An XML-Based Lightweight C++ Fact Extractor

Michael L. Collard, Huzefa H. Kagdi, Jonathan I. Maletic
Department of Computer Science
Kent State University
Kent Ohio 44242
330 672 9039
collard@cs.kent.edu, hkagdi@cs.kent.edu, jmaletic@cs.kent.edu

Abstract

A lightweight fact extractor is presented that utilizes XML tools, such as XPath and XSLT, to extract static information from C++ source code programs. The source code is first converted into an XML representation, srcML, to facilitate the use of a wide variety of XML tools. The method is deemed lightweight because only a partial parsing of the source is done. Additionally, the technique is quite robust and can be applied to incomplete and non-compile-able source code. The trade off to this approach is that queries on some low level details cannot be directly addressed. This approach is applied to a fact extractor benchmark as comparison with other, abet heavier weight, fact extractors. Fact extractors are widely used to support understanding tasks associated with maintenance, reverse engineering and various other software engineering tasks.

1. Introduction

Source code fact extraction is the process of extracting facts, entities, and the relationships, from source code given a specific query. It involves processing (e.g., parsing and/or searching) the source code to extract the particular facts, expressing a desired query, and formatting the output of the query.

Fact extractors are a vital tool for reverse, reengineering, maintenance, testing, and general development of software systems. They are used to help developers comprehend software by uncovering relationships between classes, modules, units, functions, etc. Fact extractors can be of great benefit in locating possible errors in source code as well as identify concerns of interest across a system.

In the approach presented here, we convert C++ source code into an XML representation, namely *srcML*[1] [6, 15]. This underlying representation is then leveraged via the API's, tools, and technologies of XML to give us a lightweight, robust, and tolerant C++ fact extractor. We use the term lightweight to highlight the fact that only lightweight parsing is done and a number of very low-level type facts can not be directly derived from the data source (i.e., srcML markup of the C++ source).

Our method allows the extraction of high-level entities such as functions, classes, namespaces, and templates, as well as middle-level entities such as individual statements (if, while, etc.), declarations and expressions. Lower-level entities such as variables and function calls can also be queried. Additionally, it allows the extraction of entities that are typically discarded during pre-processing such as comments, pre-processor directives, and macros. The entities are extracted with full lexical information such as white space and all original source code information.

The following section will address some of the problems encountered during fact extraction and address the related work in the field of fact extraction. We then describe srcML and our C++ to srcML translator. Additionally, we briefly address related XML source code representations. Our approach to using XML technologies to support fact extraction is then detailed and lastly the results of applying our method to a fact extraction benchmark [24] are given.

2. Extracting Facts from C++ Code

A number of challenging, well known, technical problems exist for building fact extractors for C++ [9, 11, 24]. Additionally, the work done by Sim et al [24] and the benchmark they developed uncovered a number of other problems relating to the types of questions and perspectives of the users of fact extractors. The results of researchers applying tools to this benchmark revealed that there are often many correct answers to the same question. The correctness depends on the perspective of the user and their particular software engineering task. Different tasks require different levels of detail about the system to support the particular type of comprehension necessary to complete the task. For example, a user may be interested in variable, type, or comment information while trying to understand a group of modules for reverse

[1] Pronounced, "Source ML".

engineering. Another user may need the possible level of function call nesting and dynamic typing for fault localization.

This gives credence to fact extractors with very different capabilities and complexities. While many fact extraction tools rely on a complete parsing of the entire system we have chosen another avenue that, we believe, augments those approaches.

One of the most important issues of fact extraction is the input itself (i.e., the source code). It is typically a single source code file and associated include files. In the best case you have a complete, compile-able system. In other cases there may be code fragments, compilation problems, or missing associated include files. The source code can be in a dialect of the original language(s) or it could be code that will compile under one version of a compiler but not another. Of particular interest here is that our approach allows fact extraction on most of these later situations. This can be a distinct advantage in many situations (e.g., platform change, library change).

The C++ language, in particular, is a challenging language to parse and extract facts from. This is due, in part, to the pre-processor and the numerous macro constructs that are used in conjunction with the language. Also, there are a great many versions and variations of C++ in wide use. However, the biggest problem facing someone wanting to construct a fact extractor is that C++ is defined by a non-Context Free Grammar. This makes full parsing of the language difficult and lexical analyzers can possibly produce incorrect results.

The other critical part of fact extraction is with respect to the query. The input to the fact extraction process is the specification of the desired fact. This may be in the form of a query language or it may require a specialized program to extract the answer. A simple specification input may be limited as to what facts can be extracted. However both of these approaches require a learning process on the part of the user.

Another issue for the specification is to what level of understanding the tool has for the language being processed. For example, does the tool already know how new types are introduced into the language or is it a part of the specification to list the language constructs that can form a new type. A very flexible tool may require configuration. The amount of configuration and the configuration language affect both the usefulness and the difficulty of using the tool.

Finally, we must consider the output of the fact extractor. If the extractor is being used as part of a larger process, such as for source model generation, then the output format of the fact is important to the ease of use of the result. If the extracted fact, such as a section of source code, is to undergo further processing then the tool should be able to output the extracted facts in its original format.

2.1. Fact Extractor Characterization

In [19] Murphy and Notkin describe an approach to source model extraction using the terms lightweight, flexible, and tolerant. Here, lightweight means extracting a new fact requires a relatively short specification. In the process view, this includes not only the search specification but the configuration of the tool. Flexible refers to what information from the original source code can be extracted, such as comments, macros, etc. This also includes using this information in the selection of the extraction, such as extracting functions with some particular content to their comments. Tolerant refers to how complete are the source code documents. There could be missing include files, the code may not compile, or a dialect of C++, such as from a particular version of the compiler, may have been used.

For XML to be used in the context of a lightweight fact extractor, the XML must also be processed in a lightweight form. This requirement is for both the XML markup language used and the time/memory requirements.

Our representation, srcML, and the translator we have developed uses a lexically based approach allowing the translator to be used on incomplete, non-compile-able code, and code fragments. Since this can generate incorrect results in certain cases, it has been constructed to allow for additional refinement of the translated result. These refinements include information from associated source code files and heuristics from the user.

2.3. Related Work (on C++ Fact Extraction)

Parser-based fact extractors include cppX [8], Acacia [1] and Columbus/CAN [12]. LSME (Lightweight Source Model Extraction), described in [19] is a tool for generating high-level source models using a regular-expression based specification language. The user can specify what they want to match in the source code or other system artifacts and the actions that they want performed. The system will produce a scanner that generates the system model. The specification is small and only has to be written for the needs of the particular source model that is being generated. There are no restrictions on the artifacts that the scanner can be applied to and few constraints on the condition of the artifacts.

In [19] the comparison is made between lexical and parser based approaches to source model extraction. The parser-based approaches are described as heavyweight when an extractor for a new language needs to be generated and as such are typically inflexible concerning the constraints on the kinds of artifacts and not tolerant of the (poor) condition of the source code.

The lightweight characteristic applies to the creation of a new extractor. A distinction is not made between creating a new extractor for a new source code language and creating a new extractor for extraction of a new system model.

In [7] the categorization of [21] is used to show that LSME lexically extracted unit level models but not syntactic level models. They extend this work by using both lexical and parsing techniques and comparing the results. The extraction is of individual entities such as function definitions, calls, statements, expressions, etc. Their comparison shows that lexically based approaches can produce useful results of the entity extraction that is performed.

3. C++ to srcML

In order to extract facts from C++ source code using XML tools both an XML representation of the C++ source code and a translator from C++ to the XML representation are needed. Both work in conjunction to support lightweight, flexible, and robust fact extraction. This section first discusses the particular XML representation, srcML, and the translator used, src2srcml, that support these characteristics. But first, we talk briefly about related source code representations.

3.1. Related Representations

A number of options currently exist for representing source code information (e.g., AST or ASG) in a XML data format namely, GXL [13], CppML [17], ATerms [25], GCC-XML, and Harmonia [4]. In these formats the AST (actually an ASG) of the source code, as output from a compiler intended for code generation, is stored in a data XML data format. In JavaML and GCC-XML the AST is mapped to the nested structure of XML. In GXL a graph view of the source code is stored, i.e., storing all nodes and vertices of the graph with no mapping of the nested structure of the source code to the nested structure of XML.

However, these representations are constructed as data exchange languages or for displaying program structural information. None of these representations directly supports the representation of comments or formatting information. The most widely used of these, GXL [13] is an XML-based exchange format for graph-like structures based on GraX (Graph eXchange format) [10], and RSF (Rigi Standard Format) [29]. Software systems are represented as ordered, directed, attributed, and/or typed graphs. While GXL is designed to be a standard exchange format for data that is derived from software, srcML is designed to represent the actual source code. Although srcML can be used as a standard exchange format, the underlying goal of defining and using srcML is to create an intermediate layer of representation between the source code, the developer, and tools that allows easy transformation to a standard exchange format such as GXL.

The most closely related work to srcML is Badros' work on JavaML [3], which is an XML application that provides an alternative representation of Java source code. JavaML is more natural for tools and permits easy specification of numerous software-engineering analyses by leveraging the abundance of XML tools and techniques. However, JavaML does not preserve the original source code document and discards much of the formatting information. As with srcML it keeps the comments in the text but it associates them to elements of the program. Therefore, the location of comments is not preserved. Additionally, all formatting information is lost in JavaML and the original source code document cannot be regenerated from JavaML representations.

In the same realm, the Harmonia framework [4] and cppML/JavaML developed at the University of Waterloo [17] are closely related approaches since they encode the AST itself and actual source code, rather than data extracted (such as the case in GXL). While Harmonia adds tags to source code as metadata, cppML only uses tags and records the additional information as attributes on the tags. The differences mentioned above for Badros' work stand for these approaches as well.

The XML data view of source code, since it is based on the AST, is a "heavyweight" format. It requires complete parsing of the original document and generation of the complete AST.

Other work on source code interchange formats includes the work by Malton et al [16]. While this work is not an XML application it has many of the same features as srcML and the other formats described previously. Malton's work addresses issues of design recovery through source factoring on legacy code.

3.2. srcML

As an alternative to these other representations srcML (SouRce Code Markup Language) [6, 15] is an XML application that is used to add structural information to raw source code text files. All original lexical information including comments, white space, preprocessor directives, etc. from the original source code are preserved in srcML with the syntactic information marked using elements. Figure 1 shows an example source code and figure 2 presents the source code representation in srcML.

In short, srcML is an attempt to keep the textual semantics of the source code intact while adding explicit structural information. This leaves us with a much richer representation to work with than plain text, but with all the flexibility.

The representation of the source code as structured documents directly supports the following:

- Representation of multiple levels of granularity within the AST;
- Multiple level of abstraction (or views);
- Transformation equality of source to representation and of representation to source;
- Query-able and search-able representation;
- Representation of structural information, including macros, templates, and compiler directives (e.g., #include), etc.;
- Preservation of:
    - Location of constructs;
    - Text formatting information;
    - Comments and their location;
    - File names and structure.
    - Macros and macro definitions

The feature of srcML that differentiates it from other related approaches is its ability to preserve semantic information from the source code.

3.3. srcML Translator

Translating source code to srcML is a multi-stage process and is shown in figure 3. The core unit is the srcML translator. ANTLR [20] is used to construct the srcML translator from a pred-LL(k) grammar specification and a context stack to maintain context information. Actions, both pre and post, are attached to the grammar specification to markup the source code with XML start and end tags of the appropriate syntactic structure. On identification of the beginning of syntactic structure a start XML tag is inserted in the token stream and a transition occurs in the context stack to reflect the state of the construct being parsed. When a statement or block terminating token is encountered, the context information from context stack is utilized to insert end XML tags for the appropriate closing structures. This approach of parsing is motivated by the island grammar concept and in particular the idea of an *island with lakes*[18].

An island grammar, as defined by Moonen [18] is a grammar consisting of detailed productions describing certain constructs of interest (i.e., the islands) and liberal productions that catch the reminder (i.e., the water).

```
// swap two numbers
if( a > b)
{
        t = a;
        a = b;
        b = t;
}
```

Figure 1. Source code swapping two numbers.

Island grammars can be expressed in any grammar specification formalism or parsing technique. Island grammars are suitable in identification and translation of high level (enclosing) constructs by eliminating details of low level (enclosed or constituent) constructs in the parsing phase. A variant of island grammars, *islands with lakes*, provide simple specification of constructs with recursive and nested definition such as conditional and iterative statements. Productions for a construct being parsed are considered as islands while productions for others are considered as water. Our interest is in identification of all the structural constructs of a C++ language. Therefore, all such constructs are considered as *islands*. However, all the components of these constructs are not of interest and thus only components of interest (*islands*) are parsed and marked up while others are ignored (but not discarded) as *water*. Another variant, *lakes with island*, provide simple specifications for embedded language constructs. Moonen [18] advocates the application of island grammar in generation of robust parsers for source model extraction and applications that do not need the complete parse tree.

Consider the source code fragment shown in figure 1 and its corresponding srcML representation in figure 2. At first, the comment is identified in the lexical stage, hidden from the parsing stage and is marked as first class element. On seeing the "if" token, the start tag for the if statement is inserted and context stack is updated to a state of open "if" structure. Next the conditional expression is parsed and appropriate start and end tags

```
<unit>
<comment type="line">// swap two numbers</comment>
<if>if<condition>(  <expr><name>a</name> &gt; <name>b</name></expr>)</condition><then>
<block>{
        <expr_stmt><expr><name>t</name> = <name>a</name></expr>;</expr_stmt>
        <expr_stmt><expr><name>a</name> = <name>b</name></expr>;</expr_stmt>
        <expr_stmt><expr><name>b</name> = <name>t</name></expr>;</expr_stmt>
}</block></then></if>
</unit>
```

Figure 2. srcML representation of source code swapping two numbers.

are inserted. Also, artificial start and end of tokens are introduced to support queries to the then block. In our implementation, the expression statement is a loose grammar specification. Anything ending with semicolon and not matching any other construct specification is parsed as expression statement. All the expression statements in "then" blocks are marked accordingly. On seeing the "{", block terminating token the context stack is utilized to mark the end of "if" structure.

Parsing proceeds in multiple passes with higher level entities parsed and augmented with XML tags in the first pass and lower level entities in subsequent passes. The srcML translator provides a simple Context-Free (CFG) srcML representation of C++ source code. Later processing can be used to refine srcML representation to deal with non-CFG issues. In the remaining part of the section we discuss ANTLR and multi-pass/multi-stage parsing.

3.4. ANTLR: pred-LL(k) Parser Generator

The class of languages recognized by LL(k) parsers can be extended with semantic and syntactic predicates to determine the application of a production. This results in the class of languages recognized by pred-LL(k) parsers[20]. Semantic predicates resolve syntactic ambiguities by using context information allowing for context-sensitive actions as part of the grammar specification. In our system this is only used with a context stack. No symbol table information is used in the predicates since no symbol table is maintained. Syntactic predicates resolve conflicts requiring infinite look-ahead by using selective backtracking with finite look ahead. Syntactic predicates provide a simple resolution to non-deterministic decisions. In our system, syntactic predicates provide the capability to resolve ambiguities arising due to a common left-prefix between certain C++ constructs (e.g., function declaration and definition). These extensions of conventional LL(k) grammars allows for readable and intuitive grammars for languages like C++.

ANTLR (Another Tool for Language Recognition) [2] is a tool for automatically constructing recognizers, compilers and translators in C++ or Java from a LL(k) grammar specification of a language combined with C++ or Java actions. ANTLR provide similar syntax and analysis (pred-LL(k) grammar) specifications for both parsers (tree-parsers and token stream parsers) and lexers. ANTLR considers lexical analysis to be parsing on a character stream. ANTLR allow both structural and behavioral grammar inheritance. The derived grammar can specify different actions for the same structure defined in base grammar. ANTLR supports both semantic and syntactic predicates. ANTLR also provides rich and flexible error handling and recovery. ANTLR provides a basic structure for filtering and splitting token streams between lex and parser. The ANTLR group is investigating the similar ability for parser to generate a stream of tokens as output. Multi-pass parsing would be simplified as parsers would also become stream producers.

3.5. Multi-Pass and Multi-Stage Parsing

Parsing proceeds with higher-level entities being parsed and marked appropriately before going into the details of constituent or lower level entities. The subsequent levels are processed in latter passes. This hierarchal approach enables us to control parsing at the desired level of interest. Additionally, source code irregularity in syntax present at one level does not impact parsing at other levels. Multi-pass parsing together with our partial grammar specification approach supports an event driven interface to the source code. Multi-pass parsing allows issuing events without having to wait for arbitrarily long parsing to resolve non-determinism

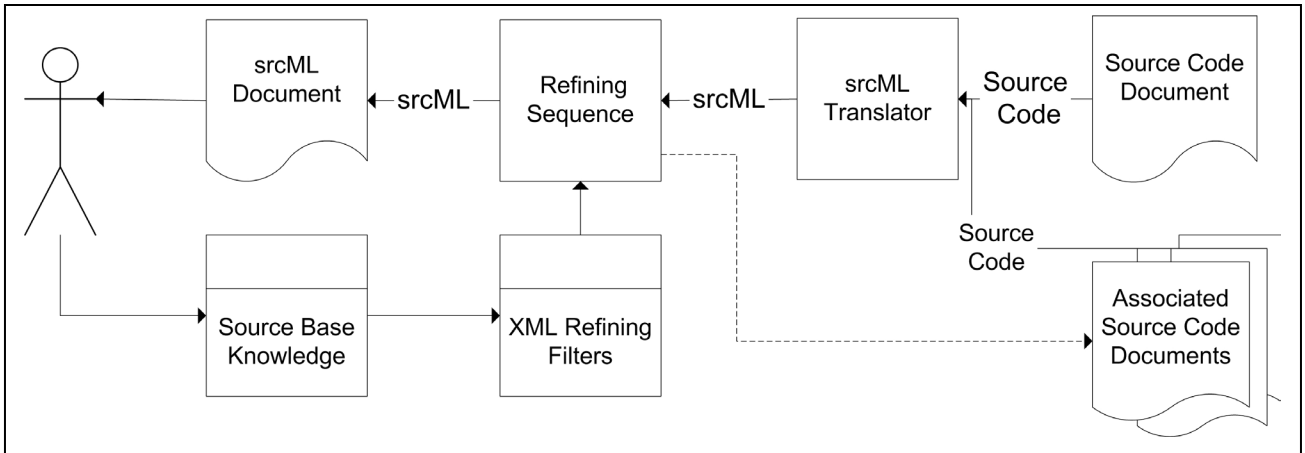

Figure 3. Translation of source code into srcML

between ambiguous structures with common left-prefix that require infinite lookup requirements (e.g., function definition and declaration ambiguity can be resolved without parsing possibly arbitrary list of parameters).

The srcML translator takes into account only a CFG view of the non-CFG C++ grammar. This transparent view of the translator introduces problems of misidentification of constructs that are syntactically identical at the context free level. This kind of ambiguity complicates the production of parsers and fact extraction tools for C++. To deal with this problem we have designed the entire translation process in stages. The first stage translates the source code into an XML representation. This basic representation may contain some inaccurate markups of ambiguous structures. The following sequence of configurable, refinement stages rectifies this inaccuracy by processing include files (if present), applying rules based on knowledge of built-in or native types, applying heuristics, and integrating user's source code knowledge. All these refinement filters are written as an XML transformation programs.

One refining filter uses keyword types to refine the base translator output. For example, we know that if the only parameter is a keyword type, such as "int", then we have a function declaration, not a variable declaration. Another filter process the include files to find out the declared types and uses that information to refine the srcML. This filter is optional since the include files are not always present. It is not necessary for all included files to be processed. The more that are available the more accurate the translation can be, but even without processing the include file we still have the base translator information.

The flexibility of using the translator comes from the stages that can be applied to the output of the srcML translator. The filters may update the existing srcML with a more accurate picture of context, or may be used to transform the results into another format. This allows the user to extend the srcML representation to better their specific task at hand.

4. Fact Extraction using XML

The srcML translator only provides an XML document view of a source code document. To actually do the fact extraction a number of standard XML tools are utilized. We now briefly cover the APIs and standards that we use to support fact extraction then we describe the use of XPath for fact extraction.

DOM (Document Object Model) [26] is an API that provides access to the XML document as a tree. User programs can use the API for sequential and random access to the XML document. The DOM is defined using a generic API with bindings to programming languages such as Java, C, C++ and Python.

A fact extraction program using the DOM has no restrictions on the order of DOM access so any fact extraction algorithm can be directly implemented. However the overhead of construction and storage of the DOM tree in memory can be costly with large source code documents or a large number of source code documents.

SAX (Simple API for XML) [22] is an event-driven Java API for XML documents. Parsing events, such as element start tags, text, element end tags, are delivered in sequential order for the user program to process. Unofficial bindings to other programming languages do exist.

For fact extraction programs SAX is the most efficient. Only the part of the tree necessary for the fact being extracted (e.g., extracting all of the types used) is constructed and stored. The disadvantage is that the program must include code to store any needed results between events. In addition the lack of an official standard in any language except Java creates portability concerns if Java is not used.

4.1. XPath and Fact Extraction

XPath [27] is a language for addressing parts of an XML document. An XPath expression is the address of a single (or multiple) part of the XML document. In addition to describing a path into the XML document, XPath expressions can also include predicates and string manipulation. XPath is normally used inside another tool, such as XSLT or STX or is used with an API to extract parts of the XML document for further processing. XPath is a subset of XQuery, an XML Query language.

4.2. XSLT and Fact Extraction

XSLT (extensible StyLesheet Language) [28] is a programming language specifically designed for transformations of XML documents. An extension of XPath is used to match and process parts of the XML document tree. Various XSLT processors exist including Xalan, Saxon, and xsltproc.

A fact extraction program requiring random (versus sequential) access to the XML document can be written in XSLT. This provides more support than using a general-purpose programming language with the DOM. However, the program has many of the same memory and time requirements of the DOM since a DOM-like tree is constructed internally by XSLT processors.

4.3. STX and Fact Extraction

STX (Streaming Transformations for XML) [5] is another programming language specifically designed for XML transformations. The difference between STX and

XSLT is that STX works off of input from a SAX interface. Only a subset of XPath expressions are supported since the entire XML document tree is not stored.

A fact extraction program requiring only sequential access to the XML document can be written in STX. This provides more support than using a general-purpose programming language with SAX.

4.4. Querying using XPath

Once in srcML, the source code is query-able using XPath. XPath expressions can be used to specify a particular point in the source code and are used to extract the fact or parts of source code.

For extracting specific language entities simple XPath expressions may be used. The XPath expression

```
/unit/function
```

finds all function definitions at the top-level of the document.

Function definitions can occur at any level in the XML document including inside namespaces and pre-processor block directives. To extract function definitions at any level the XPath expression

```
//function
```

does so by looking for the function element starting at the top and looking at any level in the XML document tree

To extract an entity in the context of other entities requires expressing a path in that context. The XPath expression

```
//function//if
```

locates any if statements at any level inside a function definition. The XPath expression

```
//function/block/if
```

finds any if statements inside of a function that are not nested inside another statement (they are at the top-level inside the block of the function). To find a particular entity by name we use XPath predicates. The XPath expression

```
//function[name='convert']
```

finds the function definition with a name of convert. We can get more specific at this point and find all if statements inside of the function with the name convert by using

```
//function[name='convert']//if.
```

XPath expressions of this type can be used with any of the entities that srcML including functions, classes, statements, types, comments, pre-processor directives, etc.

A combination of tools, starting with src2srcml, were used for the benchmark fact extraction. They include:

- src2srcml - Source code to srcML translator
- xpath - Execution of XPath statements on srcML.

The xpath tool uses the Perl module XML::XPath to query XPath statements in XML documents.

The execution of the XPath statements can be done by any XPath enabled tool. The tool xpath was chosen for its simplicity and easy integration to the fact extraction process.

The typical output of an XPath query is an XML document fragment. Fact extraction queries often return the source code itself or the line number of where the source code is located. Because of the direct traceability of srcML the document fragment can be directly translated back to the original source code fragment. The tool used to do this is srcml2src. It converts from srcML back to the original source code. This is a simple script using stripsgml, which is a part of the perlSGML package.

XPath statements refer to specific points in the srcML document. Fact extraction questions, including those in the benchmark, often ask for a line number or line count as the answer. The XPath statement can be directly translated into the line number in a particular document. The tool that was used is srcpath2line. It translates from an XPath statement to a line number in a source file and is a simple program written in the event-driven XML transformation language STX.

Any conversion from srcML to another format, such as the original source code, line number, etc., becomes the last stage of the fact extraction process.

5. Benchmark Results

In order to determine the needs of a fact extractor the CppETS 1.1 [23] benchmark for C++ fact extractors was used as a test bed. This benchmark has been applied to many of the parser-based fact extractors previously discussed and is a good choice since it helped to define exactly what was meant by fact extraction.

The benchmark consists of 19 test buckets in the category of accuracy and 10 in the robustness category. There are a total of 99 questions. The file sizes ranged from 46B to 47KB and the corresponding srcML representation ranged from 851B to 63.2KB with a ratio ranging from 1.251 to 7.586.

The srcML translator and the XML tools for extraction described in the last section were applied to

this benchmark. The remainder of this section describes these results.

5.1. Format of the answer

The benchmark had a variety of ways to format the output. In some cases, such as for a statement, the requested output should be the actual code. In other cases the line number, range of line numbers, and number of bytes is requested.

The form of the output did not affect whether the fact is extractable or not. The XPath expression to extract the answer is applied to the xpath tool when the actual code is requested and to the srcpath2line tool when the line number is requested.

There were a small number of questions that requested a byte count. We do not have a specific tool to calculate this type of value.

5.2. Entities in isolation

Many of the questions concerned the direct extraction of entities with no information in regards to their context in the source code. Since these are directly marked with tags in srcML simple XPath expressions, along with the xpath tool, were able to directly extract these entities using XPath in the manner described in the last section. For example, to extract the named namespaces the XPath expression `//namespace` was used. The entities extracted in this way include variable declarations and uses, function declarations and definitions, pre-processor directives, namespaces and templates. These were primarily questions that referred to entities that the programmer defines.

5.3. Entities in context

For other extractions the context of the entity was important. For example, the same tags are used in srcML for any type of variable declaration, both for global variables, local variables and class data members. If a class data member was requested, then the XPath expression `//class//decl` was used.

This is part of the tradeoff over using specific tags for specific uses versus general tags. In some cases the context must be used.

5.4. Scope & Type

The issue of scope was too complex in all but the simplest cases to solve in a straightforward manner with the tools that we used. An example is the linking of the use of a variable in a function with the declaration in the class that it is a friend.

A similar problem occurs with questions involving the type. Simple type questions are relatively easy to solve by extracting the type from the declaration. But given a use of a variable in any statement it is too complex to determine its type. This would have required a (partial) symbol table to be built requiring a further processing stage.

5.6. Entities extracted using string matching

There are examples of programming constructs that do not directly relate to a keyword or special symbol. For example, pure virtual functions have the "= 0" at the end of their function declaration.

Since all the original text is preserved it is possible to detect this very specific textual pattern for a pure virtual function by comparing the text at the end of the virtual function declaration. XPath does include string matching and may include regular expression matching in a future version. However, this was not included in the results since it seemed to be a hack around something missing in srcML.

5.7. Extraction with missing files

The robustness test buckets contained examples of missing files, including missing include files and libraries. With the tools that are used, the missing files did not affect the result of answering the questions. A difference would have been seen had an additional tool that applied XPath expressions across files were to be used. This would have increased the number of questions that this approach could have answered in the other buckets.

5.8. Pre-Processor

The preprocessor directives are straightforward to extract. However, it is beyond the scope of the tools used to attempt to extract the source code that is to be used given arbitrary values for the pre-processor symbols.

This is one area where a specialized tool could be built that worked on the srcML representation. The tool could go through all pre-processor directives and keep track of their current values. This new tool could have answered all of the pre-processor directive questions fully.

5.9. Dialects

The robust section included different dialects of C++ and the results were mixed. For the test bucket with MS Visual C++ extensions, the extraction was successful. For the g++ extension test bucket the extraction was not successful.

The difference is the nature of the language extension. The MS Visual C++ extension includes the addition of a

	Fact Extractor	Full Answer	Partial Answer	No Answer
Previous Benchmark Results	Acacia	32%	16%	52%
	Columbus	19%	11%	70%
	Cppx	45%	19%	35%
	TkSee/SN	28%	18%	54%
	srcML Translator	44%	8%	48%

Table 1. Summary of benchmark results compared to previous results as presented at IWPC'02

type keyword. Since srcML has a generic type and specifier element it handled the added keyword easily. The g++ language extension is not an added type or specifier keyword and did not fit into the existing grammar as easily.

The extractor failed the IBM Visual Age test bucket not due to a language extension, but due to the use of a macro.

A summary of our results applying our tool to the benchmark is combined in Table 1 to the results of the IWPC'02 Working Session.

6. Conclusions and Future Work

The results of applying our lightweight fact extractor to the benchmark are quite reasonable in comparison to the published results of the other types of tools. The other tools (Ccia, cppx, Columbus, and TkSee/SN) were able to answer approximately the same number of questions. However, the application of the tools to the benchmark resulted in [14] large improvements for a number[2] of the tools.

Since the output format is in srcML it is easily converted to other required formats, such as another XML format and even back to source code. Other tools can take multiple fact extractor results and combine them to form higher level source models. Future work will explore the conversion of the fact extractor output to the required input format of tools that can use these results, i.e., visualization tools, source model generators, etc.

Given the lightweight approach used here this is a reasonable and simple method to use for many tasks of fact extraction. Our results also indicate that the approach will scale well due to the reasonable srcML file sizes and the capability of event-driven translation.

We are working to extend our current set of tools and completely implement the C++ to srcML translator. It is fairly robust but still is not complete.

The DTD (Document Type Definition) for srcML, and our C++ to srcML translator, is available on the web page of the Software Development Laboratory <SDML>, at Kent State University (www.sdml.cs.kent.edu).

7. Acknowledgements

This work was supported in part by grants from the National Science Foundation CCR-02-04175 and the Office of Naval Research N00014-00-1-0769.

8. References

[1] Acacia, "Acacia - the C++ Information Abstraction System", AT&T, Web page, Date Accessed: 11/01/2001, http://www.research.att.com/sw/tools/Acacia/, 2001.

[2] ANTLR, "The ANTLR Translator generator", Web page, Date Accessed: 11/01/2001, http://www.antlr.org/, 2001.

[3] Badros, G. J., "JavaML: A Markup Language for Java Source Code", in Proceedings of 9th International World Wide Web Conference (WWW9), Asterdam, The Netherlands, May 13-15 2000.

[4] Boshernitsan, M. and Graham, S. L., "Designing an XML-Based Exchange Format for Harmonia", in Proceedings of Seventh Working Conference on Reverse Engineering (WCRE'00), Brisbane, Australia, November 23-25 2000, pp. 287-289.

[5] Cimprich, P., "Streaming Transformations for XML (STX) Version 1.0 Working Draft", http://stx.sourceforge.net/, Web page, Date Accessed: 11/15/2002, http://stx.sourceforge.net/documents/spec-stx-20021101.html, 2002.

[6] Collard, M. L., Maletic, J. I., and Marcus, A., "Supporting Document and Data Views of Source Code", in Proceedings of ACM Symposium on Document Engineering (DocEng'02), McLean VA, November 8-9 2002, pp. 34-41.

[7] Cox, A. and Clarke, C., "A Comparative Evaluation of Techniques for Syntactic Level Source Code Analysis", in Proceedings of 7th Asia-Pacific Software Engineering Conference (APSEC'00), Singapore, 2000, pp. 282-291.

[8] CPPX, "CPPX - Open Source C++ Fact Extractor", Web page, http://swag.uwaterloo.ca/~cppx/, 2001.

[9] Dean, T. R., Malton, A. J., and Holt, R. C., "Union Schemas as a Basis for a C++ Extractor", in Proceedings of Eighth Working Conference on Reverse Engineering (WCRE'01), Stuttgart, Germany, October 2-5 2001, pp. 59-70.

[10] Ebert, J., Kullbach, B., and Winter, A., "GraX — An Interchange Format for Reengineering Tools", in Proceedings of Sixth Working Conference on Reverse Engineering (WCRE'96), Atlanta, GA, October 6-8 1999, pp. 89 - 100.

[2] As presented during a working session at IWPC 2002.

[11] Ferenc, R., Gyimóthy, T., Sim, S. E., Holt, R. C., and Koschke, R., "Towards a Standard Schema for C/C++", in Proceedings of Eighth Working Conference on Reverse Engineering (WCRE'01), Stuttgart, Germany, October 2-5 2001, pp. 49-58.

[12] Ference, R., Magyar, F., Beszedes, A., Kiss, A., and Tarkiainen, M., "Columbus – Tool for Reverse Engineering Large Object Oriented Software Systems", in Proceedings of In Proceedings of SPLST 2001, June 2001 2002, pp. 16-27.

[13] Holt, R. C., Winter, A., and Schürr, A., "GXL: Toward a Standard Exchange Format", in Proceedings of 7th Working Conference on Reverse Engineering (WCRE '00), Brisbane, Queensland, Australia, November, 23 - 25 2000, pp. 162-171.

[14] Kienle, H., "A Benchmark for C++ Fact Extractors: Results and Observations", IWPC, Web Page, Date Accessed: 11/15/2002, http://cedar.csc.uvic.ca/kienle/view/IWPC2002/Workshop, 2002.

[15] Maletic, J. I., Collard, M. L., and Marcus, A., "Source Code Files as Structured Documents", in Proceedings of 10th IEEE International Workshop on Program Comprehension (IWPC'02), Paris, France, June 27-29 2002, pp. 289-292.

[16] Malton, A. J., Cordy, J. R., Cousineau, D., Schneider, K. A., Dean, T. R., and Reynolds, J., "Processing SOftare Source Text in Automated Design Recovery and Transformation", in Proceedings of IEEE 9th International Workshop on Program Comprehension (IWPC'01), Toronto, Canada, May 12-13 2001, pp. 127-134.

[17] Mammas, E. and Kontogiannis, C., "Towards Portable Source Code Representations using XML", in Proceedings of 7th Working Conference on Reverse Engineering (WCRE '00), Brisbane, Queensland, Australia, November, 23 - 25 2000, pp. 172-182.

[18] Moonen, L., "Generating Robust Parsers using Island Grammars", in Proceedings of 8th IEEE Working Conference on Reverse Engineering (WCRE'01), Suttgart, Germany, October 2-5 2001, pp. 13-24.

[19] Murphy, G. C. and Notkin, D., "Lightweight Lexical Source Model Extraction", ACM Transactions on Software Engineering and Methodology, vol. 5, no. 3, July 1996, pp. 262-292.

[20] Parr, T. J. and Quong, R. W., "Adding Semantic and Syntatic Predicates To LL(k): pred-LL(k)", in Proceedings of International Conference on Compiler Construction (to appear), 1994.

[21] Perry, D., "Software Interconnection Models", in Proceedings of Internation Conference on Software Engineering, March 1987, pp. 61-69.

[22] SAX, "Simple API for XML (SAX)", SAX, Web page, Date Accessed: 01/20/2002, http://www.saxproject.org/, 2001.

[23] Sim, S. E., "CppETS Benchmark", http://cedar.csc.uvic.ca/kienle/view/IWPC2002/Benchmark, Tar Zipped File, http://cedar.csc.uvic.ca/kienle/view/IWPC2002/Benchmark, 2002.

[24] Sim, S. E., Holt, R. C., and Easterbrook, S., "On Using a Benchmark to Evaluate C++ Extractors", in Proceedings of 10th International Workshop on Program Comprehension, Paris, France, 2002, pp. 114-123.

[25] van den Brand, M., Sellink, A., and Verhoef, C., "Current Parsing Techniques in Software Renovation Considered Harmful", in Proceedings of 6th International Workshop on Program Comprehension (IWPC'98), Ischia, Italy, June 24-26 1998, pp. 108 - 117.

[26] W3C, "Document Object Model (DOM) Level 1 Specification", W3C, Web page, Date Accessed: 01/20/2002, http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/, 1998.

[27] W3C, "XML Path Language (XPath) Version 1.0 W3C Recommendation", W3C, Web page, Date Accessed: 01/20/2002, http://www.w3.org/TR/1999/REC-xpath-19991116, 1999.

[28] W3C, "XSL Transformations (XSLT) Version 1.0", W3C, Web page, Date Accessed: 01/20/2002, http://www.w3.org/TR/xslt, 1999.

[29] Wong, K., "The Rigi User's Manual - Version 5.4.4." The Rigi Group, Date Accessed: 01/20, http://ftp.rigi.csc.uvic.ca/pub/rigi/doc/rigi-5.4.4-manual.pdf, 1998.