An Incremental OCL Compiler for Modeling Environments

Tamás Vajk, Gergely Mezei, and Tihamér Levendovszky

Budapest University of Technology and Economics,
Department of Automation and Applied Informatics
WWW home page: http://www.aut.bme.hu
{tamas.vajk|gmezei|tihamer}@aut.bme.hu

Abstract. In software engineering, reliability and development time are two of the most important aspects, therefore, modeling environments, which aide both, are widely used during software development. UML and OCL became industry standards, and are supported by many CASE tools. OCL code checking, which has to be performed by these tools, has a specialty, as not all of the information necessary for compilation is available from the code, the related model contains the types, navigations and attributes. The build time of OCL code fragments is increased if the development tool supports distributed modeling, because in this case, model element checking has to be performed in a model repository that cannot be held in memory. In this paper, we introduce a method that enables incremental OCL code building and therefore reduces the development time. Incremental builds require higher complexity than simple builds, thus balancing between the two methods is also considered.

1 Introduction

In the last decade, software design significantly improved with the use of modeling languages, such as the Unified Modeling Language (UML) [1]. Practice has shown that extending visual modeling languages with textual constraints is more efficient and precise than a pure graphical modeling language. UML defines the Object Constraint Language (OCL) [2] to facilitate expressing constraints and object queries on models that cannot otherwise be defined by diagrammatic notations.

Using the Model-Driven Architecture (MDA) [3] methodology, system functionality may first be defined as a platform-independent model (PIM) through an appropriate modeling language, such as UML. Then the PIM may be translated into one or more platform-specific models (PSMs) for the actual implementation. These PSMs are usually domain specific models (DSMs). The translations between the PIM and PSMs are normally performed using automated tools such as model transformation systems. The way of translating a model into another is not trivial. A standard named QVT [4] was proposed by the Object Management Group (OMG) to handle this task. During a transformation, new model elements may be created, and their relations to each other must be set. In QVT, these tasks are handled by the Imperative OCL language, which extends OCL with program control statements and imperative expressions.

In this paper an incremental OCL compiler is presented which reduces the development time of large systems using OCL constraints and increases the speed of model transformation systems. This is accomplished by not compiling every code fragment each time, but only the modified ones. The background and implementation framework for the discussed compiler is the Visual Modeling and Transformation System (VMTS) [5] developed by our research group.

The main motivation of our work was to enable faster model development in CASE tools, especially in VMTS. Most of the Integrated Development Environments (IDEs), such as Microsoft Visual Studio or Eclipse, support partial rebuilds of solutions or packages, however, in model-based development environments it is rarely available. The processing time of OCL compilers can be greatly reduced by enabling incremental builds, mainly because these compilers not only have to check the code, but also the corresponding models. If a modeling tool utilizes a shared model repository, such as a database to enable distributed model development, the speed of model validation plays a key role in the overall processing time [6]. Also, typical OCL code fragments check the attribute values of some model elements. Therefore, if one changes only a constant in the code, the modeling environment should not rebuild the whole code, only the modified parts, which in this case will not include queries to the model repository. Compared to other compilers, this dual input of the OCL translators makes a significant difference, as the compiler needs to acquire information from two sources. Thus, more attention should be paid to incremental compilation.

As OCL serves as the basis of Imperative OCL, in the future we aim to supporting incremental builds in transformations as well. In VMTS, model transformations rules are connected together into a control flow diagram [7] that can express complex transformation chains. Transformation development tends to be an incremental process; after having the skeleton of the entire process, developers fine tune the distinct transformation rules. In VMTS, these rules have no interdependencies with each other; therefore, allowing them to be separately built seems to be a logical requirement.

In our work, we attempt to find a universal method that enables incremental code building on different abstraction levels and thus, it decreases the development time of constraints and model transformations.

2 Background and Related Work

2.1 OCL and Imperative OCL

The Object Constraint Language [2] is a declarative language for describing rules that apply to UML models. OCL was intended to be a formal specification language, extending UML, but now it can be used with any kind of model type. The Object Constraint Language is a precise text language that provides constraint and object query expressions that cannot otherwise be expressed by diagrammatic notation. As OCL was designed to be a declarative query language, every OCL program construct has a value. This means that even a usual conditional statement (`if then else endif`) is an expression with a value. It is important to know that OCL does not allow for modification of model elements. Also, OCL is a side-effect free language, as no modification to the model is allowed from the code.

The QVT Operational Mappings [4] is an imperative language that supports the creation of powerful model transformations. QVT extends OCL with all the necessary programming constructs that are needed to write complex transformations in a comfortable way. The extended OCL – Imperative OCL – preserves the object-oriented aspect of OCL with all its constructs, and extends it with program control statements, such as loop and conditional statement. OCL is a side-effect free programming language but this property does not apply to the Imperative OCL as this language extension makes model element modification available to the programmer. Furthermore, the instantiation of model elements is also possible. Overall, Imperative OCL makes the Object Constraint Language a versatile object-oriented language in which transformation rules can be written in a comfortable and platform-independent way.

2.2 Visual Modeling and Transformation System

The Visual Modeling and Transformation System [5] is a metamodel-based modeling environment that supports arbitrary levels of metamodels. Models and metamodels are not differentiated, thus a model on one level can be used as a metamodel on another. For example, in case of the UML class metamodel – class diagram – object diagram, a three step metamodel-model chain has been defined, in which the class diagram is both a model and a metamodel.

In VMTS, models and transformation rules are formalized as directed, labeled graphs. And as a basis, a simplified UML class diagram has been defined as the root metamodel. VMTS is also a UML-based model transformation system, which utilizes graph rewriting techniques. Moreover, the system supports the validation of constraints defined in the transformation rules. For this purpose, VMTS utilizes the Object Constraint Language. Precise model transformations can be expressed in Imperative OCL fragments.

The basic architecture of VMTS can be seen in Fig. 1. Models are stored in a model repository – a relational database – and can be reached through the Attributed Graph Supporting Inheritance (AGSI) layer. The visualization of model elements is handled by the VMTS Presentation Framework (VPF); VPF plug-ins enable custom graphical appearances for model elements. As VMTS is a transformation tool as well, a Rewriting Engine is included that utilizes graph matching and rewriting techniques. The OCL module facilitates constraint writing on model elements and defining precise matching rules. The Imperative OCL module allows model element modification from transformation steps. The Adaptive Modeler, which provides an integrated interface to the underlying layers, is the front-end to the system. With VMTS, generating class hierarchies for model traversing and processing is also possible (Traversing Processor module), thus, external tools can also benefit from VMTS models.

2.3 OCL Compiler in VMTS

A modeling environment that supports constraints written in OCL requires source code parsing. During the parsing process, a tree representation of the source is produced, which can be further processed later (e.g. for semantic analysis). In the case of compilers, this tree is traversed to generate target code, and later the target code can be executed. While in interpreters, the parse tree is processed during execution, and based

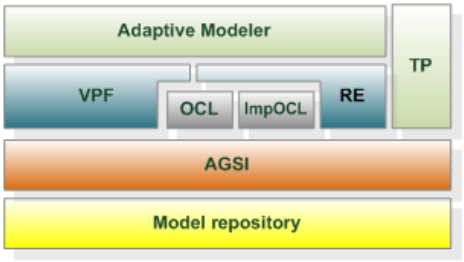

Fig. 1. The architecture of VMTS

on the tree, different code is executed. This means that compilers are usually faster than interpreters, as the necessary code analysis is performed before execution.

For source code parsing, several automatic compiler generators exist that require a formal grammar of the language. Therefore, one only has to develop the grammar to obtain a parser. In real life however, it is not this simple. Unfortunately, OCL is not specified and documented well enough for a formal grammar specification. Also, the specification contains several errors, inconsistencies and in places it can be interpreted ambiguously [8]. This also applies to Imperative OCL, but in its specification, even the syntax definition contains inconsistencies. Despite these troublesome factors, OCL grammars can be found for ANTLR [9], which generates LL(*) parsers, and for Bison [10], which is an LALR(1) [11] parser generator. However, in VMTS, we chose to implement our own grammar, partly because we needed the imperative extension and partly because the action rules that are executed during the syntax analysis have to use the VMTS framework. Action rules are executable code, placed next to the production rules, that should be executed when the given rule is applied by the parser. Usually, these routines implement semantic analysis, or in more complex cases, these are used for abstract syntax tree (AST) generation.

In VMTS, Jay [12], which is a Bison-like tool, has been used as a parser generator. This variant can generate a C# syntax analyzer from LALR(1) grammars. During the parsing process, the action rules build an AST, which is processed later in several steps. An AST is an ordered, directed tree. Ordered, because the sequence of the outgoing edges from a vertex is determined. Each inner vertex of this tree represents 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). The vertices are attributed with several properties of the tokens, such as *value*, *vertex type*, *expression type*, *token position*, etc. These attributes are evaluated partly during the parsing process, during the semantic analysis, and lastly during the code generation process. 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 these pieces of information are 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). After semantic processing, the AST is traversed once more for code generation. In VMTS, the Mi-

crosoft CodeDom [13] technology is used to produce a *CodeDom* tree, which can be transformed into C# code.

Fig. 2 illustrates a sample input OCL code and the corresponding C# code fragment generated by the VMTS OCL Compiler. The OCL code contains an invariant that iterates through a collection. In OCL, the return value of an *iterate* expression is the value of the accumulator variable after the execution of the loop. The structure of the generated C# code is obvious from the OCL code, a *namespace* is generated from the *package*, a *partial class* from the *context* and a *static method* from the *invariant*.

```
OCL:
package MyPackage
  context MyContext
    inv MyInvariant:
    collection->iterate( iterator;
      acc:Boolean=initValue | body-with-iterator-and-acc )
endpackage
```

```
C#:
namespace MyPackage {
  using ...
  public partial class MyContext {
    public static OCLBoolean MyInvariant() {
      OCLBoolean acc = initValue;
      for(int i=0; i < collection.size(); i = i+1){
        InnerType iterator = collection[i];
        acc = body-with-iterator-and-acc;
      }
      return acc;
    }
  }
}
```

Fig. 2. OCL and C# example code fragments

3 Low-Complexity Incremental OCL Compilation

We have examined what can be achieved if an OCL code is compiled and afterwards it is modified. To allow incremental builds, the basic language units that could be recompiled separately have to be determined. Obviously, the smaller the units that are handled the larger amount of overhead comes into our algorithm. In this section, we introduce some simple methods that could boost the performance of the compiler, but will not increase the complexity substantially.

3.1 Package

The largest segments of OCL code are `packages`, see Fig. 3, thus, this can serve as an obvious unit for the separation. Considering a piece of source code that contains several `packages`, during a rebuild, only the modified `packages` must be compiled again. With this very simple modification, the performance of our compiler can be immensely improved. Assume an input with length *n* that contains *k* number of units (`packages`). If the length of the modification is *m*, the average number of modified units is $\lceil \frac{k*m}{n} \rceil$. With incremental build, we can reduce the compilation time to its $(\lfloor m/n \rfloor)^{th}$. As mentioned before, `packages` are translated into C# namespaces. Therefore, the output of the incremental compilation will be valid if the original code compiled without errors, as there can be no unexpected interdependencies between `packages`.

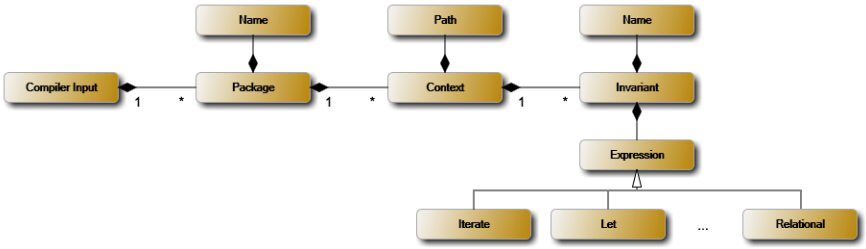

Fig. 3. Main structure of OCL

3.2 Context

The same method can be followed in the case of `contexts`, which are translated into C# partial classes. Each `package` may contain several `contexts`, which should be placed into the same C# namespace. The procedure works only because C# has partial classes that permit splitting the class definition into parts. In this way, the separate `contexts` that link to the same model element can be processed separately, although they contain the parts of the same class.

3.3 Invariant

Naturally, if a constant is changed somewhere in the OCL code, the whole unit, `package` or `context`, must be rebuilt, which takes considerable time. Therefore, it is worth trying to find a solution that can handle smaller segments of OCL. The previously illustrated examples have not introduced overhead to the compiler, however, if we consider invariants (`inv`) as separation units, further analysis and check is needed.

Invariants are compiled into C# functions placed into classes. These functions need to have unique names in the class. During a full rebuild, this can be checked, as all the generated function names are available in memory, perhaps in a symbol table, but in

case of partial rebuilds, when only individual invariants are rebuilt, this table might not be available. Obviously, without checking the names the compilation may not produce semantically correct output, as there could be name collisions. Thus, the function name table has to be produced somehow. One possible solution is to parse all of the invariants again, but only to the point where the name becomes available. In this case, the number of invariants in the source code affects the performance of the incremental build. Consider T_{full} time for a complete build of an invariant, T_{part} time for a parse that produces the name of the invariant. Then, a complete build of the input takes $T_{full} * k$ time, and in average, the cost of a partial rebuild is T in Equation 1.

$$T = T_{full} * \left\lceil \frac{k * m}{n} \right\rceil + T_{part} * \left(k - \left\lceil \frac{k * m}{n} \right\rceil \right) \tag{1}$$

Overall, this means that the overhead is inversely proportional to the number of modified invariants.

4 High-Complexity Incremental OCL Compilation

In this section, we provide an algorithm that incrementally compiles the modified expressions in the source OCL code. We give a method that produces complete semantic analysis based not on the whole source but on the modified expressions. Also the partial code generation process is illustrated. Finally, we will show that it is not always efficient to use the incremental compiler, as we introduce overhead that might slow down the incremental builder compared to the original one.

4.1 Algorithm

In OCL expressions are the smallest programming constructs that are meaningful on their own. Also, in Fig. 4 it is clearly visible that OCL expressions are recursive nonterminals in the language definition; therefore, it is hard to find a smaller – but still general – OCL fragment that can be considered a basic compilation unit.

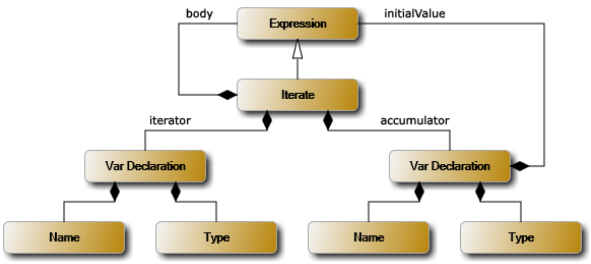

Fig. 4. A recursive OCL expression

The two properties mentioned above let us consider an algorithm that can handle OCL expressions as basic compilation units in the incremental builder, and still makes

universal handling of code fragments possible. The data flow diagram of the incremental compiler is depicted in Fig. 5.

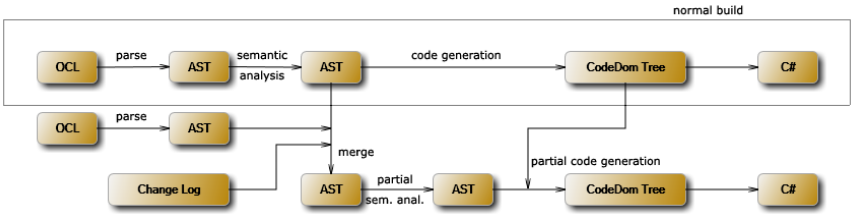

Fig. 5. Data flow of the incremental compiler

Firstly, a normal build is executed, afterwards only incremental builds are performed. This consists of a normal lexical and syntax analysis, a tree merge with the previously built AST, a partial semantic analysis and code generation, the latter two processes utilize the outputs of the previous build.

Algorithm 1 Incremental compilation algorithm

1: **procedure** $Merge(tree_{orig}, tree_{modif}, changeLog)$
2: $resetOrderIds(tree_{orig}), resetOrderIds(tree_{modif})$
3:
4: $rootId \leftarrow findChangedNode(tree_{orig}, changeLog)$
5: $tree_{root,orig}, tree_{root,modif} \leftarrow getSubtreesByOrderId(rootId)$
6: $trees_{unmod,orig} \leftarrow findUnChangedTrees(tree_{root,orig}, changeLog)$
7:
8: **for all** $tree_{unmod,orig}$ in $trees_{unmod,orig}$ **do**
9: $trees_{unmod,modif} \leftarrow matchTrees(tree_{root,modif}, tree_{unmod,orig})$
10: **for all** $tree_{unmod,modif}$ in $trees_{unmod,modif}$ **do**
11: $replaceTrees(tree_{unmod,modif}, tree_{unmod,orig})$
12:
13: $replaceTrees(tree_{root,orig}, tree_{root,modif})$
14:
15: $SemanticAnalyser.process(tree_{root,modif})$
16: $CodeGenerator.process(tree_{root,modif})$
17:
18: $maintainTokenPositions(tree_{root,orig}, tree_{root,modif})$
19: **return** $tree_{root,orig}$

Let $Tree_{orig}$ be the abstract syntax tree resulting from the previous build and regard $Tree_{modif}$ as the AST that is being processed now by the incremental compiler. Algorithm 1, which provides an implementation, is based on a simple observation: if the lowest vertex that is related to the modification made in the source code has been

found in $Tree_{orig}$, the incremental compiler only has to process the corresponding subtree ($Tree_{root,modif}$) in $Tree_{modif}$. Fig. 6 depicts the parse trees for the expressions $1 + (2 * 3)$ and $1 + (2 + 4 + 3)$ are depicted. Consider changing the $*$ sign to $+4+$, the above mentioned tree node identification results in the root of $Tree_{root,orig}$, which has the order ID 3. Now, as the ID is available, the corresponding node can be identified in $Tree_{modif}$. Moreover, if we consider several distinct modifications in the OCL code, there can be subtrees that are not modified. Overall, this means that it is enough

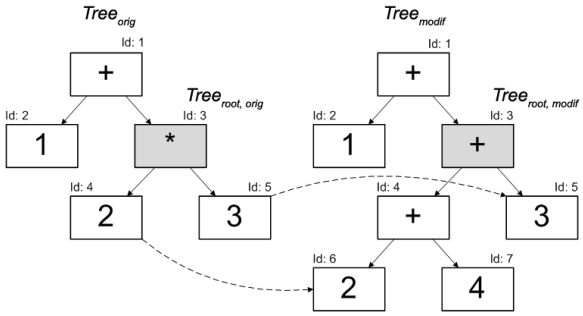

Fig. 6. Sample modification trees ($1 + (2 * 3)$ is changed to $1 + (2 + 4 + 3)$)

to find the lowest vertex that contains all the changes in $Tree_{orig}$, and if we locate the subtree which is on the same hierarchical position in $Tree_{modif}$, we have found $Tree_{root,modif}$. In our method, during a breadth first traversing of the trees each vertex is numbered. This level order numbering facilitates finding the root of the change in $Tree_{modif}$ based on the hierarchical equivalence of nodes.

After finding the root of the subtrees corresponding to the changes in the text the trees could be merged and analyzed, but this would mean that the unchanged parts of $Tree_{root,modif}$ are processed again. Thus, Algorithm 1 locates the unmodified parts in the trees and merges them together. With this extension only the modified parts of $Tree_{modif}$ have to be processed. In Fig. 6 the factors of the multiplication are not modified, thus, they have to be somewhere in $Tree_{root,modif}$. Unfortunately, they cannot be located by their ID as the structure of the original and modified subtrees are not identical, but the merging algorithm can find them.

The complexity of Algorithm 1 cannot be less than $O(n)$, where n is the number of vertices in the tree, because $resetOrderIds()$ needs to traverse the whole tree to number the vertices. However, if we consider that the complexity of the normal build is also $O(n)$, we find that smaller atomic units are required for comparing the two methods. Thus, the required units have to be determined. Let m be the number of vertices that have been changed in $Tree_{modif}$ compared to $Tree_{orig}$; assuming that during the semantic analysis k number of attributes have to be computed. Now, the running time of a normal build is

$$n*\overline{T}_{parse}+n*k*\overline{T}_{attrEval}+n*\overline{T}_{code} = n*\left(\overline{T}_{parse}+k*\overline{T}_{attrEval}+\overline{T}_{code}\right), \quad (2)$$

where $\overline{T}_x$ is the time needed by process x to handle one vertex.

In case of the incremental compiler, if we consider the time of an attribute evaluation the same as the querying time; the $resetOrderIds()$ function takes $n * \overline{T}_{attrEval}$ time. Finding the root of the changed node takes $log(n) * \overline{T}_{attrEval}$ time as we do not need to visit all the nodes, but only one branch at each vertex. Thus, the number of traversed vertices is equal to the height of the tree. Analyzing Algorithm 1 further we find that retrieving a subtree by an Id takes $n * \overline{T}_{attrEval}$ time as the tree vetices are numbered in a breadth first manner, and finding the unchanged nodes in $Tree_{root,modif}$ requires $m * \overline{T}_{attrEval}$. The $matchTrees(T1, T2)$ function, which returns all the subtrees that are similar to $T2$ from $T1$, is a recursive function. Therefore, its complexity can be easily determined by the Master theorem [14]. Also by simply considering that to find the matches to the root requires visiting all the nodes in the tree, and at each match it has to check all the nodes in $T2$. Thus, overall, it requires $\overline{T}_{attrEval} * size(T2) * size(T1)$ time. In Algorithm 1 it means $\overline{T}_{attrEval} * (m+l) * l$, where l is the number of unchanged nodes in $Tree_{root,modif}$.

4.2 Tracking Changes

This section deals with a method that can handle changes in the input code. One might ask if we need to track the changes in the source code? The answer is obviously not. As the changes are needed in the AST, it would be natural to find the changes in the trees. Several algorithms, such as [15], exist that find the differences between trees. However, these methods work only with the trees that they have to compare, although more information could be retrieved as the trees are results from a parsing method. By not considering the texts, but only the trees during the differentiation, [15] provides an $O(ND)$ algorithm to find the modifications in the trees, where N is the sum of the node-count in the input trees and D is the total number of misaligned nodes.

Programming tools that support incremental compilation keep track of changes in source code on different levels. For instance, in Microsoft Visual Studio, if a file in a project has been changed, only the project and its dependencies are rebuilt, the unrelated projects are not recompiled. During the development of a large software, this project-based incremental code building works fine. Compilation of $C++$ code with GCC [16] is based on object files, which make file-based incremental target building possible. Smaller compilation units are rarely used in industrial compilers as implementing the incremental compiler and validating the correctness of the output would not be possible in reasonable time. Finding differences between inputs based on lines is possible in $O(ND)$ [17], where N is the sum of the lengths of the two inputs being compared and D is the size of the required minimum edit script. However, OCL code is rarely long enough to be handled with line-based differentiators. Therefore, we chose to use character positions to keep track of changes. This is possible by modifying available line-based comparison algorithms. As we have an IDE for modeling, we can utilize methods that gain information from the user interface.

In the incremental compiler, we handle *KeyPressed* events of the input textbox and maintain a change log. This means that if two characters are inserted next to each other, we do not have two logs, but only one that contains a two character long modification. Also, distinct modifications are handled separately which allows finding unchanged

subtrees in the ASTs. It is important to notice that maintaining the change log runs parallel to the code writing, thus it will not increase the build time. Consider the expressions in Fig. 6. The change log contains only one entry here that shows that from character position 4 to 5 there has been a modification to character position 4 to 7. Table1 depicts how the change log is modified during code editing. This information is enough for

Table 1. Change log states based on input modifications

Current input	Current change log (*original pos* $\rightarrow$ *modified pos*)
$1 + (2 * 3)$	$empty$
$1 + (23)$	$4 - 5 \rightarrow 4 - 4$
$1 + (2 + 3)$	$4 - 5 \rightarrow 4 - 5$
$1 + (2 + 43)$	$4 - 5 \rightarrow 4 - 6$
$1 + (2 + 4 + 3)$	$4 - 5 \rightarrow 4 - 7$

the incremental compiler, however, more information could be gained. Character insertions and deletions could be distinguished. Character modifications could be tracked as a separate delete and insert log. In the case of Fig. 6 this would mean a deletion from position 4 to 5, and a three-long insertion to position 4.

A mapping between the parsed tokens and the change log is needed as the changes in the trees are used in the algorithm and not in the texts. Providing a mapping is possible if during the parsing process each token is annotated with its actual character-based position. This extension to the compiler can easily be implemented and it does not place a performance burden onto the parser. Only a counter has to be incremented with each parsed character in the compiler to implement a mapping between character position and tokens. When a token is emitted by the lexical analyzer the actual character position can be retrieved from the counter. Moreover, if the compiler provides error messages that locate the syntax errors, no extension is needed because the counter has already been implemented.

4.3 Partial Semantic Analysis

After merging the ASTs together for forming a partially analyzed tree, which is syntactically correct, the rest of the tree has to be processed. From the root of $Tree_{modif,part}$ a downwards going semantic analysis has to be performed, which is equivalent to the one used in the normal analysis process. This semantic check will leave out the previously analyzed tree nodes, thus, the unmodified parts of $Tree_{modif,part}$ will not be traversed again.

Unfortunately, the AST must be analyzed in an upward manner as well because modifications might cause changes in the types of the expressions, see Example 1. Therefore, the correctness of the whole tree has to be checked again.

Example 1. Consider *1+1+1* as the original input, the resulting parse tree is depicted in Fig. 7(a) where each node has the type *Integer*. Now, if we change the input to *1.2+1+1*

– the parse tree is shown in Fig. 7(b) –, meaning that only the first token has been modified, all of the nodes have to be reanalyzed upwards in the tree from that node, as the types have to be changed to *Real*.

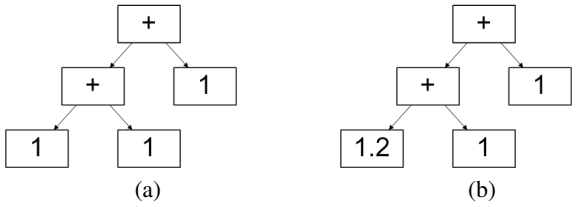

Fig. 7. The parse trees of (a) *1+1+1* (b) *1.2+1+1*

Example 1 also shows that no number can be chosen to limit the level of the upwards checking because the modification can spread to the root of the tree. However, it is clearly visible that further semantic checking is not necessary if the type of the modified element is the same as the original type on a given tree level.

In Algorithm 1 m nodes have to be analyzed in $Tree_{modif,part}$ which requires $m*k*\overline{T}_{attrEval}$ time, and upwards check might reach the root. Therefore, a maximum $log(n)$ number of nodes have to be rechecked. Overall, the required time is maximum:

$$\overline{T}_{attrEval} * k * (m + log\,(n)) \tag{3}$$

4.4 Partial Code Generation

Partial semantic checking produces a fully analyzed and correct tree, thus, code generation can be started. During a partial code generation, a *CodeDom* tree is produced from the previous *CodeDom* tree and the AST output of the semantic analysis.

To facilitate partial builds AST nodes are annotated with *CodeDom* expressions that have been generated previously. Thus, if a node in the AST contains *CodeDom* expressions, that does not have to be regenerated. This results in a similar algorithm to the one used during the semantic analysis where only the modified expression have to be processed in $Tree_{modif,part}$. However, in this case, upward processing has to reach the root as a single modification affects the output of the root.

In Algorithm 1 the required time for partial code generation is

$$\overline{T}_{code} * (m + log\,(n)) \tag{4}$$

based on the considerations made at partial semantic analysis. Formula (4) assumes that the time for code generation varies directly with the number of nodes.

5 Performance

Complexity analysis is a widely used technique to study the running times of our algorithms, but usually we still want to visualize the performance on charts or figures.

Benchmarks are efficient for comparing solutions by their relative performance. Several general purpose benchmarks exist for hardware and software performance measurements, but only a few for graph transformations, such as [18], and there is none for OCL compilers. Mainly because no typical OCL code can be chosen for performance analysis.

```
OCL:
package MyPackage
  context MyContext
   inv MyInvariant:
     (((1000+2000)+(3000+4000))+((5000+6000)+(7000+8000)))+
     (((1001+2001)+(3001+4001))+((5001+6001)+(7001+8001)))+
     ...
     (((1007+2007)+(3007+4007))+((5007+6007)+(7007+8007)))
         > 9000
endpackage
```

Fig. 8. Test input OCL code

In Fig. 8 a sample OCL code is shown that is used for comparing the running time of normal and incremental builds. The code contains an invariant that compares a sum to a constant. The left side of the expression is the sum of eight almost identical subexpressions. During the test the running time of incremental compilation has been measured in the function of the length of the change log. In Fig. 9 the measured running times can be seen. Note that the times displayed are the sums of the running times of 1000 builds.

In Fig. 9 an ascending function is depicted; the larger the modification is the more time is needed for the compilation, as we have expected. Also, eight steps are clearly visible which are caused by the recompilation of the modified identical subexpressions. After modifying about 300 characters the normal build is faster than the incremental one, mainly because of the performance overhead of the algorithm.

Naturally, the results shown in Fig. 9 are not representative as no model checking was needed and only additional operations have been used. Model checking would significantly increase the running times as model repository querying would be needed. This would result in higher steps in the diagram. Unfortunately, these cases do not facilitate measurement considerations.

6 Conclusion and Future Work

This paper showed an incremental compiler application which translates OCL pieces of code into their C# equivalents. The introduced method is utilized in a modeling environment as a constraint and transformation rule translator. The paper discussed several ways of solving the problem of incremental compilation. The methods have been separated based on the size of the basic compilation units. Also, for enabling efficient mod-

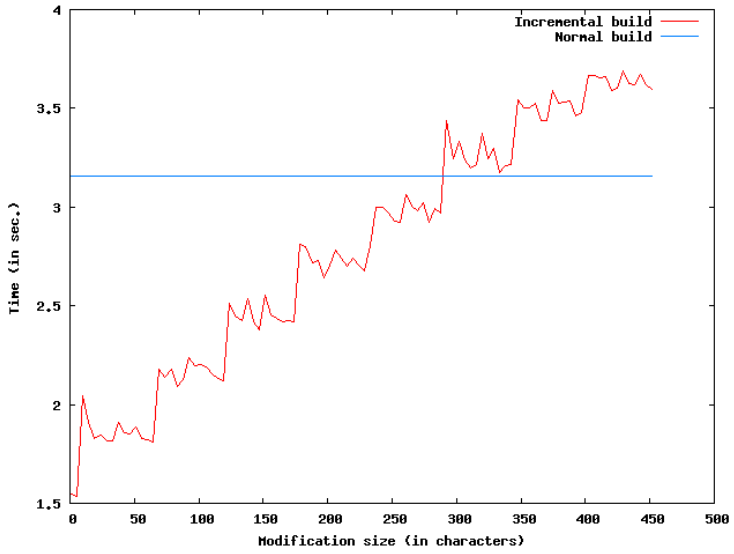

Fig. 9. Incremental and normal compilation test results

ification tracking, a method that mapped characters to tokens was introduced and the modeling environment has been extended to send information containing the positions of the changes to the compiler. These extensions to the original compiler resulted in an immense performance improvement, which makes seamless work with the modeling environment possible.

Future work includes extending the current incremental compiler to Imperative OCL code. Usually, OCL code has very few variables which does not stand for Imperative OCL. Therefore, variable referencing has to be solved in the incremental compiler. Currently we assume that a modification in the code is causing changes only in the surroundings of the actual change. But with variable references, this idea of localization does not work as the references point out from the subtrees that has been modified. This problem also arises with function declarations. Another imperfection in the current algorithm is that it does not handle modifications to the model elements, only changes to the code. This could be handled if the modeler sent notifications when a model element has been changed on the user interface. Furthermore, in the future we plan to experiment with switching between full and partial compilation, this balancing process requires more measurements.

7 Acknowledgement

This paper was supported by the János Bolyai Research Scholarship of the Hungarian Academy of Sciences and by the Mobil Innovation Centre at Budapest University of Technology and Economics.

References

1. P. Stevens and R. Pooley, *Using UML: Software Engineering with Objects and Components*. Object Technology Series, Addison-Wesley, 1999. Updated edition for UML1.3: first published 1998 (as Pooley and Stevens).
2. J. Warmer and A. Kleppe, *Object Constraint Language, The: Getting Your Models Ready for MDA, Second Edition*. Addison Wesley, 2003.
3. A. Kleppe, J. Warmer, and W. Bast, *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison Wesley, 2003.
4. OMG, *Object Management Group Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification*. Object Modeling Group, July 2007.
5. VMTS Team, "Visual Modeling and Transformation System website." `http://vmts.aut.bme.hu/`.
6. T. Vajk, G. Mezei, and H. Charaf, "Architecture of an In-Memory Transformation Engine," in *8th International Symposium of Hungarian Researchers on Computational Intelligence*, (Budapest, Hungary), pp. 573–581, November 2007.
7. L. Lengyel, T. Levendovszky, G. Mezei, and H. Charaf, "Control Flow Support in Metamodel-Based Model Transformation Frameworks," in *EUROCON 2005 International Conference on "Computer as a tool", Proceedings of the IEEE*, (Belgrade, Serbia and Montenegro), pp. 595–598, November 2005.
8. A. Hamie, F. Civello, J. Howse, StuartKent, and R. Mitchell, "Reflections on the Object Constraint Language," in *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998, Selected Papers*, vol. 1618 of *LNCS*, pp. 162–172, Springer, 1999.
9. T. Parr, *The Definitive ANTLR Reference*. Pragmatic Bookshelf, May 2007. Building Domain-Specific Languages.
10. C. Donnelly and R. M. Stallman, *Bison Manual for Version 1.875*. Free Software Foundation; 8th edition, September 2003. Using the YACC-Compatible Parser Generator.
11. A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison Wesley, Pearson Education, 1986.
12. "Jay (Language Processing)." `http://www.cs.rit.edu/~ats/projects/lp/doc/jay/package-summary.html`.
13. C. Nagel, B. Evjen, J. Glynn, K. Watson, and M. Skinner, *Professional C# 2008*. Wrox, 2008.
14. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. The MIT Press, 2nd ed., 2001.
15. S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, "Change detection in hierarchically structured information," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 493–504, 1996.
16. W. von Hagen, *The Definitive Guide to GCC*. Apress, 2006.
17. E. W. Myers, "An O(ND) Difference Algorithm and Its Variations," *Algorithmica*, vol. 1, no. 2, pp. 251–266, 1986.
18. G. Varro, A. Schürr, and D. Varro, "Benchmarking for graph transformation," in *Proc. IEEE Symp. Visual Languages (VL/HCC)* (A. Amber and K. Zhang, eds.), (Los Alamitos), University of Texas at Dallas, IEEE Computer Society Press, 9 2005.