YAAB (Yet Another AST Browser): Using OCL to navigate ASTs

G. Antoniol*, M. Di Penta*, E. Merlo**
antoniol@ieee.org dipenta@unisannio.it ettore.merlo@polymtl.ca

* RCOST - Research Centre on Software Technology
University of Sannio, Department of Engineering
Palazzo Bosco Lucarelli, Piazza Roma 82100 Benevento, Italy

**École Polytechnique de Montréal
Montréal, Canada

Abstract

In the last decades several tools and environments defined and introduced languages for querying, navigating and transforming abstract syntax trees. These environments were meant to support software maintenance, reengineering and program comprehension activities.

Instead of introducing a new language, this paper proposes to adopt the Object Constraint Language (OCL) to express queries over an object model representing the abstract syntax tree of the code to be analyzed. OCL is part of the UML lingua franca and thus several advantages can be readily obtained. Central to the idea is to shift the analysis paradigm from a tree-based to an object-oriented paradigm, and to provide a meta-model decoupling the query language from the target language.

This paper presents the current status in implementing an OCL interpreter with the ability of querying an object model representing the abstract syntax tree, as well as some interesting applications, such as extracting software metrics or computing clones.

Keywords: source code analysis, UML, OCL, AST navigation

1. Introduction

Software maintenance and program comprehension very often involve costly and tedious code browsing activities to locate data structures, code areas and, more generally, concepts related to programmers tasks. Indeed, the time pressure, an outdated or inexistent documentation, force maintainers to rely on the unique form of available and reliable documentation: the source code itself. Unfortunately, as the size and complexity of software system increases, the practice of code browsing becomes a very complex and resource demanding exercise.

Integrated development environments (e.g., Microsoft Visual C++^{TM}, Borland C++ Builder^{TM}/JBuilder^{TM}, etc.) provide a useful means supporting developer basic operations. However, these tools have limited customization and programming capability and thus they play a little or no role when concepts needs to be located in large software system.

Very promising technologies, such as program slicing [1, 2, 3] and impact analysis [4, 5, 6, 7, 8], have been sometimes integrated into industrial environments, like *GrammaTech CodeSurfer*, *Semantic Designs Source Code Browser*, *Codecrawler*, *Rigi* and many others. These tools have very powerful and well-known capabilities, though they have been often designed to solve particular categories of problems. For example, *CodeSurfer* [9] is particularly indicated for point-to analysis and slicing, *Rigi* [10, 11] was conceived as a visual tool to help program comprehension and reverse engineering, *Codecrawler* [12] is a language independent reverse engineering tool integrated with metrics computation and large software system visualization capabilities, *Semantic Designs Source Code Browser* [13] helps program comprehension by allowing navigation of source code extracted documentation and hyperlinked Java code. The drawback is that these tools may be difficult to integrate with traditional programming environments/languages to build custom tools.

Several categories of languages/toolkits for source code analysis and transformation have been developed during the last decade. Among all, those providing a powerful language oriented to program comprehension and transformation are the *Design Maintenance Systems* (DMS) produced by Semantic Designs Inc. [14, 15], the *TXL programming language* produced by TXL Software Research Inc. [16, 17], *Refine* [18] produced by Reasoning Systems

DRAFT

Inc., and *FermaT* by Martin Ward [19, 20]. These tools have powerful analysis capabilities, if compared to standard development environments; e.g., they provide pattern-matching languages and a way to query and transform the Abstract Syntax Tree (AST) produced by a parser. By providing mechanisms to query and transform an AST, they can be used to fulfill several analysis and comprehension tasks, as well as to carry out maintenance and source code transformation tasks. However, such tools require trained and skilled people, in that they define proprietary languages.

This paper proposes to adopt the Object Constraint Language (OCL) [21] as a specification language to browse, navigate, and query, via an underlying programming language object model, ASTs.

OCL is a formal language easy to read and to write, developed to specify constraints and more general expressions; OCL has been used in the UML Semantics documents and could be considered part of the UML users' background. It is not the intention of the authors to create yet another language, or a language similar to those already defined and available in the aforementioned transformation and reengineering environment.

By adopting OCL several other advantages are readily available. First and foremost, the analysis paradigm changes: the focus is no longer an AST or parse tree paradigm, rather an Object-Oriented (OO) paradigm. The idea is not new and to same extent was already introduced by tools such as *JavaCC* [22] and books such as [23]. *JavaCC* allows to generate an AST based on a class hierarchy, and it provides also a mechanism, based on the *Visitor* design pattern [24] to navigate and manipulate it. In [23] a clean and elegant OO approach to develop compilers is presented even if the author position is clear and, due to porting issues, more inclined to non-OO style in writing code. The second advantage is that UML and OCL are *de-facto* standards, thus we aim to reduce the learning curve. Third, OCL is powerful enough to express in a concise and elegant way very complex conditions. Finally, by changing the underlying language model, OCL based applications can be easily ported from one language to another (e.g., from Java to C).

This paper presents the idea, the preliminary implementation of an OCL interpreter, and some possible applications in querying an object model of the Java programming language. The aim of this paper is therefore to explain how an object model of a programming language can be defined so that the AST representation is obtained, how OCL can be used for navigating it, and what are the limitations of our current OCL interpreter implementation.

As a test application, we wrote a software metrics extractor for the Java language. OCL provides a way for navigating and computing metrics on the object model very close to the standard notation used to express constraint on any UML object model. Currently, we are extending the OCL interpreter capabilities and, in the meantime, developing a C object model and parser.

The remainder of the paper is organized as follows: Section 2 presents the language - AST object model; a brief overview of OCL basic notions is reported in Section 3. Section 4 discusses, presenting some examples, how OCL can be used as an AST navigation and query language. Section 5 presents the experience of the authors in developing the OCL interpreter; finally Section 6 gives concluding remarks and outlines directions for future work.

2. The AST Object Model

To navigate and query an AST (extracted from a chunk of source code) using OCL, an object model (representing the underlying source code programming language) has to be preliminary defined. Rules to map a programming language grammar to a class hierarchy have been defined in [23], and can be summarized as follows:

1. A tree is described by one or more abstract classes, corresponding to non-terminal symbols of the grammar;
2. Each abstract class is extended by one or more subclasses, one for each grammar rule in which the abstract class appears on the left side; and
3. For each nontrivial symbol in the right side of a rule, there will be one field (i.e., attribute, association or aggregation) in the corresponding class.

Our language object model is slightly more complex, to ease the navigability of the AST itself. In particular, it considers the way in which grammar rules are represented in *JavaCC*:

- If there are more grammar rules with the same left-hand side, they may be expressed using the option ``|'' operator, e.g., `cond_statement = if_statement | for_statement | while_statement`; this, again, is mapped, in the object model, in a class hierarchy where the left-hand side is the superclass and the options on the right-hand side the subclasses; and
- Sequences and options are expressed using the + (one or more), * (zero or more) and [ ] (optional) operators, e.g., `argument_list = expression (``,'' expression)*`. Such sequences and options will be translated, in the object model, in associations with target multiplicities `1..*`, `0..*` and `0..1`, respectively.

Classes representing the grammar are derived from a superclass `Node`; `Node` is the supertype for all nodes in the

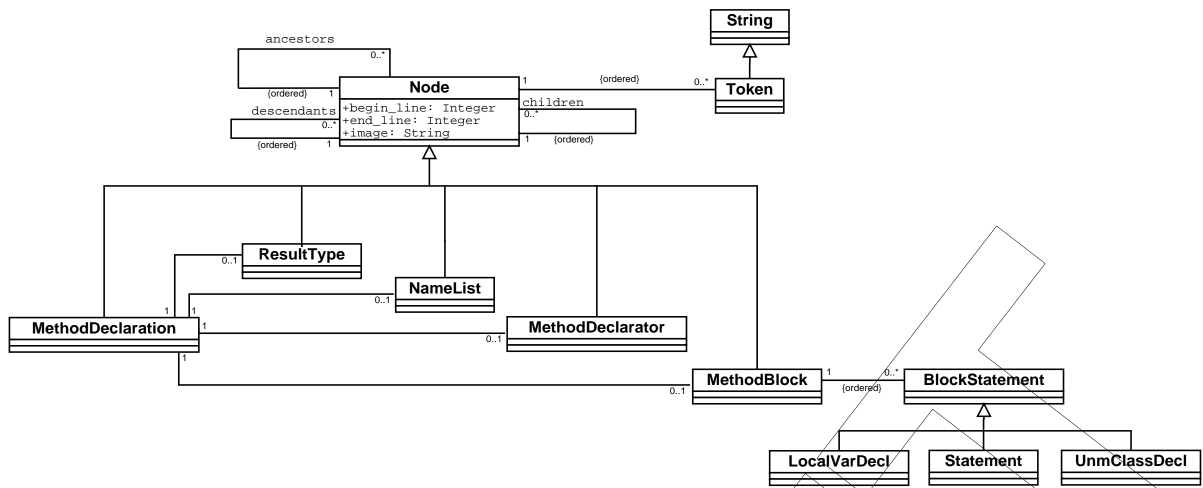

Figure 1. AST object model example: Method Declarator

language object model and it factors out the elements useful to query and navigate the AST. The following associations and attributes are defined via the superclass `Node`:

- Each node is associated to (all) its children. This will allow to consider the children of a node as an OCL Sequence (as described in Section 3, something similar to an array), and to iterate through it;
- For similar reasons, each node is associated with (all) its descendants; as shown in Section 4, this kind of links are particularly useful to locate, inside an AST, all the nodes satisfying a specified property;
- Each node is directly associated with (all) its ancestors. This is useful, for example, to perform backtracking analysis, such as retrieving (see Section 4.1) the conditional statements influencing the execution of a particular statement;
- Each node is directly associated with (all) its descendant tokens (terminal nodes);
- Each node has two attributes, `begin_line` and `end_line`, indicating at which line of the file the node begins and ends. As shown in Section 4, this can be used, for example, to compute the LOC metric; and
- Finally, each node is associated with its image (i.e., the string associated to it).

Clearly, having associations to all the descendants and ancestors does not necessarily imply the actual existence of such links. In the model implementation, each node is only associated to its parent and its children. The Sequence of ancestors and descendants is built on the fly, when needed, using appropriate visitors.

We are currently investigating the overhead imposed by the navigation structure. For the Java programming language and moderate system sizes, it does not constitute a limit. For other languages such as C and very large applications (i.e., a main plus all the linked files) some optimizations may be required.

Figure 1 shows an excerpt of the object model corresponding to the Java method declaration as represented in the *JavaCC* grammar. Several *JavaCC* grammars are available at the *JavaCC* grammar repository [22] . The grammar was modified superimposing via the grammar superclass the navigation structure: as shown, all classes are subtypes of the abstract class `Node`. A `MethodDeclaration` has associations with its children, i.e., `ResultType`, `NameList`, `MethodDeclarator` and `MethodBlock`. A `MethodBlock` contains zero or more objects of type `BlockStatement`. A `BlockStatement`, in turn, may be a local variable declarator, an unmodified class declarator or a statement.

3. OCL Overview

OCL is a formal language used to express constraints on a UML model, i.e., conditions that must hold on the system being modeled. OCL was created to overcome the limitation of UML to provide all the relevant aspects of the specification. In particular, UML does not have a formal mechanism to express constraints on diagrams and to specify pre and post conditions for class methods, class invariants, nor guard conditions on statecharts and interaction diagrams.

OCL is a pure expression language, therefore OCL ex-

pressions are guaranteed to be without side effects, i.e., any OCL expression simply returns a value, and it cannot change anything in the model.

Providing an extensive description of OCL is out of the scope of this paper; we will limit ourselves to highlight the OCL characteristics that are the most relevant to define an AST navigation language. Further details can be found in [21].

In order to express constraints, OCL allows navigability among classes in a UML meta-model. OCL supports two navigation forms. The first is indicated by "." (the same for accessing to a class attribute or method or to apply a feature to a single object), while the fact that a feature is applied on a Collection (see below) is indicated by the symbol "$->$". When navigating through an object model, class names are indicated in lower cases.

Given the AST object model shown in Figure 1, from an object `m` of type `MethodDeclaration` one can access to the line where its `MethodBlock` begins, simply writing `m.methoddeclarator.begin_line`. As shown in Section 4, this formalism is particularly useful for navigating from an AST node to its children (i.e., from a class to its associated classes).

A useful feature is the ability to verify if an object is an instance of a given type, or if it is of one of the supertypes of the given type. This can be done by applying to an object, respectively, the features `oclIsTypeOf` and `oclIsKindOf`. Thus, given a node `v` of type `Statement`, the expression `m.oclIsTypeOf(Statement)` returns *true*, as well as the expression m.oclIsKindOf(BlockStatement).

Another interesting feature is the ability to access all the instances of a class, e.g., `MethodDeclaration.allInstances`. In this case, the class name has to be written using the same case of the class diagram.

In addition to classes specified in any meta-model, OCL defines:

- Basic types: Integer, Real, Boolean, String, each one provided with a set of the most important features (e.g., basic String manipulation features are available); and
- Collection types: the result of navigation is, in general, a Collection. In particular, the result of a single navigation is a Set, the result of a combined navigation is a Bag, and the result of the navigation over an {*ordered*} association is a Sequence.

Casting between different Collection types is supported via the `oclAsType` operator; Collection types are provided with a powerful set of operator enabling the creation of new Collections from existing ones.

This is a key factor and the main motivation underlying our choice of OCL to express the navigation and the query of an AST. The idea is not new, for example, a powerful set of operators to handle collections and sets was present in the Reasoning Software Refinery language [18]. The following is a brief summary of the OCL available constructs operating over collections:

- Extract from a collection all the items (not) verifying a given property - `select` (`reject`); perform a projection (`collect`);
- Universal quantifiers: `forall`, `exists`;
- `Iterate` through all the items of a collection verifying a given conditions, possibly, updating the value of an accumulator variable (returned at the end of the iteration);
- Basic set operators: set union, intersection, difference, set cardinality as well as operator to test if an item belongs to a collection, etc.; and
- Sequence operators: accessing to the first, last, or *i-th* item, extracting a subsequence, etc.

4. OCL for Navigating ASTs

The aim of this Section is to demonstrate with some examples how OCL can be used to fulfill the typical needs of an AST navigation language supporting program comprehension. The examples will be referred to the Java programming language, and the object model corresponds to the Java grammar available at [22].

In particular, we will discuss how to implement the following features:

- Returning some subtrees of an AST: e.g., all the classes declared in a Java file or all the method declared inside a Java class;
- Matching subtrees;
- Searching for all AST nodes having a given property and, in particular, computing some metrics. We will show how some of the metrics used to perform a metric-based clone detection [25, 26, 27] can be computed.

AST navigation languages, as mentioned in the introduction, are very often integrated with construct to help AST transformation. However, OCL is a pure expression language, and thus it cannot be used to describe transformations. An extension of OCL for AST transformation is part of our future works.

DRAFT

```
extractClasses(cu: CompilationUnit):Sequence(UnmodifiedClassDeclaration)
post: result=cu.descendants->select(oclIsTypeOf(UnmodifiedClassDeclaration))


extractMethods(cu: CompilationUnit):Sequence(ClassBodyDeclaration)
post: result=c.descendants->select(oclIsTypeOf(MethodDeclaration)
                                   or oclIsTypeOf(ConstructorDeclaration))


Conditioning(s: Statement):Sequence(Statement)
post: result=s.ancestors->select(oclIsTypeOf(ForStatement)
                                 or oclIsTypeOf(WhileStatement)
                                 or oclIsTypeOf(DoStatement)
                                 or oclIsTypeOf(SwitchStatement)
                                 or oclIsTypeOf(IfStatement)
                                )
```

Figure 2. Extracting nodes from a tree

Operations performed with OCL are expressed, in this Section, using the standard notation adopted to describe post conditions:

```
MethodName ``(`` [ paramName '':''
paramType ([``,'' paramName '':''
paramType])* '')'' '':''  ReturnType
post:  oclExpression
```

4.1 Extracting Nodes Having a Given Property

The first operation to perform, prior to execute any program comprehension task, is to extract the AST subtrees subject of our analysis. Normally, what we obtain from the parser is an AST for each source file parsed or, if our parser is able to perform a multi-file parsing, an AST forest. Let us now suppose, for sake of simplicity, that our parser works, as in the former case, on a single file, then we may be interested to extract, from a source file:

- The Sequence of all declared classes;
- For each class, the Sequence of its methods; and
- For each class, the Sequence of all its attributes.

The first task can be performed, given a Compilation Unit `cu` (i.e., the root of a source file AST), by the function `extractClasses` shown in Figure 2. The `select` feature is applied to all the descendant nodes of `cu` (i.e., to all nodes of the source file AST). It selects only the nodes of type `UnmodifiedClassDeclaration` (the condition is expressed using the `oclIsTypeOf` feature), and then returns them as a Sequence.

Similarly, the function `extractMethods` extracts all methods declared inside a class `c`.

AST nodes corresponding to attributes and other fields can be similarly matched.

Finally, another interesting example: given a statement `s`, one may be interested to retrieve the Sequence of conditional statements influencing its execution. This can be done using the function `Conditioning` that, as shown in Figure 2, takes advantage of the association a node has with all its `ancestors`.

4.2 Matching Subtrees

The second feature that an AST navigation language should implement is the ability to perform some forms of tree/subtree matching. In particular, we defined two kinds of operators:

1. An operator to determine if two trees *exactly* match, i.e., if the root nodes of the two trees contain the same number of children (both terminals and non-terminals) and all children match; and
2. An operator to determine if two trees *structurally* match, i.e., if the root nodes of the two trees contain the same number of children (both terminal and non-terminals) and all non-terminal children match.

The first operator was associated with the "=" OCL operator, that returns a Boolean value indicating if two objects match. For the second operator, we extended OCL with a feature `structMatch(target: oclAny):Boolean` that applies on any OCL object, i.e., on only AST node in our case.

To better understand the details, let us consider two examples. Firstly, given an Expression `e`, we need the list of all conditional statements of a method `m` where the condition is *exactly* `e`, i.e., the condition must be expressed on the same variables with the same constants. Thus, we may write

DRAFT

```
matchExpression(m: MethodDeclarator e: Expression):Sequence(Statement)
post:result=m.methodblock.descendants->select(s: Statement | (s.oclIsTypeOf(ForStatement)
                                                              or s.oclIsTypeOf(WhileStatement)
                                                              or s.oclIsTypeOf(DoStatement)
                                                              or s.oclIsTypeOf(IfStatement)
                                                              or s.oclIsTypeOf(SwitchStatement))
                                                           and s.Expression=e)


detectClones(c: CompilationUnit):Sequence(MethodDeclarator)
post: result=c.descendants->iterate(n: Node; s:Sequence(MethodDeclarator) |
                                             if n.oclIsTypeOf(MethodDeclarator)
                                                and c.descendants->exists(n1: Node |
                                                                            not s->including(n1)
                                                                            and n1.structMatch(n))

                                                then s->append(n)
                                             endif
                                  )
```

Figure 3. Matching subtrees

the function `matchExpression` shown in Figure 3. As shown, the function iterates on all the descendants nodes of the Method Block of `m`. The feature `select` provides to return only the statements `s` satisfying the enclosed condition, i.e., `s` must be a conditional statement, and its conditional expression must *exactly* correspond to `e`.

The second example aims to find *cloned* methods contained in a file. Literature reports several methods to detect *clones*, from metric-based [25, 26, 27] to those based on matching subtrees [28]. For illustrating the *structural* matching feature, we will refer to the latter approach (i.e., tree matching based). Thus, given the `c` the *CompilationUnit*, i.e., the root node of the file parse tree, we may define the function *detectClones* shown in Figure 3.

The OCL feature `iterate` has been used. This feature iterates on all the items of the Sequence composed by all `c` descendants and, for each node `n` a test is carried out to verify whether or not a subtree rooted in `n` matches any other subtree (rooted in `n1`). The functionality requires, as shown in Figure 3, the *exists* feature. The user-defined *structMatch* feature checks if two nodes have the same number of children, and all their children having equal position also satisfy the *structMatch* feature.

4.3 Computing Metrics

Given a method `m` from a Java class, let us suppose we want to compute some metrics on it such as metrics related to size (LOC, number of statements), complexity (e.g., McCabe complexity), coupling (e.g., method passed parameters, local or global variables).

To compute the number of method's passed parameters, the function `Parameters` shown in Figure 4 can be used. The function shows a classical example of navigation through the UML model of an AST: a `NameList` is part of a `MethodDeclarator`, and `FormalParameters` on its turn, is part of a `NameList`. The value is computed as the size of the node Sequence by applying the feature `count()`.

The LOCs may be computed as the difference between the beginning and the end of a method as described in the function `LOCs`. It takes a `MethodDeclarator` node as parameter, and then returns the difference between its bounding block `end_line` and `begin_line`.

The function `Statements` (see Figure 4) computes the number of statements composing a method. The feature `select` takes all the descendants nodes of the method block (i.e., all nodes of the method block AST) and returns the Sequence of nodes satisfying the specified condition: nodes derived from the `BlockStatement` (i.e., Statement, variable declarator, or unmodified class declarator). Then, as in the previous case, the feature `count` returns the Sequence size.

Much in the same way, other elementary metrics such as the number of return statments (`ReturnStatements` function) or the number of methods calls can be computed.

The cyclomatic complexity (see function Cyclomatic in Figure 4) can be computed counting all the decision points in a block and then adding 1 [29].

Slightly more complex is the task of counting the number of variables declared in a block `b` (i.e., all the local variables of that block). This means that, for each block statement of type `VariableDeclarator`, we need to count the number of declared variables. Such

DRAFT

```
Parameters(m: MethodDeclarator): Integer
post: result=m.namelist.formalparameters.formalparameter->count()


LOCs(m: MethodDeclarator): Integer
post:result=m.methodblock.end_line-m.methodblock.begin_line


Statements(m: MyMethodDeclarator): Integer
post:result=m.methodblock.descendants->select(oclIsKindOf(BlockStatement))->count()

ReturnStatements(m: MethodDeclarator): Integer
post:result=m.methodblock.descendants->select(oclIsTypeOf(ReturnStatement))->count()

Cyclomatic((m: MyMethodDeclarator): Integer
post: result=m.methodblock.descendants->select(oclIsTypeOf(ForStatement)
                                               or oclIsTypeOf(WhileStatement)
                                               or oclIsTypeOf(DoStatement)
                                               or oclIsTypeOf(IfStatement)
                                               or oclIsTypeOf(SwitchLabel))->count()+1


Locals(b:MethodBlock): Integer
post: result=b.blockstatement->iterate(n:Node; r:Integer |
                                              if oclIsTypeOf(LocalVariableDeclaration)
                                                 then r+n.variabledeclarator->count()
                                              endif)
```

Figure 4. Computing metrics

operation can be expressed in OCL as described in Figure 4 by the function `Locals(b:MethodBlock)`. The function iterates on all items of the Sequence `BlockStatement` and, for each node `n` of type `LocalVariableDeclaration`, it counts all the items of the Sequence `VariableDeclarator`, incrementing the Integer accumulator `r` (to be returned at the end of the iteration).

4.4 Discussion

Although not explicitly designed to navigate ASTs, OCL allows easily expressing navigation and query constructs. This is not surprising, in that the goal of OCL is to express pre and post conditions, as well as guard conditions on UML diagrams. Once mapped into an object model, the grammar of a programming language corresponds to an UML object model, and thus OCL can be used to state properties and conditions over the grammar representation itself.

As shown, it has been possible to express in OCL all the features described at the beginning of this Section. Such features represent, in the opinion of the authors, the basic querying tasks to be performed on an AST. Moreover, the OCL expressions are in general, not very complex, easy to be understood and maintained. We believe that this is due to the powerful language capability of:

1. Navigating an object model; and
2. Executing query operations on collections and sets (also present is some other proprietary languages, such as *Refine*).

One (intentional) limitation of OCL is that it is not a programming language, therefore it cannot express program logic or flow control. This does not prevent to express possible queries useful for program comprehension purposes. However, to perform composite tasks or tasks with side effects (e.g., clone detection, source code transformation), OCL needs to be complemented/integrated with other tools/languages. At the time of writing, we identified, as a suitable solution, the integration with the *JavaCC* environment, where the programmer retrieves nodes/properties executing OCL queries with an approach similar to what done to executing SQL queries from any programming language.

5. Tool Development Issues

We are currently implementing an OCL interpreter with the purpose of navigating and querying a programming language object model. The interpreter must integrate the

DRAFT

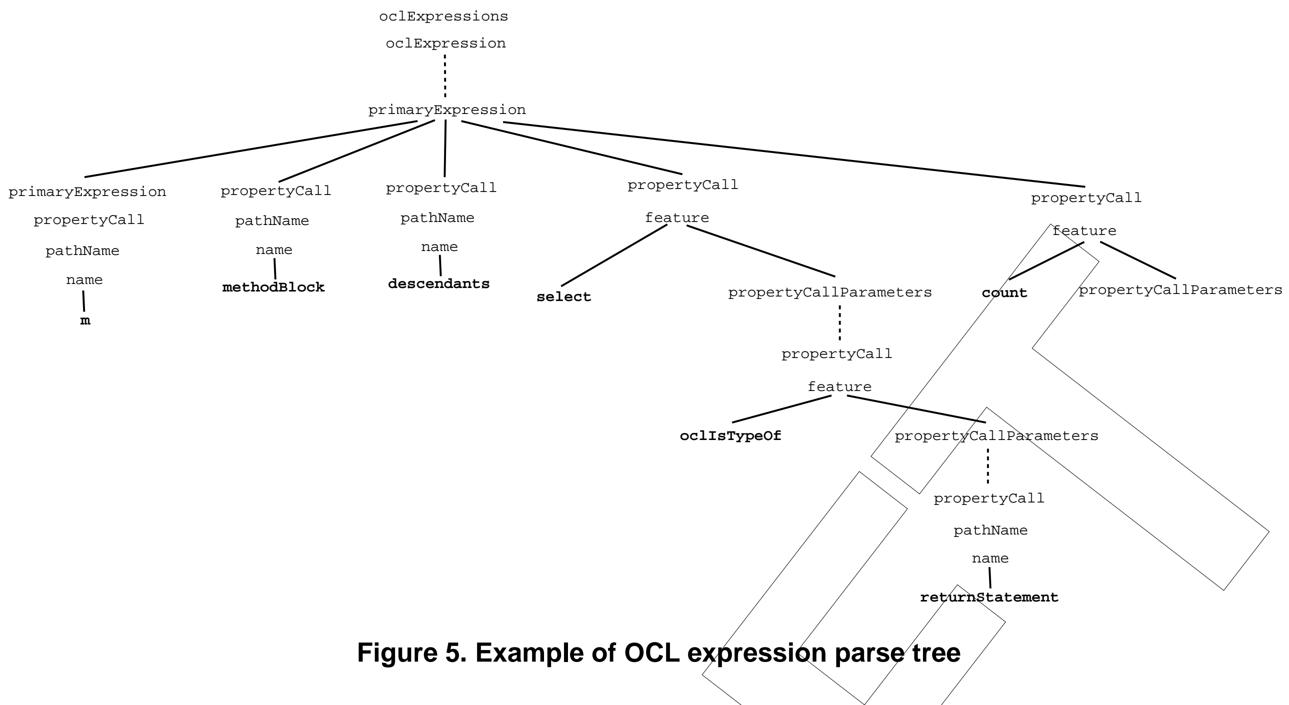

Figure 5. Example of OCL expression parse tree

knowledge of the OCL domain while being as independent as possible from the given target language domain, thus ensuring portability across different languages or programming paradigms. Furthermore, the mapping of structures or, more generally, the variable binding between the OCL domain space and the AST structures must be supported. As already mentioned, we identified in the Java programming language and Java tools (such *JavaCC*) the environment to implement the OCL interpreter. Java allows to write portable code, while providing powerful structures and algorithms via standard packages. Moreover, the reflection package allows to navigate a class structure known at run time; this is particularly useful, as shown below, for implementing the OCL interpreter's navigation over the AST object model. Finally, efficiency concerns can be addressed, if needed, via Java virtual machines implementing an aggressive optimization, such as *jrockit*, or by compiling Java source into the target machine binary code (e.g., via Java compiler such as *gcj*).

There are several classes required to implement an interpreter, the most important class is the `OclInterpreter` class. The method `OclInterpreter::execute` takes as parameters:

- A node from the AST of the target language to be interpreted;
- A string, containing the query written in OCL language; and
- A symbol table.

The method returns an Object that may contain another AST node or a scalar value.

The target language domain is composed by a lexer and a parser. The latter produces, relying on the *jjtree* tool (part of *JavaCC*) the AST (consisting, as shown in Figure 1, in a set of classes) of the source code analyzed.

The default *jjtree* AST structure has been properly modified, in order to provide each node with an attribute implementing the association with all its children.

The OCL domain is composed by:

- An OCL lexer and parser;
- A symbol table; and
- An attribute evaluator that works as an interpreter.

According to what described in [23] the symbol table binds variables to contained values. In our interpreter, to ease the development task, the symbol table stores intermediate and final computation results corresponding to nodes of the OCL expression AST; it must also provide scope mechanism and binding. The binding functionality is also needed to map AST nodes into OCL symbols belonging to the expression to be evaluated. This, in turn, brings back to the need of defining an abstract representation for symbol table entries, corresponding to OCL expressions and subexpressions. Working in Java this is readily available in that the Java `Object` class offers the method `hashCode()`. If two objects are equal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce the same integer result, thus they are mapped, as required, into the same symbol table entry.

The interpreter semantics is implemented by an attribute evaluator: an *eval* method is provided for each node of the OCL grammar. Such function evaluates the node according to:

- The passed symbol table;
- A passed node (that, often, corresponds to the part of the OCL expression already evaluated), and
- The node local values.

Let us suppose we need to count the number of return statements inside a Java method m (see the function ReturnStatement in Figure 4). A simplified version of the OCL expression parse tree is shown in Figure 5. Terminal symbols are bold-faced, while some grammar nodes (in particular, those corresponding to the different precedence levels of the expressions), for sake of simplicity, are not reported.

From an high-level point of view, evaluating such expression means to sequentially evaluate all the subtrees of the node primaryExpression on the top of the tree, except the first (primaryExpression, that simply indicates the access to the node passed as parameter). In each case, an *eval* method is invoked.

The *eval* method of the first propertyCall takes as a parameter the Object returned by the previous invocation. When the terminal node matches to ``methodBlock'', the homonym attribute of the Object m is accessed and returned. This is implemented, as above mentioned, using the Java reflection package. It is worth noting that the terminal node is reached by recursive invocation, from each node, of the *eval* method for all its children.

The second *eval* method takes again an Object as a parameter (m.methodblock) and, when the terminal node matches to ``descendants'', a visitor is executed on the node in order to retrieve all its descendants.

The third *eval* method calls the *eval* of its child node (feature), which has a different behavior depending on the feature actually called. In this case (select), it iterates on all nodes received as parameter, passing each one as parameter to its rightmost subtree, that evaluates a Boolean expression on the node. If the return value is *true*, the node is appended to the Sequence to be returned as a result to the parent node (primaryExpression).

Finally, the result of the select is passed to the *eval* method of the rightmost subtree. Similarly to the previous case, once the feature has been identified as a count, the *eval* returns the number of items contained in the Sequence received as parameter.

Other, more complex, OCL expression can be similarly evaluated. When an OCL feature defines an iterator variable or an accumulator, the scoped symbol table is used to keep track of the intermediate values.

6. Conclusions and Work-in-Progress

We have shown how OCL can be used for navigating and querying ASTs. This gives to the maintainer the following advantages:

1. The language used is not a proprietary language, but it is a *de facto* standard;
2. The grammar can be thought as an object model, thus supporting its visualization using common UML viewers/editors;
3. OCL revealed itself to be particularly effective for the tasks described in the paper; and
4. It is possible to decouple the query tool/language from the target language meta-model, therefore the tool can be easily extended to support new domains/languages.

Work-in-progress is devoted to complete the tool implementation, as well as to implement a viewer allowing to browse an AST using an object diagram.

7. Acknowledgments

Giuliano Antoniol and Massimiliano Di Penta were partially supported by the ASI grant I/R/ 091/00.

References

[1] K. Gallagher and J. Lyle, "Using program slicing in software maintenance," *IEEE Transactions on Software Engineering*, vol. 17, pp. 751–761, August 1991.

[2] F. Tip, "A survey of program slicing techniques," *Journal of Programming Languages*, vol. 3, no. 3, pp. 121–189, 1995.

[3] L. Larsen and M. Harrold, "Slicing object-oriented software," *Proc. of the Int. Conf. on Software Engineering*, pp. 495–505, 1996.

[4] R. S. Arnold and S. A. Bohner, "Impact analysis - towards a framework for comparison," in *Proceedings of IEEE International Conference on Software Maintenance*, (Montreal Quebec Canada), pp. 292–301, 1993.

[5] R. J. Turver and M. Munro, "An early impact analysis technique for software maintenance," *Journal of Software Maintenance - Research and Practice*, vol. 6, no. 1, pp. 35–52, 1994.

DRAFT

[6] K. B. Gallagher, "Visual impact analysis," in *Proceedings of IEEE International Conference on Software Maintenance*, (Monterey), pp. 52–58, 1996.

[7] M. J. Fyson and C. Boldyreff, "Using application understanding to support impact analysis," *Journal of Software Maintenance - Research and Practice*, vol. 10, pp. 93–110, 1998.

[8] L. C. Briand, J. K. Wust, and H. Lounis, "Using coupling for impact analysis in object-oriented systems," in *ISERN*, pp. 1–8, March 1999.

[9] "GrammaTech Codesurfer." `http://www.grammatech.com/products/codesurfer`.

[10] K. Wong, S. Tilley, H. A. Muller, and M. D. Storey, "Structural redocumentation: A case study," *IEEE Software*, pp. 46–54, Jan 1995.

[11] S. Tilley, K. Wong, M. Storey, and H. A. Müller, "Programmable reverse engineering," pp. 501–520, December 1994.

[12] "CodeCrawler Home Page." `http://www.iam.unibe.ch/ lanza/CodeCrawler/codecrawler.html`.

[13] "Semantic Designs Inc. Source Code Browser." `http://www.semdesigns.com`.

[14] I. D. Baxter, "Design maintenance systems," *Communications of the Association for Computing Machinery*, vol. 35, April 1992.

[15] I. D. Baxter and C. Pidgeon, "Software change through design maintenance," in *Proceedings of IEEE International Conference on Software Maintenance*, (Bari, Italy), IEEE Press, September 1997.

[16] J. R. Cordy, C. Halpern, and E. Promislow, "TXL: A rapid prototyping system for programming language dialects," in *International Conference on Computer Languages*, pp. 280–285, IEEE Press, October 1988.

[17] J. R. Cordy, T. R. Dean, A. J. Malton, and K. A. Schneider, "Source transformation in software engineering using the TXL transformation system," *Information and Software Technology*, vol. 44, pp. 827–837, October 1996.

[18] Reasoning Systems, *Refine User's Guide*.

[19] M. Ward, *Proving Program Refinements and Transformations*. PhD thesis, Oxford University, 1989.

[20] M. Ward, "Abstracting a specification from code," *Journal of Software Maintenance - Research and Practice*, vol. 5, pp. 101–122, 1993.

[21] O. M. Group, *Object Constraint Language Specification*. February 2001.

[22] "webGAIN javaCC." `http://www.webgain.com/products/java_cc`.

[23] A. W. Appel, *Modern Compiler Implementation in Java*. Cambrige University Press, 1998.

[24] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley, 1995.

[25] K. Kontogiannis, R. De Mori, R. Bernstein, M. Galler, and E. Merlo, "Pattern matching for clone and concept detection," *Journal of Automated Software Engineering*, March 1996.

[26] J. Mayrand, C. Leblanc, and E. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics," in *Proceedings of IEEE International Conference on Software Maintenance*, (Monterey CA), pp. 244–253, Nov 1996.

[27] B. Lagüe, D. Proulx, E. Merlo, J. Mayrand, and J. Hudepohl, "Assessing the benefits of incorporating function clone detection in a development process.," in *Proceedings of IEEE International Conference on Software Maintenance*, pp. 314–321, 1997.

[28] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees.," in *Proceedings of IEEE International Conference on Software Maintenance*, pp. 368–377, 1998.

[29] T. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. 2, no. 4, 1976.