Lightweight Transformation and Fact Extraction with the srcML Toolkit

Michael L. Collard, Michael J. Decker
Department of Computer Science
The University of Akron
Akron, Ohio 44325
collard@uakron.edu, *mjd52@zips.uakron.edu*

Jonathan I. Maletic
Department of Computer Science
Kent State University
Kent, Ohio 44242
jmaletic@kent.edu

***Abstract*—The srcML toolkit for lightweight transformation and fact-extraction of source code is described. srcML is an XML format for C/C++/Java source code. The open source toolkit that includes the source-to-srcML and srcML-to-source translators for round-trip reverse engineering is freely available. The direct use of XPath and XSLT is supported, an archive format for large projects is included, and a rich set of input and output formats through a command-line interface is available. Applying transformations and formulating queries using srcML is very convenient. Application use-cases of transformations and fact-extraction are shown and demonstrated to be practical and scalable.**

Keywords-component; Source Transformation, Fact Extraction, srcML

I. INTRODUCTION

srcML[1] [1-3] is an XML format for the representation of C/C++/Java source code. The representation wraps source code (text) with information from the AST (tags) into a single XML document, as shown in the example in Figure 1. All original text is preserved so that the original source code document (the programmer typed in) can be recreated. This provides full access to the source code at the lexical, documentary (e.g., comments, white space), structural (e.g., classes, functions), and syntactic (e.g., statement) levels. The format also provides easy support for fact-extraction and transformation. The srcML format is supported by the srcML toolkit and has been shown to perform scalable lightweight fact-extraction and transformation.

The first release of the tool (to other researchers) was in 2003 and the format and associated toolkit has matured steadily since that time. The toolkit is freely available under GPL and has also been licensed commercially for use by industry. Recently, we have added functionality to the toolkit, making it easier to do simple queries and transformations. This paper focuses on how to use the toolkit and the command line interface along with providing examples using srcML for practical problems.

The original intent of srcML was to support research on fact extraction and transformation in the context of very large code bases. We also found it to be very practical for developers. Because of the need for flexible usage and integration into existing tools, the srcML toolkit supports conversion to/from the srcML format through a set of two command-line tools. Users can then apply their tools to this format. However, feedback from users prompted us to include direct support for fact-extraction and transformation into the toolset via the integration of some standard XML tools. This led to a series of new features to support a rich set of input/output formats and to make the toolkit more self contained.

Transformation and fact extraction of source code begins with the conversion of source code to the srcML format. The tool src2srcml is used to convert all of the original source code files into the srcML format. This tool is robust in that it handles unprocessed and incomplete code. The tool is very efficient with a translation speed of 25 KLOCS/sec and can handle approximately 3,000 files per minute. For example, the entire Linux kernel can be converted into the srcML format in less than seven minutes. Once in srcML, XML tools and technologies can be used for fact extraction and transformation. This includes the use of XPath and XQuery for fact extraction, RelaxNG and XSchema for validation, and XSLT, DOM, SAX for transformation. Going from srcML back to source code is handled by the tool srcml2src, which is very fast with speeds over 250 KLOCS/sec. As an example of the scalability and performance of the srcML toolkit, throughout the paper we applied srcML with a fairly standard laptop (Macbook Pro 2.66 GHz i7 processor wth 4 GB memory) to version 2.6.38.3 of the Linux kernel. This version consists of approximately 13 MLOC in 29,361 source-code files, which is 353 MB of text.

The srcML project is hosted at: http://www.sdml.info/projects/srcml. The particular version used in this paper is under current development and whose release is being finalized. This can be found at http://www.sdml.info/projects/srcml/trunk.

The paper is organized as follows. The next section provides an overview of the srcML format and the associated toolkit. Section III provides details and examples of using the toolkit for conversion to/from srcML, for individual files and complete projects. Section IV discusses how srcML can be applied to transformation, followed by section V which covers applications for fact extraction. Section VI discusses how the srcML markup can be extended. This is followed by related approaches and tools, future directions for srcML and the toolkit, and finishes with our conclusions.

[1] Pronounced source M L

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<unit xmlns="http://www.sdml.info/srcML/src" xmlns:cpp="http://www.sdml.info/srcML/cpp" language="C++"
filename="rotate.cpp">
<cpp:include>#<cpp:directive>include</cpp:directive> <cpp:file>"rotate.hpp"</cpp:file>
</cpp:include>

<comment type="line">// rotate three values</comment>
<function><type>void</type> <name>rotate</name>
<formal-params>(<param><type>int&amp;amp;</type> <name>n1</name></param>,
<param><type>int&amp;amp;</type> <name>n2</name></param>,
<param><type>int&amp;amp;</type> <name>n3</name></param>)</formal-params>
<block>{
    <comment type="line">// copy original values</comment>
    <decl-stmt><decl><type>int</type> <name>tn1</name> = <name>n1</name>,
<name>tn2</name> = <name>n2</name>, <name>tn3</name> = <name>n3</name></decl>;
</decl-stmt>
    <comment type="line">// move</comment>
    <expr-stmt><expr><name>n1</name> = <name>tn3</name></expr>;</expr-stmt>
    <expr-stmt><expr><name>n2</name> = <name>tn1</name></expr>;</expr-stmt>
    <expr-stmt><expr><name>n3</name> = <name>tn2</name></expr>;</expr-stmt>
}</block></function>
</unit>
```

Figure 1. Example of the source code file rotate.cpp 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 example is for C++, however C and Java are also supported.

II. SRCML FORMAT AND TOOLKIT

The srcML format and toolkit is now described in more detail. We begin with an example and general description of the elements used in srcML, followed with details on how the source code text is preserved to allow for fact extraction of any part of source code. Finally, a brief description of the implementation of the srcML toolkit is given.

A. srcML Format

The srcML format consists of all text from the original source code file plus XML tags. Specifically, the text is wrapped with srcML elements that indicate the syntactic structure of the code. In short, this explicitly identifies all syntactic structures in the code (e.g., classes, functions, methods, if-statements, etc.).

An example of the srcML representation can be found in Figure 1. The srcML format starts with a root element *unit*. This element contains the srcML of the entire source-code file. It contains the required attribute *language* that stores the programming language of the source-code file and is used in the translation process. The optional attributes *filename* and *directory* store the path to the file. Originally, the given path to the file was split into these parts. In the current version, the path is stored in the attribute *filename* and the attribute *directory* is not automatically created. Options during translation allow these to be explicitly set. The last optional attribute is *version*, which is purely descriptive and allows for distinguishing between different files that have the same filename attribute.

A list of the srcML elements is given in Table I. Inside the element *unit* there is markup for statements, functions, classes, structs and unions, and exception handling. Comments are included in the element *comment* that has an attribute, *type*, whose value can be line or block. There are also optional elements for literals, operators, and type modifiers. Preprocessing elements are put in their own namespace *cpp*.

We must point out here that preprocessing directives and macros are marked up in srcML, much like any other part of the syntax. While the preprocessor can be run prior to conversion to srcML, most users keep the user-centric view of the code for queries and to support accurate round-trip transformations. As such, we assume that the preprocessor is not run and directives, macros, and template code remains and will be marked up as srcML.

Even with this added markup, the size of the srcML file is reasonable for an XML representation. In general, we see approximately a four-times increase in file size over the original text. For our Linux kernel example, the total text of the original source is 353 MB whereas the equivalent srcML representation is 1.2 GB. One of the reasons for this is that the markup stops at the expression level, with only variable names at calls marked.

XML uses namespaces as packaging and for disambiguation of elements. The srcML format defines a number of namespaces, as will be discussed in greater detail later in the paper. The namespace:

http://www.sdml.info/srcML/src

is used for most of the language base set of elements and is included in all srcML files. For C/C++ an additional namespace, *http://www.sdml.info/srcML/cpp*, is used for preprocessor elements. Special namespaces are used for optional markup and will be described later in the paper.

B. Addressing srcML

The srcML format allows for specific locations in the source code to be addressed for external links into the code, forming queries for fact-extraction, and in transformations. Due to the preservation of all text from the original source code, and the wrapping of parts of the text with srcML elements, addressing with srcML provides a direct way to address any of the source code text.

To refer to a specific structure in the code, the structure element, e.g., *class*, and the following name can be used. For example, to address the class Rotate, the XPath would be:

//src:class[src:name='Rotate']

The use of "//" means at any point in the document. The part in brackets is a predicate, and will apply if the XPath in it exists. Note that name of the class is not an attribute, but a nested element inside the class. This same approach can be used with the elements *function, call,* etc.

The extraction of elements in more specific locations can be obtained by giving the path to it. For example, to obtain all if-statements in the function rotate, the XPath would be:

//src:function[src:name='rotate']/src:block//src:if

Note that the body of a function (or class) is marked with the element *block*.

The limits of this often depend on how complicated an XPath the user can stand. Our work on method stereotyping applies some very complicated patterns against methods [4].

C. Preservation of Source Text

All text is preserved with as little processing as possible. However, there are some issues that arise with XML formats that the srcML representation needed to address.

Newlines, or EOL characters, are normalized in XML. The original newline characters are not stored. Therefore a srcML file can be created on one platform, and extracted on another platform that uses a different EOL representation. This is a common behavior for source code tools, such as version-control systems. Also, it is relatively easy to convert files to the desired EOL character.

TABLE I. A LIST OF THE SRCML ELEMENTS ORGANIZED INTO GENERAL CATEGORIES.

Category	srcML Elements
File/Project	unit
Statement	if, then, else, while, do, switch, case, default, for, init, incr, condition, break, continue, comment, name, type, block, index, expr_stmt, expr, decl_stmt, decl, init, goto, label, typedef, asm, macro, enum, empty_stmt, namespace, template, using, extern
Function/Method	function, function_decl, specifier, return, call, parameter_list, param, argument_list, argument
Class	class, class_decl, public, private, protected, member_list, constructor, constructor_decl, destructor, destructor_decl, super, friend
Struct and Union	struct, struct_decl, union, union_decl
Exception	try, catch, throw, throws
C-Preprocessor	cpp:directive, cpp:file, cpp:include, cpp:define, cpp:undef, cpp:line, cpp:if, cpp:ifdef, cpp:ifndef, cpp:else, cpp:elif, cpp:endif, cpp:then, cpp:pragma, cpp:error
Java	extends, implements, import, package
Extra Markup	literal, operator, modifier
Misc	escape

The XML meta-characters, '<', '>', and '&', occur frequently in source code and must be escaped. Some source code files contain special characters that cannot be stored in XML even if they are escaped. This issue is handled by including the special element *escape* with an attribute that contains the character value. One common example is the form-feed character which is represented by the element *<escape char="0xc"/>*. When converting back to source code, this element is translated back to its proper character value. Other examples are control codes that can be found in strings and comments.

We think of source code as being written in ASCII, with some exceptions such as Java with UTF-8. However, it is not unusual for programmers to use their native language for descriptions and their names in comments, and sometimes non-ASCII characters are used in literal strings. In most cases, the character set ISO-8859-1, sometimes referred to as Latin-1, is sufficient. However, there are examples of code written in Windows that use other character sets. The detection of what character set is used for a plain-text file cannot be automated. This can cause an issue because the conversion to XML, typically stored in UTF-8, requires knowledge of the original character set. The srcML toolkit supports encoding conversion for a wide variety of character sets.

D. Implementation

The srcML toolkit consists of two command-line tools. *src2srcml* provides for conversion of source code files to the srcML format, and the tool *srcml2src* provides for conversion of srcML files back to source code. In addition, srcml2src also provides options for general querying and transformation of srcML and for the handling of srcML archives. Specific features of these tools will be described with examples in later sections.

The srcML toolkit is built on a number of open-source libraries. ANTLR [2] is used for the underlying parsing. Although this was sufficient for the early versions of the toolkit, the need to handle more realistic situations of XML, encoding, and support for other formats for input and output led to the use of some additional packages. We will briefly describe these packages now and will give some example usage in following sections.

The libxml2 XML library[3] provides a full set of APIs for XML processing. For both src2srcml and srcml2src, libxml2 is used to produce the XML output, encoding conversion, and support of http: and ftp: protocols. For srcml2src it is also used to parse XML, and for XPath and XSLT evaluation. Some of the encoding conversion support of libxml2 is built-in, however, much of it is provided by the iconv library[4].

The other library that we recently integrated into the toolkit is libarchive[5]. This is a general-purpose input and output library for traditional archive formats: tar, cpio, zip, etc. It also provides support for compression formats such as gz, bzip2, etc. The greatest benefit is that the application program can write against one API, and have the automatic detection and extraction/decompression provided.

[2] http://www.antlr.org

[3] http://xmlsoft.org/

[4] http://www.gnu.org/software/libiconv/

[5] http://code.google.com/p/libarchive/

III. Single File Conversions

Before any fact extraction or transformation can be accomplished, the source must be converted to the srcML format. This conversion involves text encoding, determination of the programming language (C/C++/Java), optional markup, and other options for the conversion process. In this section, we will walk through the conversion of a single file and explain the various options that are available. The long form of options is used for comprehension, but as is typical of command-line tools, most of the options also have a short form. A complete list of the options can be found in the Appendix with Figure 5 for src2srcml and Figure 6 for srcml2src.

A. Round-Trip srcML Conversion

The simplest case is to convert a source code file to srcML. To convert the source code file rotate.cpp to the srcML file rotate.cpp.xml the command below is used:

src2srcml rotate.cpp -o rotate.cpp.xml

The speed of conversion to srcML is typically over 25 KLOC/sec. For most source-code files, the transformation to the srcML format takes fractions of a second.

To convert the srcML format file rotate.cpp.xml back to a source code file we use:

srcml2src rotate.cpp.xml -o rotatev2.cpp

The conversion back to a source-code file is even faster, and is often over 250 KLOC/second.

In general, this direct conversion to and from srcML will produce a file, rotatev2.cpp, that is identical to the original file. That is, the following pipe will find no differences:

src2srcml rotate.cpp | srcml2src | cmp rotate.cpp

where the *cmp* utility compares standard input to the file on a byte-by-byte basis. The only exception to this may be a difference in the EOL character as explained previously.

Although this direct conversion to and then from srcML is not directly useful, it does show how transformation and entity extraction can be integrated into a pipeline using the toolkit. For example, any XML transformation, here represented by the name *process*, can be integrated into the pipeline, such as in the following:

src2srcml rotate.cpp | process | srcml2src

This simple conversion will work for most cases; however there are issues that arise that can often be solved by various srcML options. The rest of this section will describe some of these options.

B. Encoding

The encoding of the original source code may need to be specified. The default is to assume ISO-8859-1 encoding and this works in most cases. If another encoding is needed, then this can be specified with the option *--src-encoding*. For example:

src2srcml --src-encoding=WINDOWS-1258 rotate.hpp -o rotate.hpp.xml

The other encoding issue concerns the resulting srcML file. The default is UTF-8 encoding, but other encodings can be specified via the option *--encoding*. For example:

src2srcml --encoding ISO-8859-1 rotate.hpp –o rotate.hpp.xml

Once in the srcML format, conversion back to source code also involves the selection of an output text encoding. The same option, *--src-encoding*, is used. For example:

srcml2src --src-encoding UTF-8 rotate.hpp.xml –o rotate.hpp

In most cases, these options are not necessary. One way to deal with the problem is to use the defaults and then check the resulting XML file to see if it is well formed.

C. Programming Languages

For conversion to srcML, the programming language of the source code is required for proper parsing (e.g., try is a keyword in C++ but not in C). Originally, the src2srcml tool assumed a default programming language of C++. Other languages could be specified with an option, however, if this was not properly specified, then incorrect markup may result. For example, there were cases where users would enter the command *src2srcml Rotate.java* and incorrectly assume that Java was being used as the language. Now, the type of programming language is automatically determined by the file extension, and usage errors due to language choice are avoided. A mapping of file extensions to languages is listed later on in the paper in Table III with the typical file extensions for the languages C, C++, and Java.

One exception that arises is for include .h files. From the extension, it may be assumed that the file is a C file. However, it is common to use the .h extension (as opposed to .hpp) for C++ files as was done until relatively recently in the source for the srcML toolkit.

In any case, the programming language can be explicitly set using the option *--language*. For example, to specify C++ for the include file rotate.h, the following command is used:

src2srcml --language=C++ rotate.h -o rotate.h.xml

D. Namespaces

A namespace prefix is used as shorthand inside of a XML file. The prefixes that srcML uses by default are the default prefix for the standard srcML elements, e.g., <*class*>, and *cpp* for the preprocessor elements, e.g., <*cpp:ifndef*>, but they can be reconfigured using the command line options. To specify a prefix the option *--xmlns* is used. For example, to declare *src* as the prefix for the standard srcML namespaces, the option would be:

--xmlns:src="http://www.sdml.info/srcML/src"

Note that the prefix is a local declaration to the file, and other prefixes can be used in queries for such things as fact extraction. This will be further demonstrated in the later section on fact extraction using srcML.

TABLE II. A LIST OF NAMESPACES FOR SRCML. THE MAIN NAMESPACE IS AT THE TOP FOLLOWED BY THE NAMESPACE FOR PREPROCESSING ELEMENTS. THIS IS FOLLOWED BY THE OPTIONAL NAMESPACES AND THEIR CORRESPONDING OPTION.

Option	*Prefix*	*Namespace*
-	-	http://www.sdml.info/srcML/src
-	cpp	http://www.sdml.info/srcML/cpp
--position	pos	http://www.sdml.info/srcML/position
--operator	op	http://www.sdml.info/srcML/operator
--literal	lit	http://www.sdml.info/srcML/literal
--modifier	type	http://www.sdml.info/srcML/modifier
--debug	err	http://www.sdml.info/srcML/srcerr

E. Optional Markup

Optional markup is available but not included by default because it can impact the size of the srcML file (i.e., increase the number of tags). Each optional markup forms a unique group that can be identified separately. These can be enabled by a specific option and each has a namespace and default prefix. A complete list of these options with their namespaces and default prefixes can be found in Table II. The use of a separate namespace allows us to tell if they are enabled. If the namespace for an option is declared in the root element, then the corresponding feature is enabled during parsing. This is a small feature to help in automating testing of the toolkit. We will briefly describe each option.

One consistent request by srcML users was for the recording of the line and column numbers for each element in the original source file. Typically, researchers wanted to integrate srcML into their line-based tools. In the past, we accomplished this by writing a separate utility for this purpose. Now, the recording of line and column position as attributes in an element can be turned on with the option *--position*. The feature adds two attributes, *pos:line* and *pos:column*. While the definition of line is clear, the definition of column is not. The exact column that the element appears on depends on the use of tabs and the tab stops. In order to deal with this, a tab stop of 8 was assumed as a default. This can be set to a different value with the option *--tabs*.

Originally, srcML marked elements only down to variables and calls at the expression level, e.g.,

```
<expr><name>a</name> = 1</expr>
```

This required further processing for applications that wanted to easily work with operators and literal values. The options *--operator* and *--literal* can be used to enable these features. The following is the same expression as before, but with the options to markup operators and literal values:

```
<expr><name>a</name> <op:operator>=</op:operator>
<lit:literal type=”number”>1</literal></expr>
```

There is a similar situation for type modifiers in declarations, i.e., ‘&’ and ‘*’ for C++, which are not marked by default. This marking can be enabled in a similar manner with the option *--modifier*.

One special option for internal debugging is the option *--debug* which adds additional elements to indicate problems in parsing.

Alternatively, declaring a prefix for the appropriate namespace can enable all of the optional markup features. For example, instead of using the option *--operator,* the namespace declaration option *--xmlns* can be used to define a prefix for the operator namespace:

--xmlns:oper="http://www.sdml.info/srcML/operator"

that enables the markup of operators, and also give them a prefix of *oper* instead of the default prefix *op*.

F. General Information

Meta-data about a srcML file can be generated using the option *--info* of srcml2src as below:

srcml2src --info rotate.hpp.xml

This option lists the namespaces and prefixes used, encoding, language, and values of the standard attributes *filename*, *directory*, and *version,* if they are used. Options exist to extract all of these items individually.

G. Miscellaneous Options

There are a few miscellaneous options that affect the behavior of the translation and are useful in various situations. First, although the srcML toolkit can handle a source code file that consists of a single statement, the attempt to convert a single expression that is not in a statement will be seen as an expression statement. The option *--expression* prevents the expression from being wrapped in a statement.

The stream processing used by the toolkit allows for the conversion to srcML on the fly. For example, the output of XML for an expression can begin as soon as the start of the expression is parsed. Due to output buffering, this is difficult to see. The option *--interactive* turns off this output buffering.

Finally, the srcML output can be compressed using the option *--compress*. A compressed file, either source code or srcML, is automatically detected when used as input with the srcML toolkit. For example, the Linux kernel can be compressed by a factor of almost 8 from 1.2 GB to 162 MB.

IV. PROJECT CONVERSIONS: SRCML ARCHIVE

The srcML toolkit provides a number of options and configurations for translation of individual files. For a large project, with a number of files, each file can be converted individually. However, we found it inconvenient to have a large number of separate XML files to process and deal with. Also, this is a limitation for fact extraction and multiple-file transformations. Therefore, the srcML archive format was created.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<unit xmlns="http://www.sdml.info/srcML/src">
<unit xmlns:cpp="http://www.sdml.info/srcML/cpp" language="C++" filename="rotate.hpp>…</unit>
<unit xmlns:cpp="http://www.sdml.info/srcML/cpp" language="C++" filename="rotate.cpp>…</unit>
<unit language=”Java" filename="Rotate.java”>…</unit>
</unit>
```

Figure 2. The outer elements for a srcML archive of three files, rotate.hpp, rotate.cpp, and Rotate.java. Each source code file is stored in a separate unit element that has its own attributes, and can be of a different language.

The srcML archive format allows for the representation (and processing) of multiple files in a single srcML file (basically a single file multi-document format). The original motivation for the format was actually for the test suite. For example, instead of a large number of separate files for fine-grained tests of parsing for if statements e.g., if1.cpp.xml, if2.cpp.xml, etc., all of them are stored in a single if.cpp.xml srcML archive file.

An example of the archive format is given in Figure 2. This shows a common situation for C++ where both the .hpp and .cpp file for a class are needed for analysis or manipulation. The srcML archive format is the default whenever multiple input files are indicated. The srcML file in Figure 2 can be formed from the following command:

src2srcml rotate.hpp rotate.cpp Rotate.java -o rotate.xml

For a large number of files, the filenames can be put into a list and the list used as input:

src2srcml --files-from=foo.txt -o rotate.xml

The generation of a srcML archive can be forced, even for a single source code file, using the option *--archive*. If the files to be converted are source code files in the same directory, then the name directory can be used:

src2srcml rotate/ -o rotate.xml

TABLE III. DEFAULT MAPPING OF LANGUAGES TO EXTENSIONS. IT IS ALSO POSSIBLE TO CHANGE THE MAPPING OF AN EXTENSION, OR TO CREATE NEW EXTENSIONS.

Language	***Extension***
C	.c, .h
C++	.cpp, .h˙pp, .cxx, .hxx, .cc, .hh, .c++, .h++, .C, .H
Java	.java

In a typical case, there are other files besides source code in a directory. This is so common that we build in direct support for it. Therefore, the tool uses the file extension to decide what is or is not a source code file. All files without registered extensions are ignored. The list of default-registered extensions is given in Table III. This also allows for multi-language projects, where each file in the srcML archive can be in a different language. Additional extensions can be registered, e.g., to register .h files as C++ use:

src2srcml --register-ext h=C++ rotate.h

In many cases the files may be stored in a standard archive format, i.e., tar, cpio, tar, etc., and may also include compression. The use of the library libarchive, internal to the toolkit, allows for the automatic direct handling of these files with no options required. The use of the libxml2 library also allows the tool to fetch remote files. For example, the following will create a srcML archive of the Linux kernel directly from kernel.org:

src2srcml http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.38.3.tar.bz2 -o linux.xml

The total time for this command was just around 7 ½ minutes on our test machine. This includes the time it took the tool to download, decompress, untar, and convert to the srcML archive.

Note that, as with input from a directory, the srcML archive is the default and is required, even with a single file in the source-code archive.

Finally, all of these sources can be combined into a single, srcML archive in one command:

src2srcml rotate/ project.tar.gz rotate.hpp rotate.cpp –o rotate.xml

The option *--info* for srcML archives provides the same set of information as it does with non-archive files. Even for very large srcML archives, it is very fast because only the root *unit* tag has to be processed. Another option, *--longinfo*, provides a count of the number of files contained in the archive. This requires the complete traversal of the archive but only takes about 15 seconds on an archive of the Linux kernel.

For extraction back to source code, the set of choices for output format are almost the same as what was supported for input formats, with minor exceptions due to limitations in the library *libarchive* used by the toolkit. To generate a gzipped tar file from the project, the command is:

srcml2src rotate.xml -o rotate.tar.gz

Individual files can be extracted from a srcML archive using the option *--unit* and by providing the number of the unit. For example, to extract the file rotate.cpp from the archive in Figure 2:

srcml2src --unit=2 rotate.xml

This extracts the text of the file. If the srcML form is needed, then the option *--xml* extracts the XML for that individual file into a non-archive srcML file with the contents. If the position

of a file in the srcML archive is not known, the --*list* option can be used to find the file.

Most of the conversion back to source code files is relatively straightforward. One that isn't is output to a directory. The model here is that of the tar utility for extraction. To specify the tar-like extraction of the complete srcML archive the command would be:

srcml2src --to-dir rotate/ rotate.xml

This also allows for round trip, from a directory of files back to the same directory, with any transformation that may be performed along the way.

V. TRANSFORMATION WITH SRCML

With the use of src2srcml and srcml2src, transformation pipelines can be created. Users can write a transformation using the various XML transformation language or API, including DOM, SAX, TextReader, LINQ, etc. XML transformation can also be performed with XSLT directly using srcml2src. To apply the XSLT program copy.xsl the command is:

srcml2src --xslt=copy.xsl linux.xml -o linux.copy.xml

One limitation of using XSLT is that it requires the complete DOM (Document Object Model) to be constructed in memory. For large projects the memory requirements for XSLT processing for the equivalently large srcML archive can exceed that of many machines. This is opposed to SAX handlers that process the XML a node at a time. Therefore, special processing of srcML archives is used with XSLT transformations. The entire document is traversed in a stream fashion using a SAX handler. When the srcML for an individual file in the archive is reached, the DOM for that particular file is created. The XSLT is applied to that individual DOM, the result is captured, and the DOM for that file is de-allocated. The result, when applied to a srcML archive, is the collection of results of applying the transformation to each individual unit in order. This is wrapped so that the collected result forms a srcML archive. This process can be applied repeatedly.

An example XSLT transformation for instrumenting code is shown in Figure 3. To each block of a function, a call to the function fprintf() is added to output the file and function names. This example shows that literal text can be output without srcML markup.

To perform a typical transformation on the entire Linux kernel takes under three minutes (approximately 50 KLOC/sec), not including conversion to/from srcML. A recursively-defined copy that simulates matching of every node to an XSLT template (simulating full processing of the XML) requires just over three minutes (200 seconds). Many common transformations work at over 80 KLOC/sec. using XSLT. Examples of transformations for adaptive maintenance changes using XSLT can be found in [5].

One problem with this approach is that the individual transformations do not have any access to the attributes on the root *src:unit* element, nor do the transformations know their position in the srcML archive. Therefore each XSLT program has access to the XPath extension function *src:archive()* and src:unit(). The function src:archive() provides access to the attributes and takes the name of the archive attribute as a parameter, e.g., *src:archive('language')*.. The other function, *src:unit()*, returns the unit number/position of an individual file in the srcML archive. These extension functions can also be used in XPath queries.

Parameters can be passed to the XSLT program through the option:

--xpathparam parameter_name=parameter_value

```
<xsl:stylesheet xmlns="http://www.sdml.info/srcML/src" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:cpp="http://www.sdml.info/srcML/cpp" version="1.0">
<xsl:output method="xml" omit-xml-declaration="yes" version="1.0" encoding="UTF-8"/>

<!-- instrument each function -->
<xsl:template match="function/block">

  <!-- copy the start of the block (first line) -->
  <xsl:copy-of select="node()[1]"/>

  <!-- insert the instrumented code -->
  fprintf(stderr, "FILENAME: <xsl:value-of select="/unit/@filename"/>\tFUNCTION: <xsl:value-of
select="../name"/>\n");

   <!-- copy the rest of the block -->
   <xsl:copy-of select="node()[position()!=1]"/>

</xsl:template>

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

</xsl:stylesheet>
```

Figure 3. An example XSLT transformation to instrument code where a trace call to fprintf() is inserted in each function. Inserted text is shown in bold. Note that literal text can be directly inserted, i.e., it does not have to be marked up with srcML elements. The call to fprintf is constructed from a mixture of literal text and XSLT to extract data from the function, i.e., the filename and function name. The template at the bottom is used for an identity copy.

This is only valid with the option *--xslt* and is analogous to the *--xpathparam* of the libxml2 utility *xsltproc*.

A. Filesystem Transformations

The srcML archive also allows for transformations that convert the file and directory structure of the program, and allows for the merging or creation of additional source files.

Certain transformations cannot be applied to each source code file individually. One example is coordinating changes to an .hpp and the corresponding .cpp file. In this case, both files should be transformed using a single XSLT program. The option *--apply-root* can be used. Note that if the srcML archive is quite large, then the transformation may take a particularly long time. For these cases, the archive should only consist of the files of interest.

Applying the transformation to the entire archive also allows for filesystem transformations. By a filesystem transformation, we mean where files are created, deleted, or moved in the directory structure, possibly with their contents moved around. The transformation can create new files by generating the element *unit* with the appropriate attributes and delete files by filtering out the appropriate element unit. Moving files around in the directory structure can be accomplished by changing the path that is stored in the attribute *filename*. This is one of the main reasons that the srcML archive stores the files in one long list and preserves the directory structure (for future extraction) in the attribute *filename*.

VI. Fact Extraction with srcML

Users can perform fact extraction in multiple ways with the srcML toolkit. They can simply use the toolkit to convert to the srcML format and then use their own XML tools for querying or traversal. For example, XPath can be used to address the needed fact in the documents. Once the result of a query is obtained, the user's tool can format the result to their requirements. This was the original intent as was described in the first paper on fact extraction using srcML [1].

However in order to do this, a user had to install and learn to use an XPath evaluation tool. Many of these are inefficient or difficult to understand/use. Further, these tools have no knowledge of the srcML format, namespaces, or structure of the srcML archive. In order to provide an alternative, we decided to build features for fact extraction directly into the srcml2src tool. This was relatively straightforward because these features are in the underlying libxml2 library. XPath queries can be directly performed with srcml2src,.e.g.,

srcml2src --xpath "/src:unit//src:expr" -o project.xml

The XPath expression is applied to each individual unit inside the srcML archive project.xml. The result is a series of fragments consisting of *src:expr* elements.

Note that in the previous example, the namespace prefix *src* is used. Many XPath evaluators interpret the XPath standard as requiring that a prefix be used, including libxml2. The explanation is that the XPath */unit* refers to all elements with the local name of *unit* that are not part of a namespace. This is different from the default namespace used inside of srcML for *unit*. However, namespace prefixes are local to an XML file and matching is based on the namespace, not the prefix. The prefixes used inside of the srcML file can be anything (and are configurable), as long as they map to the proper namespaces. A complete list of default prefixes and their namespaces can be found in Table II. In short, follow the previous example and your query should work.

Another general issue with fact extraction is the output format. The result may require further processing. Therefore, we decided to use the srcML archive format as the output format. Each result is placed into a separate *unit* element in the srcML archive along with the associated metadata, e.g., *filename* and *directory*. Another attribute, *item*, is used to provide a unique identifier for when multiple occurrences of the extracted code are found in the same file. An example is shown in Figure 4.

The time for the evaluation of an XPath depends on the amount of output. In general, the evaluation is quite fast. For example, an XPath query that examines each element of the entire Linux kernel but produces no output takes around 45

```
<unit xmlns="http://www.sdml.info/srcML/src">

<unit xmlns:cpp="http://www.sdml.info/srcML/cpp" language="C"
  filename="linux-2.6.38.3/Documentation/accounting/getdelays.c" item="1"
><condition>(<expr><name>fd</name> &lt; 0</expr>)</condition></unit>

<unit xmlns:cpp="http://www.sdml.info/srcML/cpp" language="C"
  filename="linux-2.6.38.3/Documentation/accounting/getdelays.c" item="2"
><condition>(<expr><name>rcvbufsz</name></expr>)</condition></unit>
…
<unit xmlns:cpp="http://www.sdml.info/srcML/cpp" language="C"
  filename="linux-2.6.38.3/Documentation/auxdisplay/cfag12864b-example.c" item="1">
<condition>(<expr><name>cfag12864b_fd</name> == -1</expr>)</condition></unit>
…
</unit>
```

Figure 4. A portion of the results of a fact extraction to find all conditions in the Linux kernel. The value of the filename and item number are in bold. Each match of the query is placed in its own unit element, with the unit attributes copied from the original unit that it was a portion. In order to distinguish separate occurrences, the attribute item is used as a counter.

seconds at over 150 KLOC/sec. Also, since the XPath is applied to each individual unit separately, the internal memory requirements are minimal in contrast to application on the entire nested document.

Another application of fact extraction is to filter the files in the project; to find the particular files that match a query. This is quite straightforward to apply. The XPath that matches the desired units is used as a predicate. For example, if we want to find all files that contain a call to a particular function, memcpy(), the command is:

```
srcml2src  --xpath
"/src:unit[.//src:call/src:name='memcpy'] "
linux.xml  -o memcpy.xml
```

Because the resulting XML element is *src:unit* and corresponds to a complete file, the result is placed in a srcML archive. This forms a subset of the original set of Linux files, all of which can undergo further processing.

Another way to filter files in a srcML archive is to use a RelaxNG grammar. The option *--relaxng* can be used to apply the grammar to the entire archive, as in the following:

```
srcml2src --relaxng=nestedif.rng linux.xml -o memcpy.xml
```

This applies the RelaxNG grammar nestedif.rng to each individual unit in the srcML archive and combines the individual files that match the grammar and places them into an output srcML archive. We are just starting to explore the use of RelaxNG for pattern matching with srcML.

Besides extraction of particular parts of a XML document, XPath also supports the calculation of numeric results. The XPath numeric function *count()* can be used to find the number of occurrences of an element. For numeric results, the processing of an XPath forms a sum of the individual results. For example, to find the number of conditions in the Linux kernel, the following query can be made:

```
srcml2src --xpath="count(/src:unit//src:condition)"
linux.xml
```

which, in our example Linux kernel is 738,712 and takes approximately one minute to determine. This is equivalent to extracting all the occurrences, then counting the number of resulting occurrences:

```
srcml2src --xpath="/src:unit//src:condition" linux.xml |
srcml2src --longinfo
```

VII. Extending the srcML Format

XML is an extensible format, and this philosophy is carried over into the srcML tools. It is possible to extend the markup with additional elements. This can be extended markup for the contents of strings, or additional elements to group items together.

One example of this can be found in srcDoc, a markup for structured comments, e.g., Javadoc, Doxygen. [6]. The srcML underwent additional markup in a new namespace using the srcDoc tool. Elements were added to markup the contents of comments. Another example is the srcDiff format, which is an extension of srcML that represents multiple versions of a source-code file in a single document that can be queried for analysis of code changes [7]. The toolset provides full support for these kinds of extensions.

Alternatively, users may not like some of the choices of elements names or require additional elements. Since the output is XML, in many cases a simple XML transformation will convert srcML to a markup more to their liking. The srcML tools do not validate against a DTD or any schema, so the elements can be put into the default srcML namespace.

VIII. Related Approaches and Tools

We 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 [8, 9] and tool. In [8] the concept of the *documentary structure* of source code, whose elements include all white space and comments, is presented. This documentary structure is often at odds with the linguistic structure of the program. Unfortunately for many parse-tree-based approaches, this documentary structure is completely lost. Attempts to preserve these ties often result in the documentary structure not being easily integrated back into the representation.

In contrast to these requirements, software-development tools typically take a totally compiler-centric approach of representing the source code as an abstract syntax tree. It has been observed that these approaches are often not a good match to the problems that they are trying to solve [8, 10]. There are exceptions to this problem with compiler-centric approaches, with one example being the DMS systems by Baxter [11]. Baxter has gone to great lengths to address this specific issue by storing important textual items within the underlying abstract-syntax graph. Also, as a full compiler (i.e., heavy weight) approach, it allows for static analysis to be built into the transformation. Our approach is very lightweight by comparison and uses widely available and accessible XML technologies. One approach is to move down to the level of lexical analysis and provide for the transformation at that level, as in [12]. This allows for the preservation of all of the text, but at a cost of complex regular expressions. Also, with this approach, it is not as easy to provide for abstractions that reflect static analysis. Another approach that preserves the programmer's view is to move the transformation to the level of the grammar as in TXL [13]. 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.

The Proteus system [14] addresses similar problems of performing transformations on large C++ systems while preserving the layout and handling code before preprocessing. They refer to this as "high-fidelity" transformations. An AST approach is used, with white space and comments stored in additional AST nodes. They provide their own language

YATL for transformations on the AST. Additionally, in [15] these documentary structure issues are seen as a cross-cutting concern in the form of annotated parse trees. Other approaches include using an intermediate language to describe the source, as in the case of the C Intermediate Language (CIL) [16].

The lightweight approach used in srcML preserves the documentary structure, as is done in some of these approaches, while at the same time integrates static analysis into transformations that go down to the lexical level. This results in a combination of a lightweight approach and static analysis in an efficient and useable manner.

IX. Future Directions

Development of the srcML toolkit is continuing and much of this is based on feedback and questions from users (and sponsors). Besides fixing parsing errors and incorrect markup, additional features are being developed. First, we are extending the markup and the toolkit to additional languages. One language that we are continually asked about is C#. Our analysis so far is that this should be a relatively straightforward addition. The C++0x standard introduces new language constructs and is already partially implemented in the toolkit. Due to the recent large interest in mobile apps, including iOS, another language of current interest is Objective-C. Due to its C/C++ base, much of the language is already handled. However, the Smalltalk-style syntax used for messaging in Objective-C will require additional markup.

Currently, any XML notation in the original source code is escaped. One example of this is in comments where XML is used for Javadoc and Doxygen. We first ran across this in widespread use in C# code. It may be useful to preserve this XML so that it can be referenced in queries and transformations. This presents issues for the parser in preserving this XML. We foresee adding a standard namespace/prefix for these elements, and providing an option for the user to provide a namespace and prefix.

The other area of interest is in direct support for XPath extension functions. Currently, they must be defined and used in XSLT and are not for direct queries using XPath. We see the need for some that are predefined, e.g., *src:statement*() which maps to a collection of all statements, i.e., if, while, expression statements, etc. These could then be used directly in XPath queries, e.g., */src:unit/src:unit//src:statement(.).*

XPath extension functions are typically written in other languages and then are linked into the program. Since we are using libxml2, they would have to be written using the extension API of libxml2 and statically linked into the srcML toolkit. However, many of the possible extension functions can be directly expressed in (typically very complicated) XPath. In this case, the extension function serves as a type of macro language. We are currently investigating providing the ability to declare these functions as an option to the toolkit, or stored in a file. This list of macro-type extensions would then be registered and executed dynamically as needed.

X. Conclusions

The srcML toolkit has gained many features over the years and has turned into a powerful tool for fact-extraction and transformation. As noted, the development of srcML continues. We encourage you to contact the authors if you have any questions.

XI. References

[1] Collard, M. L., Kagdi, 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.

[2] Collard, M. L. and Maletic, J. I., "Document-Oriented Source Code Transformation using XML", in Proceedings of 1st International Workshop on Software Evolution Transformation (SET'04), Delft, The Netherlands, Nov. 9 2004, pp. 11-14.

[3] 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.

[4] Dragan, N., Collard, M. L., and Maletic, J. I., "Reverse Engineering Method Stereotypes", in Proceedings of 22nd IEEE International Conference on Software Maintenance (ICSM'06), Philadelphia, Pennsylvania USA, September 25-27 2006, pp. 24-34.

[5] Collard, M. L., Maletic, J. I., and Robinson, B. P., "A Lightweight Transformational Approach to Support Large Scale Adaptive Changes", in Proceedings of 26th IEEE International Conference on Software Maintenance (ICSM'10), Timisoara, Romania, Sept 12-18 2010, pp. 10 pages to appear.

[6] Shearer, C. D. and Collard, M. L., "Enforcing Constraints Between Documentary Comments and Source Code", in Proceedings of 15th IEEE International Conference on Program Comprehension (ICPC'07), Banff Canada, June 26-29 2007, pp. 271-280.

[7] Maletic, J. I. and Collard, M. L., "Supporting Source Code Difference Analysis", in Proceedings of IEEE International Conference on Software Maintenance (ICSM'04), Chicago, Illinois, September 11-17 2004, pp. 210-219.

[8] 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.

[9] Cordy, J. R., "Comprehending Reality - Practical Barriers to Industrial Adoption of Software Maintenance Automation", in Proceedings of 11th IEEE International Workshop on Program Comprehension (IWPC'03), Portland, OR, May 10-11 2003, pp. 196-206.

[10] Klint, P., "How Understanding and Restructuring Differ from Compiling - A Rewriting Perspective", in Proceedings of 11th IEEE International Workshop on Program Comprehension (IWPC'03), Portland, OR, May 10-11 2003, pp. 2-12.

[11] Baxter, I. D., Pidgeon, C., and Mehlich, M., "DMS: Program Transformations for Practical Scalable Software Evolution", in Proceedings of 26th International Conference on Software Engineering (ICSE04), Edinburgh, Scotland, UK, May 23 -28 2004, pp. 625-634.

[12] Cox, A. and Clarke, C., "Relocating XML Elements from Preprocessed to Unprocessed Code", in Proceedings of Proceedings of the IEEE 10th International Workshop on Program Comprehension (IWPC'02), Paris, France, June 2002, pp. 229-238.

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

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

[15] Kort, J. and Lammel, R., "Parse-tree annotations meet re-engineering concerns", in Proceedings 2003, pp. 161–171.

[16] 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.

XII. APPENDIX

```
Usage: src2srcml [options] <src_infile>... [-o <srcML_outfile>]

Translates C, C++, and Java source code into the XML source-code representation srcML.
Input can be from standard input, a file, a directory, or an archive file, i.e., tar, cpio, and zip.
Multiple files are stored in a srcML archive.

The source-code language is based on the file extension.  Additional extensions for a language
can be registered, and can be directly set using the --language option.

By default, output is to stdout.  You can specify a file for output using the --output or -o option.
When no filenames are given, input is from stdin and output is to stdout.
An input filename of '-' also reads from stdin.

Any input file can be a local filename (FILE) or a URI with the protocols http:, ftp:, or file:

Options:
  -h, --help              display this help and exit
  -V, --version           display version number and exit

  -l, --language=LANG     set the language to C, C++, or Java
  --register-ext EXT=LANG register file extension EXT for source-code language LANG

  -o, --output=OUTPUT     write result to OUTPUT which is a FILE or URI
  --files-from=INPUT      read list of source file names, either FILE or URI, from INPUT
                          to form a srcML archive
  -n, --archive           store output in a srcML archive, default for multiple input files
  -e, --expression        expression mode for translating a single expression not in a statement
  -x, --encoding=ENC      set the output XML encoding to ENC (default:  UTF-8)
  -t, --src-encoding=ENC  set the input source encoding to ENC (default:  ISO-8859-1)

  -z, --compress          output in gzip format
  -c, --interactive       immediate output while parsing, default for keyboard input
  -g, --debug             markup translation errors, namespace http://www.sdml.info/srcML/srcerr
  -v, --verbose           conversion and status information to stderr
  -q, --quiet             suppresses status messages

  --no-xml-declaration    do not output the default XML declaration
  --no-namespace-decl     do not output any namespace declarations

Metadata Options:
  -d, --directory=DIR     set the directory attribute to DIR
  -f, --filename=FILE     set the filename attribute to FILE
  -s, --src-version=VER   set the version attribute to VER

Markup Extensions:
  --literal               markup literal values, namespace "http://www.sdml.info/srcML/literal"
  --operator              markup operators, namespace "http://www.sdml.info/srcML/operator"
  --modifier              markup type modifiers, namespace "http://www.sdml.info/srcML/modifier"

Line/Column Position:
  --position              include line/column attributres, namespace "http://www.sdml.info/srcML/position"
  --tabs=NUMBER           set tabs NUMBER characters apart.  Default is 8

Prefix Options:
  --xmlns=URI             set the default namespace URI
  --xmlns:PREFIX=URI      set the namespace PREFIX for the namespace URI

                          Predefined URIs and Prefixes:
                            xmlns="http://www.sdml.info/srcML/src"
                            xmlns:cpp="http://www.sdml.info/srcML/cpp"
                            xmlns:err="http://www.sdml.info/srcML/srcerr"

CPP Markup Options:
  --cpp-markup-else       markup cpp #else regions (default)
  --cpp-text-else         leave cpp #else regions as text

  --cpp-markup-if0        markup cpp #if 0 regions
  --cpp-text-if0          leave cpp #if 0 regions as text (default)
```

Figure 5. Complete list of options for src2srcml from the help output.

```
Usage: srcml2src [options] <srcML_infile>... [-o <src_outfile>]

Translates from the the XML source-code representation srcML back to source-code.

Extracts back to standard output, the disk, or to traditional archive formats,
e.g., tar, cpio, zip, and with optional gzip, bzip2 compression.
Provides access to metadata about the srcML document. For srcML archives
provides extraction of specific files, and efficient querying/transformation
using XPath, XSLT, and RelaxNG.

srcML archives contain multiple individual source code files,
e.g., an entire project or directory tree.

By default, output is to stdout.  You can specify a file for output using the --output or -o option.
When no filenames are given, input is from stdin and output is to stdout.
An input filename of '-' also reads from stdin.

Any input file, including XSLT and RelaxNG files, can be a local
filename (FILE) or a URI with the protocols http:, ftp:, or file:

The srcML files can be in xml, or compressed with gzip or bzip2 (detected automatically).

Options:
  -h, --help              display this help and exit
  -V, --version           display version number and exit

  -o, --output=OUTPUT     write result to OUTPUT which is a FILE or URI
  -t, --src-encoding=ENC  set the output source encoding to ENC (default:  ISO-8859-1)
  -z, --compress          output text or XML in gzip format
  -v, --verbose           conversion and status information to stderr
  -q, --quiet             suppresses status messages

  -X, --xml               output in XML instead of text
  --no-xml-declaration    do not output the XML declaration in XML output
  --no-namespace-decl     do not output any namespace declarations in XML output

Metadata Options:
  -l, --language          display source language and exit
  -d, --directory         display source directory name and exit
  -f, --filename          display source filename and exit
  -s, --src-version       display source version and exit
  -x, --encoding          display xml encoding and exit
  -p, --prefix=URI        display prefix of namespace given by URI and exit
  -n, --units             display number of srcML files and exit

  -i, --info              display most metadata except file count (individual units) and exit
  -L, --longinfo          display all metadata including file count (individual units) and exit
  --list                  list all the files in the srcML archive and exit

srcML Archive Options:
  -U, --unit=NUM          extract individual unit NUM from srcML
  -a, --to-dir            extract all files from srcML and create them in the filesystem

Query and Transformation Options:
  --xpath=XPATH           apply XPATH expression to each individual unit
  --xslt=XSLT_FILE        apply XSLT_FILE (FILE or URI) transformation to each individual unit
  --xpathparam NAME=VAL   passes a parameter NAME and VAL to the XSLT program
  --relaxng=RELAXNG_FILE  output individual units that match RELAXNG_FILE (FILE or URI)
  --apply-root            apply an xslt program or xpath query to the root element
```

Figure 6. Complete list of options for srcml2src from the help output.