Writing and Designing C++ Extensions with Transformers

Vincent Ordy

Technical Report n^o0913, December 2009
revision 2154

The Transformers project aims at providing source to source transformations for C and C++ languages. This consists in parsing the input source, a C/C++ source code extended to accept new syntactic rules. The input code is then transformed into standard C/C++. This is similar to the process used by the C++ ancestor, "C with classes", which was an extension of C and which was transformed into C before being compiled.

We will show how to write an extension of the C++ grammar using the Transformers project, and to transform the extended C++ input into standard C++. For this purpose, we will use extensions that have already been implemented (ContractC++, class-namespaces) as examples. We will analyse to what extent the technologies like attribute grammars used in Transformers help us.

Le projet Transformers permet de faire de la transformation source à source pour les languages C et C++. Le but est de pouvoir effectuer une analyse syntaxique d'un langage d'entrée, du C ou C++ étendu pour accepter de nouveaux éléments syntaxiques, qui seront ensuite transformés en C/C++ standard de la même manière que « C avec classes », ancêtre du C++, était d'abord transformé en C avant d'être compilé.

Nous allons expliquer comment écrire une extension de la grammaire du C++ grâce au projet Transformers puis transformer le code étendu en code standard, en nous appuyant sur des exemples d'extensions déjà écrites (ContractC++, class-namespaces). Nous montrerons les avantages et inconvénients des technologies utilisées par Transformers comme les grammaires attribuées.

Keywords
C++, Transformers, grammar extension, source to source transformations, Stratego/XT

Laboratoire de Recherche et Développement de l'Epita
14-16, rue Voltaire – F-94276 Le Kremlin-Bicêtre cedex – France
Tél. +33 1 53 14 59 47 – Fax. +33 1 53 14 59 22
vincent.ordy@lrde.epita.fr – http://www.lrde.epita.fr/

Copying this document

Contents

Introduction

The TRANSFORMERS project aims at designing a set of tools to manipulate C++ source code. This includes the generation of programs to automatically change a given pattern of code into another. Combined with the ability to extend the C++ grammar of TRANSFORMERS, we can write tools to convert a C++ program containing experimental new language features into a regular C++ program. This enables us to add such features as multi-methods, concepts, or contract programming. Some of them were described in a previous technical report (Despres, 2004).

In order to manipulate the input source code, several steps are required: the parsing, the disambiguation, the transformation (using rewriting rules), and finally an additional step to output the AST into C++ code.

An intense work has been made over the last years to implement the parsing and disambiguation in TRANSFORMERS. There were a few attempts of transformations and grammar extensions. The first attempts were on the C language (Borghi et al., 2006), because its grammar is easier than the C++ one. Finally, two grammar extensions and transformations were made for C++ (Ordy, 2008, 2009). The purpose of this report is to analyze the different approaches used in these.

First, we will describe a few related frameworks for C++ transformations. Then we will briefly explain how TRANSFORMERS works, and its current state. After that, we will explain how to make an extension of the C++, analyze the problems encountered, and the possible solutions. Finally we explain how to do transformations with and without attribute grammars, and compare these two approaches.

Acknowledgements I want to thank all the LRDE staff and students, and more specially:

- Warren Seine and Akim Demaille for proofreading this report
- Nicolas Pierron and Valentin David for their help with TRANSFORMERS and STRATEGO

Chapter 1

Rose

The ROSE Compiler Framework is a tool for building source-to-source translators. It uses the EDG parser, and SAGE III as an Intermediate Representation.

1.1 Features

ROSE permits to write transformations in C++. The AST contains a lot of informations, and the user can add more informations on the nodes if he needs to. (Attribute System.)

There are several traversal possible using the visitor pattern:

- AstSimpleProcessing,
- AstPrePostProcessing,
- AstTopDownProcessing...

However it is not possible to extend its parser grammar, because it relies on the EDG non-free/non-extensible parser.

1.2 Example

Add _foo to function names:

```
#include "rose.h"
#include "my_visitor.hh"

int main(int argc, char* argv[]) {
  // Parse the input file.
  SgProject* astNode = frontend(argc, argv);

  // Call MyVisitor::visit on each node.
  MyVisitor v;
  traverseMemoryPoolVisitorPattern(v);

  // Output the code, compile...
  return backend(astNode);
}


//
void MyVisitor::visit(SgFunctionDefinition* functionDefinition)
{
```

```
  // Get the old name and make the new.
  SgName fname = functionDefinition->get_qualified_name();
  fname.getString().append("_foo");

  // Make the new declaration of the function
  SgFunctionDeclaration* fdecl = functionDefinition->get_declaration();
  SgFunctionDeclaration* newfdecl =
    buildDefiningFunctionDeclaration(fname,
                                     fdecl->get_type(),
                                     fdecl->get_parameter_list(),
                                     fdecl->get_scope()) ;
  // Make the new definition.
  SgFunctionDefinition* newfdef =
    buildDefiningFunctionDefinition(newfdecl,
                                    functionDefinition->get_body());

// ... some code to go to the right scope, and delete old code

  // Add the new nodes.
  appendStatement(newfdecl);
  appendStatement(newfdef);
}
```

Chapter 2

OpenC++

Quoting their homepage (Chiba, 2004):

> *OpenC++ is C++ frontend library (lexer+parser+DOM/MOP) and source-to-source translator. OpenC++ enables development of C++ language tools, extensions, domain specific compiler optimizations and runtime metaobject protocols.*

The development has apparently stalled, the latest release dates from 2004 and the latest development snapshot from September 2005. It cannot even parse a recent GNU STL anymore.

Warning: OpenC++ is not to be confused with Open C++, a development environment provided by Nokia for the S60 platform, a software platform for mobile phone running Symbian OS.

2.1 Examples

Add _foo to function names:

```
/* void fct(); -> void fct_foo(); */

using namespace Opencxx;
class RenameClass : Class
{
  // ...
};

void RenameClass::TranslateMemberFunction(Environment* env, Member& m)
{
  // Get the function name.
  Ptree* name = m.Name();
  // Set the new function name.
  // `name` refers to the variable declared above.
  m.SetFunctionBody(Ptree::qMake("`name`_foo"));
}
```

Add preconditions and postconditions to C++:

```
/*
   void my_class::pre_fct();
   void my_class::post_fct();

   void my_class::fct() { fbody }
   ->
   void my_class::fct() {
     pre_fct();
```

```
    { fbody }
    post_fct();
  }
*/

using namespace Opencxx;
class ContractClass : Class
{
  // ...
};

void ContractClass::TranslateMemberFunction(Environment* env, Member& m)
{
  // Get the function body.
  Ptree* fbody = m.FunctionBody();

  // Make the name of the {pre,post}conditions that match the function name.
  Ptree* precond_name = Ptree::Make("pre_%p", m.Name());
  Ptree* postcond_name = Ptree::Make("post_%p", m.Name());

  std::string code("{");
  // If the precondition function exists, add a call to it.
  if (LookupMember(precond_name))
    code += "`precond_name`();";

  // Add the old function body.
  code += "`fbody`";

  // If the postcondition function exists, add a call to it.
  if (LookupMember(postcond_name))
    code += "`postcond_name`();";

  code += '}';
  // Set the new function body.
  m.SetFunctionBody(Ptree::qMake(code.c_str()));
}
```

2.2 CodeBoost

CodeBoost (Bagge, 2003) is a C++ source-to-source transformation framework. Like TRANSFORMERS, it relies heavily on the STRATEGO/XT toolkit.

Quoting their homepage (Bagge, 2004):

> *CodeBoost is a tool for source-to-source transformation and optimisation of C++ programs. It is intended to be used as a test-bed for various high-level optimisations [...]*
>
> *CodeBoost was created mainly for use with the Sophus numerical library.*

A notable difference between CodeBoost and TRANSFORMERS is that full compliance with the C++ standard is not the primary objective. It works only on the subset of the C++ language that is required for Sophus, the target library, using OpenC++.

Chapter 3

Proteus

Proteus (Waddington and Yao, 2005) is a C/C++ code transformation framework used by Alcatel-Lucent:

> *Proteus is a programmable system developed by Bell Labs for performing high-fidelity transformations on C/C++ code. It automates tedious and repetitive software modification tasks*

Proteus uses a proprietary language called YATL (Yet Another Transformation Language) to describes the transformations to apply.

Their are only few informations about it or how it works available.

Chapter 4

Mozilla Rewriting and Analysis tools

4.1 Pork

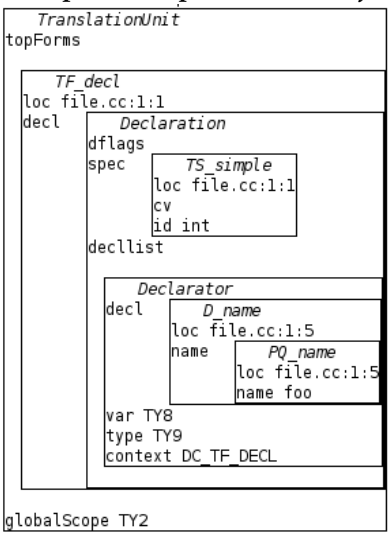
Elkhound Elkhound (McPeak, 2002) is a parser generator, similar to Bison, but using the GLR parsing algorithm in order to be able to work with any context-free grammar.
Elkhound is written in C++ and can produce parsers either in C++ or OCaml.

Elsa Elsa is an Elkhound-based C++ parser written in C++. It is not fully standard compliant, but according to its authors, it is already pretty usable:

> *Elsa can parse most C++ "in the wild". It has been tested with some notable large programs, including Mozilla, Qt, ACE, and itself.*

Elsa, compared to compiler parsers like the g++ one, preserves much information and is easy to extend. This makes it an interesting choice for C++ manipulation.

Example of output tree for *int foo;*



Oink Oink (Wilkerson et al., 2008) is a set of tools built around Elsa for C++ manipulation and more specifically static analysis.

It was supposed to be a central place to coordinate all Elsa based developments, but unfortunately it is not actively maintained.

Pork Pork (Glek, 2008b) is a set of tools used by Mozilla to rewrite source code. The main author is Taras Gleck who works at Mozilla Corporation on static analysis and automatic refactoring of the Mozilla code base.

Pork is based on the Elsa parser and on MCPP (Matsu, 2006). MCPP is a preprocessor, similar to `revcpp`, that adds comments about macro expansions, so that the process can be reverted.

Due to the lack of reactivity of the Oink community and the difficulties in getting patches applied upstream, Pork now contains a fork of Oink.

Examples

- Outparamdel rewrites code using outparameters to utilize the return value instead.
- Prcheck is a static analysis tool which produces patches as output. It is useful for correcting bugs that resulting PRBool behaving more like an int than a C++ bool.
- Garburator rewrites nsCOMPtrs allocated on the stack into stack variables to assist the XPCOMGC project.

4.2 Dehydra/Treehydra

Dehydra is a static analysis tool created by Mozilla.

Quoting from the main page (Glek, 2008a):

> *Dehydra is a lightweight, scriptable, general purpose static analysis tool capable of application-specific analyses of C++ code. In the simplest sense, Dehydra can be thought of as a semantic grep tool.*

Dehydra uses the parser of g++. It is developed as a set of patches to apply to the source code of g++. These patches add additional command line parameters to g++ so that it can accept plugins and plugin arguments.

A plugin including SpiderMonkey (Eich, 2000), the JavaScript engine of Mozilla Firefox, is then given to g++ at run time. The plugin converts the AST of g++ into a structure that is made available to the scripting engine and accepts a script written in the JavaScript language. The script is able to do simple static analysis tasks and produce compiler warning or error.

Dehydra is very handy because it does not take a long time to start using it. Once you have a first script, it is easy to learn by trial and error. Dehydra scripts do not need to be compiled, so the only thing to do to retry after a modification is to relaunch the analysis on the C++ file.

This kind of analysis tools is very useful to enforce project specific coding policies.
Dehydra is run as part of the compilation process so it is well suited for being executed automatically in a build farm.

It has been used to spot some places where to do transformations. The transformations were later done using a *sed* script.

Chapter 5

TRANSFORMERS

5.1 A State of TRANSFORMERS

TRANSFORMERS aims at being a C++ program manipulation framework, that is, a set of tools that would make it easy to create programs that do some processing on C++ code.

5.1.1 Motivation of TRANSFORMERS

When the project started, the inception was to create tools that would simplify the development of other projects of the LRDE. The dominant field of research at the LRDE is performance and genericity. Both Olena (Duret-Lutz, 2000) and Vaucanson (Lombardy et al., 2003) make heavy use of template meta-programming techniques. While template meta-programming is very good to produce generic and efficient code, it often leads to writing code which is very hard to understand code and thus error prone.

TRANSFORMERS was viewed as a way to reduce the complexity of the generic code written for these projects. The complexity of the generic and efficient code can be hidden behind new constructs added to the language, but these constructs need to be desugared before giving the program to a regular compiler. TRANSFORMERS should fulfill this task: convert a C++ program containing experimental new language features into a regular C++ program.

5.1.2 SGLR / SDF

The parser we use is SGLR (Visser, 1997b), which means Scannerless Generalized LR parser.

Generalized, in this context, means that grammars that are ambiguous and thus not suitable for standard LR parsers are acceptable. Instead of producing a single parse tree, the parser will produce a parse forest with all possible derivations of the input. Filtering the output to keep only a single tree, the "right parse" of the input program remains to be done. Filtering this output is the "disambiguation process".

Scannerless means that the lexer and parser are not two separate components: the tokens are defined directly in the grammar.
SDF, Syntax Definition Formalism (Visser, 1997a) is the declarative language used to define both the lexical and context-free syntax of the language that is to be parsed with SGLR. SDF

offers some basic disambiguation facilities like prioritizing or rejecting some productions.

5.1.3 STRATEGO/XT

STRATEGO/XT is the combination of the STRATEGO language with a set of tools (like ATerm libraries or SGLR) that are indispensable to use it.

STRATEGO is a domain-specific language dedicated to program transformation. It works by applying rewriting strategies or rules on terms and is well suited to manipulate trees. The standard input and output format of STRATEGO is ATerm.

TRANSFORMERS is based on the STRATEGO/XT bundle of tools and learning how to use them is required to work with TRANSFORMERS.

Giving a complete overview of the STRATEGO language is far beyond the objectives of this technical report and several good papers (Bravenboer et al., 2008) on the subject are available so let us move on.

5.1.4 The TRANSFORMERS pipeline

Figure 5.1: Source to source transformation pipeline

The TRANSFORMERS' pipeline defines the way the framework is meant to be used. The process is composed of successive calls to several tools, piped with each other, hence the "pipeline".

- First, the C++ source code is preprocessed. This is currently done with a regular preprocessor (the g++ one for instance) but `revcpp` should replace it as soon as it is finished.
- Then, SGLR is called on the preprocessed file and produces a parse forest.
- We need to reduce the parse forest to a single parse tree. This is the disambiguation process that is realized with attribute grammars (David, 2004) (more on this in Section 5.1.5).
- Once the disambiguation is completed, it is possible to apply transformations! Transformations are STRATEGO programs that rewrite parts of the parse tree in order to change some properties of the code. Several tools can be chained.

- At the end of the pipeline, we pretty print the result to go back to a C++ text source file.

- In the future, we may call `revcpp` again to undo the effects of the preprocessing and be able to produce patches against the original source files.

For users of TRANSFORMERS, the preprocess, parse and disambiguate steps are regrouped in a single tool, the parser (see Figure 5.1). It is currently called `parse-cxx-ng`.

5.1.5 TRANSFORMERS' Attribute Grammars

An Attribute Grammar (Knuth, 1968) system was implemented for STRATEGO/XT. It decorates all the production nodes in parse trees with attributes. There is an automatic attribute propagation (Demaille et al., 2008) for three supported attribute types: Left-Right, Top-Down and Bottom-Up. Left-Right attributes are especially useful to collect symbols and build a symbol table while we traverse the tree. The Attribute Grammar system is described in details in several reports (David, 2004; Demaille et al., 2008; Pierron, 2007).

5.2 Writing a C++ grammar extension

You first need to create an empty project. There is a script called *crap* (Kalleberg, 2008) provided in the strategoxt-utils package that can do that for you. Used with the modified templates for TRANSFORMERS C++ Tools, it will generate all the Autotools files, with the correct dependencies on STRATEGO. After this step we will be able to write a grammar extension using SDF and the attribute grammars.

5.2.1 SDF

SDF is a language for defining syntax. We use it in TRANSFORMERS because of its modularity. To extend the C++ grammar, we first need to declare our module and import the C++ grammar.

```
module MyExtension
imports
  Cxx

exports
  context-free syntax
```

Then we can write the production rules. The C++ grammar in TRANSFORMERS is exactly the one from the C++ standard. Note that we use CamelCase instead of snake_case for the symbol names.

The SDF syntax is easy to understand. For example, if we want to add a new rule to be able to parse something like that:

```
namespace class Foo {};
```

We can just use this:

```
"namespace" ClassHead "{" Declaration* "}" → Declaration
```

ClassHead and *Declaration* come from the C++ grammar. The * means that we want a list of *Declaration*.

However, when we want to override another rule, we can face a problem. For example, let us take a rule from the C++ grammar.

```
DirectDeclarator "(" ParameterDeclarationClause ")" CvQualifierSeq? ExceptionSpecification?
  → DirectDeclarator
```

The *?* indicates that the symbol is optional.

If we want to add a precondition and postcondition, we may be tempted to do something like this:

```
DirectDeclarator "(" ParameterDeclarationClause ")" CvQualifierSeq? ExceptionSpecification?
PreCondition? PostCondition?          → DirectDeclarator

DirectDeclarator "(" ParameterDeclarationClause ")" CvQualifierSeq? ExceptionSpecification?
→ DirectDeclarator { reject }
```

However, *reject* will in fact also reject the first rule. It happens because the path in the graph is tagged to be rejected. Since the tokens that we add are in the last position, the parser has already rejected the production rule before it actually try to parse the *PreCondition*.

Another way to deal with it is to use the attribute grammar system, which we will see in the next section, to always reject the rule from the standard C++ grammar. This is a bad idea,

because it has an extra cost and it prevents us from using the concrete syntax easily. We can easily rewrite the grammar in another way that works out of the box:

```
DirectDeclarator "(" ParameterDeclarationClause ")" CvQualifierSeq? ExceptionSpecification?
PrePost          → DirectDeclarator

PreCondition → PrePost
PostCondition → PrePost
PreCondition PostCondition → PrePost
```

5.2.2 Attribute Grammar

TRANSFORMERS uses an attribute grammar system to disambiguate the AST. It decorates the symbols with attributes. The values of the attributes are computed from expressions written in STRATEGO that are evaluated. If the special attribute *ok* is invalid, the symbol and its sub-tree are pruned.

```
A → B
{attributes(disamb:
  root.ok := !0
)}
```

```
A → B
{attributes(disamb:
  root.ok := !invalid()
)}
```

Some attributes are propagated all across the tree. For example, the attributes starting with *lr_* are propagated by a prefix traversal. Some others are propagated manually. When writing a grammar extension, you will likely encounter errors because of you have not declared an attribute that a parent is trying to access.

To improve modularity, there is a way to import all the attribute code from another production rule. This is useful for example in the example of the previous section: we can import them from the C++ grammar.

```
DirectDeclarator "(" ParameterDeclarationClause ")" CvQualifierSeq? ExceptionSpecification?
PrePost          → DirectDeclarator
{ inherit attributes(
    DirectDeclarator "(" ParameterDeclarationClause ")"
    CvQualifierSeq? ExceptionSpecification?
    → DirectDeclarator
)}
```

This way, we do not have to duplicate the code from the C++ grammar, and it gets updated when the C++ grammar is modified.

5.3 Writing a transformation

Now we will explain how to make a transformation with TRANSFORMERS. This offers a way to change a C++ extended source code into standard C++, or to do tedious refactoring of a large project.

There are two ways to perform a transformation:

- using the attribute system
- in STRATEGO only

We will now explain and compare them.

5.3.1 Term rewriting with STRATEGO only

We can write a transformation in STRATEGO using either *abstract syntax* or *concrete syntax*. The former is written using production constructor name to try to match the AST, whereas the later is written in the language we are trying to transform. The *concrete syntax* is in fact desugared into *abstract syntax* by the STRATEGO compiler.

Abstract Syntax

Here is a transformation using *abstract syntax* to replace every *return x;* where x is an integer by *return 42;*.

```
JumpStatement(
  return(
    Some(
      Expression2(
        [ ConditionalExpression(
            LogicalOrExpression(
              LogicalAndExpression(
                InclusiveOrExpression(
                  ExclusiveOrExpression(
                    AndExpression(
                      EqualityExpression(
                        RelationalExpression(
                          ShiftExpression(
                            AdditiveExpression(
                              MultiplicativeExpression(
                                PmExpression(
                                  CastExpression(
                                    UnaryExpression(
                                      PostfixExpression(
                                        PrimaryExpression(
                                          Literal(IntegerLiteral(INTEGER-LITERAL(x)))
                                        )
))))))))))))))) ])))))

→

JumpStatement(
  return(
    Some(
      Expression2(
        [ ConditionalExpression(
            LogicalOrExpression(
              LogicalAndExpression(
                InclusiveOrExpression(
                  ExclusiveOrExpression(
                    AndExpression(
                      EqualityExpression(
```

```
                        RelationalExpression(
                          ShiftExpression(
                            AdditiveExpression(
                              MultiplicativeExpression(
                                PmExpression(
                                  CastExpression(
                                    UnaryExpression(
                                      PostfixExpression(
                                        PrimaryExpression(
                                          Literal(IntegerLiteral(INTEGER-LITERAL(42)))
                                        )
))))))))))))))) ])))))
```

It is tedious to write this kind of rule. Moreover, adding or removing rules to the grammar can lead to a constructor name modification. We cannot accept that because it prevents us to apply different transformations with grammar extensions at the same time.

The rules are applied on the parse tree using traversal strategies such as *innermost* or *bottomup*.

Concrete Syntax

The *concrete syntax* extends the STRATEGO language itself. This is done by creating a .meta file that contains the name of the grammar file.

For example:

```
Meta ([Syntax ("Ret42Cxx")])
```

Listing 5.1: In the file Ret42.meta

```
exports
  context-free syntax
    "|[" JumpStatement "]|"      -> \StrategoTerm { cons("ToTerm"), prefer }

%% [...]

exports
  variables
    x                            -> IntegerLiteral       { prefer }
```

Listing 5.2: In the file Ret42Cxx.sdf

We are now able to write the transformation:

```
|[ return x; ]| -> |[ return 42; ]|
```

This version is easier to read and write than the abstract syntax version, but this have some drawbacks. Since we are using C++ grammar inside STRATEGO, and C++ grammar is ambiguous, we sometimes need to disambiguate the rules by hand. We can achieve this using *with*.

5.3.2 Term rewriting with the attribute grammar system

As we already saw in Section 5.2.2, there is a special attribute *ok* to tell whether the evaluator should prune the node. There is another special attribute named *replace*. When it is present, the current node is replaced by the value of this attribute. Therefore, we can use it to write some transformations in STRATEGO.

This example replaces the current node by its child:

```
A -> B
{attributes(disamb:
  root.replace := !A
)}
```

Note At the moment, we cannot use concrete syntax in the transformations done with the attributes, because of a problem with STRATEGO. When the STRATEGO compiler desugars the *concrete syntax*, it generates a code that tries to match exactly a part of the parse tree where we store the attributes.

5.3.3 Comparison

Attributes When we make a transformation with the attribute grammar, we have access to innumerable informations that we lose later. This is very useful to perform transformations that need to lookup symbols. If we want to lookup a symbol with a transformation in regular STRATEGO, we will have store the attributes required for the lookup, like we do in ContractC++:

```
"invariant" "{" a:Assertion* "}" → MemberDeclaration
{attributes(disamb:
  local.dummy := extern
  root.replace :=
    <set-attribute> (<id>, "disamb", "lr_classtable_inh",
                     root.lr_classtable_inh)
  ; <set-attribute> (<id>, "disamb", "lr_ns_inh",
                     root.lr_ns_inh)
  ; <set-attribute> (<id>, "disamb", "lr_table_inh",
                     root.lr_table_inh)
  ; <set-attribute> (<id>, "disamb", "lr_template_parameters_inh",
                     root.lr_template_parameters_inh)

  ; attr-eval
  ; <skip-attribute> (<id>, "disamb", "replace")
  ; put-results-opt
  ; <set-anno> (<id>, root.lr_ns_inh)
)}
```

And then, in the transformation:

```
trans =
  has-annos
  <
  {ns:
    where(get-annos => ns)
  ; rm-annotations
%% your code here
  }
  + id
```

This can be done only with the AsFix format. (The TRANSFORMERS Attribute Grammar system already uses the annotations in the AST mode, we would override them.) Moreover some tools does not handle very well the annotations.

Bridge When the attribute system replaces a tree by another, the new tree should have been evaluated. A common problem occurs when we try to replace a node with a node of the same kind. To prevent an infinite loop, we use the *bridge* system. This is a bit technical, people interested in this should have a look to
'`generic-tools/sdf-attribute/tests/exec/many-instantiation-traversal.esdf`'.

Time Each change in a transformation written with the attribute system will trigger a full recompilation of the grammar. However, this solution is faster at runtime, because the evaluation is done at the same time the other attributes are being evaluated, in the same traversal. With the transformation after, at least one extra pass is required depending on the traversal strategy chosen.

Conclusion

TRANSFORMERS is not mature enough to write real transformations on C++ for now. Some information such as the type of variables are missing, but would be useful for some extensions. For example on a large project someone may rename a member function, and do an extension to automatically change the name of the calls to this function without changing the function calls with the same name but from other classes.

Because TRANSFORMERS sticks to the C++ grammar, it is very painful to the extension and transformation developers to use it. Have a look to Appendix A for example: this is the Stratego code required to rename a function. A first part do only the traversal through the AST, the second part is made of convenient strategies to rebuild a valid tree with the new name, and the last actually change the name.

Moreover, it requires to know a lot of technologies before actually being able to use it efficiently, such as:

- SDF (and how SGLR works behind the scene.)
- STRATEGO
- Attribute grammars
- the C++ standard

Future Work A framework like Centaur would really help to write new transformations. Someone without a very good knowledge of STRATEGO and the C++ standard would be able to write a simple transformation with Centaur. However, it is not easy to design something like Centaur (Raud, 2008).

Meanwhile, it would be useful to have tools to visualize the grammar graph, or to check that the output tree of a transformation is a valid C++ tree.

References

Bagge, A. H. (2004). CodeBoost.org. http://www.codeboost.org/.

Bagge, O. S. (2003). CodeBoost: A framework for transforming C++ programs. Master's thesis, University of Bergen, P.O.Box 7800, N-5020 Bergen, Norway.

Borghi, A., David, V., and Demaille, A. (2006). C-Transformers — A framework to write C program transformations. *ACM Crossroads*, 12(3). http://www.acm.org/crossroads/xrds12-3/contractc.html.

Bravenboer, M., Kalleberg, K. T., Vermaas, R., and Visser, E. (2008). Stratego/XT 0.17. a language and toolset for program transformation. *Science of Computer Programming*. (To appear).

Chiba, S. (2004). Open C++, a c++ frontend library (lexer+parser+dom/mop) and source-to-source translator. http://opencxx.sourceforge.net/.

David, V. (2004). Attribute grammars for C++ disambiguation. Technical report, LRDE.

Demaille, A., Durlin, R., Pierron, N., and Sigoure, B. (2008). Automatic attribute propagation for modular attribute grammars. Technical report, EPITA Research and Development Laboratory (LRDE).

Despres, N. (2004). C++ transformations panorama. Technical report, EPITA Research and Development Laboratory (LRDE).

Duret-Lutz, A. (2000). Olena: a component-based platform for image processing, mixing generic, generative and OO programming. In *Proceedings of the 2nd International Symposium on Generative and Component-Based Software Engineering (GCSE)—Young Researchers Workshop; published in "Net.ObjectDays2000"*, pages 653–659, Erfurt, Germany.

Eich, B. (2000). SpiderMonkey, Gecko's JavaScript engine written in C. https://developer.mozilla.org/en/SpiderMonkey.

Glek, T. (2008a). Dehydra is a lightweight, scriptable, general purpose static analysis tool capable of application-specific analyses of C++ code. https://developer.mozilla.org/en/Dehydra.

Glek, T. (2008b). Pork, MCPP, Oink and Elsa... What's going on. http://blog.mozilla.com/tglek/2008/07/18/pork-mcpp-oink-and-elsawhats-going-on/.

Kalleberg, K. T. (2008). Stratego create-a-project (crap). http://strategoxt.org/Stratego/SimpleProjectCreation.

Knuth, D. E. (1968). Semantics of context-free languages. *Journal of Mathematical System Theory*, pages 127–145.

Lombardy, S., Poss, R., Régis-Gianas, Y., and Sakarovitch, J. (2003). Introducing Vaucanson. In Springer-Verlag, editor, *Proceedings of Implementation and Application of Automata, 8th International Conference (CIAA)*, volume 2759 of *Lecture Notes in Computer Science Series*, pages 96–107, Santa Barbara, CA, USA.

Matsu, K. (2006). mcpp – a portable C preprocessor. http://mcpp.sourceforge.net/.

McPeak, S. (2002). *Elkhound: A Fast, Practical GLR Parser Generator*. http://www.cs.berkeley.edu/~smcpeak/elkhound/.

Ordy, V. (2008). Implementing a C++ extension with Transformers: `class namespace`. Technical report, EPITA Research and Development Laboratory (LRDE).

Ordy, V. (2009). Adding Contracts to C++ with Transformers. Technical Report 0901, EPITA Research and Development Laboratory (LRDE).

Pierron, N. (2007). Formal definition of the disambiguation with attribute grammars. Technical report, EPITA Research and Development Laboratory (LRDE).

Raud, C. (2008). Centaur: A generic framework simplifying C++ transformation. Technical report, EPITA Research and Development Laboratory (LRDE).

Visser, E. (1997a). A family of syntax definition formalisms. Technical Report P9706, Programming Research Group, University of Amsterdam.

Visser, E. (1997b). Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam.

Waddington, D. G. and Yao, B. (2005). High fidelity C++ code transformation. In *Proceedings of the 5th workshop on Language Descriptions, Tools and Applications (LDTA 2005)*, Electronic Notes in Theoretical Computer Science, Edinburgh University, UK.

Wilkerson, D. S., Chen, K., and McPeak, S. (2008). Oink: a Collaboration of C++ Static Analysis Tools. http://www.cubewano.org/oink/.

Appendix A

Renaming a function in Stratego

```
// Traversal Rules
rules
  FunTraversalIdExpression(s) :
    |IdExpression[ ui ]| → |IdExpression[ ui' ]|
  where <FunTraversalUnqualifiedId(s)> ui ⇒ ui'

  FunTraversalUnqualifiedId(s) :
    |UnqualifiedId[ ti ]| → |UnqualifiedId[ ti' ]|
  where <FunTraversalTemplateId(s)> ti ⇒ ti'

  FunTraversalUnqualifiedId(s) :
    |UnqualifiedId[ ui ]| → |UnqualifiedId[ ui' ]|
  where not(!ui;?|UnqualifiedId[ ti ]|)
      ; <s> ui ⇒ ui'

  FunTraversalQualifiedId(s) :
    |QualifiedId[ nns ui ]| → |QualifiedId[ nns ui' ]|
  where <FunTraversalUnqualifiedId(s)> ui ⇒ ui'

  FunTraversalQualifiedId(s) :
    |QualifiedId[ nns template ui ]| → |QualifiedId[ nns template ui' ]|
  where <FunTraversalUnqualifiedId(s)> ui ⇒ ui'

  FunTraversalQualifiedId(s) :
    |QualifiedId[ :: nns ui ]| → |QualifiedId[ :: nns ui' ]|
  where <FunTraversalUnqualifiedId(s)> ui ⇒ ui'

  FunTraversalQualifiedId(s) :
    |QualifiedId[ :: nns template ui ]| → |QualifiedId[ :: nns template ui' ]|
  where <FunTraversalUnqualifiedId(s)> ui ⇒ ui'

  FunTraversalQualifiedId(s) :
    |QualifiedId[ :: ident ]| → |QualifiedId[ :: ident' ]|
  where <s> ident ⇒ ident'

  FunTraversalQualifiedId(s) :
    |QualifiedId[ :: ofi ]| → |QualifiedId[ :: ofi' ]|
  where <s> ofi ⇒ ofi'

  FunTraversalQualifiedId(s) :
    |QualifiedId[ :: ti ]| → |QualifiedId[ :: ti' ]|
  where <s> ti ⇒ ti'

  FunTraversalTemplateId(s) :
    |TemplateId[ ident < tal > ]| → |TemplateId[ ident' < tal > ]|
  where <s> ident ⇒ ident'

  FunTraversalTemplateId(s) :
    |TemplateId[ ident < > ]| → |TemplateId[ ident' < > ]|
  where <s> ident ⇒ ident'
```

```
  FunTraversalIdExpression(s) :
    |IdExpression[ qi ]| → |IdExpression[ qi' ]|
  where <FunTraversalQualifiedId(s)> qi ⇒ qi'

  FunTraversalDeclaratorId(s) :
    |DeclaratorId[ idexp ]| → |DeclaratorId[ idexp' ]|
  where <FunTraversalIdExpression(s)> idexp ⇒ idexp'

  FunTraversalClassName(s) :
    |ClassName[ ident ]| → |ClassName[ ident' ]|
  where <s> ident ⇒ ident'

  FunTraversalClassName(s) :
    |ClassName[ ti ]| → |ClassName[ ti' ]|
  where <FunTraversalTemplateId(s)> ti ⇒ ti'

  FunTraversalTypeName(s) :
    |TypeName[ cn ]| → |TypeName[ cn' ]|
  where <FunTraversalClassName(s)> cn ⇒ cn'

  FunTraversalTypeName(s) :
    |TypeName[ ident ]| → |TypeName[ ident' ]|
  where <s> ident ⇒ ident'

  FunTraversalDeclaratorId(s) :
    |DeclaratorId[ :: nns tn ]| → |DeclaratorId[ :: nns tn' ]|
  where <FunTraversalTypeName(s)> tn ⇒ tn'

  FunTraversalDeclaratorId(s) :
    |DeclaratorId[ nns tn ]| → |DeclaratorId[ nns tn' ]|
  where <FunTraversalTypeName(s)> tn ⇒ tn'

  FunTraversalDeclaratorId(s) :
    |DeclaratorId[ :: tn ]| → |DeclaratorId[ :: tn' ]|
  where <FunTraversalTypeName(s)> tn ⇒ tn'

  FunTraversalDeclaratorId(s) :
    |DeclaratorId[ tn ]| → |DeclaratorId[ tn' ]|
  where <FunTraversalTypeName(s)> tn ⇒ tn'

// Internal Strategies to rebuild the function name
strategies
  // internal
  NonDigit =
   !appl(
      prod(
        [char-class([range(65, 90), 95, range(97, 122)])]
        , lex(sort("NON-DIGIT"))
        , no-attrs()
      )
      , [<id>]
    )

  // internal
  Digit =
   !appl(
      prod(
        [char-class([range(48, 57)])]
        , lex(sort("DIGIT"))
        , no-attrs()
      )
      , [<id>]
    )

  // internal
  make-ID =
    ?(non-digit, idcharstar)
  ; !appl(
      prod(
        [lex(sort("ID"))]
      , cf(sort("ID"))
      , no-attrs()
```

```
      )
    , [ appl(
          prod(
            [ lex(sort("NON-DIGIT"))
            , lex(iter-star(alt(sort("NON-DIGIT"), sort("DIGIT"))))
            ]
          , lex(sort("ID"))
          , no-attrs()
          )
        , [ non-digit, idcharstar]
      )])

  // internal
  make-char-iter1 =
    !appl(
       prod(
         [lex(alt(sort("NON-DIGIT"), sort("DIGIT")))]
       , lex(iter(alt(sort("NON-DIGIT"), sort("DIGIT"))))
       , no-attrs()
       )
     , [<id>]
     )

  // internal
  make-char-iter2 =
    ?(c1, c2)
  ; !appl(
       prod(
         [ lex(iter(alt(sort("NON-DIGIT"), sort("DIGIT"))))
         , lex(iter(alt(sort("NON-DIGIT"), sort("DIGIT"))))
         ]
       , lex(iter(alt(sort("NON-DIGIT"), sort("DIGIT"))))
       , attrs([assoc(left())])
       )
     , [ c1, c2 ]
    )

  // internal
  make-char-iter-star1 =
    !appl(
       prod(
         [lex(iter(alt(sort("NON-DIGIT"), sort("DIGIT"))))]
       , lex(iter-star(alt(sort("NON-DIGIT"), sort("DIGIT"))))
       , no-attrs()
       )
     , [<id>]
     )

  str-to-id-list =
    (?(c,[])
  ; (<is-alpha; NonDigit; make-char-iter1> c
   <+ <Digit; make-char-iter1> c))
  <+ (?(c, idchar)
   ; (<is-alpha; NonDigit> c ⇒ non-digit
   ; <make-char-iter2> (<id>, idchar)
   <+ <Digit> c
   ; <make-char-iter2> (<id>, idchar)))

  str-to-id =
    explode-string
  ; ?[x|xs]
  ; <is-alpha; NonDigit> x ⇒ non-digit
  ; <foldr(id, str-to-id-list); make-char-iter-star1> xs
  ; <make-ID> (non-digit, <id>) ⇒ idstr
  ; !|Identifier[ idstr ]|

// The strategies to rename functions
  fun-rename(s) =
    FunTraversalDeclaratorId(get-string; s; str-to-id)

  fun-rename(|new-name) =
    where (<str-to-id> new-name ⇒ ident)
  ; FunTraversalDeclaratorId(!ident)
```