Program annotation in XML: a parse-tree based approach

James F. Power
Computer Science Department
National University of Ireland
Maynooth, Co. Kildare, Ireland
jpower@cs.may.ie

Brian A. Malloy
Computer Science Department
Clemson University
Clemson, SC, USA
malloy@cs.clemson.edu

Abstract

In this paper we describe a technique that can be used to annotate source code with syntactic tags in XML format. This is achieved by modifying the parser generator bison *to emit these tags for an arbitrary LALR grammar. We also discuss an immediate application of this technique, a portable modification of the* gcc *compiler, that allows for XML output for C, Objective C, C++ and Java programs. While our approach is based on a representation of the parse-tree and does not have the same semantic richness as other approaches, it does have the advantage of being language independent, and thus re-usable in a number of different domains.*

1. Introduction

In this paper we describe a modification of the parser generator, GNU bison, that permits the generation of an XML representation of its parse tree.

Program analysis tools are the keystone of good software reverse engineering applications. Program analysis is a key component of tasks such as program comprehension, slicing, visualisation and metrication, and acts as a foundation for more comprehensive tools that aid software maintenance, migration, transformation and re-engineering.

The levels at which a program can be analysed mirror the phases of the traditional description of a compiler as found in references [1, 15]. In particular, we can distinguish between *static analysis*, concerning information gleaned from the program code, and *dynamic analysis*, concerning information collected from running the program. At the level of static analysis, we can identify four main levels of information, associated with four phases of compilation:

1. *Preprocessing* involves dealing with conditional compilation and textual inclusion, and is mainly an issue in C and C++, although C# also has a limited form of preprocessor.

2. *Lexical analysis* collects characters into words, and eliminates comments and whitespace. Tools working at the lexical level can provide crude metrics by analysing keywords, and can often be constructed using relatively simple tools such as lex, grep or awk.

3. *Parsing-level analysis* concerns the hierarchical categorisation of program constructs into syntactical categories such as declarations, expressions, statements etc.

4. *Semantic analysis* deals with issues such as definition-use pairs, program slicing and identifier analyses.

While information from each level is needed to build a full view of a program, in many ways the parser is central to this process, and is the focus of this paper. Typically, it is the parser that drives the lexical analysis phase by requesting and organising tokens. A parser also acts as a foundation for the semantic analysis phase either directly, through events triggered on recognition of various constructs, or indirectly through the generation of some form of intermediate representation.

Most of the constructs of languages such as Pascal and Ada are context-free, and it is a straightforward matter to generate a parser for these languages, particularly using a parser generator. An exception to this easy-parse rule can be found in the language C, where a context-sensitive ambiguity exists between a declaration and an expression. This *declaration/expression ambiguity* not withstanding, parser front-ends for the C language have not been difficult to construct. However, a parser front-end for the C++ language has proven elusive and the difficulties involved have been described in references [3, 11, 12, 17, 18, 19]. Many constructs in the C++ language cannot be recognized by syntactic consideration alone; these constructs not only include the *typedef* declaration/expression ambiguity of C, but the ISO

C++ grammar also includes context-dependent keywords for *namespace*, *class*, *enumeration* and *template* declarations (see reference [2, Appendix A]). It is not surprising therefore that many program analysis tools use third-party front-ends to parse the program source.

In Section 2 we present some of the issues associated with generating XML using gcc, the GNU compiler collection, and describe how our approach seeks to address these issues. In Section 3 we present the algorithm and technical details behind our approach, and in section 4 we present some results showing this approach in action. Section 5 concludes the paper, mentioning the limitations of the approach.

2. Background and Motivation

In this section we provide some of the background to the work presented in this paper. In particular, we survey some of the issues relating to producing program analysis information from gcc, the GNU compiler collection, since this was the original goal of our work.

2.1. GNU bison and XML

Parsers are typically written using a parser generator, although smaller parsers may be directly coded, using techniques such as recursive descent. There is a large variety of parser generators available, but one of the oldest, simplest, and arguably most widely used, is the yacc parser generator, and its more modern manifestation, GNU bison.

Most parser generators, including bison, are capable of producing some kind of information on the progress of the parse, typically as some kind of parse tree. The information emitted by bison when run in debug mode consists of a list of the production rules used; a format that is not readily useful to other tools. More modern tools such as JavaCC or ANTLR have more sophisticated and flexible forms of output, but they lack the breadth of usage of bison.

In terms of choosing a more suitable output for bison, the XML markup language [23, 6] is the obvious choice. The general case for XML as a data exchange language providing for modularity between tools has been widely made. The suitability of XML and similar languages as a data format for program analysis tools has been noted in references [10, 21, 8, 13], and their arguments will not be repeated here. However, it is worth noting that there is an elegant symmetry between the hierarchical classification produced by a program parse, and the normal nested tagging used in XML.

This paper exploits this symmetry in two ways. First, by generating information at the parsing stage, we maximise the amount of available information, and this information can thus act as a basis for a variety of subsequent semantic tools. Second, by utilising the bison parser generator, we gain a technique that is generally applicable to a variety of applications and languages, and does not have to be modified for each new type of source.

2.2. gcc - The GNU Compiler Collection

The GNU compiler collection originated as the GNU C compiler, but has since expanded to include compilers for Ada, Fortran, Objective C, C++ and Java. As a robust compiler generating efficient code, it has attained wide popularity as a C and C++ compiler, and boasts a considerable code base. As an open source compiler, released under the GNU public licence, it seems an attractive starting point for the development of analysis tools for these languages. However, gcc is a large and complex piece of software, that is not immediately amenable to the production of such static analysis information. While it would be possible to modify the gcc compiler directly, such an approach has a number of difficulties.

The first difficulty with modifying gcc is the size and complexity of the compiler itself. For example, gcc version 3.0.4 is written in C, and consists of 2958 `.c` files and 1820 `.h` files. The gcc compiler has been deliberately designed to separate its front ends, which are specific to particular input programming languages, from it back ends, which are specific to target architectures. However, from the perspective of static program analysis tools, the level of modularisation is not so helpful. In particular, there is an extremely close coupling between the lexical analysis, parsing and semantic analysis phases due, in a large part, to the aforementioned problems with parsing C and C++.

A second problem with modifying the compiler directly is the evolving nature of the compiler itself. While the C programming language is fairly stable at this stage, the C++ standard is relatively new [2] and gcc, like many other compilers [22, 14], is still undergoing a process of convergence toward the standard. It is reasonable to anticipate that new releases of gcc will continue to increase standardisation and add functionality, creating a considerable maintenance overhead for anyone developing tools based on gcc. Conversely, a tool that could work for multiple versions of gcc, rather than one specific version, would be valuable for the analysis of legacy software developed using obsolete or deprecated features from these versions.

A third disadvantage of modifying the compiler directly is associated with one of the major advantages of gcc: its ability to handle multiple source languages using different front-ends. Any technique that directly modified the source code of one front end, would need to be modified and adapted for other front ends. While such modification is perhaps inevitable for language-specific applications, it

```
S      : expr
       ;
expr   : expr PLUS term
       | term
       ;
term   : factor MULT term
       | factor
       ;
factor : LBRACKET expr RBRACKET
       | ID
       | NUM
       ;
```

Figure 1. *The expression grammar*. **This is a grammar for simple expressions involving multiplication and addition. The** bison **input format is used here.**

```
<S><expr>
  <expr>
    <term>
      <factor><TOKEN type="NUM"></factor>
    </term>
  </expr>
  <TOKEN type="PLUS">
  <term>
    <factor><TOKEN type="NUM"></factor>
    <TOKEN type="MULT">
    <term>
      <factor><TOKEN type="NUM"></factor>
    </term>
  </term>
</expr></S>
<TOKEN type="$">
```

Figure 2. *XML output for the sentence "*`3 + 4 * 5`*" using the expression grammar*. **Each XML tag represents either the start or end of a region reduced to a non-terminal, or a single token. Indentation has been added to clarify the presentation.**

is reasonable to suggest that a more generic, language independent approach will avoid this maintenance problem, while also providing a useful basis for many tools.

3. Implementation

In this section we describe the modification of the bison parser generator to produce XML output. We discuss some of the problems encountered, along with our solution.

3.1. A simple example

To focus the discussion, consider the grammar given in Figure 1, a standard example of a simple grammar for expressions involving numbers with addition and multiplication.

Given a suitable lexical analyser, the sentence:

```
3 + 4 * 5
```

would be converted using the techniques described in this section to the XML representation shown in Figure 2.

In Figure 2, the terminals and non-terminals are represented as XML elements. The terminal symbols are represented as simple XML elements of type `TOKEN`, with an attribute, `type`, giving the token type. The non-terminals are represented as compound elements, with no attributes, enclosing the section of output to which they correspond.

3.2. The problem domain

The two basic approaches to parsing are *top-down* parsing which includes recursive descent and LL parsing, and *bottom-up* parsing, which includes the LALR algorithm used by bison. Generation of XML output using top-down parsing is reasonably straightforward, since typically each non-terminal corresponds to a function in the parser. Hence it is only necessary to add statements at the beginning and end of these functions to generate the proper start and end tags. Indeed, the production of XML output could be avoided altogether if desired, and the parser could simply generate corresponding SAX events for the tags. (SAX is the *Simple API for XML*, which allows an XML document to be processed serially, in an event-based manner).

Generating XML for a bottom-up parser such as bison is more problematic, however. Bottom-up parsers work by collecting tokens until a sequence is found that corresponds to the right-hand side of a grammar rule; these are then reduced by substituting the non-terminal symbol on the left-hand side of the grammar rule, and the parse continues. The problem here is that a non-terminal is only recognised *after* the corresponding symbols have been processed. While this does not cause a problem with the end XML tag, which can be inserted directly into the output at this point, it would be necessary to go back through the output to find the correct position for the start tag. At very least, this precludes the possibility of generating SAX events as the input is processed, but it also makes it difficult for simple solutions to scale to larger inputs.

For example, one possible approach would be to store the XML tags on a stack, so that the position for the start

tag could be easily determined. In our simple expression example shown in Figure 2 this is not really a problem, since the input is small. However, it should be noted that even here the last reduction is to the start symbol, so that the last XML tags generated are actually the pair `<S>` and `</S>` surrounding the whole program. While this may not seem a problem here, it can scale up quite badly to real input using C or C++. For example, a C++ program containing only the line:

```
#include <iostream>
```

expands to some 17,616 non-blank lines of C++ code, and yields an 14,481,357 kilobyte XML file containing 111,050 tags for terminal symbols, and 358,417 pairs of tags for non-terminal symbols.[1]

Clearly, for "real-world" C and C++ programs, the entire parse tree, or even a substantial part of it, is not easily stored in memory during the parse, awaiting the reduction to the start symbol. Instead it is necessary to deal with the parse output in serial form at all times.

3.3. Bottom-up generation of XML

The strategy adopted involves three stages:

1. The parse is carried out, and a single number is written to a temporary file for each shift and reduce action, uniquely identifying that action.

2. The temporary file is then read in *reverse*, and a "backwards parse" is carried out, with the start tags being inserted in the correct location

3. Since this last step reverses the order of the parse (an LR parser conducts a *rightmost* derivation), the temporary file it generates is reversed, producing the correct output.

The intermediate representation of the parse actions as single integers reduces the necessary storage, and greatly facilitates the reversing actions. It also ties in well with bison, where these integers can be used as indices into the arrays containing the production rules and the terminal and non-terminal names.

The backwards parse starts at the last action to occur, and working back toward the first, performs the following:

1. If the action is a **reduce** action, then output an end tag for that non-terminal, and push it onto the stack. Also record the number of symbols (terminal or non-terminal) on the right-hand side of the production rule as that element's *children-count*

2. If the action is a **shift** action, output a tag for the token, and decrement the *children-count* for the topmost stack element.

3. While the *children-count* for the topmost stack element is zero, output its corresponding start tag, pop it from the stack, and decrement the *children-count* for the next stack element.

Note that only numbers corresponding to production rules are pushed onto the stack. Further, the stack only contains references to rules for which a start tag is still outstanding, rather than whole sections of the input program.

A side-effect of this approach is that empty start/end XML tags are generated for ϵ-rules that have an empty right-hand side. These were included to maximise the information that is made available from the parse, but such tags could be easily filtered out.

3.4. Adapting bison

To integrate this algorithm into the operation of bison, it is necessary to change the C-code output produced by the parser generator. This is considerably facilitated by the inclusion of the `--skeleton` option since bison version 1.28a (in August 2001). This allows the user to specify an arbitrary parsing routine, while bison will generate the necessary parse tables and lists of symbols to support it.

Thus, with a suitably modified parser skeleton, it is then straightforward to run bison over a given input grammar, generating a parser that will produce a XML tagged version of any valid input program. For versions of bison which do not support the `--skeleton` option, it is necessary to recompile the bison source code, with the new code substituted for the normal parsing code.

While the modified bison parser can be used on any bison-compatible grammar to generate XML output, the immediate application was in gcc. The C, Objective C, C++ and Java components of the gcc suite all use a bison-generated parser, and so could be used with our modified parser generator (unfortunately, the Fortran 77 and Ada parsers are written by hand). The only changes needed to the entire gcc source was in the Makefile. This just involved changing two flags; one to ensure that bison was run with the new skeleton, and one to ensure that the compiler flags were turned on in order to generate the output debugging code.

The simplicity of the changes to gcc is an important feature of our approach, since it means that we are not tied to any given version of gcc, and can easily modify future versions. As mentioned above, this also allows us to use old version of gcc with equal ease; for example, we have determined that a bison-generated C++ parser has been used in gcc since at least version 1.40.3 of October 1991.

[1] These figures were obtained using gcc under RedHat Linux 7.2, run on a 350MHz Intel Pentium II system.

```
1    int x;
2    void f(int);
3    typedef int g;
4    int main()
5    {
6        f(x);  /* Expression */
7        g(x);  /* Declaration */
8    }
```

Figure 3. *An example of the declaration/expression ambiguity in C*. **Both lines in the body of** `main()` **are identical to a simple context-free syntactic analysis, yet the first is an expression, whereas the second is a declaration**

4. Results

In this section we discuss some of the pragmatics of using our approach with gcc, and present some summary results to give an estimate of the overhead of using the system.

4.1. The declaration/expression ambiguity in C

Figure 3, lines 6 and 7, illustrates the declaration/expression ambiguity that occurs in the C programming language. The problem here is that both statements in the body of `main` are syntactically identical at the context-free level - an identifier, followed by another, parenthesised identifier. Since `f` is declared previously as a function, line 6 is parsed as a call to `f` with the global variable `x` as the argument. However, because of the declaration of `g`, line 7 is parsed as a declaration of a local variable `x` of type `int`, with redundant parentheses around the declarator. This type of ambiguity complicates the production of parsers and reverse engineering tools for C, and bedevils efforts to produce tools for C++, where the problems are considerably exacerbated - see reference [2, §6.8].

Figure 4 and Figure 5 are excerpts from the XML generated for this program by our modified bison-produced output, working as a part of the gcc compiler, version 3.0.4. It can clearly be seen that in Figure 4 the first statement is being parsed as an `<expr>`, whereas in Figure 5 the second is parsed as a `<decl>`. While the XML output is just a parser tree, we note that the tags effectively encode the context-sensitive information that was derived by gcc during the parse.

4.2. Modifying the output for C and C++

One other point of note from Figures 4 and 5 is that the tags for terminal symbols are carrying an extra attribute, labeled `attrib`. This is also a relatively new feature of bison which allows users to enhance debugging information by defining a macro `YYPRINT` showing how to print these. Fortunately, the GNU C and C++ compilers take advantage of this, thus allowing variable names to appear in the XML output.

It should be noted that this feature is not specific to the C and C++ components of gcc, and will work for all bison parsers that define the `YYPRINT` macro. The Java compiler from gcc does not, unfortunately, use this at present. It is relatively easy to add this facility to the gcc Java compiler, but it does require modification of the gcc source code.

One further problem with the default output from bison is the amount of data generated from even relatively simple programs. While some of this can be attributed to the complexity of the grammar, it is due mostly to the inclusion of various header files in programs, which expands the amount of code seen by the parser quite considerably. Needless to say, this is not an issue with the Java compiler from gcc which does not use a preprocessor.

The solution we have implemented at the moment is an attempt to minimise the modifications that needed to be made to gcc, so that they will be portable over multiple versions of the compiler. To this end, we have added a facility to the compiler to process a new type of `#pragma` directive, one that has the effect of turning XML generation on and off. This has the disadvantage of meaning that the source programs need to be modified to insert the pragma directive, but has the advantage of minimising the impact on gcc. It is relatively simple to write a script to insert these directives automatically around all system file inclusions, and the modification to gcc required only a dozen lines of code.

It is arguable that the solution to place pragma directives around the system include files is somewhat messy, but it represents a compromise between the application-independence of our modifications to bison, and the need to produce realistic amounts of output for C and C++ programs. Further work is required to see if a more elegant solution may be found here.

4.3. Some Results

In order to test the output of our XML-enhanced version of gcc, we ran the compiler over a number of programs from a benchmark suite designed as part of a study to compare several C++ fact generators [20]. While our use of the benchmark suite is not strictly in keeping with its intent, we chose this suite as the source programs are freely available, and likely to be familiar to the reverse-engineering community.

We used the “Accuracy” section of the suite, since our version of gcc would not be capable of dealing with the “Robustness” section. Figures 6 and 7 summarise the

```
<simple_stmt>
  <expr><expr_no_commas><cast_expr><unary_expr><primary>
    <notype_unqualified_id>
      <TOKEN type="IDENTIFIER" attrib=" ‘f’">
    </notype_unqualified_id>
    <TOKEN type="’(’">
    <nonnull_exprlist><expr_no_commas><cast_expr><unary_expr><primary>
      <notype_unqualified_id>
        <TOKEN type="IDENTIFIER" attrib=" ‘x’">
      </notype_unqualified_id>
    </primary></unary_expr></cast_expr></expr_no_commas></nonnull_exprlist>
    <TOKEN type="’)’">
  </primary></unary_expr></cast_expr></expr_no_commas></expr>
  <TOKEN type="’;’">
</simple_stmt>
```

Figure 4. *The XML output for the expression* `f(x)`**. This XML corresponds to line 6 of the C program representing the declaration/expression ambiguity.**

```
<simple_stmt>
  <decl>
    <typespec><complete_type_name><nonnested_type><type_name>
      <TOKEN type="TYPENAME" attrib=" ‘g’">
    </type_name></nonnested_type></complete_type_name></typespec>
    <initdecls><initdcl0>
      <declarator><notype_declarator><direct_notype_declarator>
        <TOKEN type="’(’">
        <expr_or_declarator_intern><expr_or_declarator><notype_unqualified_id>
          <TOKEN type="IDENTIFIER" attrib=" ‘x’">
        </notype_unqualified_id></expr_or_declarator></expr_or_declarator_intern>
        <TOKEN type="’)’">
      </direct_notype_declarator></notype_declarator></declarator>
      <maybeasm></maybeasm><initdcl0_innards><maybe_attribute></maybe_attribute></initdcl0_innards>
    </initdcl0></initdecls>
    <TOKEN type="’;’">
  </decl>
</simple_stmt>
```

Figure 5. *The XML output for the declaration* `g(x)`.**This XML corresponds to line 7 of the C program representing the declaration/expression ambiguity.**

---

results of running the modified gcc. Four of the programs, `classmember.cpp`, `Mover.C`, `Sayer.C` and `animals.C` had to be modified slightly, to deal with gcc's insistence on qualifying names from included system header files with the `std::` qualifier. Other than this, all the files were processed without error.

The data in Figure 6 seeks to determine the overhead caused by the production of XML using gcc, including the reverse parsing and file writing. For each file we give the time taken in seconds[2] to process the file without and with XML generation, and in the last column, show the degree of slowdown. The last column is calculated by dividing the time with XML generation by the time without XML generation. As can be seen from this table, XML generation slows the parsing process by an average of 25 times over ordinary parsing.

The data in Figure 7 gives an estimate of the "size" of the generated files. The second and third columns give an estimate of the length of each file, in terms of the number of lines of code before preprocessing, and the number of non-blank lines of code after preprocessing. The fourth and fifth columns show the number of tags for terminal symbols, and the number of pairs of start/end tags for non-terminal symbols respectively. The last column, calculated by dividing the previous two, shows that on average there are three non-terminal start/end pairs for each non-terminal tag.

5. Related Work

While we have presented our work in terms of its ability to tag C and C++ programs with syntactic information in XML, it should be noted that the technique is not limited to a single programming language or compiler. By linking XML production to the bison parser generator we provide for a greater range of application of our approach, but sacrifice the ability to include language-specific semantic information. In this context, our approach is most readily comparable to that of reference [16], which outlines a rough framework for XML production from top-down parsers. It is mentioned that this technique could also be applied to bottom-up parsers, but it is not evident from the paper that this has actually been fully worked out or implemented.

While our approach is not specifically linked to C or C++, it is useful to compare it with other approaches in this area. There has already been considerable work done on developing schemas for representing facts about programs; some of the more prominent include schemas such as Datrix [9], Columbus [7] and Harmonia [4]. More generic schemas include those based on GXL [10], as well as the WoSEF effort [21] to develop a standard exchange format. Each of these approaches produces a program representation with more semantic information than ours. We see our approach as being useful in producing an initial XML representation of a program, that may then be transformed into a schema with semantic content.

At the opposite end of the scale is the work on srcML [13], which notes that information regarding file position and comments is often important for fact extraction. Thus, this approach seeks to retain information even from the preprocessing stage; a non-trivial issue given the complexity of the preprocessor for C and C++. While such information is clearly useful, it is not obvious how to integrate this into our language-independent approach, although file position can now be handled in bison using the @$ and @n attributes.

The CPPX tool [5] is similar to our work in that it uses gcc as a front-end. It differs in that it dumps information from a later phase of the compiler's operation. This has the advantage of including more semantic information, but involves a tighter coupling with the gcc sources.

Reference [20] measures four fact extractors for C++ in terms of their robustness and accuracy. Each of these four tools produces significantly more semantic information than our approach, and the tests applied analyse their performance based on their ability to retrieve semantic facts. From the perspective of our work here, it is notable that each of these extractors employs a third-party front end, and thus focuses on the "back-end" schema and fact generation. We hope that our work can contribute to the provision of a lightweight front end for C++ and other languages. We see the XML syntax tree produced by our approach as a basis for transformation into more complex schemas, helping to decouple the "compiler" related parsing issues from those issues more directly related to the field of reverse engineering.

6. Conclusion

In this paper we have outlined a general algorithm for the modification of the bison parser generator, so that it can produce a parse tree in XML format. We have also discussed an immediate application of this technique, a portable modification of the gcc compiler, that then allows for XML output for C, Objective C, C++ and Java programs.

By modifying bison rather than gcc directly, we have produced a tool that is applicable in any domain that uses the bison parser generator and, in particular, is directly applicable to multiple versions of gcc. While our approach does not have the same semantic richness as other approaches, it does have the advantage of being language independent, and thus re-usable in a number of different domains. We do not envisage it as a stand-alone product, but believe that it will be useful as a starting point for more language-specific tools.

Having outlined some of the features and advantages of

[2] All programs were run under Redhat Linux 7.2 on a PC with a 350MHz Pentium II processor and 128MB of RAM.

Filename	Time in seconds standard	With XML	Rate of slowdown
preproc/1/preproc1.C	5.43	136.86	25.2
preproc/2/preproc2.C	4.82	140.39	29.1
preproc/pragma/a04.cpp	4.74	121.20	25.6
syntax/array/main.C	6.40	124.77	19.5
syntax/array/poly.C	5.15	129.45	25.1
syntax/enum/enum.c	0.63	2.46	3.9
syntax/enum/enum.cpp	0.20	3.46	17.0
syntax/exceptions/exception.cpp	4.87	133.60	27.5
syntax/fcns/main.C	3.71	125.19	33.8
syntax/fcns/multiply.C	0.07	0.13	1.7
syntax/fcns/sort.C	4.80	121.31	25.3
syntax/fcns/squared.C	0.06	0.06	1.0
syntax/inherit/Mover.C	6.23	125.46	20.1
syntax/inherit/Sayer.C	5.23	124.52	23.8
syntax/inherit/animals.C	5.33	125.34	23.5
syntax/namespace/main.C	4.83	123.78	25.7
syntax/namespace/ns.C	3.49	175.21	50.2
syntax/namespace/ns2.C	0.05	0.09	1.9
syntax/operators/addition.cpp	5.01	124.97	24.9
syntax/operators/fcall.cpp	5.09	124.95	24.5
syntax/struct/struct.C	5.17	123.81	24.0
syntax/templates/classmember.cpp	7.24	148.88	20.6
syntax/templates/function.cpp	5.71	145.13	25.4
syntax/union/union.cpp	5.04	124.67	24.7
syntax/vars/vars.C	4.84	120.77	24.9
average	*4.17*	*105.06*	*25.2*

Figure 6. *Timing results the analysis of C++ programs.* **Here we show each of the programs tested, along with the time taken to parse the program with and without XML generation. The final column shows the slowdown factor caused by XML generation.**

Filename	orig LOC	nb, pp LOC	Terminals	Non-Terms.	T/NT
preproc/1/preproc1.C	88	17934	117111	376499	3.2
preproc/2/preproc2.C	28	17641	111204	358821	3.2
preproc/pragma/a04.cpp	110	17679	111309	359212	3.2
syntax/array/main.C	34	17671	111386	359420	3.2
syntax/array/poly.C	58	17975	117438	377442	3.2
syntax/enum/enum.cpp	19	515	3429	11710	3.4
syntax/enum/enum.c	18	407	2428	8000	3.3
syntax/exceptions/exception.cpp	49	17658	111231	358918	3.2
syntax/fcns/main.C	23	17944	117243	376876	3.2
syntax/fcns/multiply.C	16	12	66	237	3.6
syntax/fcns/sort.C	62	17703	111711	360413	3.2
syntax/fcns/squared.C	3	3	14	55	3.9
syntax/inherit/animals.C	52	17740	111642	360424	3.2
syntax/inherit/Mover.C	83	17729	111652	360384	3.2
syntax/inherit/Sayer.C	64	17692	111446	359736	3.2
syntax/namespace/main.C	25	17656	111245	359081	3.2
syntax/namespace/ns2.C	18	13	44	153	3.5
syntax/namespace/ns.C	17	17640	111140	358793	3.2
syntax/operators/addition.cpp	33	17634	111161	358801	3.2
syntax/operators/fcall.cpp	31	17633	111126	358680	3.2
syntax/struct/struct.C	18	17646	111289	359124	3.2
syntax/templates/classmember.cpp	41	19553	128305	412119	3.2
syntax/templates/function.cpp	29	19545	128247	411946	3.2
syntax/union/union.cpp	47	17659	111255	359020	3.2
syntax/vars/vars.C	64	17682	111366	359441	3.2

Figure 7. *Size data from the analysis of C++ programs.***This table lists the number of original lines-of-code and the number of non-blank, preprocessed lines-of-code in each file. It then shows the number of terminal XML tags and non-terminal pairs of XML tags. The final column is calculated by dividing the number of non-terminal pairs of tags by terminal tags.**

our approach above, it is worth noting some drawbacks. First, the output format is necessarily dependent on the non-terminals used in the corresponding bison input file. As well as not being directly in conformance with the standard grammars (e.g. for the gcc C++ parser), these are subject to change as the parser is modified. Second, the tagging is not generic between languages as, again, it is grammar dependent. Thus a language construct shared between e.g. C, C++ and Java may be tagged differently if the grammar-writer chooses to use different names for the non-terminals. Finally, our approach does not, of course, work for hand-coded parsers; in particular, the Ada and Fortran compilers from gcc use a hand-coded parser, and thus are not amenable to our approach.

However, we believe that despite these caveats, the approach described here can still prove useful in the development of program analysis tools. Our goal here was not to replace more semantically-rich schemas and tools, such as those described in the last section, but to facilitate their integration with more accurate front-ends, as exemplified by the GNU compiler collection. We hope to extend this work to provide mappings between our outputs for C++ and some of the more standard schemas described above.

References

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

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

[3] F. Bodin, P. Beckman, D. Gannon, J. Gotwals, S. Narayana, S. Srinivas, and B. Winnicka. Sage++: An object-oriented toolkit and class library for building Fortran and C++ restructuring tools. In *The second annual object-oriented numerics conference (OON-SKI)*, pages 122–136, Sunriver, Oregon, USA, 24-27 April 1994.

[4] M. Boshernitsan and S. Graham. Designing an XML-based exchange format for Harmonia. In *Seventh Working Conference on Reverse Engineering*, pages 287–289, Brisbane, Queensland, Australia, 23-25 November 2000.

[5] T. Dean, A. Malton, and R. Holt. Union schemas as a basis for a C++ extractor. In *Eighth Working Conference on Reverse Engineering*, pages 59–67, Stuttgart, Germany, 2-5 October 2001.

[6] H. Deitel, P. Deitel, T. Nieto, and P. Sadhu. *XML: How to Program*. Prentice Hall, 2001.

[7] R. Ferenc and A. Beszédes. Data exchange with the Columbus schema for C++. In *6th European Conference on Software Maintenance and Reengineering*, pages 59–66, Budapest, Hungary, 11-13 March 2002.

[8] R. Ferenc, S. Sim, R. Holt, R. Koschke, and T. Gymióthy. Towards a standard schema for C/C++. In *Eighth Working Conference on Reverse Engineering*, pages 49–58, Stuttgart, Germany, 2-5 October 2001.

[9] R. Holt and A. Hassan. E/R schema for the Datrix C/C++/Java exchange format. In *Seventh Working Conference on Reverse Engineering*, pages 284–286, Brisbane, Queensland, Australia, 23-25 November 2000.

[10] R. Holt, A. Winter, and A. Schurr. GXL: toward a standard exchange format. In *Seventh Working Conference on Reverse Engineering*, pages 162–171, Brisbane, Queensland, Australia, 23-25 November 2000.

[11] G. Knapen, B. Lague, M. Dagenais, and E. Merlo. Parsing C++ despite missing declarations. In *7th International Workshop on Program Comprehension*, Pittsburgh, PA, USA, 5-7 May 1999.

[12] J. Lilley. PCCTS-based LL(1) C++ parser: Design and theory of operation. http://www.empathy.com/pccts/, Version 1.5 13 July 1997.

[13] J. Maletic, M. Collard, and A. Marcus. Source code files as structured documents. In *10th International Workshop on Program Comprehension*, La Sorbonne, Paris, France, June 26-29 2002.

[14] B. Malloy, S. Linde, E. Duffy, and J. Power. Testing C++ compilers for ISO language conformance. *Dr. Dobb's Journal*, 337:71–78, June 2002.

[15] S. S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan-Kaufman, 1997.

[16] A. Nakhimovsky. Using parser-generators to convert legacy data formats to XML. In *XML Europe*, Berlin, Germany, 21-25 May 2001.

[17] J. Power and B. Malloy. An approach for modeling the name lookup problem in the C++ programming language. In *ACM Symposium on Applied Computing*, Como, Italy, 19-21 March 2000.

[18] S. Reiss and T. Davis. Experiences writing object-oriented compiler front ends. Technical report, Brown University, January 1995.

[19] J. Roskind. A YACC-able C++ 2.1 grammar, and the resulting ambiguities. http://www.empathy.com/pccts/roskind.html, Independent Consultant, Indialantic FL 1989.

[20] S. Sim, R. Holt, and S. Easterbrook. On using a benchmark to evaluate C++ extractors. In *10th International Workshop on Program Comprehension*, La Sorbonne, Paris, France, 26-29 June 2002.

[21] S. Sim and R. Koschke, editors. *Workshop on Standard Exchange Format (WoSEF)*, Limerick, Ireland, 6 June 2000. A workshop of the 22nd International Conference on Software Engineering.

[22] H. Sutter. C++ conformance roundup. *C/C++ User's Journal*, 19(4):3–17, April 2001.

[23] World Wide Web Consortium. Extensible markup language (XML). http://www.w3.org/XML/, Revision 1.225 23 April 2002.