The Parser Subsystem of the Dresden OCL2 Toolkit

Design and Implementation

Author: Ansgar Konermann
Author contact: <konermann@itikko.net>

Last modification: 2005-09-14 00:27:57

Revision History

DATE (yyyymmdd)	*Rev. No. (xx.yy)*	*Comment*
20050914	01.00	First public release.

Table Of Contents

1 Introduction

Software development has made a lot of progress since the first advent of formal languages and compilers. Today, not only are compilers used to transform source code into machine instructions, but software engineering already aims at automatically generating complete software systems from system models and metamodels. This goal is the driving force behind the *Model Driven Architecture (MDA)* approach [MDAw], which is fostered by the Object Management Group (OMG) [OMGw].

To describe the models used throughout MDA, the Object Management Group (OMG) has standardized and published the Unified Modeling Language (UML). UML is a graphical notation that allows specification of the structure and behaviour of software systems. It is complemented by the Object Constraint Language (OCL), which is part of the UML. OCL is a textual language that can be used to specify concise constraints over object models.

To benefit from these modeling languages, tool support is essential. Object models have to be created and constraints have to be attached to them. Constraints have to be evaluated, models to be transformed into other, intermediate models, or code.

A toolset to work with UML models containing OCL constraints has been developed and matured over the years at the Institute for Software and Multimedia Technology, Department of Computer Science at the University of Technology Dresden (TUD) [TUDw]. This *Dresden OCL Toolkit* [OCLTKw] consists of a number of libraries and standalone tools, covering a wide range of usage scenarios. Its newest version supports OCL 2.0.

To process models with OCL constraints, it is necessary to obtain textual constraints from user input or from a file and convert it into a tree-like data structure for further processing. This process is commonly called *parsing,* and a program or subsystem performing parsing is referred to as *a parser.*

This document describes the parser subsystem of the Dresden OCL2 Toolkit. In order to prevent confusion, the toolkit is called *Dresden OCL2 Toolkit,* to distinguish it from the older *Dresden OCL Toolkit,* which supports only OCL 1.x.

1.1 Structure of this document

Chapter 2 summarizes essential facts about syntax analysis. Chapter 3 introduces the architecture of the parser subsystem of the Dresden OCL2 Toolkit and explains the *process* used to build the parser. Chapter 4 describes in detail the extensions made to the open-source compiler generator SableCC and explains how these extensions are used to *generate* the desired OCL 2.0 parser to a great extent. Chapter 5 sketches selected details of the parser implementation not fitting elsewhere. Chapter 6 contains a complete reference of the changes made to SableCC's input grammar.

Appendix A contains the GNU Free Documentation License, under which this document is licensed. Appendix B contains one additional sequence diagram which might be useful for future versions of this document.

1.2 Conventions

Throughout this text, a number of conventions are used to provide the reader with additional information or simply make the text more readable.

1. *References to source code* contained in the Dresden OCL2 Toolkit are given as footnotes. The term *$BASE* refers to the base directory where the toolkit is installed.
2. *Cross references* are indicated by a small angle arrow (↯), followed by the item referenced (chapter, page number or key word).

2 Syntax analysis revisited

This document assumes the reader is familiar with the basics of formal languages and syntax analysis as introduced by any decent undergraduate course on this topic. To faciliate understanding the remainder of the text even without this prerequisite, following is a short introduction to the basic terms of syntax analysis and their relationships.

Syntax analysis is the process of transforming a linear representation of text[1] in a formal language into a structured representation of the same text. The structure of the text is *extracted* from its linear representation using a set of rules describing valid constructs in terms of characters and words.

The input to this process is modeled as a stream of characters, called *input stream.* For most formal languages[2], the transformation process comprises three steps:

1. *Lexical analysis*: the *characters* of the input stream are examined and matched against a set of rules to form groups of characters. These groups are called *tokens.* The program performing lexical analysis is called *lexer,* or sometimes *scanner.* The rules used by the lexer are typically given as regular expressions, so the lexer can only recognize regular languages. It produces a *token stream* as its output.
2. *Syntactical analysis:* the *token stream* is matched against another set of rules called *grammar.* Each rule is called a *production.* Each production describes valid tree-like structures in terms of tokens and, recursively, in terms of other productions. This processing step is performed by a program or subroutine called *parser* or *syntax analyzer.* The rules are typically specified in form of a *context-free grammar,* which would imply that only context-free languages can be parsed[3]. This class of languages allows description and efficient analysis of recursively tree-structured texts, so all *nodes* recognized by the parser are interconnected to form the *syntax tree.* It consists of a *intermediate node* for each part of the original text matching a particular *production*, and a *leaf node* for each *token*. The syntax tree can be represented with different levels of detail and optimized for different processing purposes. The *concrete syntax tree (CST)* resembles the text under analysis quite closely. It is used during parsing. An *abstract syntax tree (AST)* is a more terse form of the syntax tree, intended for further processing such as translation. In an AST, information which was only needed to properly recognize a text's structure is dropped.
3. *Context-sensitive analysis:* most real computer languages cannot be specified completely using a context-free grammar. Additional rules are necessary to define the concise borders of the language. These rules are often given as prose text in the language specification and thus have to be implemented manually.

A common technique to formalize context-sensitive analysis and to render automatic code generation for context-sensitive analysis possible is to assign *attributes* to each production of a context-free grammar, making it a so-called *attribute grammar.* An attribute is essentially a (type, name) pair. *Attribute evaluation rules* specify how to compute an attribute's value, possibly using attribute values of other productions. An *attribute evaluator* is the part of a syntax-based computer program which performs the successive computation of attribute values for a given attributed grammar.

There are several algorithms to perform attribute evaluation systematically. The one used here can be characterized as a single-sweep, left-to-right, depth-first tree walk. An *l-attribute grammar* is simply an attribute grammar which has the nice property that *all* it's attribute values can be computed by *this* special form of tree walk.

1 such as a text file or user input from the keyboard
2 more precisely: for languages which have a context-free skeleton and some additional context-sensitive constraints
3 there are also algorithms to parse context-*sensitive* languages, but they are inefficient and thus not useful for most practical applications

This of course puts a restriction on attribute evaluation rules. They may use *other attribute's values* if and only if the desired values have already been computed before, in the same tree walk. Informally speaking, this means that information can only flow *from top of the analyzed text to the bottom.*

Attribute evaluation can be used to transform a CST into an AST. To achieve this, the attribute evaluation rules describe how to build the corresponding AST node from each CST node type.

3 Architecture of the OCL 2.0 parser subsystem

The OCL 2.0 parser for the Dresden OCL2 Toolkit consists of a number of modules which interoperate to create an abstract syntax tree from textual OCL 2.0 constraints.

This chapter first gives a quick introduction to the overall structure of the parser subsystem (3.1). Section 3.2 names the *artifacts* used to create the modules of the parser subsystem and explains the *process* used to build the modules.

3.1 Overall structure

The overall structure of a system using the OCL 2.0 parser is shown in figure 1. An example of such a system is the OCL 2.0 parser GUI, which is part of the Dresden OCL2 Toolkit.

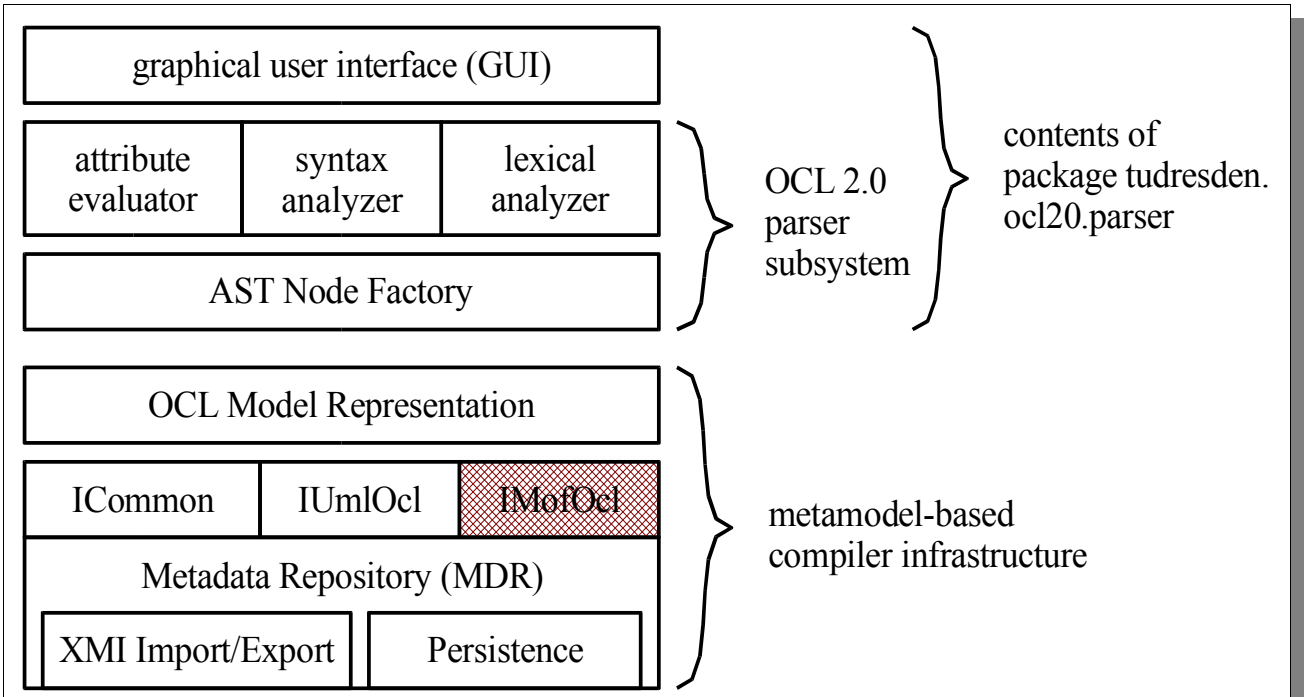

Figure 1: Layered architecture of the OCL 2.0 parser subsystem

The subsystem is build atop a metamodel-based OCL compiler infrastructure, developed by Stefan Ocke as part of his diploma thesis [Ock03]. The subset of the infrastructure relevant to the parser is shown at the bottom of figure 1. Compare this to the complete infrastructure as explained in [Ock03] (Chap. 3, Fig. 3-1). At the core of the infrastructure is a metadata repository as a central storage for metamodels and models. It allows import and export of XMI files, as well as persisting the complete content of the repository, for example in files or relational databases.

For the OCL 2.0 compiler, the three metamodels get installed in the repository during the build process (↳ 3.2). These are:

1. UML 1.5 with OCL 2.0
2. MOF 1.4 with OCL 2.0
3. A common OCL metamodel, describing the abstract syntax of OCL 2.0 and abstracting away the differences between OCL for UML 1.5 and MOF 1.4.

In figure 1, three distinct interfaces IUmlOcl, IMofOcl and ICommon allow access to the repository via three tailored APIs, one for each metamodel. Please note that the box for IMofOcl is hatched. The parser does not support MOF 1.4 models yet, so it does not use the IMofOcl API.

Atop the infrastructure begins the actual parser subsystem. It comprises the following parts:

1. a lexical analyzer *(lexer, scanner)* for OCL 2.0, transforming a character stream containing OCL constraints into a token stream
2. a LALR(1) syntax analyzer *(parser)* for OCL 2.0, transforming the token stream into a concrete syntax tree
3. an attribute evaluator for OCL 2.0, transforming the concrete syntax tree into the abstract syntax tree as defined by the OCL 2.0 specification ([OCL20]).
4. A node factory, faciliating creation of all types of OCL 2.0 AST nodes using a very simple API, namely a single *createNode* method.

Besides, the Dresden OCL2 Toolkit contains the already mentioned GUI-based demonstration program for the parser subsystem.

Please note that the term *parser* is used in two distinct meanings throughout this text. First, it is used to denote a syntax analyzer for a context-free grammar. Second, *parser* in the term *OCL 2.0 parser* is used to name the complete parser *subsystem* of the Dresden OCL2 Toolkit. So the *OCL 2.0 parser* includes a *parser* in the meaning of *syntax analyzer*.

3.2 Parser construction process

This section explains the activities and artifacts required to generate and build the OCL 2.0 parser. See figure 2 for a UML activity diagram illustrating this process.

Activities and objects tagged *«manual»* in Figure 2 are performed or created by hand. The remaining activities and objects are performed or created automatically.

The build process involves the following tools and artifacts:

1. an unmodified version of the open-source compiler generator SableCC, Version 2.18.1 ([SCCw]).
2. an extended version of SableCC, consisting of:
    1. a SableCC grammar for *extended* grammar files
    2. enhanced Java source code making SableCC capable of parsing and using extended grammar files
3. an OCL 2.0 grammar in extended SableCC syntax
4. an attribute evaluator implementation for OCL 2.0 in Java

Building the parser subsystem involves three major steps. First, an extended version of SableCC has to be created. During the next step, this extended SableCC is used to generate a lexical and syntactical analyzer for the OCL 2.0 language *(lexer* and *parser* in figure 2), and an attribute evaluator base class. By implementing the attribute evaluation rules contained in the OCL 2.0 specification [OCL20], a complete attribute evaluator can be derived. The following sections describe these three steps in detail.

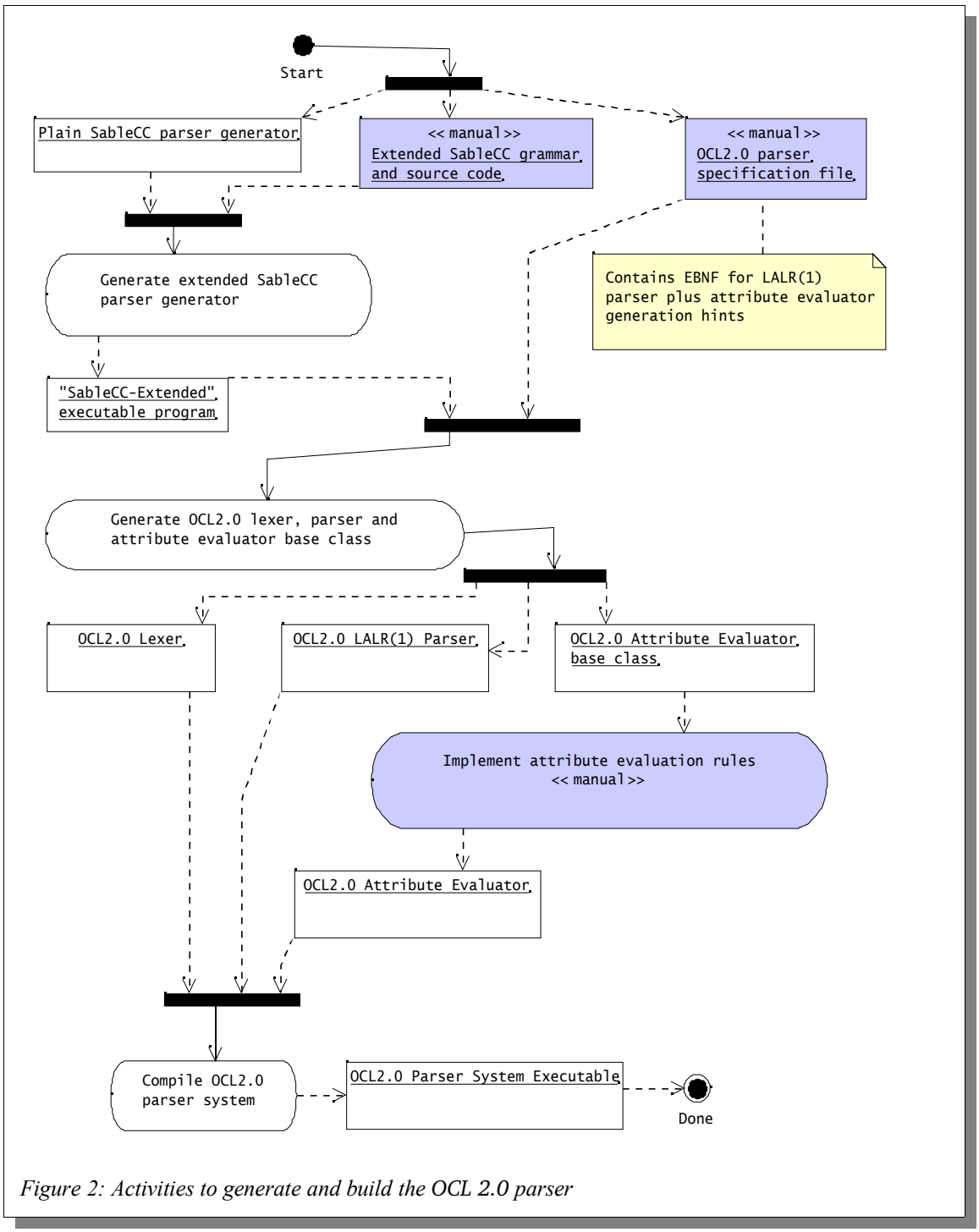

Figure 2: Activities to generate and build the OCL 2.0 parser

3.2.1 Generating an extended SableCC parser generator

One of the main goals during process design was to create most of the OCL 2.0 parser subsystem automatically from formal specifications. SableCC is well suited for generation of lexical and syntactical analyzers, but is not capable of generating evaluators for attribute grammars. Nevertheless, it comes with full source code ([SCCw]), including its own input file grammar. It was thus decided to enhance SableCC by a new code generator for attribute evaluator skeletons. A number of enhancements to SableCC's input file grammar (↳ Chap. 6) now allow specification of *hints* which are used to control generation of an attribute evaluator *skeleton* for the specified language.

In Figure 2, the extended SableCC input file grammar is fed to a plain, unmodified SableCC parser generator. It generates complete lexical and syntactical analyzers for extended SableCC input files. Furthermore, SableCC's source code was enhanced and it now also generates an abstract base class for attribute evaluators. Throughout the remainder of this text, the extended SableCC will be referred to as *SableCC-Ext.*

3.2.2 Generation of OCL 2.0 lexer, parser and attribute evaluator

The OCL 2.0 parser source code contains an OCL 2.0 grammar file[4] in extended SableCC syntax. This file is processed by SableCC-Ext (↳ 3.2.1), which produces:

- An OCL 2.0 lexer[5].
- An OCL 2.0 syntax analyzer[6] (parser), transforming the token stream generated by the lexer into a concrete syntax tree made up of typed nodes[7].
- An abstract skeleton class of the attribute evaluator[8], including JavaDoc comments for all abstract methods. The correspoding Java class is called LattrEvalAdapter, although it does not contain a real *„adapter“* as described by [GoF]. It is rather a „skeleton“. The name was however kept, since all names of abstract tree walker classes generated by SableCC end with „...Adapter“.
- Some other utility and tree walker classes[9]. See [Gag98] for details.

Except for the attribute evaluator skeleton, these artifacts are standard SableCC output files. For a detailed explanation of standard SableCC's output, see [Gag98].

The OCL 2.0 lexer, syntax analyzer and attribute evaluator skeleton are ready to compile. To get a working OCL 2.0 parser respecting all the OCL 2.0 context conditions, an *attribute evaluator implementation* has to be derived.

3.2.3 Deriving the attribute evaluator implementation

To implement a concrete attribute evaluator, a Java class has to be derived from the abstract evaluator skeleton. This class must provide implementations for all abstract methods declared in the skeleton.

Please note that the evaluator skeleton is tailored for the Dresden OCL2 Toolkit for now. It imports a fixed set of java packages of the compiler's metamodel-based infrastructure. This is necessary since the attribute evaluation methods return instances of the OCL 2.0 abstract syntax and some other temporary nodes[10]. The import statements are hard-wired inside the code generation templates[11]. For the prototypical parser described here, this procedure was deemed acceptable. For a more generic skeleton generator, the import statements should be made runtime-configurable.

4 $BASE/resources/grammars/OCL20.xgrammar
5 $BASE/src/tudresden/ocl20/parser/sablecc/lexer/*
6 $BASE/src/tudresden/ocl20/parser/sablecc/parser/*
7 $BASE/src/tudresden/ocl20/parser/sablecc/node/*
8 $BASE/src/tudresden/ocl20/parser/sablecc/analysis/LAttrEvalAdapter.java
9 other files in $BASE/src/tudresden/ocl20/parser/sablecc/analysis/*
10 $BASE/src/tudresden/ocl20/parser/astlib/*
11 $BASE/src/org/sablecc/sablecc/analyses.txt

For details on the meaning of all hook methods, how and when they are generated from the specification file, see Chapter 4 *(„SableCC Extensions for OCL2.0“)*.

4 SableCC Extensions for OCL2.0

This chapter provides detailed documentation of the SableCC extensions. After motivating the need for an extended version of SableCC in Chapter 4.1, an analysis of data dependencies in the attribute evaluation rules of the concrete syntax of OCL 2.0 follows in Chapter 4.2. This analysis allows us to decide which attribute evaluation strategy to use in the attribute evaluator *skeleton*. Finally, all features of the SableCC extensions, including the internals of the evaluator skeleton, are explained in detail and illustrated using examples in Chapter 4.3.

4.1 Motivation for the SableCC extensions

4.1.1 Limitations of SableCC's tree walker classes

The unmodified version of SableCC is capable of generating the following artifacts from a given grammar specification file:

- a lexical analyzer (lexer)
- a syntactical analyzer (parser)
- some simple tree-walker classes

The specification file essentially contains regular expressions to define tokens and context free grammar productions to define the structure of the syntax tree.

While the lexical and syntactical analyzer were found to be well suited for our needs, the tree-walker classes have a design limitation making them unusable as a basis for the attribute evaluator of the OCL 2.0 parser.

For a detailed discussion of SableCC's tree walkers, see chapter 6.4 of [Gag98]. Here, we will only name the most important facts leading to the uselessness of SableCC's tree walkers.

SableCC's simple tree walkers allow to traverse a syntax tree in a depth-first fashion either left-to right or vice versa. Both of these tree walkers contain one *visit method* for each type of syntax tree node. These methods are generated by SableCC and automatically perform the required recursive descent. An exemplified visit method for a syntax tree node representing iterator expressions is shown in Listing 1. Note that no custom code can be inserted or called between the descent into the child nodes, realized by apply() method calls. Only at the very beginning or the end is it possible to call custom code by overriding the methods *outA<NodeType>* and *inA<NodeType>* to perform actual work, instead of just calling the empty default implementations *defaultOut* and *defaultIn*.

For the OCL 2.0 attribute evaluator however, it is necessary to allow insertion of custom code *between* the descent to each child node, as inherited attributes for the child nodes need to be computed. Section 4.2 explains this more detailed.

```
public void defaultIn(Node node) { }
public void defaultOut(Node node) { }

public void inAIteratorArrowPropertyCallExpCs(
        AIteratorArrowPropertyCallExpCs node) {
    defaultIn(node);
}
public void outAIteratorArrowPropertyCallExpCs(
        AIteratorArrowPropertyCallExpCs node) {
    defaultOut(node);
}

public void caseAIteratorArrowPropertyCallExpCs(
        AIteratorArrowPropertyCallExpCs node) {
    inAIteratorArrowPropertyCallExpCs(node);
    if(node.getName() != null) {
        node.getName().apply(this);
    }
    if(node.getParenOpen() != null) {
        node.getParenOpen().apply(this);
    }
    if(node.getIterators() != null) {
        node.getIterators().apply(this);
    }
    if(node.getBody() != null) {
        node.getBody().apply(this);
    }
    if(node.getParenClose() != null) {
        node.getParenClose().apply(this);
    }
    outAIteratorArrowPropertyCallExpCs(node);
}
```

Listing 1: Example of a SableCC-generated visit method for an iterator property call and related utility methods.

4.1.2 No support for attribute handling

```
Object getIn(Node node);
Object getOut(Node node);
void setIn(Node node, Object in);
void setOut(Node node, Object out);
```

Listing 2: methods of SableCC's AnalysisAdapter to access its internal analysis data hash tables.

Besides, the default SableCC tree walkers provide only very limited support for attribute storage and attribute passing between related syntax tree nodes. All data is held in two internal hash tables, called *in* and *out*. They can be accessed using the four methods summarized in Listing 2. All methods use *java.lang.Object* as the type of the parameter representing the data object to store or as their return type. When using these methods to store and retrieve attributes, the implementor of an attribute evaluator would have to *know* the exact type of the data objects an perform explicit type-casting, which is tedious and error-prone.

4.2 Data Dependencies in the Concrete Syntax of OCL 2.0

The OCL 2.0 concrete syntax ([OCL20], Chap. 9) is specified as an attribute grammar using both *synthesized* and *inherited* attributes. Figure 3 illustrates this for a hypothetical production A → B C, where A, B and C are nonterminals, represented by squares. Rectangles attached to a square at the left side represent inherited attributes, flowing *into* a node *from* its parent node. Rectangles attached at the right represent *synthesized* attributes, flowing *out of* the node *to* its parent node. In figure 3,

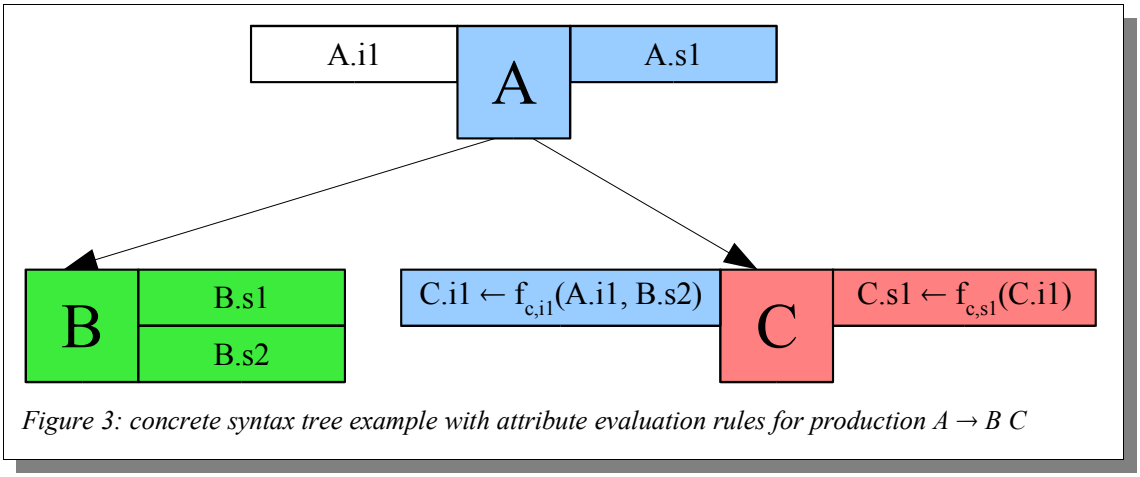

Figure 3: concrete syntax tree example with attribute evaluation rules for production A → B C

nonterminal A has one synthesized and one inherited attribute (A.s1, A.i1), B has two synthesized attributes and no inherited one. C has one inherited attribute and one synthesized attribute. Their evaluation rules are written as functions.

Objects painted in the same background color are specified as one unit of text (paragraph) in [OCL20]. It is important to note that the attribute evaluation rules for the *inherited* attribute C.i1 of nonterminal C are specified as one unit with production *A*, *not* with production C. To prove this fact with a real-world example, have a look at production LetExpCS in [OCL20]: the inherited attributes for LetExp*Sub*CS are defined in the definition paragraph of LetExpCS, *not* LetExpSubCS.

Assuming an L-attribute grammar, we choose an attribute evaluation strategy performing exactly one depth-first tree walk from left to right. This leads to a set of rules that must hold for data dependencies between attribute evaluation rules of a parent node (A) and its children (B, C):

- A's *inherited* attributes may not depend on any of its children's attributes
- C's *inherited* attributes may depend on both the inherited attributes of its parent (A) and any of its left sibling's (B) synthesized attributes.
- B's *inherited* attributes may depend only on the inherited attributes of its parent (A), because B does not have any left siblings.
- C's *synthesized* attributes may depend on any of its left sibling's synthesized attributes, *plus* the inherited attributes of C. Indirectly, they may also depend on the inherited attributes of the parent node A, since these can be passed down into child node C as an inherited attribute, now of C.
- B's *synthesized* attributes may depend on its inherited attributes and, again indirectly, on the inherited attributes of the parent node A.
- A's *synthesized* attributes may depend on *any* synthesized attributes of *any* child nodes (B and C), *plus* the inherited attributes of itself.

Note that it would be sufficient to allow C.s1 to depend on C.i1 and to prohibit any dependeny from synthesized attributes of C's left siblings, since C.i1 can be defined to contain any of B's synthesized attributes if desired. A direct dependency on B's synthesized attributes is not necessary. This is also shown in Figure 3: C.s1 depends on C.i1 only.

This is also the way to go in practice, since it removes the need to know which productions *might appear* as a left sibling when specifying C's synthesized attribute evaluation rules. Conceptually however, there can still be data dependencies between synthesized attributes of a left sibling and the synthesized attributes of the current node.

The rules described above determine the time at which computation of attributes must be performed:

- Values of inherited attributes must be known *before* descending to the respective child node.

- Since inherited attributes might depend on synthesized attributes of any of its left siblings, they cannot be computed until all synthesized attributes of any left siblings are known.
- Synthesized attributes cannot be computed before the last synthesized attribute of the rightmost child node has been computed.

Figure 4 (page 15) shows the required call sequence to visit the partial syntax tree of our example production A → B C. The attribute evaluator implementation first calls the accept method *apply* of the parent node. The parent node object performs a double dispatch[12] by calling the evaluator's *visit method* for nodes of type NodeA, called *caseNodeA* here.

Inside caseNodeA, the evaluator executes generated code to visit each of the child nodes in turn. Before descending into each child node, a hook method is called which performs computation of the inherited attributes for the current child, which are in turn B and C. Subsequently, the synthesized attributes for the parent node A are computed. In the special case of the attribute evaluator for the OCL 2.0 parser, only a single synthesized attribute is used. This is the AST node for the concrete syntax tree node currently processed (A).

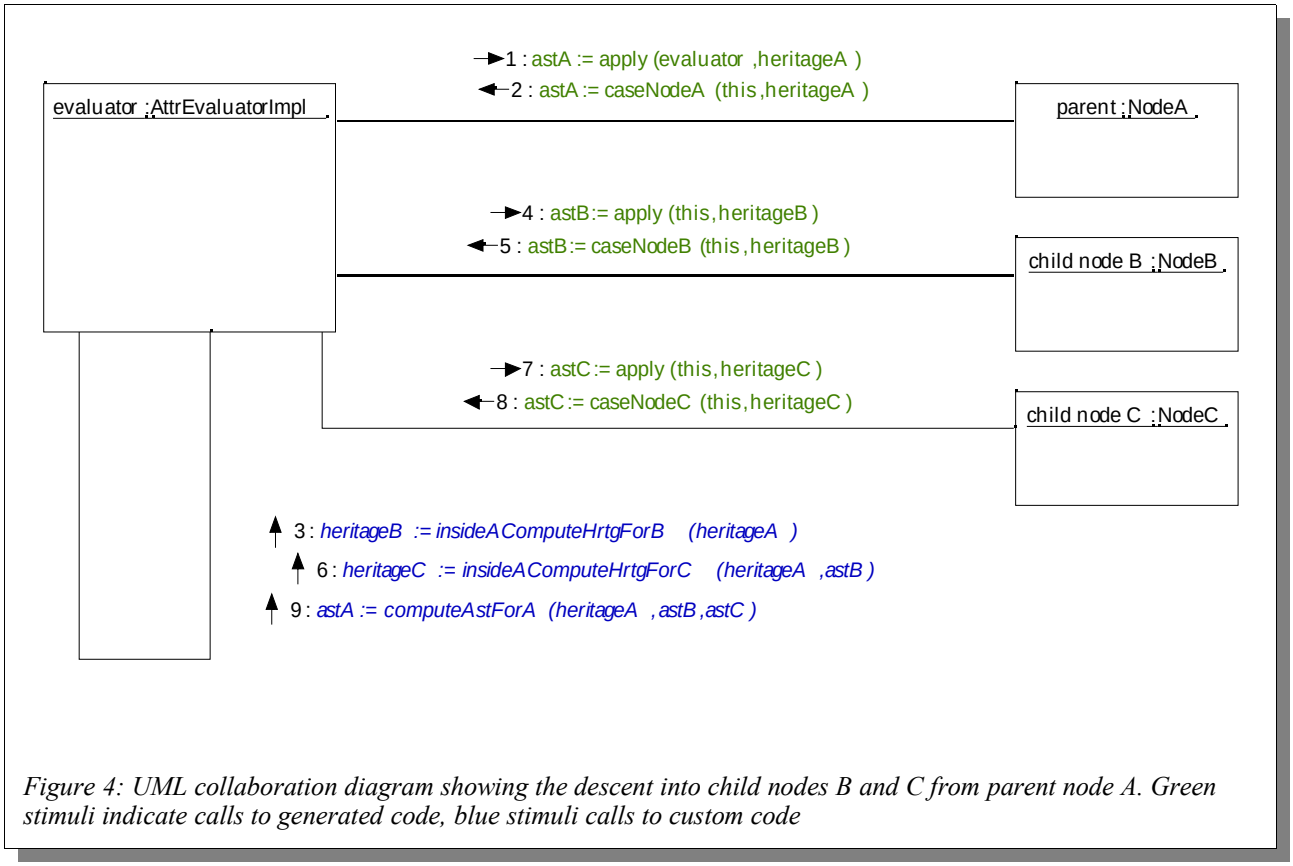

Figure 4: UML collaboration diagram showing the descent into child nodes B and C from parent node A. Green stimuli indicate calls to generated code, blue stimuli calls to custom code

12 Double dispatch is a mechanism used to implement the visitor pattern [GoF]. A detailed explanation can be found in section 4.3.2.

4.3 Features of SableCC-Ext

The SableCC extensions consist of support for generation of a basic attribute evaluator skeleton, which in turn performs automatic attribute handling and allows to add custom code where needed to fully support L-attribute grammars. This section describes all features of the SableCC extension in detail.

4.3.1 Generator for Attribute Evaluator Skeleton

The most fundamental feature of SableCC-Ext is its ability to *generate an attribute evaluator skeleton* for L-attribute grammars. The resulting skeleton performs a single sweep over the concrete syntax tree in a depth-first, left-to-right manner. The specific properties of the attribute evaluator *skeleton* are the key to efficient implementation of a working attribute evaluator.

The attribute evaluator skeleton is an extension to the existing SableCC tree walker architecture. The original architecture is based on a slightly modified version of the *visitor pattern* [GoF], called *tree walkers,* as described in [Gag98] (Chap. 6). This section describes the *extension* of the existing visitor architecture and how it is used in the Dresden OCL2 Toolkit to perform attribute evaluation for L-attributed grammars.

4.3.2 Extension of SableCC's visitor architecture

At the heart of the extended visitor architecture is a new *visit method* for a new kind of *visitor* (Fig. 5). This new visitor's *visit method* accepts an additional parameter *param* of type java.lang.Object and passes this parameter to the *accept method* of the concrete element. Besides, both the visit and the accept method now return a value. It's type can be specified in the extended grammar definition file, as described in 6.1.3. The accept method simply forwards the return value of the visit method, so the caller of the accept method can then store and use this value.

Figure 5 illustrates the usage of the visitor pattern in the parser. Note that for illustration purposes, only a small number of methods and concrete node classes have been included here. The actual parser contains about 220 concrete elements, one for each alternative or token node of the concrete syntax tree.

AnalysisWithReturn extends the tagging interface *SwitchWithReturn* and defines the *interface* of the concrete visitor. It hence forms the *abstract visitor* and contains the declaration of a *visit method* for each concrete element. Note that the visit methods are named *case<ElementName>* here to keep the naming scheme established by SableCC.

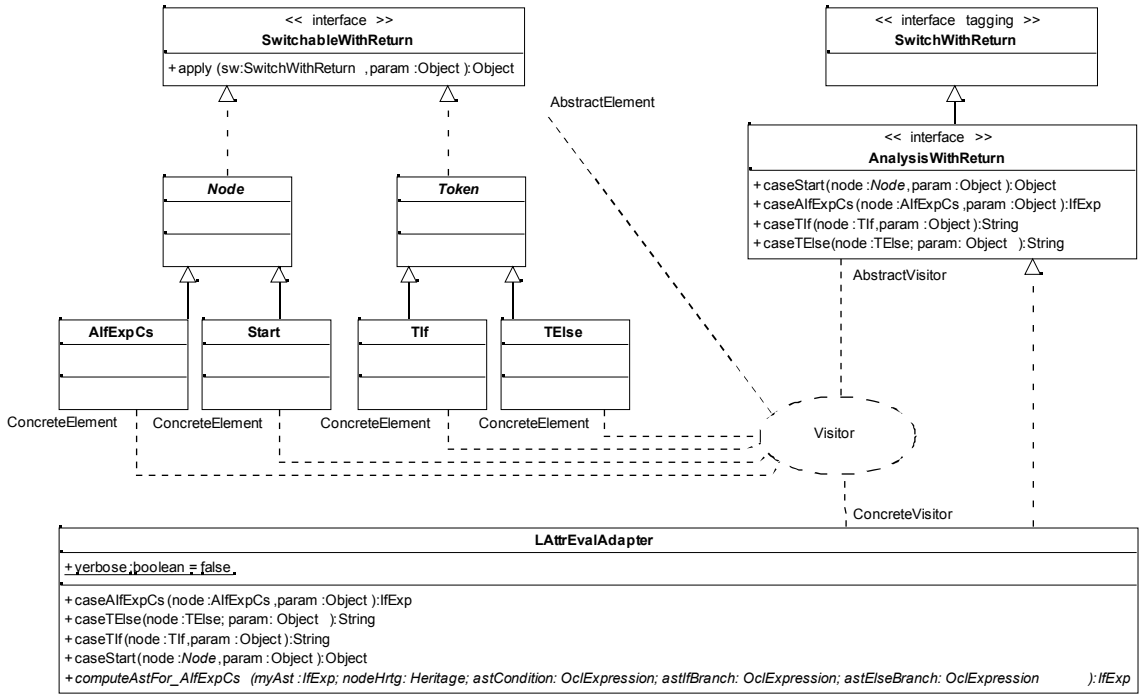

Figure 5: Visitor pattern as used by the extended SableCC syntax tree framework

SwitchableWithReturn represents the *abstract element,* and concrete subclasses form *concrete elements.* All nodes of the concrete syntax tree are *concrete elements.* There is one node for each alternative or token of the concrete syntax.

The element's interfaces have been enhanced and contain a new, overloaded method *appy,* shown in Listing 3.

```
1  public Object apply(SwitchWithReturn sw, Object param)
2          throws AttrEvalException
3  {
4     return ((AnalysisWithReturn) sw)
5          .case<MyNodeName>(this, param);
6  }
```

Listing 3: New apply() method implemented by all concrete elements.

The extended SableCC generates an attribute evaluator skeleton *LAttrEvalAdapter*, which implements interface *AnalysisWithReturn* (Fig. 5). The implementation of all the methods named *case<NodeName>* inside LAttrEvalAdapter form the first half of the double dispatch which is typical for the visitor pattern. Thus, LAttrEvalAdapter is the *concrete visitor* in this setting.

In the accept method (Listing 3), *<MyNodeName>* is the name of the node the method belongs to. For example, for the token node *TElse*, the switch's method *caseTElse* would be called here. The apply method is the second half of the *double dispatch* mechanism. This method is commonly called *accept* in design pattern descriptions.

4.3.3 Attribute Evaluation using the Extended Visitor Architecture

The extended visitor architecture described above can be used to perform attribute evaluation for L-attribute grammars in general, or for OCL 2.0 in particular. This section summarizes how it works.

The visit methods of LAttrEvalAdapter perform a *single recursive descent* into all *relevant* child nodes (↳ 6.1.1.5).

The additional parameter *param* of both the visit and accept methods are used to feed all *inherited attributes* into the concrete syntax tree. We do not model each inherited attribute as a separate data unit, but unite them all in a data container comprising *all* inherited attributes *ever* used in the complete translation process. This class is called *Heritage*[13]. The absence of particular inherited attributes is denoted by *null* values.

The *return value* of the visit and accept methods represent the *synthesized attributes* for the corresponding CST node. Again, we do not model each synthesized attribute separately. Instead, we use either use exactly *one* synthesized attribute or, where more than one is required, *one* container class comprising *all* synthesized attributes for the type of node. Note the similarity to modelling inherited attributes.

The type of return values can be specified in the extended grammar file. Thus we can instruct our extended SableCC to generate accept/visit methods which return instances of the OCL 2.0 abstract syntax classes.

The code generator of the extended SableCC generates complete visit methods named *case<NodeName>* for *all* node types of the concrete syntax tree. Whenever *custom code* is necessary to compute inherited or synthesized attributes, it generates a call to an abstract method performing the concrete computation. Thus, the generated method *case<NodeName>* is a *template method* [GoF], and the custom code methods are *hook methods.*

4.3.4 Features of the Attribute Evaluator Skeleton

As explained in Section 4.3.2, the attribute evaluator skeleton *LAttrEvalAdapter* implements interface *AnalysisWithReturn.* This section describes which code is generated for LAttrEvalAdapter, and its features with respect to attribute evaluation.

The attribute evaluator skeleton supports the following features:

Automatic attribute handling

For each *alternative* of the concrete syntax, a return type must be defined either explicitly using a special syntax or implicitly using generator defaults. It defines the type of the (one) synthesized attribute making up the abstract syntax tree node for the alternative. Attributes are passed up and down the abstract syntax tree, both for inherited and synthesized attributes. This removes the need for writing attribute passing code manually, which is tedious and error-prone.

Support for CST to AST transformation

Tokens in the CST are *automatically* converted to AST nodes of type java.lang.String by default. This behaviour can be customized by specifying a different return type. Each *alternative* of a (syntactical) production *must* be converted to their AST counterparts *by custom code*, by default to an instance of java.lang.Object. More specialized types are desirable to allow for better type checking at generator or compiler run-time.

13 $BASE/src/tudresden/ocl20/parser/astgen/Heritage.java

Automatic AST node instance creation

The generator skeleton uses a simple node *factory*[14] [GoF] to create AST nodes automatically. The evaluator implementor just has to fill all fields of the node („attributes“ in UML nomenclature) with the desired values. He does not have to cope with node instance creation. This feature can be disabled on a per-alternative basis.

To support this feature set, the generator creates an abstract attribute evaluator *skeleton* named LAttrEvalAdapter, which consists of four kinds of methods:

1. Concrete visit methods. One method named *case<NodeType>* for each production alternative and for each token type is created. For production alternatives, <NodeType> is the alternative node name, starting with „A...“. For tokens, <NodeType> is the token type name, starting with „T...“.
2. Abstract methods to compute the AST representation of production alternatives and tokens. Conceptually, the AST representation is a (complex) synthesized attribute of the respective production or token.
   For alternatives, these methods are named *computeAstFor_<AlternativeName>*. If desired, tokens can also be converted into AST nodes by *custom code*. The corresponding methods are named *createNodeFor_<TokenName>*.
3. Abstract methods to compute inherited attributes for child elements of a production alternative which require them. They are called *inside<AltName>_computeHeritageFor_<ElementName>*.
4. A fixed set of utility methods.

All abstract methods serve as *custom code hooks*. To facilitate implementation, a JavaDoc comment describing the methods parameters, return type and responsibility is generated for each of these methods.

The following sections describe the visit methods for productions and tokens in more detail, in conjunction with the abstract methods to compute synthesized and inherited attributes. To illustrate how these methods work, we will use *real code* from the toolkit, generated for an excerpt of the OCL 2.0 grammar. The said grammar extract is shown in Listing 4.

14 The skeleton uses a simplified version of the *abstract factory* pattern. There is no abstract factory used here, instead only one default implementation is provided. In later versions of this software, the factory interface should be defined explicitly as a Java interface.

```
1   Tokens
2
3   ! in  = 'in';
4   ! let = 'let';
5   ! bag                   = 'Bag';
6   ! collection            = 'Collection';
7   ! ordered_set           = 'OrderedSet';
8   ! sequence              = 'Sequence';
9   ! set                   = 'Set';
10
11  Productions
12
13    let_exp_cs <LetExp> =
14        let [variables]:initialized_variable_list_cs
15        in [expression]:expression #customheritage
16      ;
17
18    collection_type_identifier_cs <CollectionKind> =
19       {set} set #nocreate
20     | {bag} bag #nocreate
21     | {sequence} sequence #nocreate
22     | {collection} collection #nocreate
23     | {ordered_set} ordered_set #nocreate
24     ;
25
26    arrow_property_call_exp_cs <PropertyCallExp> =
27        {iterate} <IterateExp> T.iterate paren_open ...
28                  paren_close
29      | {implicit_collection} <OperationCallExp>
30                  [name]:simple_name paren_open ...
31                  paren_close
32      | {iterator} <IteratorExp>
33                  [name]:iterator_name_cs paren_open ...
34                  paren_close
35      ;
36
37    literal_exp_cs <LiteralExp> =
38       {lit_collection}   collection_literal_exp_cs   #chain
39     | {lit_tuple}        tuple_literal_exp_cs        #chain
40     | {lit_primitive}    primitive_literal_exp_cs    #chain
41     ;
```

Listing 4: Excerpt from OCL 2.0 grammar, in SableCC-Ext syntax.

Words in angle brackets are type names, defining the type of the AST node which is to be created when converting a production or token of the CST into its respective AST representation. Exclamation marks indicate that the following token or production element should never be visited during attribute evaluation. This allows to skip „syntactic sugar“ during attribute evaluation. Keywords starting with a hash sign (#) represent special generator control instructions. They are used during code generation to select among different code templates, specialized for various purposes.

4.3.4.1 Overall Structure of Visit Methods for Alternatives

For each alternative of each production contained in the grammar, the generator creates one visit method. As an example, consider production *arrow_property_call_exp_cs* from Listing 4. The generator creates three methods for this production, their signatures are shown in Listing 6. Notice that the alternative node type name is formed by prepending the production name with the alternative name and a capital 'A', and then capitalizing all characters which follow an underscore.

To show the code inside the visit methods, let's have a look at a simpler production, namely *let_exp_cs* (see Listing 4). The complete code generated for this production is given in Listing 5. Each visit method obtains the CST node instance currently visited as its actual parameter *node,* plus one additional parameter *param* (line 1). This parameter is type-casted to *Heritage*[15]*,* a class containing all inherited attributes ever used within the attribute evaluator (line 2). The return type of each visit method corresponds to the *AST node type* defined between angle brackets in the grammar (line 13 of Listing 4). The return value represents the *synthesized attributes* computed for this type of CST nodes (ALetExpCs).

Lines 5 to 22 of Listing 5 contain the code for recursive descent into each *relevant* child node. A child element can be flagged „irrelevant“ in the specification file by prepending it with an exclamation mark '!' (↳ 6.1.1.5). The evaluator skeleton won't visit children flagged in this way.

For each descent, the following code fractions are generated (line numbers refer to the descent into the first child node):

1. Get the child's CST node from the current node (line 5).
2. Declare a variable to hold the AST node(s) corresponding to the CST child node (line 6).
3. Check whether the child's CST node is not null. If so, the child was syntactically optional and must not be visited (line 7).
4. Output some verbose debug messages if a private flag *verbose* is set to true (lines 8 and 10).
5. Call the child's visit method via *double dispatch* mechanism and store resulting AST node in the prepared variable from step 2 (line 9). The child node's inherited attributes are made up of a shallow copy of the current node's inherited attributes.

After having visited all child nodes, the skeleton by default automatically creates an AST node representing the CST node currently being transformed (line 24). This behaviour can be turned off by postfixing the production with the generator control instruction *'#nocreate'*. The newly created, but otherwise uninitialized AST node is passed to a custom code hook, made up of the abstract method *computeAstFor_<NodeType>*. <NodeType> again is the node type name of the current node, *ALetExpCs* in the current example. By default, the parameter list of this method is made up of:

15 $BASE/src/tudresden/ocl20/astgen/Heritage.java

```
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           if (verbose) System.out.println( /* debug output */ );
9           astVariables = (List) childVariables.apply(this, nodeHrtg.copy());
10          if (verbose) System.out.println( /* debug output */ );
11      }
12      PExpression childExpression = node.getExpression();
13      OclExpression astExpression = null;
14      if( childExpression != null) {
15          childHrtg = insideALetExpCs_computeHeritageFor_Expression(
                node, childExpression, nodeHrtg.copy(), astVariables);
16          if ( childHrtg == null ) {
17              childHrtg = nodeHrtg.copy();
18          }
19          if (verbose) System.out.println( /* debug output */ );
20          astExpression = (OclExpression) childExpression.apply(this,
                childHrtg);
21          if (verbose) System.out.println( /* debug output */ );
22      }
23      // create AST node for current CST node here.
24      LetExp myAst = (LetExp) factory.createNode("LetExp");
25      myAst = computeAstFor_ALetExpCs(myAst, nodeHrtg, astVariables,
            astExpression);
26      return myAst;
27  }
```

Listing 5: Code generated for production let_exp_cs

1. The newly created AST node
2. The inherited attributes passed to this child node as parameter *param.*
3. AST representations of all child nodes (astVariables, astExpression), unless flagged as 'irrelevant' using an exclamation mark (↳ 6.1.1.5).

The *computeAstFor* method is responsible for computing the member values of the AST node. An implementation of the attribute evaluator must define this method according to the attribute evaluation rules for *synthesized attributes.*

Finally, the AST node for the transformed node is returned to the caller (line 26).

4.3.4.2 Tailoring Visit Methods for Alternatives

The descent described above is the most basic form of descent supported by the skeleton. It can however be be customized using generator control instructions. There are four control instructions available for visit methods pertaining to alternatives: #nocreate, #customheritage, #chain and #maketree. For documentation of the #maketree directive, see Section 4.3.4.3. The other directives are documented here.

#nocreate prevents the skeleton from creating an AST node for the currently processed CST node. Instead, responsibility to create an AST node is delegated to the *computeAstFor* method. Its signature is shortened by omitting the first parameter *myAst.* As an example, consider Listing 6, which shows one visit method for production *collection_type_identifier_cs* from Listing 4.

```
public final CollectionKind caseASetCollectionTypeIdentifierCs(
        ASetCollectionTypeIdentifierCs node, Object param) ... {
    // ...
    // create AST node for current CST node here.
    CollectionKind myAst =
        computeAstFor_ASetCollectionTypeIdentifierCs(nodeHrtg);
    return myAst;
}
```

Listing 6: Visit method for alternative 'set' of production collection_type_identifier_cs

#customheritage affects the *production element* preceeding the directive in the EBNF term. This instruction indicates that the child node recognized by the element should be visited with inherited attributes *computed by custom code.* As an example, consider the control instruction on line 15 of Listing 4 and the corresponding code on lines 15-19 of Listing 5. On line 15, a call to an abstract method is made to compute the inherited attributes for the next child to descend into, thus providing a hook to incorporate custom code to perform this computation. The method is called:

inside<NodeType>_computeHeritageFor_<ElementName>

<ElementName> is the SableCC-internal name of the EBNF term element the #customheritage instruction was appended to, *'expression'* in this case. The parameter list of this method is variable. It is made up of:

1. A reference to the currently processed CST node (parameter *node).*
2. A reference to the CST node representing the child for which inherited attributes are to be computed (parameter *child<ElementName>).*
3. A copy of the inherited attributes which were inherited by to the currently processed CST node. This allows to merely *modify* the heritage, instead of completely redefining it.
4. References to AST nodes of all left siblings of the respective child node, unless flagged as irrelevant using an exclamation mark (↳ 6.1.1.5).

The computeHeritageFor method must return an instance of *Heritage* or null. If null is returned, the skeleton code uses a copy of the current node's heritage and uses it as the child's heritage (see lines 16-18 of Listing 5).

#chain is used to simplify passing of synthesized attributes for so-called *chain rules.* Chain rules are productions that have the form A → B | B' | B'' | ..., with A and B, B', ... being nonterminals. These alternatives serve to subsume different syntactic instances (B, B', B'', ...) of the same semantic concept under a common name ('A' in this example). They delegate the definition of the concrete textual form of the syntactic instances to subordinate rules. Often, the subordinate rules (B, B', B'', ...) can be transformed into an AST node whose type is a subtype (subclass) of the AST node type of the enclosing production (A). The generator checks whether the AST type of all subordinate alternatives correspond to the AST type of the enclosing production at run-time[16].

As an example, consider production *literal_exp_cs* on lines 37-41 of Listing 4. The corresponding skeleton code of one of the alternatives code is shown in Listing 7.

The descent into the child node is as usual (lines 3-9). But on line 11, no call to a computeAstFor method is generated. Instead, the AST node for the current node is assumed to be the AST node of the (one) child node.

16 To accomplish this, it uses a hard-wired, static type map located at $BASE/src/org/sablecc/sablecc/TypeMap.java. This approach was deemed acceptable for the prototypical implementation of the parser. Future versions should aim to remove this static map in favour of a dynamically generated one.

```
1   public final LiteralExp caseALitCollectionLiteralExpCs(
            ALitCollectionLiteralExpCs node, Object param) ... {
2       // ...
3       PCollectionLiteralExpCs childCollectionLiteralExpCs =
            node.getCollectionLiteralExpCs();
4       CollectionLiteralExp astCollectionLiteralExpCs = null;
5       if( childCollectionLiteralExpCs != null) {
6           if (verbose) System.out.println( /* debug output */ );
7           astCollectionLiteralExpCs = (CollectionLiteralExp)
                childCollectionLiteralExpCs.apply(this, nodeHrtg.copy());
8           if (verbose) System.out.println( /* debug output */ );
9       }
10      // create AST node for current CST node here.
11      LiteralExp myAst = astCollectionLiteralExpCs;
12      return myAst;
13  }
```

Listing 7: Skeleton code for alternative lit_collection of production literal_exp_cs

Please note that *only one* child node is present. This is a prerequisite to use the #chain instruction, because otherwise the generator cannot know which child's node it should use as the current CST node's AST representation.

4.3.4.3 Converting Lists of CST nodes

Converting CST nodes into AST nodes also works for lists of CST nodes. They are created by the syntax analyzer whenever an input text matches an EBNF element with a multiplicity specifier '*' or '+'. If an element of an alternative *is* a list construct, the skeleton visits *each* of these nodes. The resulting AST nodes are stored in a list (java.util.List), in the order in which they appeared in the CST.

As an example, consider production *logical_exp_cs* given in Listing 8. Its alternative *binary* contains an element *tail* with a multiplicity specifier '+'. A tail element consists of an operator and an operand. For this production, the generator creates the skeleton code shown in Listing 9.

```
1   logical_exp_cs <OclExpression> =
2       {chain}  <OclExpression>    [operand]:relational_exp_cs #chain
3     | {binary} <OperationCallExp> [operand]:relational_exp_cs
4                                   [tail]:logical_exp_tail_cs+
5     ;
6
7   logical_exp_tail_cs <OclBinaryExpTail> =
8       [operator]:logic_op [operand]:relational_exp_cs
9     ;
```

Listing 8: Production logical_exp_cs

The descent into child *operand* of alternative *binary* is as usual (lines 5-11). For the *list* of CST nodes recognized by *logical_exp_tail_cs+,* the generator creates code which serves the following functions:

1. Create a new java.util.List instance to accomodate the AST nodes which will be created (line 12).
2. Declare a variable to hold the current CST node and the corresponding AST representation (lines 14-15). *OclBinaryExpTail* is an implementation-specific type not defined by the OCL 2.0 specification. It holds an operator name and an OCL expression representing the right-hand operand.
3. Convert the list of CST child nodes into an array, allowing for easier traversal (line 16).
4. Visit each child in the list of CST child nodes (lines 17-23). For each child, perform the following steps:

1. Call the visit method of the child via the *double dispatch* mechanism (line 20).
2. Add the AST node returned by the apply method at the end of the list of AST nodes (line 22).

The rest of the code is as usual. Please note that when using *#customheritage* with elements having list

```
1   public final OperationCallExp caseABinaryLogicalExpCs(
            ABinaryLogicalExpCs node, Object param) throws ... {
2       Heritage nodeHrtg = (Heritage) param;
3       Heritage childHrtg = null;
4
5       PRelationalExpCs childOperand = node.getOperand();
6       OclExpression astOperand = null;
7       if( childOperand != null) {
8           if (verbose) ...
9           astOperand = (OclExpression) childOperand.apply(
                this, nodeHrtg.copy());
10          if (verbose) ...
11      }
12      List astListTail = new java.util.LinkedList();
13      {
14          PLogicalExpTailCs childTail = null;
15          OclBinaryExpTail astTail = null;
16          Object temp[] = node.getTail().toArray();
17          for(int i = 0; i < temp.length; i++) {
18              childTail = (PLogicalExpTailCs) temp[i];
19              if (verbose) ....
20              astTail = (OclBinaryExpTail) childTail.apply(this,
                   nodeHrtg.copy());
21              if (verbose) ...
22              astListTail.add(astTail);
23          }
24      }
25      // create AST node for current CST node here.
26      OperationCallExp myAst = (OperationCallExp)
            factory.createNode("OperationCallExp");
27      myAst = computeAstFor_ABinaryLogicalExpCs(myAst, nodeHrtg,
28              astOperand, astListTail);
29      return myAst;
30  }
```

Listing 9: Skeleton code for production logical_exp_cs

multiplicity ('*' or '+'), the *left siblings* of the children's CST nodes inside the list do *not* include *any* of the AST representations of elements in the CST node list and are thus *not* present in the parameter list of the *computeHeritageFor* method. That is, the left siblings are defined to be the siblings whose definition inside the *EBNF term* is *left* to the element that defines the list which is currently being processed.

In this situation, it might be desirable to be able to access the *previous* left sibling (but not the complete list) while processing each list element. This is accomplished by using the #maketree directive.

#maketree is used to simplify transformation of *lists* of nodes in the CST into *trees* of nodes in the AST. If the #maketree directive is appended to a production element, the evaluator delegates responsibility to record and store the *previous* sibling's AST representation to the *createAst*-method of the currently processed child node. This allows the createAst-method to *directly* produce a *tree*-structured *AST* representation from a *list* construct in the *CST*.

This is especially useful for *postfix expressions,* i. e. expressions which can have another expression appended. As shown in Listing 10, a postfix expression is made up of a *leftmost expression,* and a list of *tail elements,* each consisting of a postfix *operator* and an *expression.* Each of the tail expressions carry a certain semantic, e. g. operation or attribute call, depending on the postfix *operator.* In OCL 2.0, this semantic is largely encoded in the corresponding AST node classes for the tail expressions. The only question remaining open is 'To which other expression should this semantic be applied?'

```
postfix_exp_cs <OclExpression> =
      {primary}     [primary]:primary_exp_cs #chain
    | {with_tail}   [leftmost_exp]:primary_exp_cs
                    postfix_exp_tail_cs+ #maketree #nocreate
    ;

primary_exp_cs <OclExpression> = // ...

postfix_exp_tail_cs <OclExpression> =
    {prop}       dot [prop_call]:property_call_exp_cs // ...
  | {arrow_prop} arrow_right [tail]:arrow_property_call_exp_cs // ...
  | {msg} ...    [operator]:msg_operator_cs [signal]:signal_spec_exp_cs ...
  ;
```

Listing 10: Grammar extract defining postfix expressions

To answer this question for each expression inside a postfix tail, the AST node classes for these constructs contain a reference to a *source* or *target expression.* If we can manage to set the *source* attribute of each of these expressions to its direct left sibling, we can transform the *list* of expressions into a (degenerated) *tree* of expressions.

In the Dresden OCL2 Toolkit, the parser uses this feature to put the current left sibling into the *Heritage,* so that the *computeAstFor method* of each postfix tail expression can *uniformly* query the left sibling and set the *source* or *target* references appropriately. Listing 11 shows the visit method generated for alternative *with_tail* of production *postfix_exp_cs.* Notice the difference in the parameter lists of the *computeHeritageFor* and *computeAstFor* methods. For computeHeritageFor, an additional parameter *astPreviousPostfixExpTailCs* is passed, containing the previous left sibling (line 21). Before the loop continues with the next list element, the *previous sibling* is switched to be the current one (line 27). The *computeAstFor* method is passed only the *last* element of the expression list. Since this element should contain direct or indirect references to all preceeding expressions, no information is lost.

```
1   public final OclExpression caseAWithTailPostfixExpCs(
2           AWithTailPostfixExpCs node, Object param) throws ... {
3     Heritage nodeHrtg = (Heritage) param;
4     Heritage childHrtg = null;
5
6     PPrimaryExpCs childLeftmostExp = node.getLeftmostExp();
7     OclExpression astLeftmostExp = null;
8     if( childLeftmostExp != null) {
9       astLeftmostExp = (OclExpression) childLeftmostExp.apply(this,
10        nodeHrtg.copy());
11    }
12    OclExpression astTreePostfixExpTailCs = null;
13    {
14      PPostfixExpTailCs childPostfixExpTailCs = null;
15      OclExpression astPreviousPostfixExpTailCs = null;
16      Object temp[] = node.getPostfixExpTailCs().toArray();
17      for(int i = 0; i < temp.length; i++) {
18        childPostfixExpTailCs = (PPostfixExpTailCs) temp[i];
19        childHrtg = insideAWithTailPostfixExpCs_computeHeritageFor_PostfixExpTailCs(
20          node, childPostfixExpTailCs, nodeHrtg.copy(),
21          astPreviousPostfixExpTailCs, astLeftmostExp);
22        if ( childHrtg == null ) {
23          childHrtg = nodeHrtg.copy();
24        }
25        astTreePostfixExpTailCs = (OclExpression)
26            childPostfixExpTailCs.apply(this, childHrtg);
27        astPreviousPostfixExpTailCs = astTreePostfixExpTailCs;
28      }
29    }
30    // create AST node for current CST node here.
31    OclExpression myAst = computeAstFor_AWithTailPostfixExpCs(nodeHrtg,
32      astLeftmostExp,
33      astTreePostfixExpTailCs);
34    return myAst;
35  }
```

Listing 11: Visit method for alternative with_tail of production postfix_exp_cs

Listing 12 shows the actual implementation of the computeHeritageFor method for postfix expression tails. If a previous left sibling was passed to the method, it is used to set the current *left sibling* and *source expression* references inside the *Heritage* object[17]. All *computeAstFor* methods for expression types occurring inside the expression tail can now use these references to perform computation of their synthesized attributes, i. e. their AST node. Some examples are shown in Listing 14 and Listing 13. It can be easily recognized that both methods make use of the current left sibling without having to cope with possibly different left contexts. This is handeled inside the respective *computeHeritageFor* method (Listing 12).

17 It would be sufficient to provide only *one* reference (either *current left sibling XOR source expression).* The unnecessary aliasing of these references must be attributed to the prototypical character of the current parser implementation.

```
1   public Heritage insideAWithTailPostfixExpCs_computeHeritageFor_PostfixExpTailCs(
2           AWithTailPostfixExpCs parent, PPostfixExpTailCs child,
3           Heritage parentHrtgCopy, OclExpression astPreviousSibling,
4           OclExpression astLeftmostExp) throws AttrEvalException {
5       if ( astPreviousSibling != null ) {
6           // we have a previous sibling. set it as source expression
7           parentHrtgCopy.setCurrentSourceExpression(astPreviousSibling);
8           parentHrtgCopy.setCurrentLeftSibling(astPreviousSibling);
9       } else if ( astLeftmostExp != null ) {
10          parentHrtgCopy.setCurrentSourceExpression(astLeftmostExp);
11          parentHrtgCopy.setCurrentLeftSibling(astLeftmostExp);
12      } else {
13          throw new AttrEvalException("Internal error: ... ");
14      }
15      return parentHrtgCopy;
16  }
```

Listing 12: Computing inherited attributes for postfix expression tails

```
1   public IterateExp computeAstFor_AIterateArrowPropertyCallExpCs(
2           IterateExp myAst, Heritage nodeHrtg, String astIterate,
3           List astIterators, VariableDeclaration astAccumulator,
4           OclExpression astBody) throws AttrEvalException {
5       Heritage hrtg = nodeHrtg; // ...
6       OclExpression source = hrtg.getCurrentSourceExpression();
7       assert ( source != null ): "source expression must not be null";
8       Classifier sourceType = obtainType(source);
9       // ...
10      myAst.setSource(source);
11      // ...
12      return myAst;
13  }
```

Listing 13:Computing synthesized attributes for alternative iterate of production arrow_property_call_exp_cs

```
1   public OclMessageExp computeAstFor_AMsgPostfixExpTailCs(
2           Heritage nodeHrtg, OclMessageOperator astOperator,
3           OclSignalSpec astSignal) throws ... {
4
5       OclExpression target = nodeHrtg.getCurrentLeftSibling();
6       assert ( target != null):
7           "target expression (left sibling node) must not be null";
8
9       List params = astSignal.getParameters();
10      String name = astSignal.getName();
11      if ( astOperator == OclMessageOperator.SINGLE_CARET ) {
12          OclMessageExp msgExp = createMessageExp(target, name, params);
13          // ...
14      } else if ( astOperator == OclMessageOperator.DOUBLE_CARET ) {
15          OclMessageExp result = createMessageExp(target, name, params);
16          return result;
17      } // ...
18  }
```

Listing 14: Computation of synthesized attributes for message postfix expressions (alternative 'msg' of production postfix_exp_tail_cs).

4.3.4.4 Visit Methods for Tokens

Tokens in the CST are represented by concrete subclasses of class *Token,* all located in package *tudresden.ocl20.parser.sablecc.node.* When a token node is visited, the generated code by default

```
1   public final String caseTEquals(TEquals node, Object param)
2           throws AttrEvalException {
3       Token curTk = (Token) node;
4       if (verbose) System.out.println( /* debug output */ );
5       this.setCurrentToken(curTk);
6       return node.getText();
7   }
```

Listing 15: Visit method for token '=' (equals).

transforms the token into an instance of *java.lang.String,* containing the token text as value of the string. As shown in Listing 15 for the equals sign, the default implementation merely sets the currently processed token and returns the token text.

It is possible to change the default AST node type for token representations. This works as with production alternatives: simply specify the AST node type in angle brackets after the token name. Listing 16 shows an example from the Dresden OCL2 Toolkit. For the tokens *true* and *false,* the AST node type is declared to be *Boolean.* Whenever the generator encounters a token with a custom AST node type, it generates code to compute the appropriate AST node instance from the token text.

```
1   Tokens
2     false <Boolean>       = 'false';
3     true <Boolean>        = 'true';
```

Listing 16: Grammar extract defining custom AST node types for tokens

For our example, the generated code is shown in Listing 17. Similar to computing AST nodes for alternatives, an abstract hook method named *createNodeFor_<TokenName>* is called. Its parameter list

consists of the current CST node and the current *Heritage*. The implementation of *createNodeFor_TFalse* is trivial, as shown in Listing 18.

```
public final Boolean caseTFalse(TFalse node, Object param) ... {
    Token curTk = (Token) node;
    this.setCurrentToken(curTk);
    Boolean result = createNodeFor_TFalse(node, (Heritage) param);
    return result;
}
public abstract Boolean createNodeFor_TFalse(TFalse node, Heritage nodeHrtg)
    throws AttrEvalException;
```

Listing 17: Generated code to allow for custom code to compute the AST node for token 'false' (simplified).

```
1  public Boolean createNodeFor_TFalse(TFalse node, Heritage nodeHrtgCopy) {
2      return Boolean.FALSE;
3  }
```

Listing 18: Implementation code computing the AST node for token 'false'

4.3.4.5 Utility Methods

The utility methods provided by LAttrEvalAdapter are shown in Listing 19. Method getCurrentToken() can be used to obtain the current token in the parser's input stream. This is useful for error reporting. Method setCurrentToken() should not be called by client code and is shown here only for the sake of completeness. Method hasCurrenToken() returns true if and only if the attribute evaluator knows the current token.

```
1  public Token getCurrentToken() {
2      return currentToken;
3  }
4  public void setCurrentToken(Token tk) {
5      this.currentToken = tk;
6  }
7  public boolean hasCurrentToken() {
8    return (this.currentToken != null);
9  }
10 public void setNodeFactory(NodeFactory f) {
11     this.factory = f;
12 }
```

Listing 19: Utility methods provided by LAttrEvalAdapter

Method setNodeFactory is used to set the *abstract factory* [GoF] implementation which should be used to create all AST nodes. It is mandatory to call this method and provide a node factory instance before automatic creation of AST nodes can occurr (↳ Sect. 4.3.4.1).

In the current implementation of the Dresden OCL2 Toolkit, the interface of the abstract factory is not modeled explicitly. Instead, only one implementation named *NodeFactory*[18] is provided. This factory is tailored to create AST nodes as instances of the JMI-based implementation of the OCL 2.0 abstract syntax. The attribute evaluator skeleton uses method *createNode(String name)* as the factory method, so the interface of the abstract factory in fact only consists of this method. All other methods of class NodeFactory are implementation-specific.

18 $BASE/src/tudresden/ocl20/parser/astgen/NodeFactory.java

5 Selected Details of the Attribute Evaluator Implementation for OCL 2.0

To get a working attribute evaluator implementation, an implementation class has to be derived from the attribute evaluator skeleton by *inheritance*. The derived class must define all abstract methods declared in the skeleton, as documented in Section 4.3.

This chapter sketches some of the details and problems encountered while implementing the attribute evaluator for OCL 2.0 and outlines their solutions where appropriate. The implementation class is called LAttrAstGenerator.

5.1 Abstract Syntax Enhancements

The OCL 2.0 specification [OCL20] defines a concrete syntax for context declarations (Chap. 12.13). However, no *abstract* syntax for context declarations is given. The attribute evaluator implementation for the Dresden OCL2 Toolkit's OCL 2.0 parser includes a library of additional AST node classes, allowing to convert textual context declarations into an abstract syntax tree in their entirety. These classes are contained in package tudresden.ocl20.parser.astlib. The topmost node of an abstract syntax tree generated by the attribute evaluator is a *java.util.List* of *OclPackagedConstraintList* instances. It is possible to traverse all textual context declarations using this list and the accessor methods of OclPackagedConstraintList and (recursively) its members. For even more detailed documentation of these classes, refer to the JavaDoc documentation.

Besides AST node classes for *context declarations*, the package contains some more *intermediate node* classes which are used to represent various kinds of syntax elements during attribute evaluation, but which are never contained in the final AST as defined by [OCL20]. These classes are necessary because the grammar given in the specification is not suitable for parsing and had to be restructured in large parts. The resulting grammar thus contains some elements which have no direct counterpart in the abstract syntax of OCL 2.0, but nevertheless need to be represented during attribute evaluation. Among these classes are:

- OclActualParameterListItem
- OclBinaryExpTail
- OclMessageOperator

For further details, refer to the JavaDoc documentation of the respective classes.

5.2 Model Access and Context Determination

Each OCL constraint must be defined in some *context*. Possible contexts are defined in chapter 12.13 of [OCL20]. For the OCL 2.0 parser contained in the Dresden OCL2 Toolkit, the context is always an element of a *UML* model.[19] It is mandatory to obtain a reference to this context before attribute evaluation can occurr, since many attribute evaluation rules depend on information from this *context*. This section explains how the attribute evaluator determines the context for each textual OCL constraint.

The model itself must be loaded using the interface IUmlOcl of the metamodel-based compiler infrastructure developed by Stefan Ocke ([Ock03]). A reference to this model must be passed to the constructor of the attribute evaluator LAttrAstGenerator.

During attribute evaluation, the evaluator descends into the CST and collects in the *Heritage* object all information required to select the desired context element from the model. This information consists of the current package name and the corresponding package inside the model. At each occurrence of a

19 Future versions of the parser might as well support other metamodels, such as MOF.

context declaration, it is possible to query the model for the required model element. This model element is then put into the *Heritage* object as the *contextual classifier*.

Attribute evaluation code of nodes further down in the subtree can obtain a reference to the contextual classifier through the inherited attributes and use it according to the attribute evaluation rules.

5.3 Limitations of Metamodel Class 'Namespace'

Scope of variables and other identifiers in the concrete syntax of OCL 2.0 is managed using a dedicated class *Environment,* defined by [OCL20] in Chapter 9.4. In conjunction with Environment, [OCL20] defines two more classes which are used in scope and namespace management, namely *Namespace* and *NamedElement.* The support for *Namespace* by the compiler infrastructure is insufficient. This section explains why and sketches the solution used by the OCL 2.0 parser.

Class *Namespace* is defined in UML 1.5 ([UML15], Chap. 2.5.2) and extended in [OCL20] (Chap. 9.4.3) by some specialized methods *getEnvironmentsWithoutParents* and *getEnvironmentWithParents,* converting a namespace into an *Enviromment* instance containing all elements of the namespace. These methods are supported neither by the metamodel-independent interface of the compiler infrastructure (ICommon), nor by the UML1.5 specific interface IUmlOcl. Instead of tampering with the metamodel and the compiler infrastructure, the parser uses a variant of the *decorator* pattern [GoF] to overcome this limitation.

The classes involved are contained in package *tudresden.ocl20.parser.astgen.* A new interface *Namespace* replaces the *Namespace* class from UML or OCL. This interface contains only the conversion methods mentioned above and a reference to a potential parent namespace, as defined in [UML15]. An abstract class *UML15Namespace* implements *Namespace* partially abstract, leaving implementation of the conversion methods to the concrete subclasses *UML15PackageNamespace* and *UML15ClassifierNamespace.* Class *UML15Namespace* also contains a factory method *createDecorator* creating an appropriate subclass instance for *Package* or *Classifier* instances from the UML 1.5 metamodel implementation provided by the compiler infrastructure.

Whenever the attribute evaluation code must convert a namespace into an *Environment* instance, it first wraps the *Namespace* into one of the concrete subclasses of *UML15Namespace,* using the factory method. The resulting subclass instance can then be used to perform the desired conversion.

5.4 Managing Identifier Scope

Names of variables and other identifiers in the concrete syntax of OCL 2.0 are managed using nested *Environment* instances. Class *Environment* is defined in [OCL20] (Chap. 9.4.1). This section explains how this class is implemented in the Dresden OCL2 Toolkit.

There are two main differences between the specification of class Environment and its implementation: support for additional lookup operations and a split read-only/read-write interface.

Additional Lookup Operations

The implementation supports additional lookup operations for implicit model elements which do not only return *either* the implicit model element *or* the source of the implicit model element, but both references in a small data container.

Since queries for an implicit element *plus* its source are commonly used throughout the attribute evaluation code and each of the operations involve costly search operations, computational resources can be cut to half if both values are searched and returned in one step.

The relevant operations added by the implementation are:

- *lookupImplicitAttributeWithSource*
- *lookupImplicitAssociationEndWithSource*

- *lookupImplicitAssociationClassWithSource*
- *lookupImplicitOperationWithSource*

Split Read-Only/Read-Write Interface

The implementation of *Environment* has a split interface separating read-only operations from read-write operations. This prevents implementors of attribute evaluation code from accidentally modifying the *Environment* instance pertaining to a parent or higher-level CST node.
The *Heritage* object of a CST node contains a reference to the read-only interface of the current environment. This interface part supports creation of a *nested* environment using method *nestedEnvironment(),* which returns a *WritebleEnvironment.* The nested environment obtained this way can be modified and stored in the *Heritage* object for child nodes. Since the type of the respective member of *Heritage* is *Environment* and thus read-only, no child can ever accidentally modify this environment.

6 SableCC Grammar File Syntax Enhancements

This section contains a very terse description of the enhancements made to SableCC's grammar syntax, using the terms defined by the SableCC manual to refer to parts of the specification file.

If you would like to understand the enhancements and are not yet familiar with the SableCC grammar specification syntax, read the SableCC documentation [Gag98] *now***.**

On the other hand, you can skip this section if you are not interested in the details of the extended grammar file syntax. Back references in the following chapters will take you back to specific subsections if required.

6.1.1 New Tokens

Some new tokens were introduced to support the syntax enhancements.

6.1.1.1 Token 'chain' (chain rule indicator)

```
chain            = '#chain';
```

Token *chain* is used to indicate that an alternative must contain *exactly one* relevant child and the synthesized attribute of the child should be returned as synthesized attribute of the current alternative.

6.1.1.2 Token 'customheritage' (custom heritage indicator)

```
customheritage   = '#customheritage';
```

Token *customheritage* is used to flag an element of an alternative as requiring custom code to compute the inherited attributes when descending into the corresponding child node of the concrete syntax tree.

6.1.1.3 Token 'maketree'

```
maketree         = '#maketree';
```

Token *maketree* is used to indicate that the attribute evaluator skeleton should create code which supports direct creation of a tree-structured AST for a list of CST nodes, without an intermediate *list* of AST nodes.

6.1.1.4 Token 'nocreate'

```
nocreate         = '#nocreate';
```

Token *nocreate* is used to indicate that the attribute evaluator skeleton should *not* create an instance of the synthesized attribute's type for a production's alternative.

6.1.1.5 Miscellaneous tokens

```
exclam           = '!';
l_abkt           = '<';
r_abkt           = '>';
```

Token *exclam* is used to flag productions and tokens as irrelevant during attribute evaluation. The opposite of irrelevant is *relevant.* Tokens are relevant by default. Irrelevant elements won't be visited during attribute evaluation. This can save some computational resources.

Tokens *l_abkt* and *r_abkt* are used to delimit the AST node type specification for a production or alternative.

6.1.2 New helpers

Helpers are regular expressions which can be used inside token definitions. Two new helpers were introduced.

```
external_name_start    = [uppercase+lowercase];
external_name_char     = [[uppercase+lowercase]+['_'+digit]];
```

External names External names are names of entities external to SableCC. These helpers are used for Java identifiers of the AST types specified for attribute evaluation. In contrast to external names, internal names are names of entities used inside SableCC to generate the parser, such as tokens, productions and alternatives.

6.1.3 Modified productions

This section lists all *modified productions* of the grammar used by the SableCC to generate a parser for grammar files.

6.1.3.1 Production 'token_def' (token definition)

```
token_def = state_list? exclam? id ast_type? equal reg_exp look_ahead?
      semicolon;
```

A token definition may contain an optional exclamation mark and an optional AST type specification.

An exclamation mark is used to flag the token as irrelevant for AST generation. Tokens with this option set will be ignored during AST generation. Their CST nodes will not be visited.

A token having an AST type specification will be converted to a node of the given type during AST generation. The attribute evaluator generator will create an abstract method

```
<AstNodeType> createNodeFor(<CstTokenType> t)
```

which the implementor must implement accordingly. If not specified, the attribute evaluation generator will create a default conversion method which converts tokens to their string representation (i. e., to instances of java.lang.String).

Example:

For a token named 'if', the corresponding CstTokenType would be *TIf* (SableCC convention), and the AstNodeType would be the type specified in the grammar file.

6.1.3.2 Production 'prod' (production)

```
prod = exclam? id ast_type? equal alts semicolon;
```

Similar to a token definition, a production definition may also have an optional exclamation mark and an optional AST type specification.

An *exclamation mark* flags the complete production as irrelevant during AST generation. If the exclamation mark is present, the attribute evaluator skeleton won't descend into the elements of this production's alternatives. This flag is valid for all alternatives of the production, meaning none of the child nodes of any alternative will be visited.

The *ast type specification* is similar to the one for tokens. The conversion method is called

```
createAstFor_<AlternativeName>(...);
```

and by default has return type java.lang.Object. You probably want a more specific AST node type, so specify one for your productions and alternatives. For details on the conversion method, see chapter 4.3.4.1. *<AlternativeName>* is the Java name of the alternative, as generated by SableCC. See chapter 5.3 of [Gag98] for details of this name transformation.

6.1.3.3 Production 'alt' (alternative)

```
alt =   {parsed} alt_name? ast_type? [elems]:elem* chain? nocreate?
      | {ignored} l_par alt_name? [elems]:elem* r_par;
```

Only alternative *'parsed'* (first line) was changed to allow specification of an AST node type on a per-alternative basis. This allows the grammar writer to define AST node types more fine-grained if the language (OCL2.0) specification requires it.

Besides, it is often convenient to specify more specialized AST node types for alternatives, although the production containing the alternative already has a node type specification. It allows to delegate creation of the correct AST node instance to the attribute evaluator skeleton.

The AST type specified for an alternative must conform to the ast type specified for the enclosing production. Conformance means: it must either be the same type, or a subclass of the AST type specified for the production. The generator checks this at generator runtime.

6.1.3.4 Production 'elem' (element of an alternative)

```
elem = elem_name? exclam? specifier? id un_op?
       [custom_heritage]:customheritage? [make_tree]:maketree?;
```

A production's alternatives are made up of *elements*. An element is either a production name or a token name. Each element may be postfixed with a star '*', a plus '+' or a question mark '?' to define the acceptable multiplicity for this element to be 0..inf, 1..inf or 0..1 respectively.

The production defining an element syntactically has been added three new features:

1. Elements may be flagged irrelevant on a per-element basis. This is done by preceeding the element's specifier with an exclamation mark. If you omit the specifier, put the exclamation mark before the element's id. This is useful if you *need* a specific token's or production's CST node transformed to an AST node in *one* alternative, but want to *omit* it in *another one.* In this situation, you cannot simply prefix the token *definition* or production *definition* with an exclamation mark, since this would lead to the attribute evaluator skeleton ignoring the corresponding token or node *everywhere* in your grammar.

   Example:

   The *equals* token is used to recognize a comparison operator in OCL 2.0. We need the textual representation of each comparison operator to construct the AST for a relational expression. So we *must* transform the concrete syntax representation of the *equals* token to its abstract syntax counterpart. We cannot flag out the token completely.
   On the other hand, we do not want the attribute evaluator to descend into the token node during attribute evaluation of an initial value definition for a variable or a definition constraint, since the equals sign is only syntactic sugar here:

   ```
   definition_constraint_cs <OclDefinitionConstraint> =
         [entity]:defined_entity_decl_cs
         !equals [definition]:ocl_expression_cs;
   ```

2. You can specify the need to compute the inherited attributes for this element by custom code during attribute evaluation. Use the flag '#customheritage' to achieve this. By default, each element is passed a copy of the parent node's inherited attributes. For details on custom inherited attributes, see chapter 4.3.4.1.

3. It is possible to have the code generator create code which enables the implementor of the attribute evaluator to convert list constructs in the concrete syntax tree to tree structures in the abstract syntax tree *directly*, without creating a list of AST nodes first and then iterating over this list to construct a tree from it. By default, the attribute evaluator skeleton converts lists of concrete syntax nodes to lists of their abstract syntax counterparts. For details on this direct list-to-tree conversion, see chapter 4.3.4.3.

6.1.4 New productions

6.1.4.1 Production 'ast_type' (AST node type specification)

```
ast_type = l_abkt P.external_name r_abkt;
```

This production defines how an AST type specification must be constructed. Examples of AST type specifications:

```
<OclExpression>
<Boolean>
```

See source code of generator's build-in type map[20] and node factory[21] for supported AST node types.

6.1.4.2 Production 'external_name' (identifiers external to SableCC)

```
external_name =
            {simple} T.id
          | {extended} T.external_name
          | ({generic} T.id ) ;
```

The lexers generated by SableCC use a *first match* strategy when resolving ambiguities in the charcter stream. If multiple token definitions match a given input character sequence, the lexer uses the definition which occurs first in the language specification file.

Since internal and external names (T.id and T.internal_name, respectively) in fact *are* ambiguous, T.external_name will be *only* recognized if the token text *does not match* T.id. P.external_name is used to unite both sublanguages.

You are not required to understand this paragraph unless you want to enhance the SableCC extensions further and modify the language allowed for external names.

20 $BASE/src/org/sablecc/sablecc/TypeMap.java
21 $BASE/src/tudresden/ocl20/parser/astgen/NodeFactory.java

Appendix A: GNU Free Documentation License

Version 1.2, November 2002

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (c) YEAR YOUR NAME.
> Permission is granted to copy, distribute and/or modify this document
> under the terms of the GNU Free Documentation License, Version 1.2
> or any later version published by the Free Software Foundation;
> with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
> A copy of the license is included in the section entitled "GNU
> Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the
> Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Appendix B: Additional Recursive Descent Example

This appendix contains a UML sequence diagram showing the recursive descent performed by the attribute evaluator skeleton for if-expressions (production if_exp_cs). It is included here to provide additional material for future maintainers of this document or derived documentation.

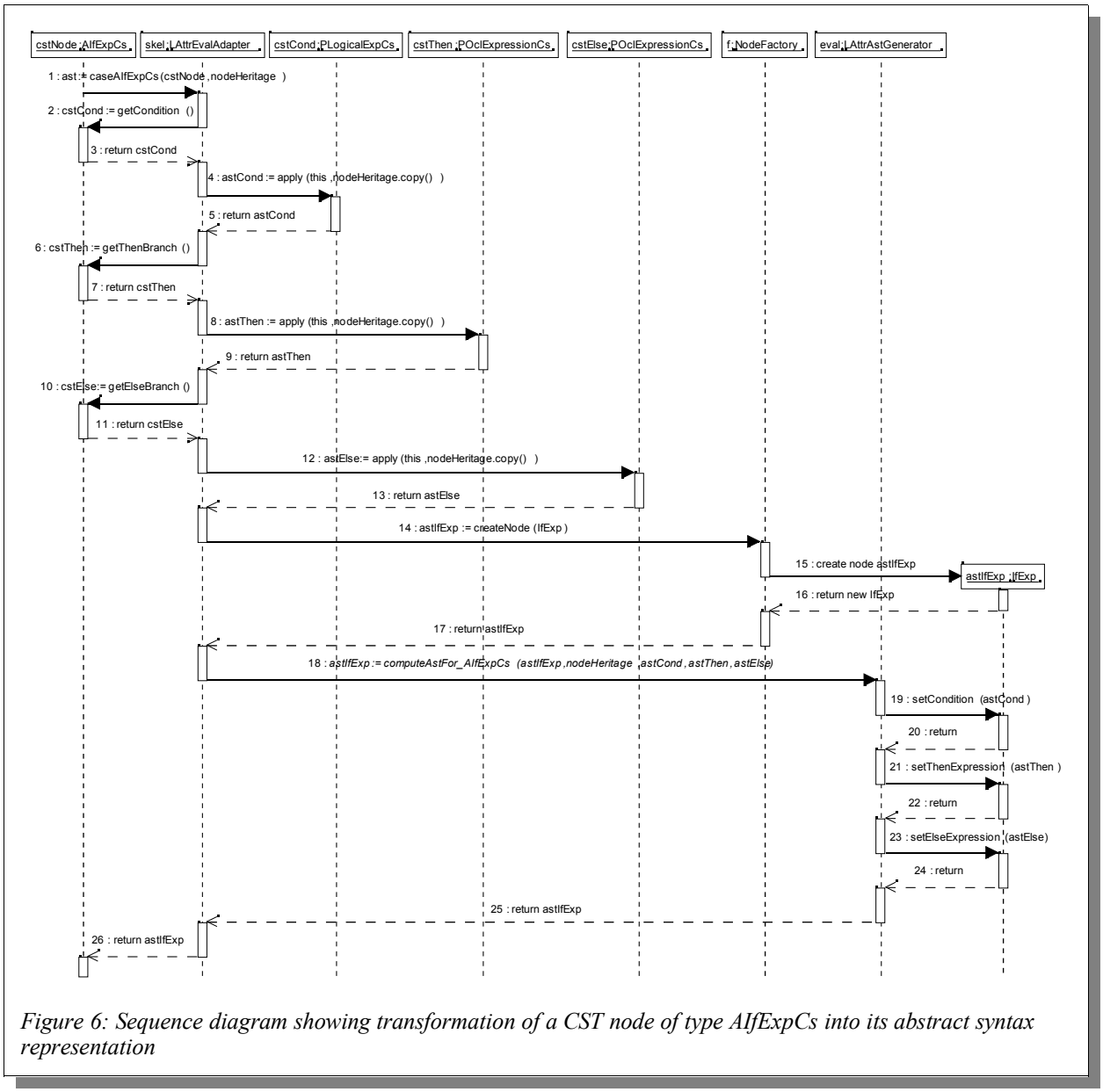

Figure 6: Sequence diagram showing transformation of a CST node of type AIfExpCs into its abstract syntax representation

References

[Gag98] Gagnon, É., SableCC, An Object-Oriented Compiler Framework, Master's Thesis, McGill University, Montreal 1998

[GoF] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., Design Patterns: Elements of Reusable Object-Oriented Software, 1994, Addison-Wesley

[MDAw] Object Management Group (OMG): OMG Model Driven Architecture (Homepage), 2005-06-02, 2005, http://www.omg.org/mda/

[Ock03] Ocke, Stefan, Entwurf und Implementation eines metamodellbasierten OCL-Compilers, Masters Thesis, Dresden University of Technology, 2003 (in German)

[OCL20] Object Management Group (OMG), UML 2.0 OCL Specification, OMG Document ptc/03-10-14, 2003

[OCLTKw] Dresden University of Technology, Department of Computer Science, Institute for Software and Multimedia Technology: Dresden OCL Toolkit - Welcome Page (Homepage), 2005-06-02, 2005, http://dresden-ocl.sourceforge.net/

[OMGw] Object Management Group (OMG): Object Management Group (Homepage), 2005-06-02, 2005, http://www.omg.org/

[SCCw] Gagnon, Étienne M., SableCC parser generator, 2005, http://sablecc.org/

[TUDw] Dresden University of Technology, Department of Computer Science: Chair of Software Engineering (Homepage), 2005-06-02, 2005, http://www-st.inf.tu-dresden.de/

[UML15] Object Management Group (OMG), OMG Unified Modeling Language Specification, Version 1.5, September 2002