Design and Implementation of a Language-Complete C++ Semantic Graph

Edward B. Duffy and Brian A. Malloy,
School of Computing
Clemson University
Clemson, SC 29634, USA
{eduffy,malloy}@cs.clemson.edu

Abstract

In this paper, we describe a system, Hylian, for construction of a language-complete abstract semantic graph that can be used for statement-level analysis, both static and dynamic, of a C++ application. We begin by extending the GNU gcc parser to generate parse trees in XML format for each of the compilation units in a C++ application. We then provide verification that the generated parse trees are structurally equivalent to the code in the original C++ application. We use the generated parse trees, together with an augmented version of the gcc *test suite, to recover a grammar for the C++ dialect that we parse. We use the recovered grammar to generate a schema for further verification of the parse trees and evaluate the coverage provided by our C++ test suite. We then extend the parse tree, for each compilation unit, with semantic information to form an abstract semantic graph, ASG, and then link the ASGs for all of the compilation units into a unified ASG for the entire application under study.*

1. Introduction

The process of software maintenance, including comprehension, modification, and refactoring of complex multi-paradigm systems, requires extensive and detailed information about the system under study. However, software artifacts that provide this information are frequently unavailable and for large, open-source applications, they are virtually nonexistent. Thus, much of the research in software maintenance has focused on the development of inquiry and analysis tools to automate the process of generating information to improve comprehension, and to facilitate analysis, modification and testing of the application under study.

However, the C++ language has proven to be particularly problematic for maintenance engineers interested in developing tools to facilitate analysis and modification of C++ applications. The difficulty in developing tools for C++ is mostly due to the scope and complexity of the language; for example, the grammatical representation of C++ has been shown to be larger and more complex than other, commonly used languages [15]. A particularly perplexing problem for C++ maintenance engineers entails the correct recognition of the language constructs as specified in the ISO standard, for example class template partial specializations and argument-dependent lookup [12, §A.8]. Moreover, statement-level analysis, which is required for pointer analysis and program slicing [2, 10, 11, 17], relies on the correct recognition of expressions such as *expression*, *postfix-expression*, and *unary-expression* [12, §A.4].

Nevertheless, the C++ language is frequently used and recently has been shown to outperform other, commonly used, languages by a large margin [4]. Therefore, to support software maintenance and other software engineering efforts for the C++ language, it's important to develop analysis tools for the language.

In this paper, we describe our extension of a system, Hylian, whose initial phases were described in references [6, 7, 8, 9, 13]. We describe our construction of an abstract semantic graph (ASG), that is *language-complete*, which is a semantic graph that includes all aspects defined in the language standard. For this paper, our focus is an ASG that is language-complete for the 2003 revised ISO C++ language standard; our current implementation does not include the C++11 standard, recently ratified [1]. A semantic graph that completely defines C++ must include: evaluation and lookup of constants, full type resolution for names, determination of type equivalency and type promotion, full and partial template instantiation, operator overload resolution, function and method resolution including argument dependent name lookup, implicit method invocation commonly used in class construction and destruction. Our current implementation includes all of the above.

In the next section, we provide an overview of the three phases of our system and then provide details about construction of ab-

stract semantic graphs for C++ applications. In Section 4, we provide some statistics that summarize efficiency considerations for the construction process, including some size results for an ASG for a popular video game. Finally, in Section 5 we draw conclusions.

2. Overview of The System

In this section, we describe Hylian, the system that we developed to empower a researcher or developer to perform statement-level analysis of a program written in a gcc dialect of the C++ language. Figure 1 is an overview of Hylian, which consists of three phases: (1) Parse tree extraction and grammar recovery, (2) development and generation of an abstract semantic graph (ASG), and (3) transformation of the ASG.

The first phase is summarized at the top of Figure 1 by the rectangles labeled I, Parse Tree Generation; II, Parse Tree Post-processing; III, Parse Tree Validation; and IV, Grammar Recovery. In module I of the first phase, we generate parse trees for each compilation unit in the application by augmenting the gcc C++ parser with probes whose output generates an XML representation for each of the respective parse trees. The *gcc* parser performs tentative parsing and then backtracks to recover from incorrectly chosen alternatives. Thus, in module II, the parse subtrees that were emitted as part of an incorrect alternative are deleted and the committed subtrees are written to a file in XML format. In module III, the generated parse trees are validated. Validation entails recovery of a grammar for the gcc C++ grammar, module IV, and then use of the grammar to generate a schema, in Relax NG format.

The second phase, generation of an ASG, is summarized by the two rectangles, V, Abstract Semantic Graph generation, and VI, ASG verification, are summarized in the middle of Figure 1. The focus of this paper is ASG generation, and this will be described in more detail in Section 3.

There are many advantages attached to the use of a language-complete system, such as Hylian, the language-complete ASG construction system that we describe, One of these is the subsequent feasibility of statement level transformations and subsequent code generation of the transformed ASG. In reference [13], We show that the use of parse trees does not provide sufficient information to fully automate the process of generating interface protocols for the classes in a library and that using the language-complete Hylian ASG, the process can be fully automated and, in fact, there are even more benefits of using a Hylian ASG. However, a full explanation of these additional features is beyond the scope of this paper.

3. Methodology for ASG Construction

In this section, we provide an overview of the second phase of the construction of ASGs, illustrated in the middle of Figure 1 by the rectangles labeled V and VI. We first describe the problems in building an ASG for the C++ language and then we describe the procedure we use in Hylian to construct a language-complete ASG.

The attachment of type information to the names used in an application is fairly straightforward if the program is written in a procedural language such as C or Fortran. However, attaching semantic information to names used in applications written in a multi-paradigm, composite language [16, 3], such as C++, present unique challenges. These features include data hiding, generics, multiple inheritance and other constructs that make the attachment of semantic information much more difficult than in the processing of procedural languages. These challenges comprise the bulk of the effort required for the construction of the Hylian analysis system.

The most imposing of these challenges, name lookup, actually sounds uniquely trivial, yet a solution to this problem entails resolving type information for virtually every production in the C++ grammar. Thus, name lookup is difficult firstly because of the breadth of the problem, since the C++ grammar is one of the largest grammars in use, yet more importantly, is the most complex grammar [15]. For example, the impurity metric applied to the C++ grammar shows that, at 85%, the C++ grammar contains a considerable density in the edges in the closure of the call graph, especially compared to the grammars for C, Java and C#. The impurity metric, together with the application of McCabe's metric to the C++ grammar, provide further evidence of the complexity, and the tight coupling of the C++ grammar productions [14, 15].

As an illustrative example of the issues involved in name lookup in C++, consider resolution of the simple three-operand expression for printing "Hello World," illustrated in Figure 3.

```
int main() { std::cout ≪ "Hello, World!" ≪ std::endl; }
```

Figure 3. A Hello World **program**

The first operand is std::cout and its type is std::basic_ostream-<char>. The second operand, a string literal, is type **const char**∗. Resolving the correct implementation of **operator**≪ involves examining seventeen methods of the instantiation of std::basic_ostream, and five function templates in the std namespace, that accept a std::basic_ostream object as its first argument. The Hylian ASG construction algorithm compare the parameters for the partially-specialized function template in std::basic_ostream and find those that most closely match the types of these first two operands. When found, that function is instantiated, inserted into the ASG, and the type of the subexpression is std::basic_ostream. The full ASG for a hello world program is illustrated in Figure 2.

3.1 Parsing parse trees expressed in XML

The algorithm in Figure 4 parses the XML representation of the parse-tree of the original user application code. The parse-tree parser is a top-down SAX XML parser, but we prefer to handle the grammar productions in bottom-up fashion. Thus, to simulate a bottom-up parse we must buffer the XML tags that represent terminals and non-terminals from the original user application code, and delay the semantic actions until a complete sentence encountered. The semantic action for each production is handled by a function that accepts two lists: (1) a list of the syntax symbols, and (2) a list of their corresponding semantic values. The return value of the semantic action function is the semantic value of that

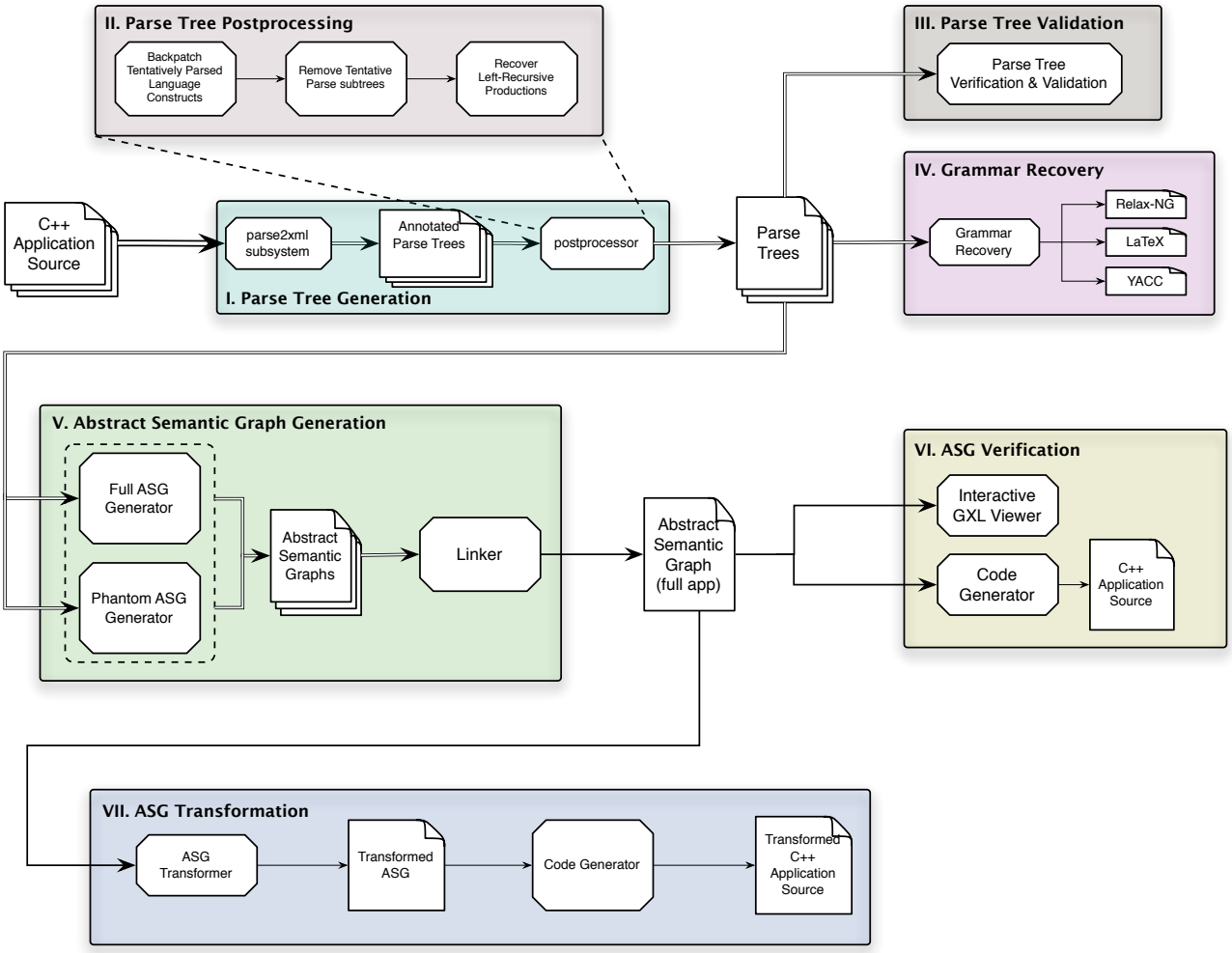

Figure 1. System Summary. This figure encapsulates the important modules in the Hylian system.

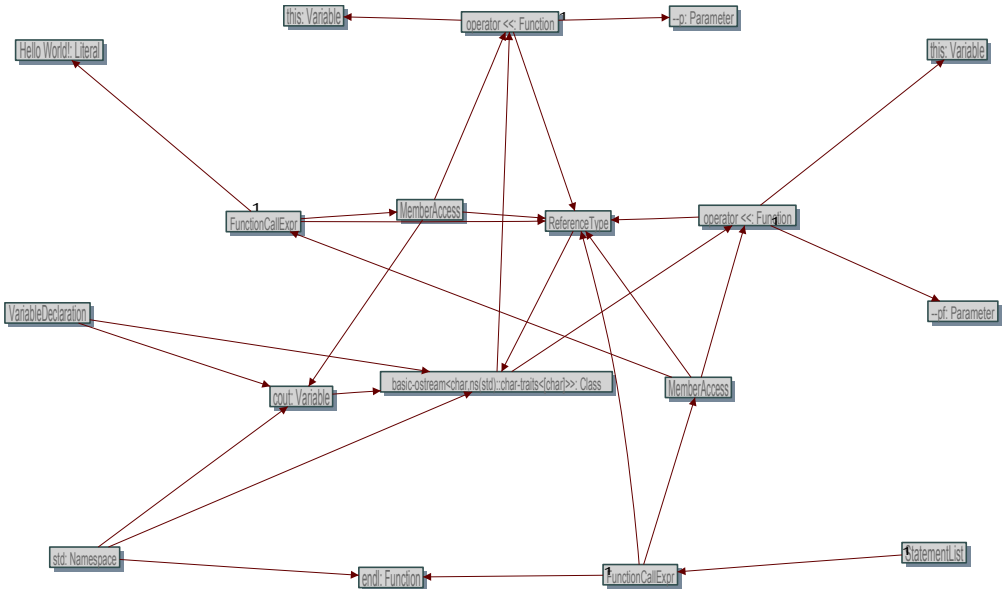

Figure 2. A language-complete Abstract Semantic Graph for a Hello World **program**

production.

To correctly handle template classes and functions, we delay total evaluation of their subtree until we encounter a use later in the parse tree. When a template definition is encountered, we must parse enough of its tree to determine its declaration all the while buffering the entire subtree for later use. Once we determine the declaration, we stop handling semantic actions until the matching close tag for the template declaration is reached. We then store the template declaration and its corresponding parse tree in a dictionary until a usage of the template requires instantiation.

3.2 Overview of ASG construction

To generate an abstract semantic graph, ASG, we first prune the generated parse trees to eliminate unnecessary non-terminals and empty productions. We then annotate the pruned parse tree with semantic information, such as the type or scope of a variable. In the case of templates, we must build an ASG representation of the instantiated template. After we have extended each of the parse trees with semantic information to produce an ASG for each compilation unit, we then *link*, or merge, the ASGs into a single ASG for the entire program. Exposition of the full algorithm for ASG construction is beyond the scope of this paper. In this section, we provide an overview of this construction process. The interested reader may consult reference [5] for details, and an algorithm, for ASG construction.

The algorithm for ASG construction uses seven data structures and a while loop that examines each XML tag in turn. The first data structure, tag, is the current XML tag. The next two data structures facilitate a bottom up parse using the tags from the top-down XML parser: syntaxStack, a stack of lists that contain terminals and non-terminals encountered in the bottom-up parse; and semanticStack, a stack of lists that contain semantic values encountered in the bottom up parse of the parse tree. The final four data structures facilitate template declaration buffering and instantiation: parsetreeBuffer, a dictionary that maps a template declaration to its respective parse tree; currentParsetreeBuffer, a buffer for a parse tree of a template so that when a template declaration is encountered the parse action is delayed until the template is instantiated and fully defined by its usage; isBuffering, a boolean indicating that the current tag is the child of a template declaration; and, finally, currentTemplateDecl, a handle for a declaration that contains information to facilitate name lookup for the currently buffered template declaration, which contains scope information, name, arguments for the template, and possibly information about parameters to a function template.

4. Results

In this section we provide some interesting results for ASGs that we have built using Hylian. Table 1 summarizes these results for three programs in our test suite: a Hello World program written in C and C++, and AlephOne, a rather large video game translated into C++ that uses the Simple Direct Media Layer (SDL) API, as well as the Lua embedded scripting engine. There are two sections of data in the table: the top section that summarizes results for the construction of Parse Trees, and the bottom section that summarizes results for the construction of ASGs. We are comparing parse trees and ASGs in the top and bottom sections to show the reduction in the size of ASGs compared to parse trees, and the reduction in size when the ASGs for each compilation unit in a program are *linked*, or merged, into a single ASG for the entire program.

In comparing the various results, we first consider the results for the three "Hello World" programs. The first column in the table describes the information in the particular row of the table, and the

	Hello World		Video Game
	In C	In C++	AlephOne
Parse Trees			
No. Compilation Units	1	1	176
No. System Terminal	2,930	132,122	30,339,853
No. User Terminal	13	16	3,441,975
No. Total Terminal	2,943	132,138	33,781,828
No. Non-Terminals	13,404	518,788	128,818,128
Max Branch Length	223	374	3,671
Avg. Branch Length	123.47	215.33	652.71
Abstract Semantic Graph			
No. Vertices	984	21,861	1,914,869
No. Vertices (linked)			478,642
No. Edges	2,098	45,444	3,632,959
No. Edges (linked)			1,003,413
No. Statements	137	2,263	1,512,638
No. Statements (linked)			115,534

Table 1. Some statistical results comparing Parse Tree and ASG construction

next three columns represent information for the "Hello World" programs written in C and C++. Our first interesting result shows that both the C and C++ versions of "Hello World" consist of a single Compilation Unit, as shown by the first row of data in the table for columns two, three and four; however, the No. System Terminals in the C version has 2,930 terminals in the parse trees, but the C++ versions have 132,122 terminals in their parse trees, so that the C++ version has almost two orders of magnitude more terminals than the C version. This is due to the fact that the C++ version of "Hello World" includes the iostream library and much of the C++ standard library. Also, the No. System Terminals in the AlephOne program is 30,339,853 (first row, last column of the table), illustrating the large number of system terminals needed for a standard video game. The remaining rows in the top section of Table 1 reflect similar comparisons.

The bottom section of Table 1 summarizes results for the ASGs that Hylian builds. The second and third columns in the bottom section compare the ASG for a full parse of a C and a C++ "Hello World" program. For example, the third row of the bottom section of the table shows that there are 984 vertices in the C version, and 21,861 vertices in the C++ version, an order of magnitude increase. Since there is only one compilation unit in each "Hello World" program, there are is only one ASG and therefore no vertices are linked; thus, the second and fourth rows in the bottom section of the table do not list any values for No. Vertices (linked) or No. Edges (linked).

5. Conluding Remarks

In this paper, we described Hylian, a system for construction of a language-complete ASG that provides statement-level analysis, both static and dynamic, of a C++ application. We performed various verification and validation metrics to the ASG, including a viewer that can visualize any graph in GXL format, to provide assurance for a developer that the ASGs that Hylian builds correctly represent the program under investigation. To evaluate space considerations for Hylian ASGs, we have provided some results that describe the size of the generated ASGs. Our future work entails using the Hylian system to perform transformations on the ASG, and generate C++ code for the transformed ASG.

6. References

[1] Iso/iec 14882:2011. *International Organization for Standardization*, February 2012.

[2] D. Binkley, S. Danicic, T. Gyimóthy, M. Harman, A. Kiss, and B. Korel. Theoretical foundations of dynamic program slicing. *Theor. Comput. Sci.*, 360(1):23–41, 2006.

[3] J. E. Denny. PSLR(1): Pseudo-Scannerless minimal LR(1) for the deterministic parsing of composite languages, 2010. PhD Dissertation.

[4] D. du Preez. C++ clear winner in google language tests, 2011.

[5] E. B. Duffy. The design & implementation of an abstract semantic graph for statement-level dynamic analysis of c++ applications, 2011. PhD Dissertation.

[6] E. B. Duffy and B. A. Malloy. A language and platform-independent approach for reverse engineering. *Proceedings of The 3rd ACIS International Conference on Software Engineering Research, Management & Applications*, August 2005.

[7] E. B. Duffy and B. A. Malloy. An automated approach to grammar recovery for a dialect of the C++ language. In *Proceedings of the Fourteenth Working Conference on Reverse Engineering*, 2007.

[8] E. B. Duffy and B. A. Malloy. An automated approach to grammar recovery for a dialect of the c++ language. In *Proceedings of the 14^{th} Working Conference on Reverse Engineering*, pages 11–20. IEEE Computer Society, 2007.

[9] J. O. H. Edward B. Duffy and B. A. Malloy. Reverse engineering interface protocols for comprehension of large C++ libraries during code evolution tasks. In *Proceedings of the The 20th International Conference on Software Engineering and Knowledge Engineering*, 2008.

[10] K. Gallagher and D. Binkley. An empirical study of

```
1   declare tag: XML tag
2   declare syntaxStack: stack
3   declare semanticStack: stack
4   declare parsetreeBuffer: dictionary
5   declare currentParsetreeBuffer: string
6   declare isBuffering: boolean
7   declare currentTemplateDecl: template declaration
8
9   push an empty list onto semanticStack and syntaxStack
10  tag ← getTag()
11  while ( more tags in XML Parse Tree ) :
12      if (tag is startTag) :
13          if ( tag is template declaration and not isBuffering ) :
14              currentParsetreeBuffer ← empty parse tree
15              currentTemplateDecl ← NULL
16              isBuffering ← TRUE
17          if ( isBuffering ) :
18              store tag in currentParsetreeBuffer
19              if (currentTemplateDecl is not NULL) :
20                  continue at top of while
21          append tag to list at top of syntaxStack
22          if ( tag is tokenTag ) :
25              // keyword, constant, symbol, or identifier
26              if ( tag is literal ) :
27                  determine representation and type
28                  append value of literal to list at top of semanticStack
29              else if ( tag is name ) :
30                  if ( tag is keyword ) :
31                      replace 'token' with actual keyword in list at top of
32                          syntaxStack (e.g., class)
33                      append NULL to list at top of semanticStack
34                  else :
35                      push identifier onto list at top of semanticStack
36              else if ( tag is symbolTag ) :
37                  replace 'token' with symbol value (e.g., replace with '+')
38                  append NULL to list at top of semanticStack
39          else :
40              append NULL to list at top of semanticStack
41      else if ( tag is endTag ) :
42          if ( isBuffering ) :
43              store tag in currentParsetreeBuffer
44          if (not isBuffering xor currentTemplateDecl is not NULL) :
45              call function to perform semantic-action on_tag_name
46                  i.e., for the tag "</class_head>" call on_class_head()
47              pop syntaxStack
48              pop semanticStack
49              replace NULL in list at top of semanticStack with result of
50                  semantic-action, i.e. the semantic value of the production
51
52          if (isBuffering and currentTemplateDecl is not NULL) :
53              add to parsetreeBuffer, where the key is the declaration, and the
54                  value is the partially created buffer (in progress).
55          if (matching close of template_declaration that started buffering) :
56              isBuffering ← FALSE
57      tag ← getTag()
```

Figure 4. The algorithm used to convert a parse tree into a bottom-up style parse

computation equivalence as determined by decomposition slice equivalence. In *WCRE '03: Proceedings of the 10th Working Conference on Reverse Engineering*, page 316, Washington, DC, USA, 2003. IEEE Computer Society.

[11] M. J. Harrold and G. Rothermel. Syntax-directed construction of program dependence graphs. In *Technical Report OSU-CISRC-5/96-TR32, Department of Computer and Information Science, The Ohio State University (OSU-CISRC-5/96- 1996)*, May 1996.

[12] ISO/IEC JTC 1. *International Standard: Programming Languages – C++*. Number 14882:1998(E) in ASC X3. ANSI, first edition, September 1998.

[13] B. A. Malloy, E. L. Lloyd, J. O. Hallstrom, and E. B. Duffy. Capturing interface protocols to support comprehension and evaluation of C++ libraries. *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*.

[14] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976.

[15] J. F. Power and B. A. Malloy. A metrics suite for grammar-based software. *Journal of Software Maintenance and Evolution: Research and Practice*, 16(6):405–426, 2004.

[16] B. Stroustrup. *The design and evolution of C++*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1994.

[17] M. Weiser. Program slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984.