srcML: An Infrastructure for the Exploration, Analysis, and Manipulation of Source Code
A Tool Demonstration

Michael L. Collard[1], Michael John Decker[2], Jonathan I. Maletic[2]

[1]Department of Computer Science
The University of Akron
Akron, Ohio
collard@uakron.edu

[2]Department of Computer Science
Kent State University
Kent, Ohio 44242
mdecker6@kent.edu, jmaletic@kent.edu

***Abstract*—srcML is an XML representation for C/C++/Java source code that forms a platform for the efficient exploration, analysis, and manipulation of large software projects. The lightweight format allows for round-trip transformation from source to srcML and back to source with no loss of information or formatting. The srcML toolkit consists of the src2srcml tool for robust translation to the srcML format and the srcml2src tool for querying via XPath, and transformation via XSLT. In this demonstration a guide of these features is provided along with the use of XPath for constructing source-code queries and XSLT for conducting simple transformations.**

***Keywords*—srcML; static code analysis; source transformation**

I. Introduction

The research and practice of software maintenance and evolution almost always, in some manner, requires the exploration, analysis, or manipulation of source code. Here we demonstrate the features of a powerful infrastructure, namely srcML, which supports these tasks via an underlying format, parser, and tool set.

While srcML [3, 4] has been publicly available to the research community since the mid 2000's it has recently gained more traction in both industry and the research community. Much of this is due to a number of new usability features recently added, expanded language support, scalability, and robustness of the platform.

In short, srcML is an XML format for source code. Specifically, the parsing technology supports C/C++ and Java. However, we will soon be releasing support for C#. The XML markup identifies elements of the abstract syntax for the language. This allows us to leverage XML tools to support various tasks of exploration, analysis, and manipulation.

A number of underlying features make srcML particularly useful for evolution and maintenance. The main philosophy is to take a programmer-centric view of the code rather than a compiler-centric one. First, the conversion from source code to srcML is lossless. That is, no formatting, comments, or actual code is lost. There is a round-trip equivalency from source code to srcML and back to the original source code. Additionally, macros, templates, and preprocessor statements are marked up. That is, the preprocessor is not run (or need not be run) prior to conversion to srcML. This also implies that code with missing includes, libraries, or code fragments can be converted to well-formed srcML. Lastly, the conversion to srcML is extremely efficient, running faster than a compiler (over 25KLOC/sec).

srcML has been used for a variety of maintenance problems. This includes, but not limited to, such things as the analysis of large systems to automatically reverse engineer class and method stereotypes [9], supporting syntactic differencing [12], and applying transformations to support API and compiler migration [5].

In the demonstration we will provide an introduction of how to use srcML. It is a command-line tool with a number of options for converting one file, multiple files, or complete source archives to srcML. Then, a number of basic features to explore and analyze source code using the infrastructure and XPath are presented. Means to develop specialized programs using such things as XML utility libraries and Python programs are also demonstrated. Lastly, simple ways to manipulate the source code to produce complex transformations are given.

The srcML infrastructure, including the format, parsing technology, and a select set of tools is currently open source and licensed under GPL. It is available for download at:

http://srcml.org

II. The srcML Platform

The srcML platform is based on the srcML format, which is an XML format where the syntactic aspects of the source code are marked with srcML elements. An example of srcML is given in Figure 1. The main element is *unit* which contains some optional attributes about the source code, including filename, directory, and version. The required attribute *language* includes the programming language, which must be specified when the srcML is created. The *unit* element contains the srcML form of the source code. This includes all of the original source code for that translation unit, and the srcML markup elements. srcML includes elements for all syntactic statements, e.g., *if*, *while*, *for*, *expr_stmt, decl_stmt*, and program structural elements, e.g., *function*, *class*, *namespace*, all in the namespace *http://www.sdml.info/srcML/src*. C-preprocessor statements are also marked, e.g., *cpp:include*, in their own namespace *http://www.sdml.info/srcML/cpp*. Special elements include escape for invalid XML characters, e.g., formfeeds. Optional markup is provided for literals, operators, and type modifiers. The complete list of srcML elements are on the website.

```
<unit xmlns="http://www.sdml.info/srcML/src" xmlns:cpp="http://www.sdml.info/srcML/cpp" language="C++"
filename="ex.cpp">
<comment type="line">// copy the input to the output</comment>
<while>while <condition>(<expr><name><name>std</name>::<name>cin</name></name> &gt;&gt;
<name>n</name></expr>)</condition>
  <expr_stmt><expr><name><name>std</name>::<name>cout</name></name> &lt;&lt; <name>n</name> &lt;&lt;
'\n'</expr>;</expr_stmt></while>
</unit>
```

Figure 1. A code fragment in the srcML format. Note that all original text is preserved, including white space and comments. The XML markup is placed to indicate syntactic context. The srcML format can also represent complete source code files and even complete projects in a srcML archive.

The main architectural elements of the srcML infrastructure are presented in Figure 2. Underlying everything is the srcML format. Directly on top of that is the parsing technology. Then XML technologies can be used to build various tools and applications. On the left are the abstract models we use as software engineers to conduct maintenance and evolution. These models cross the various layers. On the right we have the srcML community and support.

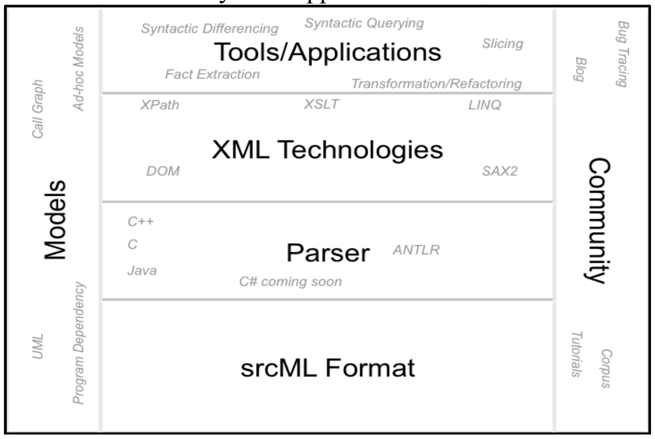

Figure 2. Architecture of the srcML infrastructure.

A. *srcML Archive*

For simple code fragments and individual files, the basic srcML format works well. But, as is done for source-code archives such as the tar format for systems with large numbers of files, it is much more convenient to combine these separate files into one large srcML file, which is a srcML archive. It basically consists of separate *unit* elements for each source code file, wrapped in an outer *unit* element. For multiple file input, the srcML archive is the default. The srcML toolkit automatically adjusts to basic and archive srcML files.

B. *Implementation*

The srcML parser takes full advantage of the srcML format. The parser is custom built using a modified version of the LL(k) ANTLR 2.7.7 parsing toolkit. A separate parsing stream is used for white space and comments, allowing them to be completely preserved. Another separate parsing stream is used for preprocessor statements. Both of these separate streams allow for the primary parser to work with the syntax of the language. Before output, the tokens from the separate parsing streams are merged back into the primary parser stream. Preprocessor statements can affect the primary parsing, so information from the preprocessor stream is used with a set of heuristics to deal with these cases.

In addition to ANTLR, many of the features are provided by the libraries *libxml2* and *libarchive*. The primary use of *libxml2* is for generating XML when creating srcML, and for conversion of srcML back to source code. *Libxml2* is also used for accepting http URLs, executing XPath queries, and in encoding issues (using the *libiconv* library). The *libxml2* associated libraries *libxslt* and *libexslt* are used for performing XSLT transformations.

The other library used extensively is *libarchive*, which supports encoded and archive file types, e.g., gz, tar, cpio, etc. *libarchive* is combined with *libxml2* to provide for remote access of these file types, e.g., an http tar.gz file.

III. EXPLORATION & ANALYSIS

The first step in using the srcML toolkit is to translate the original source code into the srcML format. This can be the entire system, a few select files, or a code fragment. The system may be a single language or multi-language. The following command can be used to convert all the source code in *KOffice* into srcML directly from the url of the *KOffice* archive:

```
src2srcml --register-ext h=C++
http://download.kde.org/stable/koffice-2.3.3/koffice-
2.3.3.tar.bz2 -o koffice.xml
```

In this example, files with the extension '.h' are treated as C++. At this point a srcML version of the entire *KOffice* source code project is generated and put into the file *koffice.xml*. The remote access and conversion of the source code archive (tar.bz2) is handled automatically. The resulting srcML file is about 164 MB, while the original source code (as text) takes about 48 MB, producing a reasonable 3.42 times increase in size.

In addition to an input source from a URL, *src2srcml* can be given an individual file, a list of files, or a directory. In either case, the language of an input source is determined automatically from the filename's extension. *src2srcml* also has multiple options for specifying input source-code encoding (e.g., Latin1) and optional markup for literals, type modifiers, and operators.

Once in the srcML format, the code can be explored using XPath. To find the number of a particular syntactic item, we can directly count them. For example, to find the total number of *for-statements* in *KOffice*:

```
srcml2src --xpath "count(//src:for)" koffice.xml
```

In this case, the result is 6,813. Elements in the XPath are by default prefixed with *src* for the syntactic elements and *cpp* for c-preproccesor elements. To find a list of particular items, we can also use XPath. The result is a srcML archive with each query result in a separate unit element. The following will find these names, and then count the number of them using the --longinfo option:

srcml2src --xpath "//src:class/src:name" koffice.xml | srcml2src --longinfo

This query took about 6 seconds on a MacBook Pro 2.66 GHz Intel Core i7 with 8GB RAM.

```
 1 reader =
libxml2.newTextReaderFilename(sys.argv[1])
 2 while reader.Read():
 3   if reader.NodeType() == 1 and
          reader.Name() == 'function':
 4      # expand the subtree
 5      node = reader.Expand()
 6      # setup for XPath evaluation
 7      ctxt = node.doc.xpathNewContext()
 8      ctxt.setContextNode(node)
 9      ctxt.xpathRegisterNs("src",

"http://www.sdml.info/srcML/src")
10     ctxt.xpathRegisterNs("cpp",

"http://www.sdml.info/srcML/cpp")
11     # output the function name
12     print
ctxt.xpathEval("src:name")[0].getContent() + ',',
13     # extract the call names
14     calls =
ctxt.xpathEval("src:block//src:call/src:name")
15     calllist = [call.getContent() for call in
calls]
16     # output the list of calls
17     print ','.join(set(calllist))
18     # finish up
19     ctxt.xpathFreeContext()
20     # delete this subtree
21     reader.Next()
```

Figure 3. Main loop from a Python program using the libxml2 xmlTextReader API to generate data for a call graph. The program reads nodes (Line 2) until a function element is found (Line 3). The tree for that function is expanded (Line 5), and XPath extracts the name of the function (Line 12) and the list of calls (Line 14). This function subtree is then deleted (Line 21).

When it is necessary to further analyze the system, XPath querying can still be used, either entirely alone, or as a first step of further analysis. This further analysis can be performed by any XML tool or API. As an example, the Python program in Figure 3 takes each function in a system and generates call information. The output is each function name followed by a list of calls made by that function.

The Python program (Figure 3) uses the *libxml2 TextReader* interface. The main loop (Line 1) reads each node in the srcML file. When a function element is detected (Line 2), the entire XML subtree for that function is created by the call to the method *Expand()* (Line 5). Since we now have the complete tree for this function, we use XPath to extract the necessary information. First, the XPath evaluation is setup (Lines 7 – 10). Then the function name is extracted using the XPath *src:name* (Line 12). Note that the XPath expressions are written from the context of this *src:function* element. Lastly for this function, all of the calls are extracted using the XPath *src:block//src:call/src:name* (Line 14). It is relatively straightforward to extend this example to the extraction of other parts of methods/functions. Additional cases for other elements are easy to add.

This approach is highly scalable, as it takes about a minute to run this on a srcML archive of the entire linux kernel. General evaluation of XPath requires a complete XML tree in memory (similar to DOM). However, in this case, we only create a tree for the current function, and then remove it when finished. This general approach can be used with other programming languages and other XML APIs, especially with pull parsers. For instance, a similar approach is built into the srcML toolkit for XPath and XSLT evaluation.

IV. Manipulation

We now show a simple XSLT transformation that uses srcML to instrument source code to gather and report basic profiling information. Figure 4 shows parts of the XSLT program. The full program is available at the demo website. The source code is transformed in three ways: 1) access to a global profiling object is added to each source file, 2) functions are instrumented to update the global profile object, and 3) the main function is modified to report the profiling information. A sample program is provided on the accompanying website to illustrate the program. The instrumentation of the sample program can be performed using the following command:

src2srcml sort.cpp sort_lib.cpp | srcml2src –xslt profile.xsl | srcml2src –to-dir profiled

First, the source files are converted to the srcML format. Then the XSLT program *profile.xsl* is applied to all of the files in the srcML archive. After the profile code is inserted, the transformed source code files are extracted.

The *profile.xsl* transformation program consists of separate templates to insert the required code. Note that any inserted source can be put as plain text, i.e., it is not necessary to insert srcML markup. It also contains an identity copy XSLT template that assures that any unmodified code is copied over.

For efficiency, the XSLT program is applied to the srcML of each individual source code file one at a time. The result of applying the XSLT to each individual unit is then merged into the output srcML archive. This provides scalability of the transformation to very large systems.

Although our example was an XSLT program (using the builtin *srcml2src* feature), transformations can be written using any XML tool or API, e.g., LINQ, SAX2, DOM, etc. For example, the approach used in the Python program with *TextReader* (as shown previously) can be extended into a transformation. A potential complexity is an identity copy of all of the unchanged parts of the source code being transformed. In XSLT, the identity template handles this.

V. Related Work

It has been observed that automated source code transformations intended to be handed back to a developer must preserve the programmer's view of the document, i.e., preserve white space, comments, and the expressions of literals, and failure to do so may mean the rejection of the result [6, 14] and tool. It has been observed that these compiler-centric approaches are often not a good match to the problems that they are trying to solve [10, 14]. There are exceptions to this problem with compiler-centric approaches, with one example being the DMS system by Baxter [2].

```
<!-- Insert call to global profile object to record call of this function -->
<xsl:template match="src:function/src:block">
<xsl:copy-of select="node()[1]"/>
<xsl:text>functions.count(__LINE__, "</xsl:text><xsl:value-of select="../src:name"/><xsl:text>");
</xsl:text><xsl:apply-templates select="node()[position()!=1]"/></xsl:template>

<!-- identity copy -->
<xsl:template match="@*|node()"><xsl:copy><xsl:apply-templates
select="@*|node()"/></xsl:copy></xsl:template>
```

Figure 4. Templates from an XSLT program to insert profiling information into source code. Other templates insert the proper includes, and output at the return of the main() function. The identity copy template makes sure that all other source code is preserved.

Baxter has gone to great lengths to address this specific issue by storing important textual items within the underlying abstract-syntax graph.

One approach is to move down to the level of lexical analysis and provide for the transformation at that level, as in [8]. This allows for the preservation of all of the text, but at a cost of complex regular expressions. Another approach that preserves the programmer's view is to move the transformation to the level of the grammar as in TXL [7]. Using this approach, the transformations are written as part of the grammar for parsing the language. The approach shares many of the advantages of our approach: preservation of programmer's view, scalability, robustness, etc. The difference is in the format of the transformation. Instead of grammar rules, our approach treats the text of the source code as data in XML, and the transformations are XML transformations. ASF+SDF and Rascal [11] use a similar approach. Stratego [15] also support various means to apply transformation rules.

The Proteus system [16] addresses similar problems of performing transformations on large C++ systems while preserving the layout and handling code before preprocessing. Other approaches include JavaML [1] and others using an intermediate language to describe the source, as in the case of the C Intermediate Language (CIL) [13].

VI. Conclusions and Future Work

The srcML platform has shown itself to be highly useful for the exploration, analysis, and transformation of source code. The principle of the format and tools to preserve all elements of the original source code text allows for any information present in the code to be extracted, whether lexical, documentary, or syntactic. Although all of this information may not be needed for a particular task, using XPath, the srcML toolkit, or a particular XML tool, unneeded information is easy to avoid. The reasonable increase in file sizes of srcML over that of the original source code lessens the cost of this potentially unneeded information.

For the future, plans are to increase language support for Java and the new features of C++11. At the request, primarily from industry, the plan is to include language support for C#. As the number of supported languages increases, the use with mixed-language systems becomes more important, and the plan is to include full support for this. The current internal architecture of the system makes it difficult for anybody (except the coauthors) to fix bugs or support new languages and language features. The plan is to change to a plug-in parser architecture. We envision that srcML parsers for new languages would be specified using a ANTLR-type declaration. Our goal is to first implement this for DSLs, and then eventually move currently supported programming languages (e.g., C++) to this format. This would greatly increase the ability of other developers to support new language features and versions.

One of the main roadblocks to usage of any system is the support in the form of examples, tutorials, documentation, etc. The plan is to increase and organize these materials at our new website (srcml.org). We especially want to support more integration into other APIs and tools. Finally, for srcML to grow in both capability and usage, a broader community needs to be formed.

References

[1] Badros, G. J., "JavaML: a markup language for Java source code", Computer Networks, vol. 33, no. 1–6, 2000, pp. 159-177.

[2] Baxter, I. D., Pidgeon, C., and Mehlich, M., "DMS: Program Transformations for Practical Scalable Software Evolution", in ICSE, May 23 -28 2004, pp. 625-634.

[3] Collard, M. L., Decker, M., and Maletic, J. I., "Lightweight Transformation and Fact Extraction with the srcML Toolkit", in SCAM, Sept 25-26 2011, pp. 10 pages.

[4] Collard, M. L., Kagdi, H. H., and Maletic, J. I., "An XML-Based Lightweight C++ Fact Extractor", in IWPC, May 10-11 2003, pp. 134-143.

[5] Collard, M. L., Maletic, J. I., and Robinson, B. P., "A Lightweight Transformational Approach to Support Large Scale Adaptive Changes", in ICSM, Sept 12-18 2010, pp. 10 pages.

[6] Cordy, J. R., "Comprehending Reality - Practical Barriers to Industrial Adoption of Software Maintenance Automation", in IWPC, May 10-11 2003, pp. 196-206.

[7] Cordy, J. R., Dean, T. R., Malton, A. J., and Schneider, K. A., "Source transformation in software engineering using the TXL transformation system", Info and Software Technology, vol. 44, no. 13, 2002, pp. 827-837.

[8] Cox, A. and Clarke, C., "Relocating XML Elements from Preprocessed to Unprocessed Code", in IWPC , June 2002, pp. 229-238.

[9] Dragan, N., Collard, M. L., and Maletic, J. I., "Automatic Identification of Class Stereotypes", in ICSM, 2010, pp. 10 pages.

[10] Klint, P., "How Understanding and Restructuring Differ from Compiling - A Rewriting Perspective", in IWPC, May 10-11 2003, pp. 2-12.

[11] Klint, P., Storm, T. v. d., and Vinju, J., "RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation", in SCAM, 2009, pp. 168-177.

[12] Maletic, J. I. and Collard, M. L., "Supporting Source Code Difference Analysis", in ICSM, September 11-17 2004, pp. 210-219.

[13] Necula, G. C., McPeak, S., Rahul, S. P., and Weimer, W., "CIL: Intermediate language and tools for analysis and transformation of C programs", Lecture Notes in Computer Science 2002, pp. 213-228.

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

[15] Visser, E., "Stratego: A Language for Program Transformation Based on Rewriting Strategies", in Proceedings of the 12th International Conference on Rewriting Techniques and Applications, 2001, pp. 357-362.

[16] Waddington, D. and Yao, B., "High-fidelity C/C++ code transformation", Science of Computer Programming, vol. 68, no. 2, 2007, pp. 64-78.