Leveraging XML Technologies in Developing Program Analysis Tools

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

Abstract

XML technologies are quickly becoming ubiquitous within all aspects of computer and information sciences. Both industry and academics have accepted the XML standards and the large number of tools that support manipulation, transformation, querying, and storage of XML objects. Thus, tools and representations based on XML are very attractive with respect to adoption. This paper describes the experiences of the authors in the development and application of srcML, a XML application to support explicit markup of syntactic information within source code. Additionally, XML technologies are leveraged along with srcML to support various program analysis, fact extraction, and reverse engineering tasks. A short description of these tools is given along with the motivation behind using an adoption centric XML approach.

1. Introduction

While conducting research in program understanding, reverse engineering, and software visualization we ran into a common technical problem namely, it is necessary to parse and analyze large amounts of source code. Furthermore, none of the existing program analysis tools worked very well for our particular problems. While many of the existing analysis tools are successful in a number of ways, they are typically difficult to integrate or extend into new research and products. The existing tools are typically: tightly coupled with other tools, language dependent, or embody a methodology orthogonal to the specific problem. Additionally, these tools are given little support by the original developers and/or require specific (older) OS versions, platforms, and libraries. These inherent problems have also been described by others [6, 12]. Oftentimes using and modifying an existing tool is as difficult as building your own from scratch.

Many of our colleagues run into this same problem. A number of these researchers expressed the simple need for an easy to use C++ parser that allows them access to the abstract syntax tree. Of course, one can hack g++ and recent versions allow some access to this information but it is still very difficult to integrate into other tools. Also, the intermediate output from a compiler is only part of the information we (and others) need. A large amount of information is lost in the compilation process and this is often vital to support such things as program understanding tasks.

To address our own problems we felt that building a tool (or set of tools) that is easy to integrate and easy to use was necessary to the long term goals of our research. Also, such a tool should be easily shared with others working on similar problems. Our need to extract syntactic information from C++ came to a head in early 2001. While we previously used quick-and-dirty solutions, these types of methods no longer supported our problem. So, we decided to build our own generic tool(s). At the same time, XML technologies had been around for a good duration and were quite mature. We'd been thinking about using XML to represent source code since the late 90's and some research in this direction had been conducted [1, 5].

These days, any type of new data storage or manipulation is using, is considering, or being compared to XML. This is due to the wide variety and availability of XML technologies including formats, tools, and standards. In the case of access to XML data there are pull and push parsers and XML transformation languages (e.g., XSLT, STX, and TextReader). Choosing an XML representation allows the use of many (if not all) of these powerful tools.

Many developers and researchers are using or are planning to use XML technologies for the applications they are constructing. Although some of these people may not be completely familiar with the intricacies of XML Schema and XQuery they do understand the basics and can quickly develop applications using the easy to use technology.

Our approach is not so much a tool as it is a representation. We designed a XML format, *srcML* [4, 8], to mark up source code (C/C++/Java) with explicit syntactic information. In essence the abstract syntax of the program is directly imbedded into the source code. So there is no need to do parsing after the source has been translated into srcML. Importantly, the representation does not alter the original source code in any inherent manner. It preserves all the formatting, comments, macros, etc. This is vital to support our own research and quite different from most other approaches along with being a stated requirement for many software engineering tasks [13].

With source code marked up in srcML we argued that all of the existing XML tools and technologies can be leveraged to construct simple tools in an opportunistic manner to solve a particular problem. We developed a C++ to srcML translator and demonstrated our approach by addressing the problem of C++ fact extraction in the context of a lightweight XML approach [3]. Using our approach has proven to be quite successful with regards to flexibility, integration with other tools, platform independence, ease of use, and adoption of our tools.

In the remainder of this paper we will briefly describe srcML and how we utilize XML tools to solve various problems. We also discuss the requirement of our approach within the context of adoption as we believe many of our decisions directly support this nonfunctional requirement. How we support srcML currently and plan to in the future is described along with the requirements of using our approach. We conclude with a summary of issues that support the construction of adoption centric software.

2. XML and Source Code

A number of options have been investigated for representing source code information (e.g., AST or ASG) in a XML data format namely, GXL [7], CppML [9], ATerms [14], GCC-XML, and Harmonia [2]. In these formats the abstract syntax tree (AST) (actually a graph) of the source code, as output from a compiler intended for code generation, is stored in a data XML data format. These XML data views of source code, since they are based on the AST, are a "heavyweight" format that requires complete parsing of the original document and generation of the complete AST. The work most closely related to srcML is Badros' work on JavaML [1], which is an XML application that provides an alternative representation of Java source code.

However many of these approaches do not take full advantage of XML technologies. They may provide XML output or input, but not use it internally, take a data view of XML at the expense of a document view, or provide an XML API for accessing the data but not allow for the full range of XML application and they may not be designed for the full variety of XML processing (i.e., pull-parsing, push-parsing etc.). The use and development of XML tools is an active area of research and it is difficult to predict what tools or languages a developer may choose to use with an XML format now or in the future.

2.1. srcML

srcML (SouRce Code Markup Language) [3, 4, 8] is an XML application that supports both document and data views of source code. The format adds structural information to raw source code files. The document view of source code is supported by the preservation of all lexical information including comments, white space, preprocessor directives, etc. from the original source code file. This permits transformation equality between the representation in srcML and the related source code document.

A lightweight data view of source code is supported by the addition of XML elements to represent syntactic structures such as functions, classes, statements, and entire expressions. Other structural information including macros, templates, and compiler directives (e.g., #include), are also represented. The data view stops at the expression level with only function calls and identifier names marked inside of expressions thus allowing reasonable srcML file sizes. The srcML for the simple program below is given in Figure 2.

```
#include <iostream>

// A function
void
f(int x)
{
  std::cout << x + 10;
}
```

In this example we see that all of the original text is present, including the preprocessor directive `include` and all original comments. Some of the original text that are meta-characters in XML, e.g., '<', have been encoded

```
<unit xmlns="http://www.sdml.info/srcML/src" xmlns:cpp="http://www.sdml.info/srcML/cpp">
<cpp:include>#<cpp:directive>include</cpp:directive><cpp:file>&lt;iostream&gt;</cpp:file></cpp:include>

<comment type="line">// A function
</comment><function><type>void</type>
<name>f</name><formal_params>(<param><type>int</type> <name>x</name></param>)</formal_params>
<block>{
  <expr_stmt><expr><name>std::cout</name> &lt;&lt; <name>x</name> + 10</expr>;</expr_stmt>
}</block></function>
</unit>
```

Figure 1. Example of srcML. Notice that preprocessor directives are marked up and not expanded. The format preserves all textual context entered by the developer.

but all of the other text is preserved. The documentary structure of the original text including spacing and lines are also preserved. The inserted XML tags allow for the addressing and location of textual elements according to their location in the XML document.

The data view allows for a search-able and query-able representation. This can be mixed with a document view to permit multiple levels of abstraction (or views), and allows a data view of the document without losing any of the document information. The reverse is also allowed with document information, e.g., white space, comments, etc., used in the data view for searches or queries.

Once the document is in srcML locations in the srcML document (and corresponding locations in the textual source code document) can be referred to using the XML addressing language XPath. For example, to refer to the second if statement inside of a function named foo we can use the following XPath: //function[name="foo"]/block/if[2]. Using XPath we can address the document using a variety of paths to refer to locations. XPath can also be used to represent groups of elements, such as all functions.

The capability to use XPath addresses is built into most XML tools and is used extensively in XML transformation languages such as XSLT. The XPath standard forms the base of the XML query language XQuery.

Unlike the physical representation of an address that a line number references, XPath addresses describe one of many possible paths to a location. The XML elements serve as reference points along the path. This makes XPath addresses much more resilient to changes in other parts of the document, unless they change the nested XML elements.

2.2. Adoption of srcML

In the design and construction of srcML there are a number of factors that have an effect on the adoption of this type of approach. In short, selecting a light weight philosophy is behind many of these issues. We discuss each issue as a postmortem in the context of adoption.

Choice of document view over data view. XML applications (e.g., XML formats) and processing falls into the categories of a document view (e.g., DocType, etc.) or of a data view (e.g., SOAP). These two views influence the layout, semantics, and tools for the format (e.g., from a document view white space is typically of importance, from a data view white space is of no importance). Source code, especially in an unprocessed state, is closer to a document than to data. That is, we program by writing documents, not by specifying parse trees. When required to make a decision between the two views, the document view is the clearest choice in this case. This does not lead to any loss since a data view can simultaneously be supported with careful handling during processing and analysis.

Preservation of the original data (i.e., source code). Allowing all of the original data to remain in any kind of transformation process is an accepted data manipulation design principle. This is true even if the data is not needed. In srcML the preservation of all original text allows for the restoration of the complete original document. The format does not try to reorganize or change the text. It only augments the text with markup that extends its capabilities while still allowing the added information to be easily extracted.

Markup only what is of interest. In srcML the markup stops at the expression level (i.e., expressions and the identifiers contained in the expressions are given markup). The full AST of the expression is not given markup because this does not meet the requirements of the uses that we saw. While marking full expressions down to the operator and parentheses level may be useful for expression rewriting and other compiler-oriented tasks, it is not useful for the identified tasks.

Tag Names based on programmers view. Programmers know the basic syntax of the language that they are programming. They also know the syntactic name of many of the program structure, i.e., block, function, type, etc. They typically do not know the exact distinction between the finer points of syntax naming. For example, in the function declaration: const int foo(); most programmers most likely will identify the return type of the function as const int. In srcML the type const int is marked using the tag <type>. Technically however, the type is int and const is a type specifier. Most programmers only care about this distinction when reading syntax diagrams.

Minimization of meta-data. There are very few attributes used in srcML. The markup in the source code is to provide navigation and access to the content. Information that may be derived from other parts of the program, e.g., the type of a variable used in an expression, are derived versus stored as attributes.

Easy conversion to and from the original format. Source code text is extracted from directly from srcML by removing markup and some output un-escaping. This is done by a variety of XML tools or even by simple Perl or Python programs.

Lightweight schema. There is a tendency to produce a source code format that is very stringent, i.e., any documents produced in the representation can only represent compilable C/C++ programs. Our philosophy is that this is unnecessary since there are compilers to test if a program can compile. We see this as a hindrance since the source code may be in a state that can not be compiled (e.g., it is under maintenance or during refactoring). This implies that we can do analysis on incomplete programs, programs with missing libraries, and source code that is under construction.

Format efficiency. One of the criticisms of XML formats is their size. Data represented in an XML format can balloon up as levels of markup and attributes are used. This may cause an increase in size of hundreds of times over the original document and is especially problematic to DOM approaches (including XSLT) which store the entire tree in memory before processing is started. In practice, a srcML document is on average 5 times larger than the original source code document. Full AST markup in XML can result in hundreds of times increase of file size [11].

3. srcML Translation

Translating source code into srcML allows the source code to be integrated into an XML infrastructure. The translation process directly influences how completely the XML infrastructure can be utilized and what applications can take advantage of the format. We now describe the C/C++ to srcML translator and the influences with regards to adoption.

3.1. srcML Translator

The srcML translator takes as input C/C++ source code text and inserts XML around the syntactic structures to produce srcML. The translator has additional requirements over that of a traditional parser (e.g., in a compiler) in preservation of source code text and ability to work with code fragments and incomplete code. Design decisions were made that meets these requirements to support all of the features of the srcML representation. In this section the important design considerations are briefly discussed.

Existing compiler-centric parsers have difficulties in the preservation of the original source code text especially with regard to white space, comments, and preprocessor directives. In addition they are not designed to handle incomplete code. Regular expression pattern matching approaches do not have these problems, but have difficulties in the context of what they are matching.

Typical parsers take a LR(k) or bottom-up approach (e.g., parsers generated by *yacc*). They start with parsing lower level components and use multiple production rules for reduction. Lower level syntactic elements are parsed and identified before higher level structural elements, e.g., contents of a block are parsed before the block structure itself.

A top-down parser generated by the compiler generator ANTLR was used to identify and markup the elements in the source code as per the natural structural order of the components. Minimal parsing was used to identify top level structural elements before constituent elements (e.g., identify that a function definition had started without having to parse all of its contents).

The translator employs a selective parsing approach based on the concept of island grammars [10]. The parsing of a language construct such as function definition occurs in multiple passes with the identification and markup of the start of the function definition done in the first pass.

The C and C++ languages have a non-CFG (Context Free Grammar). Traditional compilers construct a symbol table which can be used to resolve non-CFG ambiguities. However a complete symbol table is not possible with code fragments and incomplete code. The translator handles this issue by considering a CFG view of the non-CFG C/C++ grammar.

Most parsing and markup methods follow a batch sequential architecture: parse the source code, generate the entire AST, insert tags at appropriate nodes, and output the tree with the markup. The top-down parsing of ANTLR was extended to stream parsing where XML tokens were inserted into the stream of text tokens as soon as a markup element is identified. XML tags were only wrapped around syntactic elements of (high) interest in a language construct (e.g., in case of an expression statement identifier names are marked up while arithmetic operators are left unmarked).

The translator is the first stage from C++ source code to srcML representation. The srcML output is designed to be refined with external parsing stages based on processing of associated source code files, user defined heuristics, and user knowledge.

3.2. Adoption of the Translator

In addition to the common requirements of an application such as accuracy, reliability, and speed, the srcML translator had the additional requirement that it be able to be fully integrated with XML technologies.

Responsiveness. The decision to use event-parsing allows for output as soon as a statement is detected. Since output is immediately available the translator can be integrated into a stream XML processing (i.e., SAX, STX, TextReader) allowing for the efficiencies of memory that these approaches allow. Although not as important for tree XML processing (i.e., DOM, XSLT) it does allow for simultaneous translation and tree building.

Flexibility. It is important to support all possible forms of usage of any tool. Programmers still often use CLI tools (e.g., grep) because they are fast, portable, and easy to use once learned. They also are easily scriptable leading to low-level productivity tools. CLI tools can also be used in a GUI.

Scalability. The event-driven translation approach allows for its use on large source code files and large collections of source code files.

Extensibility. The srcML translator's CFG view of the source code and output in XML provides a base for further source code processing. This processing may more accurately markup the source code based on external information (i.e., a symbol table), or transform it for another usage entirely.

Portability. The srcML translator is a CLI program that can be used on both MS Windows and Linux.

Robustness. The translator must generate a well formed srcML document no matter what the state of the input source code. This is related to the lightweight view of schema conformance.

Once the translator met this list of requirements it was able to be used with the complete range of XML technologies. In the next section we will describe some of the uses of XML with srcML for program analysis tasks.

4. Applications using srcML

As with the many applications where XML is utilized we see a wide range of applications for srcML within the context of development, analysis and transformation of source code. The following sections cover the current applications that leverage the XML infrastructure technology with our srcML representation and translator. The first section discusses the direct use of the srcML representation and the next section covers applications that extend the srcML representation.

4.1. Directly Leveraging srcML

XML technologies have been combined directly with the srcML representation to provide applications ranging from viewing, searching, and editing source code to source code transformation. XML began as a document format with support for style-sheets formats such as CSS and XSL-FO, therefore source code viewers are a natural application using srcML. CSS style-sheets have been used to pretty print source code, hide or emphasize particular program elements, and perform simple abstract visualizations. Web browsers that support XML (e.g., Internet Explorer, Mozilla/Netscape) can be directly used as source code viewers.

Moving beyond viewing is of course editing using srcML. The user edits the source code normally, i.e., by editing text and in the background the srcML translator generates the corresponding srcML. The srcML can be used to control the view of the source code using style-sheets. Editing has additional requirements over viewing. In editing we cannot assume that the source code is in a compilable, complete state and so srcML must be able to represent source code that is not in a compilable, complete state. The translator must also accept this realistic view of source code, while at the same time providing the responsiveness that we expect in an interactive application. An editor that utilizes these features of srcML is currently in development.

The srcML format is being used to extend the capabilities of source code search tools. Current search tools are based on regular expressions in which it is difficult to define the proper context of the matching in terms of the syntactic structure, e.g., an identifier name in a comment.

The srcML tags provide XML reference points for addressing locations in source code documents. Searches are expressed using the XML addressing language XPath. Many XML tools evaluate XPath expression, e.g., the command line utility *xpath*. Full queries can also be performed on source code by the application of XPath. This has already been used for our C++ Fact Extractor that combined the srcML translator, a command line XPath tool (*xpath*), and XPath expressions. If a full query language is needed XQuery (XPath is a subset of XQuery) can be used, or any other XML query language that is developed.

The preservation of the original source code text in the srcML allows source code transformations to be performed using XML tools. Source code is converted to srcML, transformed using XML transformation languages and API's, and converted back to source code. An identity XML transformation of srcML is an identity transformation of the original source code text preserving the programmer's view of the source code.

Source code transformations with srcML can use any XML transformation language or tool including the XML transformation language XSLT, or the use of an API in a more traditional programming language, e.g., Python using a DOM API. Because of the responsiveness of the translator memory efficient pull-parsing XML API's such as SAX and languages such as STX can be used.

4.2. Extensions of the srcML Representation

A specific, and exciting, transformation application is to support aspect weaving. Aspect Oriented Programming (AOP) is a programming paradigm that addresses the problem of advanced separation of concerns. Aspects, written separately from the base code, are woven into the base code based at specific locations, e.g., before a function return. The XWeaver Project at ETH (http://control.ee.ethz.ch/~ceg/XWeaver) uses srcML to define aspects in the format AspectX. srcML is used to represent the code to insert (aspect) and the location in the code were the aspect is to be inserted. The aspects are woven into the source code using a XML transformation for non-intrusive source code transformations.

The srcML format is being used for differencing applications (results of this work are submitted to

ICSM'04). Current differencing tools and approaches are either efficient with a low-level of information (e.g., the utility *diff*), or inefficient with a higher-level of information (e.g., semantic differencing). The srcML format has been extended to support Meta-Differencing, i.e., the extraction of higher level information from the differences in source code documents. This information is stored in srcDiff, a multi-version extension of the srcML format.

The srcDiff format extends the querying capabilities of srcML for queries involving both the contents of the source code and versioning information. The srcDiff document is constructed by weaving the two versions of the srcML documents together with the utility *diff* indicating where differences occur.

The srcML format can also be used for representing higher-level structures in the source code as with source models. Source models provide an abstraction of the source code that focuses on the concept of interest hiding unnecessary details. They are often constructed after further analysis of the source code and may include information that is not contiguous or singular in the source code.

5. Conclusions

Using XML in the context of adoption presents a number of advantages including support for multiple query languages, support of complex transformations, support for a document format as well as a data format, broad based usage and acceptance of standards, and a large interest from the open source community. The general academic and industry communities have embraced XML and its associated tools.

In our experience spending additional effort to construct an underlying XML format that takes advantage of all the tools is critical to a successful application. Additionally, we focused not on solving a particular task but instead we built an infrastructure to address a general set of problems. This allows the user to opportunistically use XML tools to address their particular problem. We feel adoptable software should address general problems and allow for specialization to particular tasks.

6. Acknowledgements

This work was supported in part by grants from the National Science Foundation CCR-02-04175.

7. References

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

[2] 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, November 23-25 2000, pp. 287-289.

[3] Collard, M. L., Kagdi, H. H., and Maletic, J. I., "An XML-Based Lightweight C++ Fact Extractor", in Proceedings of 11th IEEE International Workshop on Program Comprehension (IWPC'03), Portland, OR, May 10-11 2003, pp. 134-143.

[4] 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.

[5] Cox, A., Clarke, C., and Sim, S., "A Model Independent Source Code Repository", in Proceedings of Proceedings of CASCON'99, November 8-11 1999, pp. 381-390.

[6] Favre, J.-M., Estublier, J., and Sanlaville, A., "Tool Adoption Issues in a Very Large Software Company", in Proceedings of 3rd International Workshop on Adoption-Centric Software Engineering (ACSE'03), Portland, Oregon, USA, May 9, 2003 2003, pp. 81-89.

[7] 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, Australia, November, 23 - 25 2000, pp. 162-171.

[8] 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.

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

[10] 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.

[11] Power, J. F. and Malloy, B. A., "Program Annotation in XML: a Parse-tree Based Approach", in Proceedings of 9th Working Conference on Reverse Engineering, Richmond, Virginia, October 2002 2002, pp. 190-198.

[12] Tilley, S. R., Huang, S., and Payne, T., "On the Challenges of Adopting ROTS Software", in Proceedings of 3rd International Workshop on Adoption-Centric Software Engineering (ACSE'03), Portland, Oregon, USA, May 9, 2003 2003, pp. 3-6.

[13] Van De Vanter, M. L., "The Documentary Structure of Source Code", *Information and Software Technology*, vol. 44, no. 13, October 1 2002, pp. 767-782.

[14] 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.