Generation of an OCL 2.0 Parser

Birgit Demuth
Technische Universität Dresden
53 PUBLICATIONS **426** CITATIONS

Heinrich Hussmann
Ludwig-Maximilians-University o…
175 PUBLICATIONS **1,512** CITATIONS

Generation of an OCL 2.0 Parser

Birgit Demuth[1], Heinrich Hussmann[2], and Ansgar Konermann[3]

[1] Technische Universität Dresden, Department of Computer Science
bd1@inf.tu-dresden.de
[2] Ludwig-Maximilians-Universität München,
Faculty of Mathematics, Computer Science and Statistics
heinrich.hussmann@ifi.lmu.de
[3] Technische Universität Dresden, Department of Computer Science
ansgar.konermann@gmx.de

Abstract. The OCL 2.0 specification defines explicitly a concrete and an abstract syntax. The concrete syntax allows modelers to write down OCL expressions in a textual way. The abstract syntax represents the concepts of OCL using a MOF compliant metamodel. OCL 2.0 implementations should follow this specification. In doing so emphasis is placed on the fact that at the end of the processing a tool should produce the same well-formed instance of the abstract syntax as given in the specification. This offers the possibility to implement OCL-like languages with the same semantics that are for example easier to use for business modelers. Therefore we looked for a parser technique that helps us to generate an OCL parser to a large extent. In this paper we present the technique we developed and proved within the scope of the Dresden OCL Toolkit. The resulting Dresden OCL2 parser is especially characterized by using a generation approach not only based on a context-free grammar but on an attribute grammar to create the required instance of the abstract syntax of an OCL expression.

1 Introduction

The OCL 2.0 specification [1] defines explicitly a concrete and an abstract syntax. The abstract syntax represents the concepts of OCL using a MOF compliant metamodel. In the following, this model is also referred to as *OCL metamodel*. The definition of such a *model-based abstract syntax* is often used for modeling languages [2, 3]. The concrete syntax allows modelers to write down OCL expressions in a textual way. OCL 2.0 implementations should follow the OMG specification. In doing so emphasis is placed on the fact that at the end of the processing of an OCL expression a tool should produce the same well-formed instance of the abstract syntax as given in the specification. This offers the possibility to relatively cost-efficient implement OCL-like languages that are for example easier to understand and write for business modelers. In [4], an example of a *Business Modeling Syntax for OCL* is described. Implementing a parser by hand however is a tedious and error-prone process. Thus, instead of manual implementation, all parts of a parser should be *generated* from one or more

formal specifications. Therefore we developed a parser technique that helps us to generate an OCL parser to a large extent. This includes the generation not only based on a context-free grammar but on an attribute grammar to create the required instance of the abstract syntax of an OCL expression. As a result of this technique an *extended SableCC* version of the well-known parser generator SableCC [5] for lexical and syntactical analyzers has been developed.

Besides the concrete syntax a further subject of variability are the UML and MOF metaclasses that are used in the OCL metamodel. The current OCL 2.0 specification refers to the UML 1.4 metamodel. In future the OCL metamodel has to be aligned with the UML 2.0 or other metamodels. Therefore we extended our generation approach to the overall compiler architecture as explained in [6, 7]. All metamodel classes are implemented by the JMI (JavaTM Metadata Interface [8]) based generation of their Java Interfaces. Metamodels can be incorporated by XMI files.

To the best of our knowledge our approach currently provides maximal flexibility and high productivity in the OCL 2.0 parser construction process. We developed and proved the parser generation approach within the scope of the Dresden OCL Toolkit [9]. Our solution differs from comparable implementations such as Octopus [10] by providing a clean separation of code which *computes* inherited and synthesized attributes, and code which performs tree walking and attribute *passing* to and from nodes. This separation eases implementation of semantic analysis, since the implementor is not required to deal with the tedious and error-prone task of attribute handling. Instead, a clean and elegant API to the attribute evaluator skeleton is provided. In addition, this approach simplifies the development of the generator for the attribute evaluator skeleton. The approach also facilitates maintenance of the attribute evaluator in case of changes or extensions to the OCL language, since large parts of the implementation can be generated from an L-attribute grammar. Due to this separation, our solution is superior to those ones mixing tree walking, attribute handling and attribute computation, as it is often the case with compiler generators employing semantic actions to specify semantic analysis.

In the following, we analyze the current OCL 2.0 specification and point out its major problems with respect to automatic parser construction (Section 2). In Section 3 we propose solutions for them. The main part of the paper (Section 4) explains how we implemented our parser generation approach in the Dresden OCL2 Toolkit (the reengineered Dresden OCL Toolkit). In Section 5 we summarize the results and the experience with the parser construction process. We also give an outlook on further development plans within the scope of the Dresden OCL2 Toolkit.

2 Deficiencies of the OCL 2.0 Concrete Syntax

The concrete syntax of OCL 2.0 exhibits some properties which complicate automatic parser construction. This section names those properties and explains why they make parser construction difficult.

Mixed recognition stages. The concrete syntax specification mixes specification means for all three stages of a parser, that is lexical, the syntactical and the context-sensitive analysis. For example, there is no precise definition of names.[1] Thus, a parser built based on this grammar would not take full advantage of the capabilities provided by the lexical analysis stage.

Instead, valid names are often only recognized by taking context information into account.[2] This also makes syntactical analysis context-sensitive, which is hard to handle with efficient parsing algorithms. Names can easily be recognized using regular languages, so the current approach of the specification easily leads to complex, inefficient parsers. As another example, valid binary operators are defined using disambiguation rules, typically executed during context-sensitive analysis. This prohibits successful analysis of binary expressions during the syntactical analysis stage.

In summary, the specification overstrains syntax analysis while underutilizing lexical analysis and using context-sensitive analysis inefficiently.

No analytic grammar. As above explained, the specification is structured around concepts of the *abstract syntax*. For the vast majority of elements of the abstract syntax, the specification contains one *production*, potentially consisting of more than one *alternative*. Each alternative defines a language making up valid textual representations of the abstract syntax element. These languages often bear little to no syntactic similarity. As an example, consider productions *IteratorExpCS* or *AttributeCallExpCS*. For parsing, it is important to group languages which are syntactically *similar*. This reduces the risk of parsing conflicts and is often the key to rendering efficient deterministic parsing possible.

Ineffective disambiguation. On the other hand, alternatives which *do* bear syntactic similarity are scattered across the specification. For each ambiguous production, a set of disambiguation rules is given, intended to disambiguate the regarding productions. The spatial dispersion of the rules makes it difficult to check whether they really make the grammar unambiguous. Our analysis showed in a similar manner as in [11] that in many cases, they do not. Since the disambiguation rule sets do not impose a defined order of evaluation, each set of rules must separate the language of the regarding production from *all* other productions involved in the ambiguity. An explicit evaluation order would alleviate the situation, since it introduces an additional implicit rule into each rule set except the first, ensuring that all previous rule sets did not match.

No model of input artifacts. The transformation from concrete to abstract syntax is specified in terms of an attribute grammar. Attribute evaluation and syntactic disambiguation usually take place during context-sensitive analysis and are specified on top of the concrete syntax. The concrete syntax tree hence is

[1] of classifiers, attributes etc.

[2] namely, checking whether a name exists in the model

an input artifact for the attribute evaluator. To allow for concise specification of context-sensitive analysis, an explicit model of the concrete syntax is required. The specification does not define an explicit model of the concrete syntax, leaving derivation of the eventually existing implicit model to the user of the specification. This situation is unsatisfactory, as it involves guessing. It is especially true in the presence of inconsistent use of this model. This manifests itself in the specification in different notations for referring to the abstract syntax tree node *ast* or the inherited attribute *env*. Most attribute evaluation rules use *ProductionName.ast* to denote the current AST node, but TupleLiteralExpCS uses *tuplePart*, some attribute evaluation rules also use *ProductionName*. Most rules use *ProductionName.env* to refer to the inherited attribute, but VariableExpCS uses *env.lookup()*.

Not machine-readable. The specification of the concrete syntax is provided by the OMG as a PDF file. Although being *machine-readable* in a narrower sense, the file format can neither be understood by parser generators, nor be easily transformed into an appropriate format automatically. The relevant parts of the specification have to be extracted and converted into the desired input file format *manually*. Older versions of the OCL included a link to a machine-readable grammar of OCL [12] (Chap. 6.9). To alleviate automatic parser generation, future versions of the specification should again include links to machine-readable versions of the concrete syntax.

3 Overcoming the Limitations of the Concrete Syntax

During the development of the OCL 2.0 parser for the Dresden OCL2 Toolkit, we encoutered the problems described in Section 2. This section explains the measures we took to solve or circumvent those problems.

Separate specifications for each transformation stage. From the OCL 2.0 specification, distinct specifications of lexical and syntactical language structure were derived, in essence manually. Both were written in SableCC [5] syntax. This allowed for subsequent testing of completeness and absence of parsing conflicts by simply feeding the specifications to the generator. The disambiguation and attribute evaluation rules were first dropped completely and re-introduced later.

Removal of syntactical ambiguities. The resulting grammar exhibited numerous parsing conflicts, resulting from syntactic ambiguities. It can be proved that no algorithm exists which computes for any given context-free grammar whether it is ambiguous or not ([13], Chap. 9.10). To allow for systematical removal of ambiguities, it is useful to construct the appropriate LR(k) automaton and remove any parsing conflict. If no parsing conflicts exist, the grammar is guaranteed to be unambiguous. We followed this algorithm to iteratively remove conflicts. Whenever a run of the SableCC parser generator revealed parsing conflicts, we modified the grammar by merging ambiguous productions. The

language described by a grammar thus modified is usually larger than the one described by the original grammar. To keep the recognized language identical, all merged productions were noted. During context-sensitive analysis, they need to be differentiated and sentences not allowed by the original grammar need to be sorted out. This quickly led to a grammar partially resembling the OCL 1.x grammar quite closely [12]. Recognizing this, parts of the OCL 1.x grammar and the grammar from our older Dresden OCL Toolkit [14] were used as references during the remaining process of grammar restructuring.

This step resulted in an analytical LALR(1) grammar representing OCL 2.0, available in SableCC syntax and ready for automatic parser generation. It is a firm basis for definition of context-sensitive analysis.

Introducing an explicit model of the concrete syntax. Thanks to the use of SableCC, an explicit model of the concrete syntax comes for free. SableCC is capable of generating an object-oriented framework of classes representing the syntax tree. The transformation from grammar to framework classes is explicitly defined ([15], Chap. 5). Using the API of the framework classes, we can access each node of the syntax tree in a well-defined manner, including navigation to child and parent nodes as well as retrieval of token texts.

Redefining context-sensitive analysis. Having modified the OCL 2.0 grammar heavily, the disambiguation and attribute evaluation rules from the specification did not fit the new grammar any more. A new definition of the context-sensitive analysis stage had to be derived by hand. We stipulated that the resulting attribute grammar should be an L-attribute grammar [16]. This allows attribute evaluation to be performed in a single depth-first, left-to-right tree walk. We divided the context sensitive analysis stage into two alternating substages. The first one performs the walk over the concrete syntax tree, automatically passing attributes up and down in the tree. By default, it automatically creates ASM node instances for each synthesized attribute. It calls hook methods pertaining to the second substage whenever computation of attribute values or disambiguation is required. The specification of the first substage was incorporated into the tailored SableCC grammar, allowing complete generation of implementation code for this substage. The second substage comprises of implementations for the hook methods. These were implemented manually.

4 The OCL Parser of the Dresden OCL2 Toolkit and Its Generation Process

The *Dresden OCL Toolkit* is a well-established software package [9] providing OCL support, either through standalone tools or through libraries which can be integrated into tools by third parties. It has been developed at the Technische Universität Dresden and underwent a reengineering process to accomodate it to the new OCL 2.0 standard, with the new version called *Dresden OCL2 Toolkit.*

This section describes the build process used to create the OCL2 parser of the Dresden OCL2 Toolkit. It sketches important aspects of the parser's architecture, followed by a detailed explanation of the features of both a SableCC extension used to generate an attribute evaluator skeleton and of the resulting parser implementation. A few examples illustrate the features.

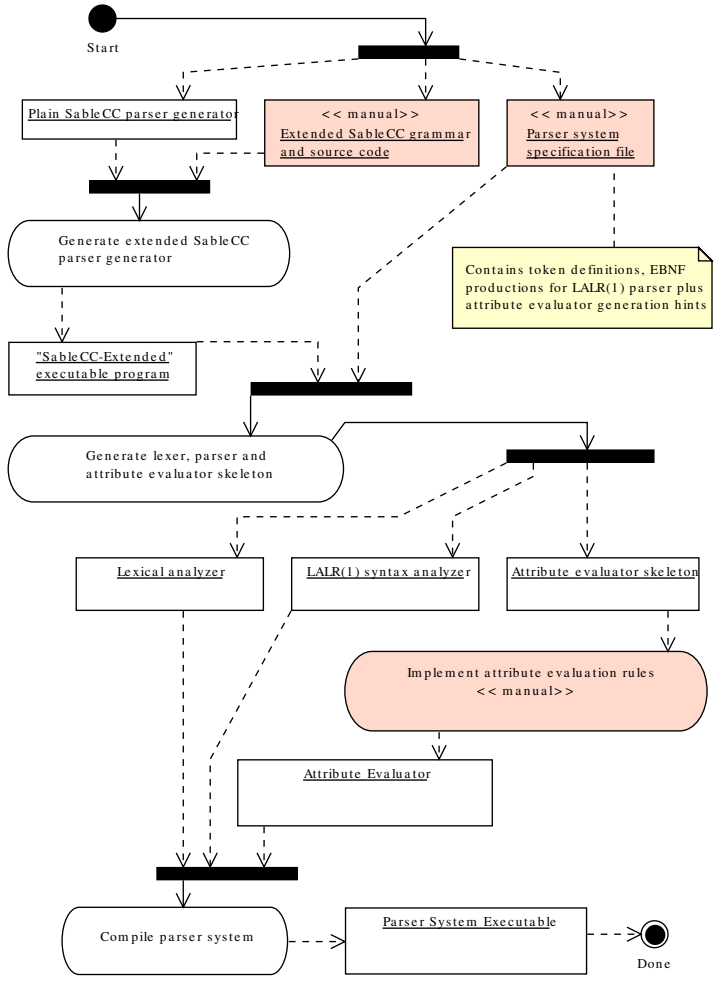

Fig. 1. Generation process used to generate the OCL 2.0 parser

4.1 Generation Process

The process employed to generate the parser is outlined by the activity diagram given in Figure 1. Activities and objects tagged *manual* are performed or created

by hand, the remaining ones automatically. The process requires the following tools and input artifacts:

- an unmodified instance of the open-source compiler generator SableCC, version 2.18.1 [5]
- an extended version of SableCC, consisting of
  - a SableCC grammar for *extended* grammar files
  - enhanced Java source code making SableCC capable of parsing and using extended grammar files
- an OCL 2.0 grammar in extended syntax
- an attribute evaluator implementation for OCL 2.0

Building the parser system involves three major steps. First, an extended version of SableCC has to be created, allowing it to act as a *generator* for the context sensitive analysis stage implementation. During the next step, the extended SableCC is used to generate a lexical and syntactical analyzer for OCL 2.0, plus an attribute evaluator skeleton. This is an abstract base class and has to be implemented according to the OCL 2.0 specification to derive a working attribute evaluator.

4.2 The Parser Architecture

The resulting parser uses two passes to transform the input text into an Abstract Syntax Model (ASM). The ASM represents an instance of the OCL metamodel as defined by the OCL 2.0 specification. During the first pass, it *constructs* the concrete syntax tree (CST). This is performed automatically by the SableCC-generated parser code. In the second pass, an attribute evaluator transforms the CST into an ASM. The attribute evaluator makes use of a modified visitor design pattern [17] called *tree walkers*, as do all of the tree walker classes generated by SableCC ([15], Chap. 6).

The attribute evaluator is designed to successfully perform attribute evaluation for any L-attribute grammar. Thus it performs a single-sweep, depth-first, left-to-right tree walk. All visit methods now take one additional argument, representing the inherited attributes. The synthesized attributes are passed to their parent node as return value.

The parser is integrated into the existing metamodel-based OCL 2.0 compiler architecture of the Dresden OCL2 toolkit [6, 7]. All ASM nodes are created using a variant of an abstract factory [17], which in turn uses the API exposed by the compiler architecture to create instances of appropriate OCL 2.0 metamodel elements.

4.3 Features of the Extended SableCC Generator

Implementing tree walking code for the context-sensitive analysis stage by hand was soon identified as a tedious and error-prone task. This is even more true since tree *walking* alone is not sufficient for an attribute evaluator. It must pass

inherited attribute values into child nodes and collect synthesized attributes produced for child nodes. Besides, it must perform *computation* of attribute values, observing data dependencies between them. Finally, all synthesized attributes must be *stored* for later use, and possibly passed to attribute evaluation code for sibling child nodes.

Lacking a generally available generator for efficient attribute evaluators in Java, we decided to add some simple attribute evaluation features to SableCC 2.18.1. We also considered using SableCC 3.x and its built-in CST-to-AST transformation techniques. It supports transformation of a CST into an AST made up of classes *generated* from appropriate grammar productions. Since we needed to use our own JMI-based ASM classes, we quickly discarded this approach. Besides, SableCC 3.x does not support using context information during transformation from CST to AST. This feature is however elementary for OCL 2.0, since many transformations depend on model information and are thus context-sensitive.

The section motivates the need for a more elaborate tree walking code than currently generated by SableCC. It illustrates the main features of our SableCC extension.

Support for attribute handling. SableCC allows generation of simple tree-walker classes based on the visitor design pattern [15]. They allow calling of custom code at the beginning and end of a visit method for each node, but nowhere in between. This is not sufficient for L-attribute grammars, where inherited attribute values may need to be computed between sibling child nodes. The code does not store the ASM nodes computed during attribute evaluation of child nodes. The only possibility to achieve this using original SableCC are two generic maps, called *in* and *out*. They are intended to hold data items associated with ASM nodes, the ASM node objects acting as map keys. If using this facility, code to store and retrieve objects from these maps, as well as any type casting from java.lang.Object, needs to be written by hand. This is error-prone and should be prevented.

Figure 2 shows the grammar extract we use to describe OCL let expressions. It is written in extended SableCC syntax. Words in angle brackets, exclamation marks and keywords starting with a hash sign (#) are part of the syntax extensions, which can be ignored for now.

Figure 3 shows tree walking code generated by the original SableCC for the production introduced in Figure 2. Custom code can be utilized in line 2 and 13 by overriding methods $\{in|out\}ALetExpCs$. For let expressions however, the OCL 2.0 specification stipulates that all variable declarations must be passed to the attribute evaluation code of the body expression as part of the inherited attribute *env*. This means that somewhere between line 7, where attribute evaluation for the variable declarations occurs, and line 11, where the same occurs for the body expression, custom code to compute the correct value for *env* must be incorporated. Besides, the generated code does allow neither passing inherited attribute values *into* the attribute evaluation method, nor passing synthesized attribute values *back* to the parent node. Please also note that the code descends

```
1  Tokens
2    ! in  = 'in';
3    ! let = 'let';
4  Productions
5    let_exp_cs <LetExp> =
6        let [variables]:initialized_variable_list_cs
7        in [expression]:expression #customheritage
8      ;
```

Fig. 2. Grammar extract describing *let expressions*

into child nodes representing irrelevant syntactic sugar, like the 'let' token. This is a waste of computational resources and should be prevented.

```
1  public void caseALetExpCs(ALetExpCs node) {
2    inALetExpCs(node);
3    if(node.getLet() != null) {
4      node.getLet().apply(this);
5    }
6    if(node.getVariables() != null) {
7      node.getVariables().apply(this);
8    }
9    // ...
10   if(node.getExpression() != null) {
11     node.getExpression().apply(this);
12   }
13   outALetExpCs(node);
14 }
```

Fig. 3. Original SableCC tree walker code traversing a *let expression* node (simplified)

Figure 4 shows the code generated by our extended SableCC. First note that an additional parameter *param* was introduced, which is automatically casted to type *Heritage*. This type is a data container comprising all inherited attributes ever required during the tree walk, including *env*. Attributes not used in a specific context contain null values. The return type of the visit method is now *LetExp*, as specified on line 5 of Figure 2. The ASM nodes of child nodes are returned by their visit methods (lines 8, 17) and *stored* in a variable. This even works for *lists* of CST nodes, which are converted to lists of their ASM nodes.

Computing inherited attributes. Incorporation of custom code for computation of inherited attributes is performed on demand if the corresponding production element is followed by the keyword *#customheritage*, as it is the case for *expression* in Figure 2. The generated code in Figure 4 (lines 14-15) calls an ab-

```
1   public final LetExp caseALetExpCs(ALetExpCs node, Object param) {
2     Heritage nodeHrtg = (Heritage) param;
3     Heritage childHrtg = null;
4
5     PInitializedVariableListCs childVariables = node.getVariables();
6     List astVariables = null;
7     if(childVariables != null) {
8       astVariables = (List) childVariables.apply(this, nodeHrtg.copy());
9     }
10
11    PExpression childExpression = node.getExpression();
12    OclExpression astExpression = null;
13    if(childExpression != null) {
14      childHrtg = insideALetExpCs_computeHeritageFor_Expression(node,
15        childExpression, nodeHrtg.copy(), astVariables);
16      // ...
17      astExpression = (OclExpression) childExpression.apply(this,
18        childHrtg);
19    }
20
21    LetExp myAst = (LetExp) factory.createNode("LetExp");
22    myAst = computeAstFor_ALetExpCs(myAst, nodeHrtg,
23      astVariables,
24      astExpression);
25    return myAst;
26  }
```

Fig. 4. Tree-walker code generated by extended SableCC traversing a *let expression* node (simplified)

stract method *inside<Alternative>_computeHeritageFor_<Node>*. The parameter list of this method has a variable length, determined at generator run-time. It contains the current CST node, the CST node of the child we are about to visit, a copy of the current heritage, and the ASM nodes of all left siblings. The latter is required to fully support L-attribute grammars.

Skipping irrelevant child nodes. The code in Figure 4 does *not* descend into all child nodes given in the grammar production. The exclamation mark in front of a token definition or a production element prevents the attribute evaluator generator to create code for *irrelevant* nodes, such as tokens merely used as syntactic markers (e. g. 'if', 'let', etc). This can save some computational resources.

Creation and computation of ASM nodes. *Creation* of ASM nodes is performed automatically by default, using a factory (Figure 4, line 21). This can be switched off on demand. *Computation* of the ASM node's member values is delegated to an abstract method called *computeAstFor_<AlternativeName>*. This method is basically responsible for computation of synthesized attributes according to the attribute evaluation rules defined in [1]. Again, the parameter list is variable, allowing not only to pass the ASM node to be initialized and the current Heritage, but also the ASM nodes of *all* left sibling nodes. This is required to support L-attribute grammars. Besides, the implementation can take context information into account, obtained as inherited attribute *Heritage*, to perform proper disambiguation.

Automatic node creation can be switched off for each element of a production by appending the keyword *#nocreate*. This will result in a slightly modified signature for the corresponding *createAstFor_Xxx* method. The feature is particularly useful for efficient conversion of recursively defined lists into their equivalent ASM counterparts.

As an example, the relevant grammar extract for context declaration lists is shown in Figure 5. The grammar recursively describes a simple list of context declarations. Figure 6 shows the corresponding generated code. The ASM node type for a context declaration list is a *List* of OclContextDeclaration instances, a type specific to our implementation. It is not defined in the OCL 2.0 abstract syntax.

```
1   context_declaration_list_cs <List> =
2      [context]:context_declaration_cs
3      [tail]:context_declaration_list_cs? #nocreate
4    ;
```

Fig. 5. Grammar extract for context declaration lists

```
1   public final List caseAContextDeclarationListCs(...) {
2     if( childContext != null) {
3       astContext = (OclContextDeclaration) childContext.apply(...);
4     }
5     if( childTail != null) {
6       astTail = (List) childTail.apply(this, nodeHrtg.copy());
7     }
8     List myAst = computeAstFor_AContextDeclarationListCs(nodeHrtg,
9       astContext, astTail);
10    return myAst;
11  }
```

Fig. 6. Generated code for context declaration lists (simplified)

In contrast to Figure 4, where the ASM node is created just before the call to the *computeAstFor* method (line 21), there is no such call in Figure 6 (line 7-8). Instead, the responsibility to create the ASM node is delegated to the *computeAstFor* method (line 8-9). This method will be called for the last context in the input text *first*, since the tree walker performs a depth-first descent and all child nodes are evaluated first, including potential list tails. Thus, the skeleton code allows for the attribute evaluation code of the last context declaration to create a list instance containing the ASM couterpart for the currently processed node. All preceding context declarations can then be added to the head of this list. Figure 7 shows the actual implementation code for this example.

```
1   public List computeAstFor_AContextDeclarationListCs(
2       Heritage nodeHrtgCopy, OclContextDeclaration astContext,
3       List astTail)
4   {
5     List result = null;
6     if ( astTail != null ) {
7       astTail.add(0, astContext);
8       result = astTail;
9     } else {
10      result = new LinkedList();
11      result.add(astContext);
12    }
13    return result;
14  }
```

Fig. 7. Attribute evaluation code for context declaration lists (simplified)

Automatic attribute passing for chain rules. The concrete syntax specification contains numerous productions comprising alternatives of the form $A \rightarrow B$, with A and B being nonterminals. Such rules serve to subsume various syntactical instances of a generic semantic concept under a common production, delegating definition of the actual syntax to subordinate productions. We call these productions *chain rules.* One example in our modified grammar is *literal_exp_cs* (Fig. 8).

The attribute evaluator code can be simplified for this type of productions. It is not necessary to compute an ASM node, if the ASM node type of subordinate, chained alternatives is conforming to the ASM node type of the embracing production. Our attribute evaluator generator supports this simplification through the keyword *#chain* appended to respective alternatives. The ASM node type for the embracing production is *LiteralExp*, which is a supertype of all ASM node types of the chained alternatives. The generator checks type conformance at generator run-time and issues an error message if the types do not match.

```
1  literal_exp_cs <LiteralExp> =
2     {lit_collection}   collection_literal_exp_cs   #chain
3   | {lit_tuple}        tuple_literal_exp_cs        #chain
4   | {lit_primitive}    primitive_literal_exp_cs    #chain
5   ;
```

Fig. 8. Production for literal expressions (simplified)

```
1  public final LiteralExp caseALitCollectionL...(...) throws ... {
2    // ...
3    CollectionLiteralExp astCollectionLiteralExpCs = null;
4    if(childCollectionLiteralExpCs != null) {
5      astCollectionLiteralExpCs = (CollectionLiteralExp)
6        childCollectionLiteralExpCs.apply(...);
7    }
8    LiteralExp myAst = astCollectionLiteralExpCs;
9    return myAst;
10 }
```

Fig. 9. Generated code for chained alternative *lit_collection* of literal_exp_cs (simplified)

Figure 9 shows the generated code for this example. After descending into the (only) child node (lines 4-7), the visit method simply returns the ASM node created for the child (lines 8-9). This completely removes the need to implement ASM node computation manually.

4.4 Features of the Attribute Evaluator Implementation

Some features of the parser belong to the overall implementation and are not limited to the attribute evaluator. This section sketches them.

Balanced syntactical and semantic analysis for leaner implementation. During implementation of the attribute evaluator, the structure of the grammar was further modified to minimize implementation effort. The generator creates one visit method per alternative. Context-sensitive analysis for each alternative is to be performed in the corresponding visit method. In situations where it *is* possible to distinguish similar languages *syntactically*, care must be taken not to overuse this possibility. It can easily lead to a large number of alternatives describing nearly identical languages. By experience we learned that context-sensitive analysis for similar alternatives tends to require similar context-sensitive checks. This would result in sections of the same code in a large number of visit methods. To prevent code-duplication, we tried to balance exploitation of syntactical analysis and redundancy of context-sensitive analysis. Thus, some recognition effort was shifted from syntactical to contextual analysis, reducing the number of visit methods and thereby leading to a slightly leaner implementation.

Support for multiple iterator variables. According to [11], it is hard to implement multiple iterator variables syntactically using the grammar given in [1]. We were able to solve this problem using different syntactic constructs for variable declarations *with* or *without* initializer values, as assumed by [11].

5 Summary

We have learned that to generate an OCL 2.0 parser according to the OMG specification is a challenging task. Experimenting with the OCL 2.0 concrete syntax and changing it, we found a technique that allows to a large extent automated generation of a parser creating the ASM for a given OCL expression. The algorithm is based on a L-attribute grammar and has been implemented as extension of the SableCC parser generator. We plan to prepare this extension as *User Contributed Tool* to the Open Source Community ([5]). We implemented and tested the attribute evaluation based on an L-attribute grammar as part of the Dresden OCL2 Parser. Furthermore, our OCL2 Parser is integrated into the Dresden OCL2 Toolkit architecture. A first use case of the parser can be demonstrated by the OCL22SQL tool that generates SQL code as explained in [18]. We are currently starting a few new projects around the Dresden OCL2 Toolkit. Among other things we will investigate techniques for code generation of procedural/object-oriented (e.g. Java or C#) and declarative (e.g. SQL or XML query languages) code.

Acknowledgment. We would like to thank all people who have contributed over several years to the Dresden OCL Toolkit project. The project has been initiated in 1999. In the following years many students accounted both with research ideas and implementations to the Dresden OCL Toolkit and made their modules available to the open source community. Concerning the Dresden OCL2 Toolkit, Stefan Ocke created a solid basis for the new toolkit version by his Dresden OCL2 Repository that manages models and metamodels and has been an important prerequisite for the running Dresden OCL2 Parser.

References

1. Object Management Group: UML 2.0 OCL Specification. (2004) OMG Document ptc/2004-10-14.
2. Greenfield, J., Short, K., Cook, S., Kent, S.: Software Factories. Assembling Applications with Patterns, Models, Frameworks, and Tools. Wiley (2004)
3. Clark, T., Evans, A., Sammut, P., Willans, J.: Applied Metamodelling. A Foundation for Language Driven Development Version 0.1. albini.xactium.com (2005)
4. Warmer, J., Kleppe, A.: The Object Constraint Language Second Edition. Getting Your Models Ready for MDA. Addison-Wesley (2003)
5. Gagnon, E.M.: Sablecc parser generator. (www.sablecc.org)
6. Ocke, S.: Entwurf und Implementation eines metamodellbasierten OCL-Compilers. Master's thesis, Technische Universität Dresden, Department of Computer Science (2003)
7. Loecher, S., Ocke, S.: A Metamodel-Based OCL-Compiler for UML and MOF. Electr. Notes Theor. Comput. Sci. **102** (2004) 43–61
8. JCP: The JavaTM Metadata Interface (JMI) Specification. (www.jcp.org/en/jsr/detail?id=40)
9. Technische Universität Dresden, D.o.C.S.: Dresden OCL Toolkit. (dresden-ocl.sourceforge.net)
10. Klasse: Octopus: OCL Tool for Precise Uml Specifications. (www.klasse.nl/english/research/octopus-intro.html)
11. Akehurst, D.H., Patrascoiu, O.: OCL 2.0 - Implementing the Standard for Multiple Metamodels. Electr. Notes Theor. Comput. Sci. **102** (2004) 21–41
12. Object Management Group: Unified Modeling Language Specification Version 1.4.2. (2004) OMG Document formal/04-07-02, www.omg.org.
13. Grune, D., Jacobs, C.J.H.: Parsing techniques: a practical guide. Ellis Horwood, Upper Saddle River, NJ, USA (1990)
14. Finger, F.: Design and Implementation of a Modular OCL Compiler. Master's thesis, Technische Universität Dresden, Department of Computer Science (2000)
15. Gagnon, E.: SableCC, an Object-Oriented Compiler Framework. Master's thesis, McGill University (1998)
16. Grune, D., Bal, H.E., Jacobs, C.J., Langendoen, K.G.: Modern Compiler Design. Wiley (2000)
17. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1995)
18. Demuth, B., Hussmann, H., Löcher, S.: OCL as a Specification Language for Business Rules in Data Base Applications. In Gogolla, M., Kobryn, C., eds.: UML 2001 - The Unified Modeling Language. 4th International Conference. LNCS 2185, Springer (2001)