An Automated Approach to Grammar Recovery for a Dialect of the C++ Language

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 present the design and implementation of a fully automated technique for reverse engineering or recovering a grammar from existing language artifacts. The technique that we describe uses only test cases and a parse tree, and we apply the technique to a dialect of the C++ language. However, given test cases and a parse tree for a language or a dialect of a language, our technique can be used to recover a grammar for the language, including languages such as Java, C, Python or Ruby.

1. Introduction

The role of programming languages in software development is well established and there has been a progression of languages to facilitate development and maintenance of the software life cycle [3, 8, 9, 22]. To promote language stability and conformance across compilers, platforms, and applications, a language standard is usually developed along with a language. However, in spite of efforts at standardization and conformance, dialects of a language are frequently developed before the language standard is even ratified, where a dialect typically contains constructs not specified by the language standard. For example, the C++ language has dialects such as *gcc*, *Visual* C++, .NET C++ and Borland C++, which do not conform to the ISO C++ language standard [3, 20]. Even Java, developed by Sun Microsystems, has dialects such as *J#* and J++ [14, 26].

Language dialects are rarely accompanied by a grammar specification and sometimes are not even accompanied by a language manual or ad hoc specification. Nevertheless, the importance of grammarware and grammar engineering has been established and a grammar for a language dialect is an important element in the construction of tools for analysis, comprehension, maintenance and visualization of applications developed in a new or existing language [12, 15]. Frequently, the only representation of a dialect is the grammar contained in the source code of a compiler; however, a *compiler grammar* is difficult to comprehend since parser generation algorithms place restrictions on the form of the compiler grammar [16, page 10].

Another problem with language dialects is that there has been little research, to date, addressing the problem of reverse engineering a grammar or language specification for a language dialect from existing language artifacts. Lämmel and Verhoef have developed a technique that uses a language reference manual and test cases to recover a grammar for a language or dialect [16]. However, their technique requires user intervention along most of the stages of recovery and some of the recovery process is manual. Bouwers et al. present a methodology for recovering precedence rules from grammars [5]. However, there is no existing technique to enable a developer to automatically reverse engineer a grammar for an existing language or language dialect.

In this paper we present the design and implementation of a technique for reverse engineering, or recovering, a grammar from existing language artifacts. Throughout this paper we use the term *grammar recovery* to refer to the extraction, assessment and testing of a grammar from existing language artifacts [16]. The technique that we describe uses only test cases and parse trees generated from the test cases, and we apply the technique to a dialect of the C++ language, GNU *gcc* version 4.0.0. However, given parse trees generated from the test cases for a language or a dialect of a language, our technique can be used to automatically recover a grammar for the language, including languages such as Java, C, Python or Ruby. To generate the parse trees, we manually augment the *gcc* parser to produce parse trees in XML format.

In the next section we provide background about terminology and concepts in our paper. In Section 3 we provide

The Fourteenth IEEE International Working Conference on Reverse Engineering (WCRE 2007)

an overview of our system and in Section 4 we present our methodology for automated grammar recovery. In Section 5 we provide some results from our recovery of the GNU *gcc* C++ dialect and in Section 6 we review related work. Finally, in Section 7 we draw conclusions.

2. Background

In this section we define terminology associated with grammars, languages, scanning and parsing. A general description of languages, context-free grammars and parsing can be found in reference [1]. We also review the terminology for grammar recovery and the use of schemas for validation of files in the extended markup language format (XML) and some of the schema languages that are used for validation of schema-based XML.

2.1. Terminology

Given a set of words (known as a lexicon), a *language* is a set of valid sequences of these words. A *grammar* defines a language; any language can be defined by a number of different grammars. When describing formal languages such as programming languages, we typically use a grammar to describe the *syntax* of that language; other aspects, such as the semantics of the language typically cannot be described by context-free grammars.

A grammar defines a language by specifying valid sequences of derivation steps that produce sequences of terminals, known as the *sentences* of the language. One procedure for using a grammar to derive a sentence in its language is as follows. We begin with the start symbol S and apply the production rules, interpreted as left-right rewriting rules, in some sequence until only non-terminals remain. This process defines a tree whose root is the start symbol, whose nodes are non-terminals and whose leaves are terminals. The children of any node in the tree correspond precisely to those symbols on the right-hand-side of a production rule. This tree is known as a *parse tree*; the process by which it is produced is known as *parsing*.

2.2. Grammar Recovery

Grammar recovery comprises the concepts involved in the derivation of a language's grammar from some available resource such as a language reference or compiler source code [15]. A recovered grammar G for a language L is likely to be an approximation of G, since G is likely to be incomplete or incorrect with respect to L. Grammar recovery is more involved than other forms of grammar reengineering [12, 25].

2.3. Schemas

The term schema can assume several different meanings but in computer science a *schema* is a model. An XML schema is a specific, well-documented definition of the structure, content and, to some extent, the semantics of an XML document. An XML schema provides a view of the document type at a relatively high level of abstraction. Relax NG is an expressive XML schema language written specifically for grammar specification [11].

3. System Overview and Partial Verification

In this section, we first provide an overview of the design and implementation of our system for grammar recovery. Our grammar recovery depends on a parse tree that correctly captures the structure of the language or language dialect under consideration. Thus, in Section 3.2, we describe the code regeneration process that we use to partially verify the structure of the parse tree generated by our system. Finally, in Section 3.3, we provide an example that illustrates that the partial structural verification provided by code regeneration is necessary, but not sufficient, for structural verification of the generated parse tree.

3.1. Grammar Recovery System Overview

Figure 1 provides an overview of our grammar recovery system, which takes a test case or application written in a C++ language or dialect as input, shown as a rectangular note on the left side of the figure, and produces a recovered grammar in three formats, Relax NG, Latex and YACC, shown as rectangular notes on the right side of the figure. The large rectangle at the top of Figure 1 illustrates the parse tree generation phase and the large rectangle at the bottom of the figure further illustrates the postprocessor phase of the parse tree generation phase.

For the parse tree generation phase, we use an augmented version of the GNU *gcc* parser version 4.0.0, labeled parse2xml, and shown in the upper left corner of Figure 1. The parse2xml component generates an annotated parse tree, Annotated Parse Tree, for the input C++ test case or application by inserting probes into the file parser.c to generate a trace of grammar terminals and non-terminals. The parser.c file contains the core of the backtracking recursive-descent parser included in the source of the GNU *gcc* C++ compiler; we use version 4.0.0 in our study but our technique can be applied to any *gcc* C++ parser or to any language whose corresponding parser generates a parse tree.

The postprocessor component, shown in the middle of Figure 1, processes the Annotated Parse Tree by completing three tasks: (1) Backpatch Tentatively Parsed

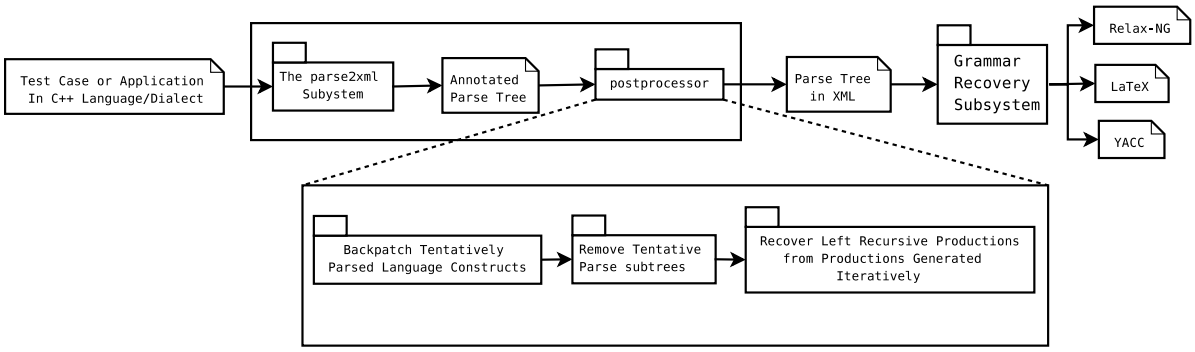

Figure 1. Overview of our Grammar Recovery System. *This figure provides an overview of our grammar recovery system, which takes a test case or application written in a C++ language or dialect as input, shown as a rectangular note on the left side of the figure, and produces a recovered grammar in three formats,* Relax NG*,* Latex *and* YACC*, shown as rectangular notes on the right side of the figure.*

Language Constructs, (2) Remove Tentative Parse Subtrees, and (3) Recover Left Recursive Productions Generated Iteratively. The *gcc* parser emits productions for member function bodies and default parameter lists after the associated class is parsed. Thus, the first task entails backpatching member function bodies and default parameter lists. Also, the *gcc* parser performs tentative parsing and then backtracks to recover from incorrectly chosen alternatives. Thus, the second task entails deleting parse subtrees that were emitted as part of an incorrect alternative and writing committed subtrees to a file in XML format. Finally, the *gcc* parser is a recursive-descent parser and productions that are expressed left recursively in the ISO C++ standard must be parsed iteratively by *gcc*. Thus, the third task that the postprocessor component performs is to recover left recursive productions from productions that were generated iteratively by the recursive-descent *gcc* parser.

The output of the postprocessor component is a Parse Tree in XML format, shown on the upper right section of Figure 1. This parse tree is then used in our Grammar Recovery Subsystem, which is described in Section 4.

3.2. Partial Structural Parse Tree Verification

Our grammar recovery technique requires a parse tree that correctly represents the particular language or language dialect under recovery. Thus, an important aspect of our work entails verification that the structure of the generated parse tree correctly reflects the structure of the particular language or language dialect under consideration. The first phase of our verification is a Partial Structural Verification of the Parse Tree, a process that entails a traversal of the generated parse tree to regenerate the code for the input test case or application. Comparison of the regenerated code with the original code for the test case or application will verify that the tokens appear in the parse tree in the correct order and that there are no extra tokens, nor have any tokens been omitted.

The partial structural verification phase of our system is illustrated in Figure 2, where the input C++ test case or application is processed by the *gcc* preprocessor, shown as gcc preprocessor in the upper left corner of the figure. The preprocessor performs macro substitution, inserts include files, and replaces trigraphs. However, the source code for the C++ test case or application cannot be compared directly to the source code that is regenerated from the parse tree since white space, comments and compiler directives are not included in their parse tree representation. Thus, the preprocessed application, must first be processed by the Token Normalizer, shown in the upper right corner of Figure 2. One function of the Token Normalizer is to "normalize" white space, so that the statement **int** f(); is converted into a normalized form, with one token emitted on each line of the output.

The last row in Figure 2 illustrates the components involved in code regeneration using the Parse Tree in XML, which is processed by the Code Regenerator, shown in the bottom left side of the figure. The Code Regenerator is a SAX parser, which traverses the Parse Tree in XML and regenerates the C++ source code by emitting the tokens in the tree; the regenerated source is then normalized using the Token Normalizer described above. Finally, we use the $diff$ utility to compare the preprocessed normalized C++ application, with the source code produced by normalizing the emitted tokens during a traversal of the Parse Tree in XML.

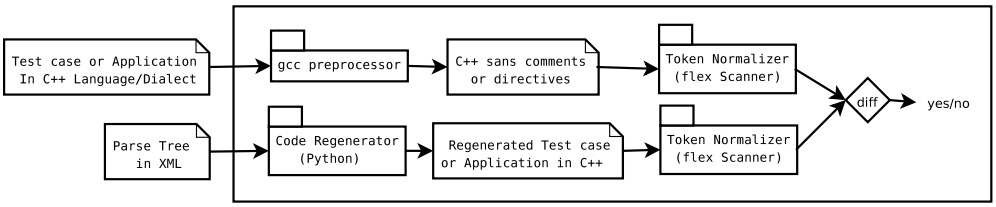

Figure 2. Partial Structural Verification of Parse Tree. *This figure summarizes the initial steps for verifying that our generated parse tree represents the language dialect under recovery. In this initial step we traverse the generated parse tree to regenerate the source code for the test case or application to verify that the tokens in the tree are complete and that their order of appearance in the tree matches the order of appearance in the test case or application.*

1	Improper nesting:	Proper nesting:
2	<unary_operator>	<unary_operator>
3	</unary_operator>	<token />
4	<token />	</unary_operator>

Figure 3. A Structurally Incorrect Parse Tree. *The XML sequence on the left side of the figure illustrates an improper nesting and the sequence on the right illustrates a proper nesting.*

3.3. Further Parse Tree Verification is Required

The partial structural verification of the parse tree provided by source code regeneration is necessary because the structure of the parse tree may be incorrect, causing tokens to be omitted, regenerated in the wrong order, or regenerated incorrectly. However, source code regeneration does not guarantee that the parse tree is correct. In particular, after completion of our source code regeneration validation, our subsequent Relax NG schema-based verification uncovered errors in the parse tree. For example, Figure 3 illustrates improper nesting of tokens within their corresponding production symbols: the XML sequence on the left side of the figure illustrates an improper nesting and the sequence on the right illustrates a proper nesting. An improper nesting of terminal and non-terminal grammar symbols is not exposed by code regeneration since, in regenerating the code, only the terminal symbols are considered.

An important result of our work is that generation of a parse tree in XML format, the correct regeneration of the source code, and the determination that the XML tags are properly nested will not expose a faulty parse tree such as the one represented by the XML sequence illustrated on the left side of Figure 3. For example, the *xmllint* tool, without a corresponding schema, will validate the XML sequence on the left side of Figure 3. Our grammar recovery and Relax NG schema generation techniques, described in Sections 4 and 5, provide further structural verification of the parse tree, but does not provide semantic verification that the structures represented in the test case or application are correctly represented in the parse tree; semantic verification of the parse tree is beyond the scope of this paper.

4. Methodology for Grammar Recovery

In this section, we describe our technique for recovering a grammar using a parse tree and a testsuite. In Section 4.1 we provide an illustrative example of grammar recovery and an algorithm that describes the technique. In Section 4.2 we describe our technique for recovering left recursion from a parse tree generated by a recursive descent parser.

4.1. Grammar Recovery and Schema Generation

In Section 3 we provided an overview of the recovery process and described our approach for generating a parse tree in XML format. We now exploit element nesting in the generated parse tree to recover a grammar for the *gcc* dialect of C++. Figure 4 shows an illustrative example of our grammar recovery process, where a parse tree in XML format is listed on the left side of the figure, and a grammar that corresponds to the parse tree is listed on the right side of the figure. The opening and closing element tags, $<A>$ and $</A>$, listed on lines 2 and 8 of Figure 4, indicate the left side of a grammar production; the elements nested within the $<A>$ element are the right side of the grammar production. In our example, the right side of the grammar production whose left side is A is indicated by the opening and closing element tags on lines 3 and 6, and the empty element tag on line 7. Thus, our first grammar production is $A \rightarrow BE$, shown on line 2 on the right side of Figure 4. The production $B \rightarrow CD$, listed on line 3 on the right side of the figure is obtained similarly. Since all empty XML elements indicate empty grammar productions, the empty elements on lines 4, 5 and 7 generate the empty productions, $C \rightarrow \lambda$, $D \rightarrow \lambda$, $E \rightarrow \lambda$.

To obtain parse trees for use in our grammar recovery,

```
1  Parse Tree in XML:          Corresponding Grammar:
2  <A>                         A → B E
3     <B>                      B → C D
4        <C />                 C → λ
5        <D />                 D → λ
6     </B>
7     <E />                    E → λ
8  </A>
```

Figure 4. Sample Recovery of a Grammar from A Parse Tree. *This figure illustrates our grammar recovery process, with a parse tree in XML format listed on the left side of the figure, and a grammar recovered from the parse tree listed on the right side of the figure.*

we use the GCC test suite that accompanies *gcc* version 4.0.0, to expose the terminals, non-terminals and productions in the grammar. Figure 5 summarizes the important steps, written in Python syntax, in our algorithm for exploiting test cases to recover a grammar for the *gcc* dialect of C++. The Python code in the figure uses a SAX parser to traverse the Parse Tree in XML and uses the set listed on line 1 of Figure 5, productions, to collect the generated productions. The productions list is initialized to empty on line 5 of the constructor Handler. The callback function, startElement listed on lines 8–16 of the figure, begins the construction of a production. On line 9, the grammar symbol for the left hand side of the current production is pushed into productions. The callback function, endElement listed on lines 18–24 of the figure, completes the collection of the current production by popping it into the set productions, on line 21 of Figure 5. The **for** loop on lines 27–29 processes the list of input files containing parse tree candidates for validation. On line 28 we test the suffix of the file; since the XML parse trees for most of the applications in our test suite are large, we use the *gzip* compression utility.

The final step in grammar recovery is to convert the recovered grammar into three schemas that might be useful to a developer for validation, comprehension or parsing. The three schemas are expressed in (1) Relax NG, (2) LaTeX, and (3) YACC format. We use the Relax NG XML schema language because it is designed especially for grammar specification [11]. Relax NG provides three choices for grammar specification; we use the flattened XML format, because of the numerous references to nonterminals and the recursive nature of the C++ grammar. We use the Relax NG schema for verification of the grammar against the test cases. We use the LaTeXschema for to compare the recovered grammar to the ISO C++ grammar and the YACC schema for deployment of the recovered grammar.

```
1   productions = set()
2
3   class Handler(xml.sax.ContentHandler):
4       def __init__(self):
5           self.productions = [ ]
6           self.token = None
7
8       def startElement(self, tag, attrs):
9           self.productions.append([tag])
10          if tag == 'token':
11              if attrs.has_key('expr_value'):
12                  self.token = attrs['expr_value']
13              elif attrs.has_key('type'):
14                  self.token = attrs['type']
15              else:
16                  self.token = None
17
18      def endElement(self, tag):
19          productions.add(self.productions.pop())
20          if self.token:
21              self.productions[-1].append('token(%s)' % self.token)
22              self.token = None
23          else:
24              self.productions[-1].append(tag)
25
26  ch = Handler()
27  for f in sys.argv[1:]:
28      if f[-3:] == '.gz': xml.sax.parse(gzip.open(f), ch)
29      else: xml.sax.parse(open(f), ch)
```

Figure 5. Grammar Recovery Algorithm. *This figure illustrates the important steps, written in Python syntax, to recover a grammar for the gcc dialect of C++.*

To generate the Relax NG schema, we first designate the $<translation-unit>$ element as the root element of the schema. We then group the productions by the left-hand side nonterminal and, for each unique left-hand side, we create $<define>$ and $<element>$ tags; if there is more than one production for this nonterminal, we also create a $<choice>$ tag. For each production of this nonterminal we generate a $<ref>$ tag and attach the terminal or nonterminal on the right-hand side to a $<group>$ tag. We continue processing all of the productions in productions until the schema representation includes all productions.

4.2. Recovering Left Recursion from a Parse Tree Generated by a Recursive Descent Parser

One of the goals of our work is to compare the recovered grammar for the C++ dialect of *gcc* version 4.0.0 with the grammar presented in the ISO C++ standard [3]. However *gcc* versions 4.0.0, and subsequent versions, use recursive descent parsing where productions expressed using left recursion in a grammar must be expressed using itera-

	Recovered Production								Example
1	*postfix-expression*	→	*primary-expression*						x
2		→	*primary-expression*	*expression-list*					f(1)
3		→	*primary-expression*	*expression-list*	.	*id-expression*			f(1).x
4		→	*primary-expression*	*expression-list*	.	*id-expression*	*expression-list*		f(1).g(2)

Figure 6. Recovered Productions. *The column on the left side of this figure lists four productions extracted from a recursive descent parser. The column on the right side lists an example that might generate the production.*

tion in the parser. Thus, grammar productions for postfix-expression will appear as illustrated in Figure 6. However, the ISO C++ grammar uses left recursion to specify postfix-expression. We have designed and implemented a technique to recover left-recursion for a grammar extracted using a recursive descent parser; however, a full explanation of our technique is beyond the scope of this paper.

5. Results of our Study in Coverage Adequacy & Grammar Recovery for a Dialect of C++

In this section we report the results of our study in recovering a grammar for the GNU *gcc* dialect of C++, version 4.0.0. An important aspect of our work is the coverage of the language or language dialect provided by the test suite that we use in our grammar recovery system. In particular, if there is no test case that exercises a given terminal or non-terminal in the dialect, then that terminal or non-terminal will not be recovered. That is, we can only recover as much of the grammar as the test suite can expose. Thus, we have developed three different test suites: the first test suite is *gcc* and is packaged with version 4.0.0 of the *gcc* compiler, the second test suite consists of three commonly used open-source applications, and the third test suite consists of three modules in the Mozilla application suite.

We begin our study, in Section 5.1, by describing the three test suites that we have developed. We follow, in Section 5.2, with an investigation of the coverage of terminal and non-terminal symbols provided by each of the three test suites and show that the *gcc* test suite provides better coverage than the other two test suites. In Section 5.3 we present some timing results for each of the three test suites and show that the *gcc* test suite was also more efficient than the other two test suites. Thus, we use the *gcc* test suite to initially recover a grammar for the *gcc* C++ dialect; we then extend the *gcc* C++ grammar using the other two test suites. In Section 5.4 we present results for our recovery of a grammar for GNU *gcc* C++ version 4.0.0, including a comparison of the *gcc* grammar with the grammar from the ISO C++ standard [3]. Finally, in Section 5.5, we present some results for the three schemas that we automatically generate from our grammar recovery system (see right side of Figure 1).

Test Suite		Version	# TUs	LOC	Size (MB)
1	GCC	4.0.0	1265	237,547	13
2	Doxygen	1.5.2	75	618,092	22
	FiSim	1.1b	23	734,865	25
	Fluxbox	1.0rc3	74	1,916,398	58
3	Gecko	1.9a	237	4,902,309	204
	Necko	1.9a	109	985,876	51
	XPCOM	1.9a	152	840,613	33

Table 1. The Three Testsuites. *This table lists the three test suites that we use in our study, together with statistics about the individual test cases in each suite.*

All experiments reported in this section were executed on a workstation with an *AMD Athlon64 X2 5200+* processor, 2048 MB of PC3200 DDR RAM, and a 7200 RPM SATA hard drive, running the Ubuntu version 7.04 operating system. The programs were compiled using *gcc* version 4.1.2 and our SAX parser used Python version 2.5.1. All of the experiments and statistics that we report in this section are automated using Python scripts and the *make* utility version 3.81 [23, 31]. We use an open source program, *xmllint*, to validate our XML [18]. However, the XML files containing parse trees for each translation unit were large, so that the size of the *xmllint* stack had to be extended from 1K to 64K to permit validation of the XML parse trees for each translation unit.

5.1. The Three Test Suites

Table 1 lists statistics for the three test suites that we use to evaluate our validation system; the column labeled Test Suite in the table contains two sub-columns: the first sub-column lists the number of the test suite and the second sub-column lists the applications in the respective test suite. The first test suite is labeled GCC and consists of

the test cases packaged with the GNU *gcc* compiler, version 4.0.0, taken from the directory *g++.dg*. We chose the test cases in the *g++.dg* directory because they test the C++ compiler of the *gcc* collection. The first row of data in the table indicates that we are using version 4.0.0 of *gcc*, that the test cases produce 1265 translation units (TUs), consist of 237,547 lines of code (LOC) and occupy 13 megabytes (MB) of data. A *translation unit* consists of a source file, together with its include files (files that are included using the *#include* preprocessor directive) but not including sections of code removed by conditional-compilation directives such as $\#if$.

The second test suite consists of three commonly used applications: Doxygen, FiSim and Fluxbox [30, 6, 7]. Doxygen is a documentation system for C, C++, and Java [30]; FiSim is a scientific modeling tool for advanced engineering of fiber and film [6]; and Fluxbox is a lightweight X11 window manager built for speed and flexibility [7]. The Fluxbox application, version 1.0rc3, is the largest of the three test cases consisting of 74 TUs, comprising 1,916,398 LOC and occupying 58MB. The Doxygen application, version 1.5.2, is the smallest of the three test cases consisting of 75 TUs, comprising 618,092 LOC and occupying 22MB. However, even though Fluxbox consists of three times as many LOC as Doxygen, Doxygen consists of more TUs, indicating that the Doxygen application is partitioned into more files than the Fluxbox application.

The three test cases in this second test suite were chosen for their variety of size and application, including a language tool, a scientific application and a window managing system. The FiSim application utilizes the Loki and Boost libraries, which rely on C++ templates [2, 27].

The third test suite consists of three modules from the *Mozilla 1.9a* open source, cross-platform web and e-mail application suite [28]. The three modules are Gecko, Necko and XPCOM. Gecko is a cross-platform rendering engine that forms the core of the Mozilla browser. Necko is Mozilla's network library, providing a platform-independent API for several layers of networking ranging from transport to presentation layer. XPCOM is a cross-platform variant of the well-known Component Object Model (COM).

The Gecko module, version 1.9a, is the largest of the three mozilla test cases, consisting of 234 TUs, comprising 4,902,309 LOC and occupying 204MB. The XPCOM module is the smallest of the three mozilla test cases consisting of 152 TUs, comprising 840,613 LOC and occupying 33MB.

We have chosen the Mozilla modules for their popularity as both an application and a frequently used test suite. For example, *iPlasma* is an integrated environment for analysis of object-oriented software systems written in the C++ or Java languages. The scalability of the *iPlasma* environment is demonstrated by its ability to handle large-scale projects in the size of millions of lines of code including Eclipse and Mozilla [19, 17]. Similarly, the TUAnalyzer system and the Columbus parser use Mozilla as a test case [4, 10].

Test Suite		Non-terminals	Productions
1	GCC	137	453
2	Doxygen	113	324
	FiSim	129	400
	Fluxbox	127	384
	Suite Total	129	405
3	Gecko	123	376
	Necko	122	358
	XPCOM	123	376
	Suite Total	123	381

Table 2. Grammar Coverage. *This table lists statistics about the coverage of the grammar non-terminals and productions provided by each of the three test suites.*

In the next two sections we further analyze the three test suites for their efficiency and their ability to provide coverage of the non-terminals and productions used to build the parse tree.

5.2. Results for Grammar Coverage

To investigate the grammar coverage for non-terminals and productions, we report coverage results for each of the three test suites. Table 2 lists these results where the first row of the table shows that the first test suite, GCC, exercised 137 non-terminals and 453 productions of the *gcc* C++ grammar.

Coverage results for the second test suite composed of Doxygen, FiSim and Fluxbox are listed in the middle of Table 2. The best coverage of the grammar is provided by the FiSim test case, which exercised 129 non-terminals and 400 grammar productions. The fifth row in the table, labeled **Suite Total**, shows that the combined coverage provided by all three test cases in the second test suite was 130 non-terminals and 415 productions. The FiSim test case exercised 129 non-terminals and, since the results in the **Suite Total** row are cumulative, we know that the FiSim test case exercised more non-terminals than any of the other two test cases and the other two test cases exercised no additional non-terminals. The FiSim test case also exercised the most grammar productions, with 400 productions; however, the other two test cases exercised 15 additional productions since the cumulative total listed in the **Suite Total** row is 415 productions.

Coverage results for the third test suite composed of

Test Suite		LOC	Parse Tree Generation		Token Verification	
1	GCC	237,547	3m	15s	2m	56s
2	Doxygen	618,092	10m	6s	4m	27s
	FiSim	734,865	10m	59s	4m	37s
	Fluxbox	1,916,398	24m	54s	10m	29s
	Suite Total	3,269,355	45m	59s	19m	33s
3	Gecko	4,902,309	70m	2s	34m	40s
	Necko	985,876	15m	7s	7m	22s
	XPCOM	840,613	10m	25s	5m	53s
	Suite Total	6,728,798	95m	34s	47m	55s

Table 3. Efficiency Considerations. *This table reports timing results for each of the three test suites.*

Gecko, Necko and XPCOM are listed at the bottom of Table 2. The best coverage of the grammar is provided by the Gecko and XPCOM test cases, which exercised 123 non-terminals and 376 grammar productions. The ninth row in the table, labeled **Suite Total**, shows that the combined coverage provided by all three test cases in the third test suite was 123 non-terminals and 381 productions.

5.3. Results for Measuring Efficiency

To investigate the efficiency of grammar recovery, we report timing results for each of the three test suites. Table 3 lists these results with the test suites listed in the first column, lines of code (LOC) listed in the second column and timing results for parse tree generation and token order verification listed in columns three and four respectively.

The first row of Table 3 lists results for the GCC test suite, which required 3 minutes and 15 seconds to generate parse trees and 2 minutes and 56 seconds to verify tokens; thus, the GCC test suite required a total of 6 minutes and 11 seconds for both phases.

Timing results for the second test suite composed of Doxygen, FiSim and Fluxbox are listed in rows 3 through 6 of Table 3. The largest test case, Fluxbox, required the longest time to generate and verify, using 24 minutes and 54 seconds for parse tree generation, and 10 minutes and 29 seconds for token verification. To generate the parse tree using all three test cases in the second test suite required 45 minutes and 59 seconds for parse tree generation, and 19 m and 33 seconds for token verification. Thus, to complete generation and verification using the second test suite required a total of 1 hour, 5 minutes and 32 seconds.

Timing results for the third test suite composed of Gecko, Necko and XPCOM are listed in rows 7 through 10 of Table 3. The largest test case, Gecko, required the longest time to generate and verify, using 70 minutes and 2 seconds for parse tree generation, and 34 minutes and 40 seconds token verification. To generate and verify the parse tree using all three test cases in the third test suite required 95 minutes and 34 seconds for parse tree generation, and 47 m and 55 seconds for token verification. Thus, the complete process using the third test suite required a total of 2 hours, 23 minutes and 29 seconds.

In comparing the three test suites, our results show that the first test suite, GCC, only required 6 minutes and 11 seconds, the second test suite required 1 hour, 5 minutes and 32 seconds, and the third test suite, Mozilla, required 2 hours, 23 minutes and 29 seconds. Thus, the second and third test suites required considerable more time to generate and verify the parse trees. In our next section, we investigate the coverage of the grammar provided by each test suite to determine if the additional time for generation and verification is warranted for complete coverage.

Grammar	Terminals	Non-Terminals	Total Productions	Productions for Extensions	Recovered by Applications	Unrecovered Productions
ISO C++	116	141	516	n/a	n/a	n/a
GNU C++	132	137	545	44	21	17

Figure 7. Results for Recovered Grammar. *We compare our recovered GNU gcc C++ grammar with the ISO grammar for C++.*

5.4. Results for Grammar Recovery

To evaluate our recovered grammar for the *gcc* dialect of C++, we compare the recovered grammar to the grammar in the ISO C++ standard [3], as presented in reference [24]. The table in Figure 7 summarizes these results. The first row of the table lists results for the ISO C++ grammar with 116 Terminals, 141 Non-Terminals, 516 Productions and no extensions. The second row of the table lists results for the recovered *gcc* grammar with 132 Terminals, 137 Non-Terminals, 545 Productions and 44 Productions for extensions to the ISO C++ grammar.

The first column of data in Figure 7 shows that there are 16 more terminals in the recovered *gcc* grammar due to the extensions to the C++ language. For example, GNU *gcc*

C++ includes terminal, thread, for threads and __imag__, __real__ for complex numbers. However, the second column of data shows that there are 4 fewer non-terminals in the recovered *gcc* grammar, partly because there are 14 non-terminals and 38 grammar productions in the ISO C++ grammar to specify precedence. Previous research has shown that for many languages the most natural grammar is not accepted by the parser generators used in practice and that precedence rules are frequently resolved in the semantic actions of the parser rather than specified by the grammar [5]. This is the case with the *gcc* parser used in version 4.0.0 so that the 14 non-terminals used in the ISO C++ grammar to specify binary operator precedence are reduced to a single non-terminal in the *gcc* grammar.

The third column of Figure 7 shows that there are 29 more productions in the *gcc* grammar and the fourth column shows that 44 productions in the *gcc* grammar are used for extensions to the C++ language. These last two results indicate that the *gcc* grammar required 15 fewer productions than the ISO C++ grammar to specify the C++ language. The fifth column of the figure show that 21 additional productions were recovered using the second and third test suites. Finally, the last column shows that 17 productions in the ISO C++ grammar were not recovered by any of the test suites; most of these unrecovered productions specify seldom used language constructs such as anonymous enums with no enumerations or overloading the ->* operator. Other unrecovered productions in the ISO C++ grammar exercise scoping rules for the ISO C++ language.

5.5. Results for Generated Schemas

Lämmel and Verhoef assert that a recovered grammar must be transformed so that it is useful for another application [16, page 5]. In this section we describe transformations of our recovered grammar for GNU *gcc* version 4.0.0. into three different schemas: Relax NG, LaTeX and YACC.

The Relax NG Schema: We use the Relax NG schema to verify that the recovered grammar conforms to the parse tree representation of the grammar. We do this by using *xmllint*, together with the Relax NG schema language, to validate the generated schema for the recovered grammar.

The LaTeX Schema: We automatically generate a LaTeXversion of our grammar and we found our LaTeXversion of the grammar for GNU *gcc* dialect invaluable for comparison to the ISO C++ grammar. Our removal of iteration from and incorporation of left recursion into the recovered grammar further enhanced readability. We also found the *gcc* compiler source version of the grammar difficult or impossible to comprehend and we agree with Lämmel and Verhoef that compiler grammars are not optimal reading for humans [16, page 10].

The YACC Schema: Grammar deployment is the process of turning a given grammar specification into a working parser [13]. We used our recovered grammar to automatically generate a YACC schema and we deployed the schema to the *Bison* parser generator to generate an LALR(1) parser. The LALR(1) parser contained 9,603 shift/reduce conflicts and 30,637 reduce/reduce conflicts. We also deployed the grammar to a *Bison GLR* parser generator and the GLR parser contained no conflicts. Although the Bison GLR parser for the GNU *gcc* dialect requires disambiguation rules, the deployment comparison supports the use of GLR parsing for ambiguous grammars [29].

6. Related Work

Lämmel and Verhoef present a semi-automatic approach to grammar recovery [16]. Their approach requires a language manual and a test suite. They use the manual to construct syntax diagrams for the language, they correct the diagrams, write transformations to correct connectivity errors, and then use the test cases to further correct the generated grammar. One advantage of their approach is that their grammar recovery is not connected to a specific parser implementation. The disadvantage of thier approach is that many phases of the grammar recovery are manual. The approach that we propose uses a parse tree and a test suite and our grammar is recovered automatically.

Bouwers et al. present a methodology for recovering precedence rules from grammars [5]. They describe an algorithm for YACC and implement the method using tools for YACC and SDF. The methodology for precedence recovery could be incorporated into our grammar recovery system and used to transform our recovered grammar for GNU *gcc* into a grammar that more closely resembles the grammar specified in the ISO C++ standard [3].

Elkhound is a parser generator, similar to Bison, whose generated parsers use the Generalized LR (GLR) parsing algorithm [21]. Elkhound is used to write a parser, Elsa, which attempts to accomodate several dialects of C++ including *gcc* and Visual C++. However, Elsa does not fully accomodate either dialect. For example, 89% of the test cases in the *gcc* test suite, described in Section 5, were successfully parsed by Elsa. However, for the Fluxbox test case only 32 of the 74 translation units were parsed by Elsa and for FiSim only 2 of the 23 translation units were parsed by Elsa. Thus, we have chosen to recover a grammar for *gcc*, a more commonly used C++ dialect than Elsa.

7. Concluding Remarks

We have described the design and implementation of an automated technique to recover a grammar for a language or language dialect. We applied the approach to recover a dialect of the GNU *gcc* C++ grammar, version 4.0.0. Our approach uses a parse tree and a test suite to recover the grammar and we have shown the importance of coverage in the recovery process. We first used the *gcc* test suite packaged with the GNU *gcc* C++ compiler and have shown that this test suite provides better coverage of terminals and non-terminals than the other two test suites composed of more commonly used open source applications. However, the open source applications generated 21 additional productions not recovered by the *gcc* test suite. Nevertheless, there are 17 productions in the GNU *gcc* C++ grammar that none of the test cases exercised and the corresponding productions were not recovered. Our recovery system generates three schemas for the recovered grammar in Relax NG, LaTeX, and YACC format, and we reported the results of using these schemas.

Acknowledgements

This work was supported by the ERC program of the National Science Foundation under award number EEC-9731680.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] A. Alexandrescu. *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[3] American National Standards Institute. *International Standard: Programming Languages - C++*. Number 14882:1998(E) in ASC X3. ISO/IEC JTC 1, Sept 1998.

[4] A. Beszédes, R. Ferenc, and T. Gyimóthy. Columbus: A reverse engineering approach. In *STEP 2005*, pages 93–96, September 2005.

[5] E. Bouwers, M. Bravenboer, and E. Visser. Grammar engineering support for precedence rule recovery and compatibility checking. In *LDTA'07*, pages 82–96, March 2007.

[6] C.L. Cox, E. B. Duffy, and J.B. von Oehsen. Fisim: an integrated model for simulation of industrial fibre and film processes. *Plastics, Rubber and Composites*, 33(9-10):426–437, November 2004.

[7] Fluxbox Project. FluxBox version 0.9.12. Available at http://www.fluxbox.org.

[8] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1983.

[9] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Prentice Hall PTR, third edition, 2005.

[10] T. Gschwind, M. Pinzger, and H. Gall. Tuanalyzer – analyzing templates in c++ code. In *WCRE'04*, pages 48–57, Washington, DC, USA, 2004. IEEE Computer Society.

[11] J. Clark and M. Makoto. RELAX NG Specification. *http://www.oasis-open.org/committees/relax-ng/spec-20011203.html*, 2007.

[12] P. Klint, R. Lämmel, and Chris Verhoef. Toward an engineering discipline for grammarware. *ACM Transactions on Software Engineering Methodology*, 14(3):331–380, 2005.

[13] J. Kort, R. Lämmel, and C. Verhoef. The grammar deployment kit. In Mark van den Brand and Ralf Lämmel, editors, *Electronic Notes in Theoretical Computer Science*, volume 65. Elsevier Science Publishers, 2002.

[14] S. R. Ladd. *Active Visual J++*. Microsoft Press, Redmond, WA, USA, 1997.

[15] R. Lämmel. Grammar testing. In *FASE*, pages 201–216, 2001.

[16] R. Lämmel and C. Verhoef. Semi-automatic Grammar Recovery. *SP&E*, 31(15):1395–1438, December 2001.

[17] M. Lanza and R. Marinescu. *Object-Oriented Metrics in Practice*. Springer, 2006.

[18] libxml. The XML C parser and toolkit of Gnome, 2007. http://www.xmlsoft.org/.

[19] LOOSE Research Group. iPlasma: An Integrated Platform for Quality Assessment of Object-Oriented Design. *http://loose.upt.ro/iplasma/index.html*, 2007.

[20] B. A. Malloy, T. H. Gibbs, and J. F. Power. Progression toward conformance for C++ language compilers. *Dr. Dobbs Journal*, pages 54–60, November 2003.

[21] S. McPeak. Elkhound: A GLR parser generator and Elsa: An Elkhound-based C++ parser, 2007. http://www.cs.berkeley.edu/ smcpeak/elkhound/.

[22] Object Management Group (OMG). Unified modeling language (uml), version 2.0, 2006. http://www.omg.org/technology/documents/formal/uml.htm.

[23] A. Oram. *Managing Projects with Make*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1992.

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

[25] A. Sellink, H. Sneed, and C. Verhoef. Restructuring of cobol/cics legacy systems. *Sci. Comput. Program.*, 45(2-3):193–243, 2002.

[26] J. Sharp, A. Longshaw, and P. Roxburgh. *Microsoft Visual J# .NET*. Microsoft Press, Redmond, WA, USA, 2002.

[27] J. G. Siek, L. Q. Lee, and A. Lumsdaine. *The Boost Graph Library*. Addison-Wesley, 2002.

[28] The Mozilla Organization. The Mozilla Application Suite. *http://www.mozilla.org*, 2007.

[29] M. G. J. van den Brand, A. Sellink, and C. Verhoef. Current parsing techniques in software renovation considered harmful. In *IWPC*, page 108, Washington, DC, USA, 1998.

[30] D. van Heesch. Doxygen version 1.3.9.1. Available at http://stack.nl/ dimitri/doxygen.

[31] G. van Rossum. *Python Library Reference*. Python Software Foundation, 2001.