Specifying Transformation Sequences as Computation on Program Fragments with an Abstract Attribute Grammar

Markus Schordan
Institute of Computer Languages
Vienna University of Technology
1040 Vienna, Austria
markus@complang.tuwien.ac.at

Daniel Quinlan
Centre for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, CA 94551, USA
dquinlan@llnl.gov

Abstract

We present a grammar based approach for specifying a transformation as a sequence of transformation operations that operate on an intermediate representation. The transformation sequence is specified in the semantic actions of an abstract attribute grammar. The mapping between the object-oriented design of the intermediate representation and the abstract grammar directly reflects the object-oriented design in the structure of the grammar. It has properties that permit utilizing grammar based tools at arbitrary abstraction levels of the language representation. The program fragments can be both source strings and fragments of the intermediate representation that can be used interchangeably in the specification.

We demonstrate the applicability of the approach by using available attribute grammar tools and the source-to-source infrastructure ROSE for specifying and performing transformations of C++ programs. The results of data flow analysis tools using fixed point algorithms is integrated as available attributes that can be used for enabling or disabling transformation sequences. With the abstract attribute grammar the transformation is computed as an attribute value that represents a sequence of restructure operations. The composition of different transformation sequences permits the reuse of sub-transformation specifications. Eventually we discuss the correspondence to rewrite tools permitting a pattern based restructuring of the program representation.

1 Introduction

Our goal is to introduce optimizing transformations on large scale scientific applications. Our approach is to define mechanisms to simplify the development of transformations to support optimizations on such applications where they are using C or C++. Within our work we have focused on domain-specific optimizations, where there is significant potential to leverage the additional semantics of domain-specific abstractions from libraries or within the applications directly, and which is unknown to vendor compilers. As a direct result, our target audience is both library developers and sophisticated applications people supporting large scale application development, without specific compiler background. Clearly our approach must thus present as easy and powerful a general mechanism as possible. This context for our research work has significantly shaped the mechanisms that we propose for the specification of general program transformations.

In order to simplify the specification of transformations we propose a combination of attribute grammars, rewrite operators, and the combined use of source fragments and AST fragments in specifications. Grammars have proven useful to define the structure of a language. As a result, several grammar based tools exist for recognizing languages and associating actions with the recognition of language constructs. To make transformations available to different clients, a transformation is usually applied to some intermediate representation (IR) of a single or multiple input language. This permits the generation of code from the intermediate representation for different target languages. Our approach uses grammars as basis for the specification of transformations.

The approach presented here, can be applied to any IR but we focus on the object-oriented design of annotated ASTs and a mapping that permits to see a direct correspondence of the object-oriented design of the AST, the grammar that defines its language, the abstract grammar, and the utilization of existing grammar based tools for specifying transformations. In the semantic actions we compute a sequence of powerful transformation operations.

The belief in the practicability of the approach is based on the experience in transforming C++ programs in source-to-source translators built with ROSE, which we describe in section 2, and some early experience from [1]. Meanwhile we have extended this effort in building tools for grammar transformations of abstract grammars and generate grammars as required by different tools such as ox [2], yacc, Coco [3], and the program analysis generator (PAG) [4].

In this paper we present the theory on which such mappings are based. In particular we shall discuss the constraints being placed on the grammars and the object-oriented design. They permit the presented mappings being applied such that a direct correspondence of the IR grammar, object-oriented design of the IR, and the source language can be established.

In section 2 we describe the ROSE infrastructure that permits building source-to-source translators for C++ and which essential features ROSE offers for transforming arbitrary C++ programs. The capabilities of the infrastructure are required to permit the use of transformation operators that we discuss in section 3. The transformation operators are used in the semantic actions of our abstract attribute grammar. In section 4 we present the mapping between the object-oriented design of the AST and our abstract C++ grammar. Since we permit to use both source strings and AST fragments in the semantic actions to specify a transformation, it is important to make the relation between the AST design and the abstract grammar as straight-forward as possible. In section 6 we present a transformation example that shows how the abstract grammar, the transformation operators, and program fragments can be used to obtain a compact specification for a parallelization of C++ iterators by introducing OpenMP directives. In section 7 we discuss the related work of object-oriented grammars and other transformation frameworks.

2 ROSE Infrastructure

The ROSE project defined a compiler infrastructure specifically for the development of source-to-source optimizing translators. Full C++ support is proved as required to compile large laboratory applications at Lawrence Livermore National Laboratory exceeding a million lines of code and using complex templates. ROSE generates an object-oriented annotated abstract syntax tree (AST) as an intermediate representation. Transformations are performed on the AST. Several components can be used to build the Mid End: a predefined traversal mechanism, attribute grammar tools, transformation operators to restructure the AST, and pre-defined optimizations. Support for library annotations is available by analyzing pragmas, comments, or separate annotation files. A C++ Back End can be used to unparse the AST and generate C++ code. An overview of the architecture is shown in Fig. 1). Steps 1-7 are explained in detail when describing the transformation operators in section 3.

2.1 Front End

We use the Edison Design Group C++ Front End (EDG) [5] to parse C++ programs. The EDG Front End generates an AST and performs a full type evaluation of the C++ program. This AST is represented as a C data structure. We translate this data structure into an object-oriented abstract syntax tree, Sage III, based on Sage II and Sage++[6]. Sage III is used by the Mid End as intermediate representation. The AST passed to the Mid End represents the program and all the header files included by the program (see Fig. 1, step 1 and 2).

2.2 Mid End

The Mid End permits the restructuring of the AST. Results of program analysis are made available as annotations of AST nodes and as accessible attributes in the abstract grammar. ROSE also includes a scanner which operates on the token stream of a serialized AST so that parser tools can be used to specify program transformations in semantic actions of an attribute grammar. The grammar is the abstract grammar, generating the set of all ASTs.

An AST transformation operation specifies a location in the AST where code should be inserted, deleted, or replaced. In Fig. 1 steps 3,4,5 show how the ROSE architecture permits the use of source code fragments and AST fragments in the specification of program transformations. A fragment is either a source string or an AST. A program transformation is defined by a sequence of AST restructuring operations. Transformations can be parameterized to define conditional restructuring sequences. This is discussed in detail in section 3.

2.3 Back End

The Back End unparses the AST and generates C++ source code (see Fig. 3, steps 6 and 7). It can be specified to unparse either all included (header) files or only the source file(s) specified on the command line. This feature is important when transforming user-defined data types, for example, when adding generated methods. Comments are attached to AST nodes and unparsed by the Back End.

3 Transformation Operators

In this section we present the transformation operators that are used in the semantic actions of the abstract attribute grammar. A restructuring sequence consists of fragment operators, and as operands are used AST fragments (subtrees), strings (concrete pieces of code), or AST locations (denoting nodes in the AST).

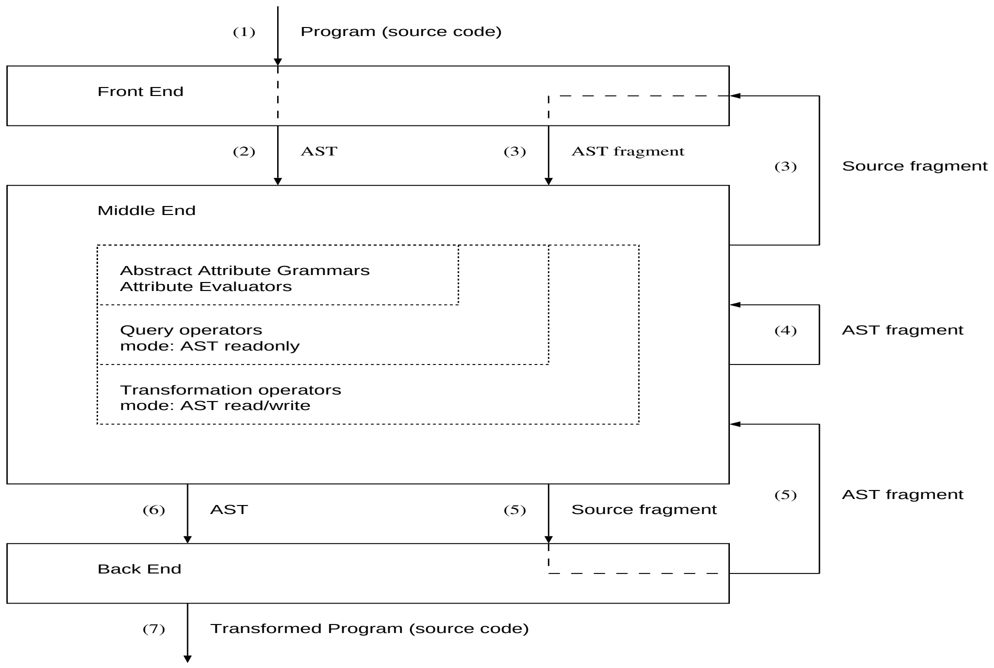

Figure 1. ROSE Source-To-Source infrastructure for parsing C++ programs (steps 1 and 2), generating C++ code (steps 6 and 7) and handling of code fragments (steps 3-5).

To permit the use of source fragments in specifications it is required to translate source fragments to AST fragments. We also permit that source strings are computed as attribute values from other source strings. Therefore source fragments are completed to valid programs such that the programs can be parsed by the Front End and then we extract the corresponding AST fragment. The components that permit us utilizing the existing Front End for parsing source-fragments are the Fragment Concatenator and Extractor. Based on this functionality the fragment operators are defined.

3.1 Fragment Concatenator and Extractor

In general, a source-fragment cannot be parsed by the Front End because it is an incomplete program. Therefore it needs to be extended by a source-prefix and a source-postfix to a complete program such that it can be parsed by the Front End. The computation of the prefix and postfix is automated. The user only specifies the fragment and the target location of the corresponding IR-fragment. In our IR, the target location, L_{abs}, is a node in the AST. The prefix and postfix are automatically generated. The source-prefix consists of all declarations and opening braces of scopes before the target location, and the function or method headers for which the target location is specified. The source-postfix consists of all closing braces of scopes after the target position.

The Front End returns a program in IR. From this the corresponding IR-fragment needs to be extracted. A source string shall be denoted as S and an intermediate representation as I. We shall denote any prefix by $\lhd$, any fragment by $\Box$, and any postfix by $\rhd$.

A given source-fragment, $S_\Box$, is translated to an IR-fragment, $I_\Box$, by invoking the Front End. The fragment concatenator concatenates the source-prefix $S_\lhd$, the source-fragment $S_\Box$, and the source-postfix $S_\rhd$. Information necessary to extract the IR-fragment, $I_\Box$, corresponding to the source-fragment, $S_\Box$, from the IR of the completed program, shall be denoted L_{sep}. It represents separators that are inserted by the concatenator before invoking the Front End, and used by the extractor to separate the fragment from the prefix and postfix.

$$(S, L_{sep}) = \text{concatenator}(S_\lhd, S_\Box, S_\rhd)$$

The completed program S can be parsed by the Front End

$$I = \text{frontend}(S)$$

to obtain the program in intermediate representation I. From this program I, the IR-fragment, $I_\Box$, is extracted by the fragment extractor.

$$I_\square = \text{extractor}(I, L_{sep})$$

The fragment extractor strips off the IR-prefix, $I_\triangleleft$, corresponding to $S_\triangleleft$ and $I_\triangleright$ corresponding to $S_\triangleright$. Information on where these parts are separated, L_{sep}, which is returned by the fragment concatenator, is used to find start and end points of $I_\triangleleft$ and $I_\triangleright$.

We have shown how we can obtain the corresponding IR-fragment $I_\square$ for a given source-fragment $S_\square$ by invoking the Front End. The inverse operation, by invoking the Back End, is

$$S_\square = \text{backend}(I_\square).$$

Since both representations, $I_\square$ and $S_\square$, can always be translated one to the other, both can be used interchangeably in the definition of a transformation.

3.2 Fragment Operators

A fragment operator permits performing a basic restructuring operation such as insert, delete, or replace AST fragments. The target location in the AST can be absolute or relative. The fragment to be inserted can be specified as source fragment or AST fragment.

Whether a source fragment is valid with respect to an absolute location is determined automatically. From the syntactic context of the absolute location the prefix, $s_\triangleleft$, is computed such that all declarations, opening scopes, and function headers are included in the prefix. The postfix, $s_\triangleright$, consists of all the syntactic entities of closing scopes (for nested scopes such as for-loops, while-loops, function definitions, etc.).

Definition 1 (Valid Source Fragment) *A source fragment* $s_\square$ *is* valid with respect to an absolute location l_{abs} *in an AST if it can be completed to a legal program from the syntactic and semantic context of the absolute location such that the completed program has correct syntax and semantics, symb.* $s_\square$ *is valid with respect to* l_{abs} *if* $frontend(s_\triangleleft + s_\square + s_\triangleright)$ *with* $s_\triangleleft = prefix(l_{abs})$, $s_\triangleright = postfix(l_{abs})$ *succeeds.*

For a valid source fragment $s_\square$ we can always generate a corresponding AST fragment, $ast_\square$. In our C++ infrastructure the AST fragment $ast_\square$ has all templates instantiated.

Based on the handling of code fragments, the transformation operators can be defined as follows. Let $ASTs$ denote the set of ASTs, L_{rel} the set of relative locations in an AST, L_{abs} the set of absolute locations, i.e. the nodes in an AST, and S the set of valid source fragments with respect to an absolute location in the AST. Then the transformation operators can be defined as

insert: $L_{rel} \times L_{abs} \times ASTs \rightarrow ASTs$
: Insertion of AST fragment at relative location (step 4 in Fig. 1)

delete : $L_{abs} \times ASTs \rightarrow ASTs$
: *Deletion of AST subtree at absolute location in AST (step 4 in Fig. 1)*

fragment-frontend : $L_{abs} \times ASTs \times S \rightarrow ASTs$
: *Translate source fragment with respect to absolute location in AST to corresponding AST fragment (steps 3,5 in Fig. 1)*

fragment-backend : $L_{abs} \times ASTs \rightarrow S$
: *Unparse AST fragment at absolute location in AST to source fragment (step 5 in Fig. 1)*

locate : $L_{rel} \times L_{abs} \times ASTs \rightarrow L_{abs}$
: *Map relative location with respect to absolute location in AST to absolute location in same AST*

replace : $L_{rel} \times L_{abs} \times ASTs \times ASTs \rightarrow ASTs$
: *Replacement of AST fragment at relative location (step 4 in Fig. 1)*

replace : $L_{abs} \times ASTs \times S \rightarrow ASTs$
: *Replacement of AST subtree at absolute location in AST by AST fragment corresponding to source fragment (steps 3,4,5 in Fig. 1)*

unsafe-replace : $L_{abs} \times ASTs \times S \rightarrow ASTs$
: *Replacement of string that is unparsed by the Back End at absolute location in AST by source fragment (i.e. the Back End unparses this string instead of the code that is represented by the AST subtree at the absolute location)*

The fragment operators allow rewriting the AST by specifying absolute or relative target locations. A relative location l_{rel} permits specification of a target location in an AST relative to an absolute location l_{abs}. The operator $location$ can map a relative location l_{rel} with respect to an absolute location l_{abs} and a given AST containing the absolute location l_{abs}, to another absolute location in the same AST according to L_{rel}. Relative locations are used to simplify the specification of the target location of a fragment operation. For example, if a statement can be hoisted out of a loop it suffices to specify as target location the "statement in outer scope right before the current loop". We have defined several classifications of such relative target locations which are useful in making transformations more compact. The insert-operation is an example of using a relative target location. The operator *fragment-frontend* permits translation of source fragments to AST fragments as explained above. It also requires step 5 to compute the necessary prefix and postfix to complete the source fragment to eventually call the Front End for the completed program. The unparsing of an AST fragment, *fragment-backend* requires invoking the Back End. The second replace operator listed, permits specification of the new AST fragment, ast, which replaces an AST subtree at location L_{abs} in this AST, to be specified by a source fragment, s. This requires all three steps 3,4,5 (see Fig. 1). Step 5 is required to unparse parts of the AST to form the prefix, $s_{\lhd}$, and postfix, $s_{\rhd}$. In Step 3 the completed source fragment is translated to an AST and the corresponding AST fragment, ast, is extracted. Step 4 is the actual rewriting of the AST and the replacement of the AST subtree with the new AST fragment is performed. Based on this basic operations on fragments, transformation operators can be defined.

The last listed replace operation, *unsafe-replace* only "patches" the Back End such that if the unparser is invoked, this string is unparsed instead of the AST subtree at the specified location. Consequently, fragments that are inserted using *unsafe-replace* are not checked (no syntactic or semantic analysis). In some transformations this feature helps to keep a transformation specification compact because we can decompose a transformation in sub-transformations where not every single operation is checked but the concatenated strings (or combined ASTs) are checked. In our example in section 6 the parameterized sub-transformation `derefToIndexBody` performs only an unsafe-replace because we cannot replace the access, `*i`, only but also need to replace the for-loop-header. Hence, when several operations need to be performed in "parallel", some operations may be specified as unsafe, and at some point, in our example when the subtree representing a for-statement is replaced, we perform a safe replace (called replace) that includes all the other unsafely replaced fragments. Note that a safe operation fails if any syntactic or semantic error is encountered.

The combination of different source-fragments and AST fragments is specified in semantic actions associated with productions of an abstract grammar which we describe in the next section.

4 Abstract Grammar

We use as intermediate representation an annotated abstract syntax tree (AST). The AST is generated by the Front End and the transformations are specified as restructure operations on the AST. In this section we shall define a mapping between the object-oriented design of the AST and the abstract grammar such that the abstract grammar generates the set of all ASTs according to the abstract data types used for implementing the AST. Then we shall provide mappings to context free grammars and tree grammars as they are required by different grammar based tools that we use for specifying transformation sequences.

```
SgNode : SgSupport
 | SgLocatedNode
 ...
 ;

SgLocatedNode : SgStatement
 | SgExpression
 ;

SgStatement : SgExprStatement ( SgExpressionRootNT )
 | SgReturnStmt ( SgExpressionRootNT )
 | SgForInitStatement ( SgStatement* )
 | SgScopeStatement
 ...
 ;

SgScopeStatement : SgBasicBlockNT
 | SgIfStmt ( SgStatement,
              SgBasicBlockNT,
              SgBasicBlockNT )
 | SgForStatement ( SgForInitStatementNT,
                    SgExpressionRootNT,
                    SgExpressionRootNT,
                    SgBasicBlockNT )
 ...
 ;
```

Figure 2. Fragment of our abstract C++ grammar used in ROSE. The full grammar consists of 199 productions.

4.1 Class Hierarchy

Let a class hierarchy,H, with single inheritance and a single root class, be a set of pairs (C, C') such that C is a super class (direct base class) of C'. Let H_I denote the set of inner nodes and H_L denote the set of leaf nodes of the hierarchy H. set of all classes be denoted as H_C.

Further, let the children information that is associated with a class C_0, be represented by a pair $(C_0, (C_1, \ldots, C_n)) \in \mathcal{C}$ where C_i, $1 \leq i \leq n$, is the class of a child of C_0 where $n = 0$ represents the fact that no children are associated with class C_0. The class hierarchy is *unfolded* such that the children information is associated with the leaf nodes only in the hierarchy.

Property 1: It holds that $\forall (C_0, (C_1, \ldots, C_n)) \in \mathcal{C} : n > 0$ implies $C_0 \in H_L$, i.e. children information is only associated with leaf nodes of the unfolded class hierarchy.

We define the object-oriented abstract grammar such that $G = (T, N, P, S)$ where the set of terminals represents the leaf nodes of the class hierarchy, $T = H_L$, the set of non-terminals represents the inner nodes of the hierarchy, $N = H_I$, $S \in N$ and the productions are defined such that

$P = \{A \rightarrow B \mid A \in H_I, B \in H_I, (A, B) \in H\} \cup \{A \rightarrow b(C_1, \ldots, C_n) \mid (A, b) \in H, (b, (C_1, \ldots, C_n)) \in \mathcal{C}\}$ generates the set of all ASTs.

The grammar represents the object-oriented design such that chain productions, $A \rightarrow B$, represent inheritance (class B inherits from class A) and production $A \rightarrow b(C_1, \ldots, C_n)$ represents the fact that class b inherits from class A and C_i, $1 \leq i \leq n$, are classes of member variables in class b that hold a reference to an object of class C_i.

Property 2: A valid AST consists of instances of classes $b \in H_L$ only.

Therefore each concrete class of the class hierarchy in the AST is represented by a terminal b in the abstract grammar. The non-terminals of the abstract grammar never represent concrete classes of AST nodes.

A fragment of our abstract C++ grammar with these properties is shown in fig. 2. The chain production `SgStatement`$\rightarrow$ `SgScopeStatement` represents the fact that class `SgScopeStatement` inherits from class `SgStatement` (is-a relationship). The last production in the grammar fragment represents the fact that `SgScopeStatement` is the base class of class `SgForStatement`. Because `SgForStatement` is a terminal, it corresponds to a concrete AST node class. The list of four non-terminals on the right-hand-side represents the fact that class `SgForStatement` has four children in the AST (has-a relationship).

5 Use of Abstract Grammar

The combination of several different grammar based tools permits to specify a transformation in the most compact form. In this section we show how we have integrated a variety of different tools into the ROSE infrastructure. The design of the abstract grammar, which is a direct consequence of the mapping presented in the previous section, has proven to be the best format to permit a straight-forward translation for different grammar based tools. The generation of the external representation, as required by some tools, is also specified as attribute computation.

5.1 Attribute Grammar Tools

We now turn attention to LL and LR grammars describing our abstract syntax to utilize existing parser tools. For LL and LR grammars several properties must hold such that no parsing conflicts exist.

A (non-redundant) context-free grammar is $LL(1)$ iff for every variable A the set of the corresponding productions $\{A \rightarrow Q_1, \ldots, A \rightarrow Q_n\}$ satisfies the following two conditions:

1. $first_1(Q_i) \cap first_1(Q_j) = \emptyset$ for $i \neq j$
2. If $Q_i \overset{*}{\Rightarrow} \epsilon$ then for all $j \neq i$ $(1 \leq j \leq n)$: $first_1(Q_j) \cap follow_1(A) = \emptyset$

Our abstract grammar has only two kinds of productions. They are represented in LL(1) tools as $A \rightarrow B$ and $A \rightarrow b(C_1 \ldots C_n)$ where b is a terminal and A,B, C_i are non-terminals. The terminals '(' and ')' are added to the token stream, which therefore is called *enhanced token stream*. Since each terminal shows up at most once in the grammar and every right-hand-side of a production that is not a chain production has such a terminal as first symbol, the first condition is satisfied and since there are no empty productions the second conditions is satisfied as well.

If an AST has optional children, for example that a reference to a child is either null or refers to some object, then we would have empty productions and condition 2 would not necessarily be satisfied. This problem can be eliminated by replacing this node class in the AST with two different classes, one holding the reference and the other does not.

Our abstract C++ grammar, shown in fig. 2, satisfies properties 1 and 2. The generation of LR and LALR grammars is straight-forward from this grammar by generating left-recursive productions for all lists (denoted by *). Beside Coco, we also use the LALR tool ox [2], which is based on yacc.

5.2 Program Analysis Tools

The Program Analysis Generator (PAG) [4] requires an abstract grammar as input, so called syn files. The results of an analysis with PAG is attached as attributes to AST nodes and can therefore be accessed in the transformation specification as attributes, enabling or disabling transformation sequences. We use PAG for flow-sensitive analyses requiring fixed point algorithms and make the results of an analysis available as information attached to AST nodes. The AST nodes are accessible in the semantic actions of the abstract grammar because pointers to the nodes are provided as an attribute in the grammar. This gives access to all information that is attached to AST nodes and permits making an attribute computation dependent on the results obtained by PAG or other program analysis components.

5.3 Bottom Up Rewrite Tools

We use burg and iburg [7] for applications of code selection. Burg requires trees that consist of nodes of arity less or equal two and null pointers are not allowed. Nodes with an arbitrary number of children need to be represented by a binary tree (i.e. a list representation). The grammar only differs in how we represent lists, because we need to introduce an auxiliary node to the abstract grammar that acts as a list separator. We use this grammar to apply semantic actions according to an optimal selection of instructions based on a constant that is associated with each production. We use it for treating overloaded user-defined operators in C++ like built-in operators and optimize them similar to expression trees for built-in operators.

5.4 Term Rewrite

To permit usual term rewriting of the AST we generate as external format a Prolog representation of the AST. An AST is represented by a single term. Using Prolog, users can specify pattern based transformations and make use of the more powerful mechanism of unification. After a term-based transformation is performed, the new Prolog term is processed and a Sage III AST is created. A similar approach is used in the JTransformer Framework, a query and transformation engine for Java source code, by using a Prolog representation of Java programs.

6 Program Transformation Example

In the example source in fig. 3 we show a C++ code before and after transformation. In the original code a sequential iteration on a user-defined container is performed. This pattern is frequently used in applications using C++98 standard container classes. We parallelize this code by introducing OpenMP directives and by transforming the loop header and loop body such that it conforms to the required canonical form of an OpenMP parallel for. After transformation the code can be executed in parallel. The transformation is specified as semantic actions of our abstract C++ grammar which corresponds to the object-oriented AST as discussed in section 4. In the semantic actions we use the transformation operators presented in section 3 and compute a transformation sequence consisting of such operations. Eventually the sequence of transformation operators is applied, the AST is transformed, and the unparsed AST is unparsed as C++ program.

In fig. 3 the object `a` is an instance of the user-defined class `Range`. The transformation we present takes into account the semantics of the type `vector<Val>` and the semantics of class `Range`. The transformation is therefore specific to these classes and its semantics.

For the type `vector<Val>` we know how the type `iterator` is defined in the class `vector` because it is one of the C++98 standard classes. For the type `Range` we know that the method `update` is thread safe from a library annotation. We show the core of a transformation to transform the code into the canonical form of a for-loop as required by the OpenMP standard. We also introduce the OpenMP pragma directive. Note that the variable `i` in the transformed code is implicitly private according to the OpenMP standard 2.0 . If the generated code is compiled with an OpenMP compiler, different threads are used for executing the body of the for-loop.

Our abstract grammar covers full C++ and we use a successor of Coco/R [3], the C/C++ version ported by Frankie Arzu, as attribute grammar tool. Coco/R is a compiler generator that permits to specify a scanner and a parser in EBNF for context free languages. The grammar has to be LL(1). We use this tool to operate on the token stream of AST nodes. Therefore we do not use the scanner generator capabilities of Coco/R and implemented a scanner to operate on a token stream of AST nodes.

In the example in fig. 4 the production of `SgScopeStatement` is shown. The terminal `SgForStatement` corresponds to an AST node of type `SgForStatement`. The variable `astNode` is a pointer to the respective AST node of the terminal and assigned by our supporting system when the scanner accesses the token stream. Note that every terminal in the grammar corresponds to a node in the AST, except the parentheses.

Methods of the object `transformationSequence` allow to insert new source code and delete subtrees in the AST. The transformation object `transformationSequence` buffers pairs of target location and string. The substitution is not performed before the semantic actions of all subtrees of the target location node have been performed. This mechanism allows to check whether substitutions would operate on overlapping subtrees of the AST (in the same attribute evaluation). In case of overlapping subtrees an error is reported.

The object `query` is of type `AstQuery` and provides frequently used methods for obtaining information stored in annotations of the AST. These methods are also implemented as attribute evaluations.

The inherited attribute `loopNestingLevel` is used to handle the nesting of for-loops. It depends on how an OpenMP compiler supports nested parallelism whether we want to parallelize inner for statements or only the outer for statement. In future this decision will be made more specific to OpenMP compilers on different platforms and the boolean attribute will be replaced by an object to provide more information about the context of OpenMP for-loops.

The object `query` of type `AstQuery` offers methods to provide information on subtrees that have proven to be useful in different transformations. In the example we use it to obtain the name of the iterator variable and to obtain the node of the declaration of the iterator variable. Note that these functions must return valid values because it has been tested before that the for-loop qualifies for transformation.

The example shows how we can decompose different aspects of a transformation into separate attribute evaluations. The methods of the query object are implemented by using the attribute evaluation. For that reason we allow to call any method of the recursive descent parser gen-

Before transformation

```
for(vector<Val>::iterator i=l.begin(); i!=l.end(); ++i) {
   a.update(*i);
}
```

After transformation

```
#pragma omp parallel for
for(int i = 0; i < l.size(); ++i) {
  a.update(l[i]);
}
```

Figure 3. An iteration on a user-defined container `l` that provides an iterator interface. The object `a` is an instance of the user-defined class `Range`. Object `l` is of type `vector<Val>`. In the optimization the iterator is replaced by code conforming to the required canonical form of an OpenMP parallel for. The user-defined method update is thread-safe. This semantic information is used in the transformation.

```
SgScopeStatement<int loopNestingLevel>
  = SgBasicBlockNT<loopNestingLevel>
  | SgIfStmt
    "(" SgStatement<loopNestingLevel>
        SgBasicBlockNT<loopNestingLevel>
        SgBasicBlockNT<loopNestingLevel>
    ")"
  | SgForStatement
        (.
          bool isOmpFor = ompTransUtil.isUserDefIteratorForStatement(astNode,loopNestingLevel);
        .)
    "(" SgForInitStatementNT<loopNestingLevel>
        SgExpressionRootNT
        SgExpressionRootNT
        SgBasicBlockNT<loopNestingLevel+1>
    ")"
        (.
          if(isOmpFor) {
            string beforeForStmt = "#pragma omp parallel for\n";

            string ivarName = query.iteratorVariableName(astNode);
            string icontName = query.iteratorContainerName(astNode);
            string modifiedBodyString = ompTransUtil.derefToIndexBody(ivarName,icontName);
            string newForStmt = "for(int " + ivarName + "=0;"
                              + ivarName + "<" + icontName + ".size();"
                              + "++" + ivarName + ") " + modifiedBodyString;

            transformationSequence.replace(astNode,beforeForStmt + newForStmt);
          }
        .)
  | ...
```

Figure 4. A part of the SgScopeStatement production of the abstract C++ grammar with the semantic action specifying the transformation for SgForStatement.

erated by Coco to parse a sublanguage, and start an evaluation at a certain node in the AST. Multiple grammar files can also be used for such cases and each file contains a version of the abstract C++ grammar. In the example, the method `isUserDefIteratorForStatement` is a wrapper function of another attribute evaluation generated by Coco that starts at a SgForStatement node and implements a conservative test.

In fig. 3 the generated code is shown. The access uses the notation for random access iterators. Even if the access is not of complexity O(1) the parallelization can still provide speedup. The user who implements the transformation has to take such trade offs into account in a test function to decide whether a transformation should be applied or not.

7 Related Work

An object-oriented view on attribute grammars that is similar to our mapping in some aspects was already presented by Koskimies [8] in 1991. He used two notions of non-terminals, so called superclass non-terminals and basic non-terminals. The concept of superclass non-terminals and the use of chain productions to express the inheritance relation is the same in our approach. But we do not use the concept of basic non-terminals to specify the syntactic composition of basic language constructs because our grammar is an abstract grammar and is normalized in the sense that a concrete class always corresponds to a terminal in the grammar. A basic non-terminal on the left-hand-side of a production and the so called slots in [8] correspond in our grammar to a terminal and list of non-terminals on the right hand side of a production. On the left-hand-side of a production our non-terminals correspond to superclass non-terminals only.

An interesting comparison of concrete and abstract syntax and its use with Stratego can be found in [9]. Eelco Visser discusses meta programs that manipulate ASTs of programs and as a case study the addition of concrete syntax to the program transformation language Stratego is presented. It turns out that the abstract syntax version is much more verbose and harder to read and write, especially the definition of large code fragments. Therefore the syntax of the object language is embedded in the meta-language. We address this aspect by using source-strings in the semantic actions associated with the productions of the abstract grammar. This makes the definition of large code fragments trivial because the code can simply be written in C++ (the object language) whereas the structure of the program is represented at the abstract level of an AST.

Stratego/XT [10] is a framework for the development of transformation systems aiming to support a wide range of program transformations. The framework consists of the transformation language Stratego and the XT collection of transformation tools. Stratego is based on the paradigm of rewriting under the control of programmable rewriting strategies. Because rewrite rules can only use information obtained by pattern matching on the subject term, for transformations that also require information from the context of a program Stratego also provides an extension of strategies with scoped dynamic rewrite rules. This functionality can be achieved in our approach by using attributes for computing context information and using absolute or relative target locations for transformations. Stratego can also be used for instruction selection. For this purpose we use the mapping of the abstract grammar to a grammar suitable for burg [7], a tool for code selection, which we utilize for operation on the AST.

Codeboost [11, 12], a source-to-source transformation tool for domain-specific optimization of C++ programs, offers the specification of transformations using Stratego or user-defined rewrite rules embedded within C++ programs. In [11] is discussed that the use of user-defined rules is advantageous over using Stratego when the goal is to optimize calls to specific functions in C++ which may be overloaded. Therefore the name of the function is not sufficient but the full signature must be specified. This is reported to be tedious and error-prone, particularly when working with the abstract syntax in Stratego. In our approach we also use an abstract syntax but provide all type information, including the signature of functions, as annotations of the AST which can be accessed as attributes in the semantic actions of the abstract C++ grammar.

Similar to our approach DMS [13] also provides attribute grammars to determine properties of programs to decide where to perform transformations for a given program. Transformations are specified as source-to-source rewrite rules where left hand side and right hand side represent source language patterns with variables to represent arbitrarily long well formed language substrings. An additional optional condition can be used to determine whether a transformation should be applied. In our approach we do not only use the attribute grammar for analysis but also for specifying the transformation sequence and combine it with the use of source strings. Parameters have a special syntax in DMS, they are preceded by a backslash. We permit to build source strings by using text concatenation and define parameters by either using computed attributes or separate query objects which permit to decompose a transformation in different objects (see object `query` in fig. 4).

8 Conclusions

We presented an approach to simplify the specification of program transformations. We defined an abstract grammar for C++ such that the object-oriented design of the ROSE AST is directly represented in the grammar. The transformation sequences were specified in the semantic actions of our abstract attribute grammar. As example for a grammar based tool we used Coco for operating on a serialized AST and performing attribute evaluation. The transformation sequence consisted of transformation operations that are supported by the ROSE infrastructure and we demonstrated the use of source strings and AST fragments in specifying transformations.

We see advantages in the presented grammar based approach because of its explicit representation of the structure of the IR, the high-level specification of computing attributes, and the use of source fragments and AST fragments in the specification. We leveraged the parsing technology of an existing Front End and grammar tools and applied them at the abstract level of the AST. The presented example was decomposed into two transformations, the header of the for-loop and the body of the for loop. Further work is required regarding the optimization of decomposed transformations.

The combination of grammar based tools, program analysis tools, and the ROSE infrastructure for restructuring C++ programs, permitted a compact specification, which we demonstrated in an example by specifying a transformation sequence for parallelizing C++ programs.

References

[1] Dan Quinlan, Markus Schordan, Qing Yi, and Bronis de Supinski. Semantic-driven parallelization of loops operating on user-defined containers. In *LCPC'03: 16th Annual Workshop on Languages and Compilers for Parallel Computing*, volume 2958 of *Lecture Notes in Computer Science*, pages 524–538. Springer Verlag, 2004.

[2] Kurt M. Bischoff. Design, implementation, use, and evaluation of ox: An attribute-grammar compiling system based on yacc, lex, and C. Technical Report TR92-31, Iowa State University, Department of Computer Science, December 1992.

[3] Hanspeter Moessenboeck. Coco/R - A generator for production quality compilers. In *3rd Int. Workshop CC'90*, volume 477 of *LNCS*, pages 42–55. Springer Verlag, 1991.

[4] Florian Martin. PAG – an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer*, 2(1):46–67, 1998.

[5] Edison Design Group. http://www.edg.com.

[6] Francois Bodin, Peter Beckman, Dennis Gannon, Jacob Gotwals, Srinivas Narayana, Suresh Srinivas, and Beata Winnicka. Sage++: An object-oriented toolkit and class library for building fortran and C++ restructuring tools. In *Proceedings. OONSKI '94*, Oregon, 1994.

[7] Todd Proebsting. Burg, iburg, wburg, gburg: so many trees to rewrite, so little time (invited talk). In *RULE '02: Proceedings of the 2002 ACM SIGPLAN workshop on Rule-based programming*, pages 53–54, New York, NY, USA, 2002. ACM Press.

[8] K. Koskimies. Object Orientation in Attribute Grammars. In H. Alblas and B. Melichar, editors, *Proc. International Summer School on Attribute grammars, Applications and Systems (SAGA'91), Prague, Czechoslovakia*, volume 545 of *LNCS*, pages 297–329. Springer-Verlag, June 1991.

[9] Eelco Visser. Meta-programming with concrete object syntax. In Don Batory, Charles Consel, and Walid Taha, editors, *Generative Programming and Component Engineering (GPCE'02)*, volume 2487 of *Lecture Notes in Computer Science*, pages 299–315, Pittsburgh, PA, USA, October 2002. Springer-Verlag.

[10] Eelco Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Spinger-Verlag, June 2004.

[11] Otto Skrove Bagge, Karl Trygve Kalleberg, Magne Haveraaen, and Eelco Visser. Design of the CodeBoost transformation system for domain-specific optimisation of C++ programs. In Dave Binkley and Paolo Tonella, editors, *Third International Workshop on Source Code Analysis and Manipulation (SCAM 2003)*, pages 65–75, Amsterdam, The Netherlands, September 2003. IEEE Computer Society Press.

[12] Otto Skrove Bagge and Magne Haveraaen. Domain-specific optimisation with user-defined rules in CodeBoost. In Jean-Louis Giavitto and Pierre-Etienne Moreau, editors, *Proceedings of the 4th International Workshop on Rule-Based Programming (RULE'03)*, volume 86/2 of *Electronic Notes in Theoretical Computer Science*, Valencia, Spain, 2003. Elsevier.

[13] Ira D. Baxter, Christopher Pidgeon, and Michael Mehlich. DMS: Program transformations for practical scalable software evolution. In *ICSE*, pages 625–634. IEEE Computer Society, 2004.