VMTS.

AGSICommon

AGSICommon (Attributed Graph Architecture Supporting Inheritance) [3] is the namespace of the most frequently used functions in VMTS. Every model information can be queried through this namespace; as the models are stored in a relational database, the model queries require database connection.

AGSIImperativeOCLInterface

AGSIImperativeOCLInterface is the main interface for checking model information. The main aim of this interface is to hide the complex structures used in the *AGSICommon* namespace and provide a simpler connection to the modeling elements.

ImperativeOCLInterface

ImperativeOCLInterface is a simple interface to reach the Imperative OCL compiler. It has only one important public method, the *GetCSharpCode()*, which gets an array of unique identifiers of models, and returns the C# code of a compilable class that implements the functions of the input source codes. This function calls the *Compile()* method of the *ImperativeOCLCompiler* class for the number of the length of the input array, and the returned *CompileJob* objects that contains the *CodeDom* trees are then compiled together to make up a single class at the end of the process.

ImperativeOCLCompiler

ImperativeOCLCompiler is the main class of the implemented compiler. This class processes the input source language code with passing it to the *ParseWrapper* (the syntax analyser), then to the *SemanticAnalyser* and at the end to the *DBCompiler* (the code generator). The passage between the managed C#

code and the unmanaged C code generated by Bison is handled in the *ParseWrapper* with a custom marshalling method. The semantic analyser uses the *ImpOCLFunctionHelper* class during the pre-processing phase to create the symbol table of the defined functions. The type checking methods are implemented in the *AGSIImperativeOCLInterface*, which is reached through the *TypeHandler* class to hide the differences between model elements.

ImperativeOCLRuntime

A C# class had been implemented for each defined OCL type, these classes are placed into the *ImperativeOCLRuntime* namespace. The implemented hierarchy follows the type hierarchy specified in the OCL specification [6]. OCL does not specify model type classes, but the simple code generation requires it; this class is implemented separately from the base types.

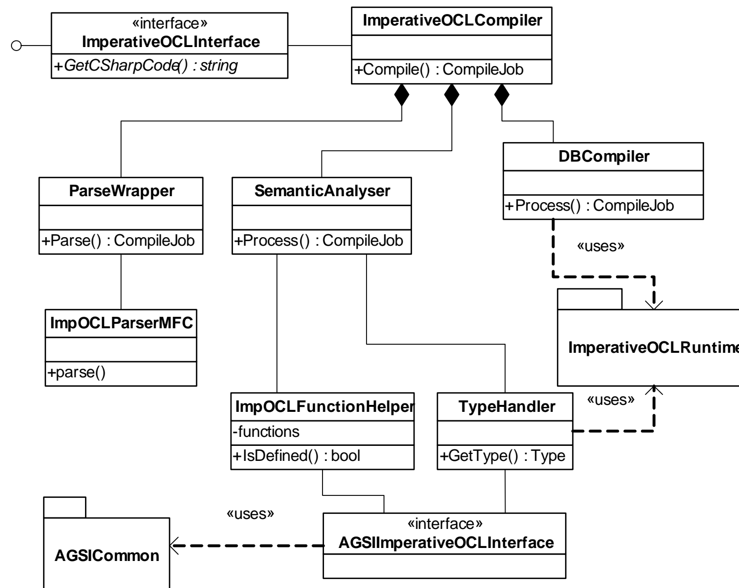

Figure 3
Architecture of the implemented compiler

4.2 Analysis

Context-free grammars are powerful enough to describe the syntax of most programming languages. Fortunately, the parsers can be generated automatically from the grammar and action rules. These actions can be target language code (but more generally the action rules creates the intermediate representation of the source code), which is executed when a reduce action is needed in the parsing

process. The automatic generation is a flexible way of creating a parser, because the modifications in the grammar can be handled easily. Bison [11] and Flex [11] (for the lexical analysis) are the most traditional tools, Bison is a general-purpose parser generator that converts a context-free grammar into an LALR(1) (Look Ahead LR) parser. Bison generates C language target code, which can easily b used in C#, thus we chose this compiler-generator for our implementation.

The architecture of the implemented compiler allows the total separation of the syntax and semantic analysis. During the syntax analysis a tree is created, which is then passed to the semantic analyser. The semantic analyser is the last part of the compiler that can modify the parse tree, because no optimization process had been implemented yet. A simple modification is the addition of default values to simply type variables that have not been initialized. In semantic analysis symbol tables have been implemented for function and (global) variable handling. The type checking is based on attribute grammars, however no attributes are written in the production rules. The attributes are added to the parse tree in our solution.

4.3 Code Generation

The code generation is based on the *CodeDom* [12] namespace of the .NET framework. This means that the parse tree is converted into a CodeDom tree, and later from this tree, the framework can generate the C# language code. This target language code is then compiled into a dynamic-link library (DLL) for execution. The code generation leans on the *ImperativeOCLRuntime*, which collects together C# classes based on OCL types. Between simple types and their OCL equivalents implicit type conversion is implemented to make the code generation easier and simpler.

4.3.1 Variables

As previously mentioned, OCL is a side-effect free programming language, which has no real variables in it. The previously implemented OCL compiler created a C# function for each expression. This is a reasonable solution there, but in case of Imperative OCL, it causes a lot of problem, because the defined variables can not be accessed from a function as the visibility and the scope is different.

As Imperative OCL and normal OCL, expressions can be mixed in the code a general solution is needed for this problem. The implemented solution is the following: a unique name is assigned to each variable as an attribute when it is created. In the source code, this unique name will be the real name of the variable. The target language variables are placed into a separate class and are defined as *static* variables. Therefore, they can be accessed from any point in the target language code. Semantic analysis previously checked whether the program is semantically correct or not, thus a variable reference can only be placed where it makes sense.

4.3.2 Break and Continue

Almost the same problem appears with the use of *break* and *continue* expressions. The problem is that if these expressions are in a generated function, the loop may be elsewhere, thus the generated *break* and *continue* C# expressions are semantically incorrect in the created code. Unfortunately, this cannot be solved with the previously applied method. Furthermore, these expressions should be in a loop expression and they should only affect the innermost loop expression if there are nested ones.

The implemented solution uses C# exceptions to deal with the complication. For each *break* and *continue* Imperative OCL expression an *exception* throwing statement is created in C#. In addition, every loop expression has a *try-catch* block in it, which catches the *break* and *continue* exceptions, and in the *catch* part a *break* or *continue* C# expression is generated. This solution satisfies the needed restrictions: semantically correct (the new *break* or *continue* expression is placed inside a C# loop), works between functions and only the innermost loop is affected, because that catches the exception and does not throw a new one.

Conclusion and Further Work

This paper has shown a viable way of developing a compiler based on a high-level programming framework. After understanding the basics of compiler theory one can create powerful translator applications, which can be used for several different purposes. This paper has illustrated how a compiler can be used in a modelling and transformation system. With this Imperative OCL compiler, the development time of a transformation has been decreased into a fraction of the previously needed time. Our former solution based on XSL transformations did not allow us to easily modify the transformations, as those were quite long and complex. With Imperative OCL, the causalities can be expressed in a compact and legible format.

Future work includes several directions. Finishing the whole specification is not an optional future work, but a must. The structure of the compiler can be easily extended with an optimization block. Simple optimization could vary from the use of cache tables to unreachable code elimination, but other platform-specific optimization could also be applied as the base of VMTS will probably not change. Another direction of future work is based on the power of the Imperative OCL language. OCL supports object-oriented application development, and with imperative extension, it is capable of expressing common event handling functions. In VMTS, there is a mobile user interface designer plug-in, which supports different mobile phone platforms, such as Symbian, J2ME or .NET Compact Framework. With this plug-in, the different user interfaces can be managed, but the event handling functions have to be implemented in different languages (C++, Java, C#). The idea of using a common language to express the functions and then translate it automatically to each target language is simple and could be solved with the compiler. The modular structure of the compiler only

necessitates the modification of the code generation phase of the compilation process.

References

[1] Object Management Group: Model-Driven Architecture Specification http://www.omg.org/docs/omg/03-06-01.pdf

[2] Object Management Group: MOF QVT Specification http://www.omg.org/docs/ptc/05-11-01.pdf

[3] VMTS Homepage, http://www.vmts.aut.bme.hu

[4] Wikipedia The Free Encyclopaedia, http://www.wikipedia.org/

[5] L. Lengyel, T. Levendovszky, H. Charaf: Realizing QVT with Graph Rewriting-Based Model Transformation, Electronic Communications of the EASST, 2006

[6] Object Management Group: Object Constraint Language Specification http://www.omg.org/docs/ptc/03-10-14.pdf

[7] J. P. Bennett: Introduction to Compiling Techniques, McGraw Hill Publishing Company, 1996

[8] Bach Iván: Formális nyelvek, Typotex Kiadó, 2002

[9] Charles N. Fisher, Richard J. LeBlanc, Jr.: Crafting a Compiler, The Benjamin/Cummings Publishing Company, 1988

[10] Alfred V. Aho, Revi Sethi, Jeffry D. Ullman: Compilers, Addison-Wesley Publishing Company, 1986

[11] Bison and Flex, http://www.gnu.org/software/bison/manual/ http://www.gnu.org/software/flex/manual/

[12] Microsoft Developer Network, http://msdn2.microsoft.com