Source Code Files as Structured Documents

Jonathan I. Maletic, Michael L. Collard, Andrian Marcus
Department of Computer Science
Kent State University
Kent Ohio 44242, USA
jmaletic@cs.kent.edu, collard@cs.kent.edu , amarcus@cs.kent.edu

Abstract

A means to add explicit structure to program source code is presented. XML is used to augment source code with syntactic information from the parse tree. More importantly, comments and formatting are preserved and identified for future use by development environments and program comprehension tools. The focus is to construct a document representation in XML instead of a more traditional data representation of the source code. This type of representation supports a programmer centric view of the source rather than a compiler centric view. Our representation is made relevant with respect to other research on XML representations of parse trees and program code. The highlights of the representation are presented and the use of queries and transformations discussed.

1. Introduction

While program source code is intrinsically structured, the manner in which it is stored has almost no structure at all. That is, source is stored as simple text. While this works quite well for writing code and a little less well for reading code, text is, frankly, a poor medium when it comes to explicitly describing structure. A common solution to this problem, in the field of document engineering, is to add structural information into the text by inserting special characters or tags into the document. The text can then be more easily searched, parsed, and transformed with the aid of these tags. The current standard for marking up documents and information is the Extensible Markup Language (XML), which is being used on a wide variety of document and software engineering problems.

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. 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 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, most obvious 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 excellent representation for many types of tools for both programming (development) and program comprehension. The primary end product of the software engineering process is usually the compile-able source code and its associated documentation. We propose to use srcML as the form for this product. It is compile-able, with simple pre-processing, while it allows the user to view the source code at a more abstract level.

We now discuss our motivation behind this work and survey the related literature on this topic.

2. Why a structured representation?

Why do we need a document-oriented representation? The parser for a typical programming language (e.g., C++, C, Java) generates an Abstract Syntax Tree (AST) and a symbol table. The format and contents of this output are great for the needs of the compiler but greatly lacking with respect to the needs of software engineering. This decidedly compiler centric representation lacks large amounts of important information with respect to comprehension, most obvious are comments. Parsing and preprocessing removes this "non-essential" information. However, to the programmer this information is often semantically very important.

Integrated development environments that support the various software engineering tasks (e.g., maintenance, reverse engineering) require a more powerful

[1] Pronounced, "Source ML".

representation of the source to be more computationally viable. This is the main purpose of the srcML representation.

A number of options currently exist for representing source code information (e.g., AST or ASG) in XML namely, GXL [5], CppML [8], ATerms [9], and Harmonia [3]. 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 [5] is an XML-based exchange format for graph-like structures based on GraX (Graph eXchange format) [4], and RSF (Rigi Standard Format) [10]. 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 an 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 [2], 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 [3] and cppML/JavaML developed at the University of Waterloo [8] 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.

3. Features of srcML

XML can be used as a document representation (e.g., DocBook, XHTML, etc.) or alternatively as a data representation (e.g., SOAP, SMIL, and countless domain specific formats). srcML is an XML application for representing source code as structured documents and as such has both document and data representation characteristics.

As a document representation, srcML preserves the information of the original text. That is, srcML preserves all information present in the original source code (e.g., formatting and spacing); information that is typically not stored in other representations. Elements occur in the same document order as they do in the original document (as typed by the developer). Document formats often encourage the separation of content from view. This typically means ignoring the white space of the document, since that is considered part of the view. In source code, formatting (e.g., the use of white space) is part of the content; not content that the compiler is interested in but content inserted explicitly by the programmer who wrote the source code. A srcML document can be used to generate the original source code it came from or it can be transformed into another srcML document.

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. Many comprehension activities do not require a complete parse tree or AST of the source code. Atkinson [1] 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. Parsing the source to a certain granularity level (e.g., expression) still offers the developer sufficient information to carry out most comprehension 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).

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

1. Representation of multiple levels of granularity within the AST;
2. Multiple level of abstraction (or views);

3. Transformation equality of source to representation and of representation to source;
4. Query-able and search-able representation;
5. Representation of structural information, including macros, templates, and compiler directives (e.g., #include), etc.;
6. Preservation of:
    a) Location of constructs;
    b) Text formatting information;
    c) Comments and their location;
    d) File names and structure.
    e) 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.

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. This is typically a file or module but can represent any piece of code (POC). A *POC* is any set of contiguous lines of source code. 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.

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., `<preproc-stmt>`) 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 POCs. The issue of syntactical correctness is a compiler problem and is not of supreme importance to the representation.

Each statement in the source code, down to the expression level, has its own element and is marked accordingly. The relationship of a language item in the srcML document to its corresponding item in the associated source code file is:

- All language items appear on the same line in the `<unit>` element of the srcML document as they would in the source code file.
- White space between srcML elements is exactly the same as the white space between the language elements in the source code document.

White space in XML includes spaces, tabs, and blank lines. While many XML applications consider white space between elements insignificant and normalize them, they 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.

Preserving the white space is what allows reformatting to the original source and presenting data and source in readable format. Often times the layout of the source code, as intended by the original developer, conveys a great deal of information (e.g., association or relation by physical proximity).

The issue of associating comments with structural elements of the source code is important in analyzing the semantic information embedded into the source code [7]. The association of comments to the program elements they describe can vary from programmer to programmer. Some programmers like to place comments describing function before its implementation, while others at the end of the implementation, or right after the header. This prompted us to design srcML such that it stores the comments without associating them with a particular program element.

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

Sometimes there is a further structure to the comments, as in the case of JavaDoc, and precondition comments. The comments stored in srcML do not extract the content of this structure directly. However, srcML does provide the comment in a convenient form for extraction and querying.

Being XML applications, srcML documents are easily search-able 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.

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. The DTD for srcML will be made available on the web page of the Software Development Laboratory <SDML>, at Kent State Univ. (www.sdml.cs.kent.edu).

5. 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. In general, a program-understanding tool should support the level of granularity of the comprehension task at hand. Representations in srcML can generate views at different granularity levels to support such tools, while these views also support concepts such as literate programming [6].

A widely used approach to program understanding is plan recognition, bottom-up from the source code to a more abstract description. Essentially, this is done by pattern matching on an internal representation of the code, which leads to detecting patterns of higher-level plans or concepts in a lower-level code. Both programmers and tools use this strategy. We strongly believe that representations of the source code such as provided by srcML directly supports this comprehension strategy. We are building tools that rely on the srcML representation and future research will address the issue of support for program understanding more directly. In addition, ways to represent textual and graphical external documentation within the srcML representation are being investigated.

A set of tools for partial parsing is being developed (a C++ to srcML translator). Also, tools to help the user specify views, queries, and rules of association between source code elements (e.g., comments and functions). Since srcML stores any POC, 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.

References

[1] 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, pp. 16-27.

[2] 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.

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

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

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

[6] Knuth, D., "Literate Programming", *The Computer Journal*, vol. 27, no. 2, 1984, pp. 97-111.

[7] 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, pp. 103-112.

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

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

[10] 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.