

Supporting Document and Data Views of Source Code

Michael L. Collard
Department of Computer Science
Kent State University
Kent, Ohio 44242, USA

collard@cs.kent.edu

Jonathan I. Maletic
Department of Computer Science
Kent State University
Kent, Ohio 44242, USA

jmaletic@cs.kent.edu

Andrian Marcus
Department of Computer Science
Kent State University
Kent, Ohio 44242, USA

amarcus@cs.kent.edu

ABSTRACT

The paper describes the use of an XML format to store and represent program source code. A new XML application, srcML (SouRCe Markup Language), is presented. srcML presumes a document view of source code where information about the syntactic structure is layered over the original source code document. The resultant multi-layered document has a base layer of all the original text (and formatting). The second layer is the syntactic information, derived from the grammar of the programming language, and is encoded in XML. This multi-layered view supports both the creation and viewing of the source code in its original form and the use of XML technologies (for tasks such as analysis and transformation of the source). Although directed at source code documents, (particularly C++) srcML is also applicable to other programming languages and to languages with a strict syntax. srcML represents a departure from the compiler centric manner in which source code is commonly stored, instead a document point of view is taken thus better supporting the manipulation and management of the large numbers of source documents typical in modern software systems.

Categories and Subject Descriptors

D.2.3 [Software Engineering] Coding Tools and Techniques
I.7.2. [Document and Text Processing] Document Preparation

General Terms

Documentation, Languages

Keywords

XML, source code, markup language, program analysis, abstract syntax tree

1. INTRODUCTION

It is a bit contradictory that most software engineers use very primitive document management tools in the development and maintenance of source code and associated documentation. However currently, software engineers really don't have a choice in the matter. The most important artifact (i.e., product) they produce, source code, is totally constrained with regard to format.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DocEng'02, November 8-9, McLean, Virginia USA.
Copyright 2002 ACM 1-58113-594-7/02/0011...\$5.00.

That is, source code must be in plain text – flat files. The most sophisticated tools for dealing with the contents of such a format are search via regular expression. Of course, the intended user of this format is the compiler (lexical analyzer, parser, linker etc.). It is an extremely powerful tool but is only good at generating executable code (and identifying a restricted set of possible errors). A compiler is not very useful to manage the source code and other software documents.

Our proposition is to change the underlying representation of source code to facilitate the use of more powerful document management methods and tools. More importantly, however, we do not propose to change the business model. That is, developers will still be able to write source code in plain text and compilers will still read plain text as input. The twist on the model is that the source code will be marked up in an XML format and translated to plain text for the compiler. The developer will develop the code in a special editor that automatically marks up the source or alternatively a translator can be used to mark up existing plain text source code files.

In this paper, we describe an XML application, *srcML*¹ (SouRCe Code Markup Language), which is used to add structural information to unstructured source code text files. Program source code is intrinsically structured but the manner in which it is stored has almost no structure at all. srcML adds much of the syntactic information found in an abstract syntax tree derived from parsing. However, the representation does not remove the programmer-centric parts of the source, that is, comments, spacing, formatting, macro definitions, etc. are retained in srcML. srcML combines the structure of the syntax (derivation) tree with the generality of the text file. The text can then be more easily searched, parsed, and transformed with the aid of these tags. Additionally, links to external documentation, other parts of the source, and multi-media descriptions can easily be added into the source. A preliminary discussion of srcML is also presented in [7].

As said previously, the programmer does not work directly in srcML, rather they work in a translation (or view) of a srcML document. This view can be exactly what the programmer originally typed, or alternatively, the programmer could define a variety of views - an obvious example is a pretty-print version.

While srcML may be a convenient representation for reformatting code, it is not our ultimate goal. Source code marked up in srcML is already parsed (at least partially) and the comments are intact. Therefore, doing static analysis, slicing, deriving call graphs, etc. becomes drastically simpler. That is, srcML is an

¹ Pronounced, "Source ML".

excellent representation for tools that support programming (development), maintenance, re-engineering, and software document management.

This change in underlying representation directly supports a number of ongoing research endeavors in the software engineering community. Our research group is particularly interested in this type of representation to better support our research on the development of better static analysis methods [9], identification of clones during re-engineering [12], and the visualization of design and architectural information of software systems [8].

Other research that would benefit from this type of underlying source code representation includes work on requirements traceability, adding hyperlinks into source code to support document management, and advanced integrated development environments. This relates to the work of Storey on the software visualization environment ShriMP [16], Munson and the software concordance project [14], work by Antoniol on recovering traceability links [1], and the Harmonia [4] to mention a few.

```
/*
 * rotate.h
 */

#ifdef ROTATE_H
#define ROTATE_H

// rotate three values
void rotate(int&, int&, int&);

#endif

#include "rotate.h"

// rotate three values
void rotate(int& n1, int& n2, int& n3)
{
    // copy original values
    int tn1 = n1, tn2 = n2, tn3 = n3;

    // move
    n1 = tn3;
    n2 = tn1;
    n3 = tn2;
}
```

Figure 1.a Original Source Code Files

```
void rotate(int& n1, int& n2, int& n3);

void rotate(int& n1, int& n2, int& n3) {
    int tn1 = n1;
    int tn2 = n2;
    int tn3 = n3;
    n1 = tn3;
    n2 = tn1;
    n3 = tn2;
}
```

Figure 1.b Single AST View of Source Code

The next section describes our underlying philosophy behind srcML followed by a discussion of other XML representations of software products. This related work motivated our many of our

discussions in the ongoing development of srcML. Finally we describe the main features of srcML.

2. CURRENT VIEW OF SOURCE CODE

Implicitly, there are two views of source code documents; namely the programmer view and the compiler view. In the programmer view, source code is a set of documents represented as plain-text files and this representation supports program creation and editing. In the compiler view, source code is transformed into an abstract form that supports the compilation process. Symbol tables and abstract syntax trees (AST) are created and effectively replace the plain-text files. The programmer view is a document view of the source while the compiler view is a data view of the source.

2.1. Document View of Source Code

The current document view, the one that a programmer sees and works with, is a simple text file with a sequence of characters arranged in lines. Typical programmer application tools reflect this fundamental view of source code. For example the UNIX utility `diff`, which forms the basis of many of the source code version control tools such as CVS and RCS, works on source files and finds differences between lines in these files.

Regular expression tools, (e.g., `grep`) search text files to find lines containing a match to a given regular expression. Again this reinforces the line view of a file. Regular expressions are very useful in locating small sections of text that match a particular pattern, such as identifiers of a particular pattern. They are powerful enough to locate some syntactic elements, such as if-statements, however these applications require very complex regular expressions are difficult for a programmer to create for a specific situation.

This type of complex syntactic matching is done in context-sensitive editing tools, for example `emacs` font-lock modes provides highlighting and other features based on regular expression matching. However these regular expressions are difficult enough to come up with that they are rarely used for programmer specific problems.

The limitation of the current document view to sequential text files restricts applications based on them to lexical analysis of the source code. This provides limited support for applications that depend on the syntactic structure of the source code

2.2. Data View of Source Code

Parsing and the lexical analysis for a typical programming language generate an Abstract Syntax Tree (AST) and a symbol table. The format and contents of this output are designed as input for code generation. This representation is based on the syntactic features of the language and it provides a richer view for applications. However, much of the information of the original source code is lost as shown in Figure 1.

The first thing that a parser does to achieve a compiler-centric view of the source code is to strip the comments out of the document. In languages such as C and C++ a separate program, the pre-processor, is used to remove the comments before the parser itself works on the document. While some work on ASTs does keep comments, they are often stored as attributes of a given syntactic element of the language. Although it is easy to decide

that a comment inside of a syntactic element, such as a block, belongs to the block, it is not easy to decide which of the comments neighboring syntactic elements it belongs to so it could be wrong.

Another loss of lexical information is white space. Although in some cases the line and column information is preserved in the AST, usually for debugging purposes, this still ignores the exact content of the white space: spaces, tabs, or a combination of both.

The AST view of the source code also loses information such as how variables are declared: individually or as a group. Not only is lexical information lost, it makes regeneration of the original source code impossible.

Since the AST it is produced by a compiler it requires source code that is syntactically and semantically valid. Even if a code fragment were extracted from a valid program file, it would probably not be semantically valid, so only complete programs can be handled in this manner.

Integrated development environments that support the various software engineering tasks (e.g., maintenance, reverse engineering) require a more powerful representation of the source to be more computationally viable.

3. XML AND SOURCE CODE

Although XML started out as a document format based on SGML (i.e., XML 1.0) it has also developed into a data format. This produces two views of XML one as a document format, another as a data format. Document formats, such as DocBook [19], TEI [15], etc., have concentrated on structuring written text and the problems of multiple displays of that text. The data view of XML treats the document as a source of data. In an extreme data view there might not be any text, instead only numeric data.

3.1. XML as a Document and Data Format

XML document formats are designed for original generation of XML, i.e., the user, possibly through some tool, generates and edits the text directly in the XML format. A document format may also be concerned with the conversion of the document from one XML format to another, but the goal is to get the generation of the original text in an XML format as early in the document creation process as possible. The only interest in non-XML formats, such as MS Word, is for export to that format or possibly in importing into the XML format. There is little interest in an XML document format in supporting the viewing or editing of the document in a possibly original non-XML format.

XML data formats provide a database view of the data as a collection of nested information. If the information was ever from a document in another format there is little of the original format left, only an abstraction of the original format. Written documents are not usually stored in XML data formats, only the information extracted from them.

XML formats are full of differences between the document and data view. Comments in an XML document are not considered part of the documents character data and an XML processor does not have to permit that retrieval of text of comments [18]. However the XSLT model (see www.w3.org/TR/xslt), primarily thought of as providing document versus data processing,

includes comments. For document view application we might want access to the comments, for a data view application we might not.

3.2. XML Data Formats for Source Code

A number of options currently exist for representing source code information (e.g., AST or ASG) in a XML data format namely, GXL [6], CppML [11], ATerms [17], GCC-XML, and Harmonia [4]. In these formats the AST (actually an ASG) of the source code, as output from a compiler intended for code generation, is stored in a data XML data format. In JavaML and GCC-XML the AST is mapped to the nested structure of XML. In GLX a graph view of the source code is stored, i.e. storing all nodes and vertices of the graph with no mapping of the nested structure of the source code to the nested structure of XML.

However, these representations are constructed as data exchange languages or for displaying program structural information. None of these representations directly supports the representation of comments or formatting information. The most widely used of these, GXL [6] is an XML-based exchange format for graph-like structures based on GraX (Graph eXchange format) [5], and RSF (Rigi Standard Format) [20]. Software systems are represented as ordered, directed, attributed, and/or typed graphs. While GXL is designed to be a standard exchange format for data that is derived from software, srcML is designed to represent the actual source code. Although srcML can be used as a standard exchange format, the underlying goal of defining and using srcML is to create an intermediate layer of representation between the source code, the developer, and tools that allows easy transformation to a standard exchange format such as GXL.

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

In the same realm, the Harmonia framework [4] and cppML/JavaML developed at the University of Waterloo [11] are closely related approaches since they encode the AST itself and actual source code, rather than data extracted (such as the case in GXL). While Harmonia adds tags to source code as metadata, cppML only uses tags and records the additional information as attributes on the tags. The differences mentioned above for Badros' work stand for these approaches as well.

In short, srcML is an attempt to keep the textual semantics of the source code intact while adding explicit structural information.

This leaves us with a much richer representation to work with than plain text, but with all the flexibility.

The XML data view of source code, since it is based on the AST, is a “heavyweight” format. It requires complete parsing of the original document and generation of the complete AST. Many comprehension activities do not require a complete parse tree or AST of the source code. Atkinson [2] argues that generating the entire AST is often times impractical and performing incremental parsing or “as needed” parsing is a better approach for many analysis tasks.

The AST & symbol table format provides little more support for most of these tasks. This is due to the loss of source code textual information. For tasks in which source code is a final result, such as viewing, editing, and software visualization, the resultant view differs too much from the source code view. For document linking it limits what we can link to in the source code itself. The only task for which this mapping doesn’t hinder is in static analysis. Here we are trying to extract abstract data, such as a call graph, from the source code and this type of representation is necessary. However this format is still lacking in that it typically cannot perform analysis on comments or preprocessor statements.

Table 1. The degree of support each representation lends to different tasks. There are two groups of tasks, the document aspects and the software aspects.

In

	Document Engineering		Software Engineering	
	Viewing/ Editing	Linking/ Querying	Software Visualization	Static Analysis
Plain-Text Source Code	Medium	None	None	None
AST & Symbol Table	Low	Low	Low	Medium
srcML	High	High	High	Medium

Although these source code formats have specific uses and permit the use of the XML technologies we believe that they don’t address all of requirements dictated for source code document engineering.

Other work on source code interchange formats includes the work by Malton et al [10]. While this work is not an XML application it has many of the same features as srcML and the other formats described previously. Malton’s work addresses issues of design recovery through source factoring on legacy code.

4. Requirements of the Representation

In Table 1 we see the range of degrees of support for the various representations of source code with respect to particular tasks. These tasks can be broken into two broad categories, document engineering tasks and software engineering tasks. Document engineering tasks reflect the issues of the written documents such as viewing, editing, and managing the documents. This also include tasks such as querying and linking to other documents. Software engineering tasks reflect concepts such as version control, debugging, and analysis.

For these groups of tasks we have three possible source code formats: plain-text that the programmer typically works with, the AST and symbol table that a compiler produces after it has parsed the plain-text source code, and srcML (i.e., the hybrid of the two).

The only support that the plain-text provides is through regular expression matching to text information. This provides limited access to lexical information thus it can be utilized for simple viewing/editing tasks. However access to syntactic information is through complex regular expressions. Access to syntactic information is necessary for linking elements in the document or to querying to extract subparts. As such this format is very awkward for these types of tasks.

addition, the AST format is limited to syntactically correct code fragments (at best). This format requires a complete parse and requires a syntactically correct source code document. This precludes applying this to any code fragments. On the other hand, the existence of a complete symbol table is the only way that complete analysis can take place.

The srcML format has a high degree of support for all of these tasks except for static analysis. The direct mapping of srcML back to the source code and the availability of syntactic information make this possible. The only deficiency is the lack of a complete symbol table that makes some tasks of static analysis difficult.

5. OVERVIEW OF srcML

The previous sections motivate why we developed srcML and why we decided simply overlay the source code’s syntactic structure, in XML, over the original source code text – thus supporting both a document view of the original text, and a data view with the XML markup. The driving principle behind srcML is to provide the user (human or tool) with the ability to view those elements and features of the source code that are needed for their task. The representation of the source code as structured documents directly supports the following:

- Representation of multiple levels of granularity within the AST;
- Multiple level of abstraction (or views);
- Transformation equality of source to representation and of representation to source;
- Query-able and search-able representation;
- Representation of structural information, including macros, templates, and compiler directives (e.g., #include), etc.;
- Preservation of:

- o Location of constructs;
- o Text formatting information;
- o Comments and their location;
- o File names and structure.
- o Macros and macro definitions

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

The srcML representation of Figure 1.a's source code is given in Figure 2 and will be discussed in the following sections.

5.1. Text

In srcML all original text is stored as text in the XML document. Elements occur in the same document order as they do in the original document (as typed by the developer). Generating the original source document can be done by output of text nodes only. While it isn't necessary to store the keywords in some applications, such as viewing or editing, they will be needed and may as well stay in the document (we found no good reason to remove them).

White space in XML includes spaces, tabs, and blank lines. While in many XML applications white space is normalized it can be preserved. White space inside of attributes, however, is normalized to a single space and is not preserved. Thus, we only store meta-information about the code in attributes. White space between srcML elements is exactly the same as the white space between the language elements in the source code document.

All language items appear on the same line in the <unit> element as they would in the source code file.

While explicitly storing white space could be replaced by storing the line and column that each element starts, as is done in JavaML, this would lose what the white space was composed of. Also the amount of storage seems to be about the same as storing these two attributes for every element. And while it isn't necessary to store characters such as the beginning of a comment or the semi-colon at the end of a statement, they are preserved for consistency with the storage of other text.

Program comments are stored in a <comment> element treating them as first-class syntactic elements with all formatting and location preserved. The user can define rules on how the comments associate with other elements of the source (e.g.,

```
<unit filename="rotate.h" dir="." xmlns:cpp="http://www.sdml.cs.kent.edu/cpp">
<comment type="block">/*
  rotate.h
*/</comment>

<cpp:ifdef>#<cpp:directive>ifdef</cpp:directive> <cpp:token>ROTATE_H</cpp:token><cpp:then>
<cpp:define>#<cpp:directive>define</cpp:directive> <cpp:token>ROTATE_H</cpp:token></cpp:define>

<comment type="line">// rotates three values</comment>
<func-decl><type>void</type> <name>rotate</name><formal-params>(
<param><type>int&amp;</type> <name>n1</name></param>,
<param><type>int&amp;</type> <name>n2</name></param>,
<param><type>int&amp;</type> <name>n3</name></param>)</formal-params>;</func-decl>
</cpp:then><cpp:endif>#<directive>endif</directive></cpp:endif></cpp:ifdef>
</unit>

<unit filename="rotate.cpp" dir="." xmlns:cpp="http://www.sdml.cs.kent.edu/cpp">
<cpp:include>#<cpp:directive>include</cpp:directive> <cpp:file>"rotate.h"</cpp:file></cpp:include>

<comment type="line">// rotate three values</comment>
<func-defn><type>void</type> <name>rotate</name><formal-params>(
<param><type>int&amp;</type> <name>n1</name></param>,
<param><type>int&amp;</type> <name>n2</name></param>,
<param><type>int&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> = <expr><name>n1</name></expr></decl>;</decl-
stmt>
  <decl-stmt><decl><type>int</type> <name>tn2</name> = <expr><name>n2</name></expr></decl>;</decl-
stmt>
  <decl-stmt><decl><type>int</type> <name>tn3</name> = <expr><name>n3</name></expr></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>tn2</name></expr>;</expr-stmt>
  <expr-stmt><expr><name>n3</name> = <name>tn1</name></expr>;</expr-stmt>
}</block></func-defn>
</unit>

</unit>
```

Figure 2. srcML of Original Source Code Files

methods, classes, etc), or define special types of comments (e.g., PRE and POST conditions). Once these rules are defined, the user can obtain a view from the srcML document that shows these relations between comments and their associated source code elements.

5.2. Elements

Every srcML document has a corresponding source code file. This is represented in the srcML document by the element `<unit>` representing a single compilation unit. The attributes of the element `<unit>` store the file name and directory. Include files (i.e., a `.h` file in C++) are also stored in their own `<unit>` element; the contents of the include file are not automatically inserted or applied to the source code files in which they are included. This allows further processing of the documents from the programmer centric view.

Each statement in the source code, down to the expression level, has its own element and is marked accordingly.

As a data format, srcML includes much of the information from an AST of the parsed source. The syntactic structure of the source code is marked up to allow for easy extraction of structural information of the source. srcML encodes data extracted from a partial derivation of the syntax tree. Parsing the source to a certain granularity level (e.g., expression) still offers the developer sufficient information to carry out most software engineering tasks. If a finer granularity level is desired, then only the parts of the source that were not previously parsed need to be examined and parsed (e.g., the expressions).

5.3. Preprocessor Constructs

A practical interest here is the difficulty dealing with macros, templates, and other preprocessor constructs in languages such as C++. srcML does not require complete parsing of this type of information. These types of constructs are simply marked up with specific tags in srcML (e.g., `<cpp: *>`) and not run through the preprocessor. The source is not completely parsed for translation into srcML. We use a partial derivation, stopping before we reach a particular level of syntactic abstraction. For example, in our case we do not completely parse expressions. This allows for on-the-fly generation of srcML. srcML can also represent syntactically incorrect code fragment. The issue of syntactical correctness is a compiler problem and is not of supreme importance to the representation.

5.4. Non-Context-Free-Grammar

Since C++ does not have a Context Free Grammar some semantic parsing must be done for full identification of program structural elements. A specific example is finding the difference between a function and a variable declaration, take the source code fragment below:

```
int f(a);
```

In order to solve the ambiguity of whether `f` is a function (this could be a function specification) or variable identifier we must know whether “`a`” is a type name or a variable name. Since the declaration of an identifier must occur before the use of an identifier we must have the complete compilation unit and symbol table for this ambiguity to be resolved. Not all symbol table information is needed, only whether the symbol is a variable

or a type. If “`a`” was declared in an include file we would have to process all include files. Include files can be translated first so their srcML version are available, and standard include files could be done beforehand.

If we didn’t want to process the include file, or if we have a code fragment where this can’t be done, an assumption can be made that if “`a`” is not in our symbol table then it most probably is a type since it is much more common to declare types rather than variables in include files. This produces the correct answer in many cases and in some applications will be perfectly fine. Note that the context-sensitive editing features that rely upon regular expression matching have this same limitation. Another approach is to allow the user to clear up the ambiguity with some configuration information. Any tools used to transform source code to srcML has the option of explicitly marking this up as a variable or a function declaration, or of leaving it as an ambiguous declaration for further processing.

6. APPLICATIONS OF srcML

Being XML applications, srcML documents are easily searchable and query-able with standard XML tools. These queries can generate different views of the source code where each view helps the developer solve a particular task. Other source code browsers allow definition of views based on structural information of the source code such as inheritance, visibility, calls, etc. While srcML supports all these elements, it adds the possibility of combining both structural and semantic information extracted from the source code and its associated documentation in one view. Extension of srcML to encode external documentation is currently under investigation.

srcML can be easily used to extract and modify information from source code using the DOM, SAX or XSLT. Selection can be done with XPath using names that directly relate to the language elements themselves. It is now simple to construct XPath expressions from a programmer centric view rather than a compiler centric graph view.

The use of XML technologies, such as Xpath, with srcML allows us to use it for a large variety of tasks. One example would be an Xpath expression for srcML to refer to all private members of a class: “`//class//private`”. For viewing and editing we can use this to highlight, show in a different font, or completely hide all private members of a class. For querying we use the result to examine all the private members of any class in the project. For software visualization we can visualize the private members as a different shape or color then non-private members. For static analysis we can measure of the number of private members and compare it to the total number of members of a class. Since srcML provides access to all source code text the Xpath expression can be extended to selectively include private members of class(es) that had a particular name in the comment that directly precedes the class declaration.

Since elements of srcML are stored in the same order/location as the corresponding source code, and the elements are nested in XML as they would be nested in the source code, filter and extraction processing is straightforward. The srcML manipulation tools can use an event-based interface, such as SAX, to the XML document rather than a DOM interface that

requires the storing in memory of the entire document tree. This is very useful for very querying large (sets of) source code files.

7. CONCLUSIONS AND FUTURE WORK

Although srcML relates to research efforts in the standard exchange format community, it proposes a somewhat different approach. We are representing the source code as structured documents. Experiences from the research communities of standard exchange formats, reverse engineering, and document engineering are combined in this proposed format. The srcML document representations can be used as an interface between the developer and the development environment, as well as between analysis tools. The emphasis in srcML is in combining text with both structural and textual information of the source code.

The syntax of the programming language should take two forms: external – easy to understand for the user; and internal – for tool exchange and processing. srcML is our means of internal representation. Through querying, it can generate views in the external representation.

We are currently developing a C++ to srcML translator. It utilizes the ANTLR Translator generator (see www.antlr.org) that generates a top-down LL(k) translator. The design of this translator is based on an island grammars paradigm [13]. srcML markup is added whenever possible to a stream of text and incorrect, incomplete, and low level details are skipped. The translator is stream-oriented to support incremental parsing. Careful choices in the grammar allow us to buffer very little input text before a start tag for an element is produced. This allows an event-driven translator that easily supports a SAX interface.

Tools to help the user specify views, queries, and rules of association between source code elements (e.g., comments and functions) are also in the planning stage. Since srcML stores any partial programs (or pseudo-code) can be represented. This allows us to develop an editor that can generate much of the srcML on the fly. While not all of srcML may be supported in this manner it will facilitate much of the features now seen in advanced source code editors.

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

8. ACKNOWLEDGEMENTS

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

9. REFERENCES

- [1] Antoniol, G., Canfora, G., Casazza, G., and De Lucia, A. Information Retrieval Models for Recovering Traceability Links between Code and Documentation in Proceedings of IEEE International Conference on Software Maintenance (ICSM'00) (San Jose, CA, October 11-14, 2000), 40-51.
- [2] Atkinson, D. C. and Griswold, W. G. The design of whole-program analysis tools in Proceedings of 18th International Conference on Software Engineering (ICSE'96) (Berlin, Germany, March 25-30, 1996), 16-27.
- [3] Badros, G. J. JavaML: A Markup Language for Java Source Code in Proceedings of 9th International World Wide Web Conference (WWW9) (Asterdam, The Netherlands, May 13-15, 2000).
- [4] Boshernitsan, M. and Graham, S. L. Designing an XML-Based Exchange Format for Harmonia in Proceedings of Seventh Working Conference on Reverse Engineering (WCRE'00) (Brisbane, Australia, November 23-25, 2000), 287-289.
- [5] Ebert, J., Kullbach, B., and Winter, A. GraX — An Interchange Format for Reengineering Tools in Proceedings of Sixth Working Conference on Reverse Engineering (WCRE'96) (Atlanta, GA, October 6-8, 1999), 89 - 100.
- [6] Holt, R. C., Winter, A., and Schürr, A. GXL: Toward a Standard Exchange Format in Proceedings of 7th Working Conference on Reverse Engineering (WCRE '00) (Brisbane, Queensland, Australia, November, 23 - 25, 2000), 162-171.
- [7] 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), 289-292.
- [8] Maletic, J. I., Leigh, J., Marcus, A., and Dunlap, G. Visualizing Object Oriented Software in Virtual Reality in Proceedings of International Workshop on Program Comprehension (IWPC01) (Toronto, Canada, May 21-13, 2001), 26-35.
- [9] Maletic, J. I. and Marcus, A. Supporting Program Comprehension Using Semantic and Structural Information in Proceedings of 23rd International Conference on Software Engineering (ICSE 2001) (Toronto, Ontario, Canada, May 12-19, 2001), 103-112.
- [10] Malton, A. J., Cordy, J. R., Cousineau, D., Schneider, K. A., Dean, T. R., and Reynolds, J. Processing Software Source Text in Automated Design Recovery and Transformation in Proceedings of IEEE 9th International Workshop on Program Comprehension (IWPC'01) (Toronto, Canada, May 12-13, 2001), 127-134.
- [11] Mamas, E. and Kontogiannis, C. Towards Portable Source Code Representations using XML in Proceedings of 7th Working Conference on Reverse Engineering (WCRE '00) (Brisbane, Queensland, Australia, November, 23 - 25, 2000), 172-182.
- [12] Marcus, A. and Maletic, J. I. Identification of High-Level Concept Clones in Source Code in Proceedings of Automated Software Engineering (ASE'01) (San Diego, CA, November 26-29, 2001), 107-114.
- [13] Moonen, L. Generating Robust Parsers using Island Grammars in Proceedings of 8th IEEE Working Conference on Reverse Engineering (WCRE'01) (Stuttgart, Germany, October 2-5, 2001), 13-24.
- [14] Munson, E. The Software Concordance: Bringing Hypermedia to Software Development Environments in Proceedings of SBMIDIA '99: V Simposio Brasileiro de Sistemas Multimidia e Hipermidia (Anais. Goiania, Goias, Brasil, June, 1999).

- [15] Sperberg-McQueen, C. M. and Burnard, L. TEI P4: Guidelines for Electronic Text Encoding and Interchange. Text Encoding Initiative Consortium 2002.
- [16] Storey, M.-A. D., Best, C., and Michaud, J. SHriMP Views: An Interactive Environment for Exploring Java Programs in Proceedings of International Workshop on Program Comprehension (IWPC'01) (Toronto, Ontario, Canada, May 12-13, 2001), 111-112.
- [17] 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), 108 - 117.
- [18] W3C XML Information Set Version 1.0. Webpage, Date Accessed: 6/2002, <http://www.w3.org/TR/2001/REC-xml-infoiset-20011024>, 2001.
- [19] Walsh, N. *Docbook: The Definitive Guide*. Reilly & Associates, Inc., 2002.
- [20] Wong, K. The Rigi User's Manual - Version 5.4.4. The Rigi Group,, Date Accessed: 01/20, <http://ftp.rigi.csc.uvic.ca/pub/rigi/doc/rigi-5.4.4-manual.pdf>, 1998.