Addressing Source Code Using srcML

Michael L. Collard
Department of Computer Science
Kent State University
Kent Ohio 44242
collard@cs.kent.edu

Abstract

The plain-text representation of source code is limited in addressing locations in the source code, providing context to locations, and integrating higher-level information such as from source models. The representation, srcML, attempts to solve these problems by providing an enhanced, XML view of the source code while at the same time preserving the current textual view. This paper discusses the problems that the plain-text representation has, and the philosophy behind srcML, which provides a rich addressing language to source code. The intent is to further the discussion of what is needed to enhance textual views of source code for comprehension.

1. Introduction

Source code contains information at a variety of levels. At its most basic it is a sequence of characters arranged in lines and contained in a file. Since it is just a sequence of characters, it has no restrictions. It can (and often does) represent source code at any stage of development including non-compilable code, code fragments, and code intermixed with other information.

The advantages of the textual representation come from its lack of any higher-level knowledge of structure, syntax, etc. This has led to the set of tools that are typically used by developers, e.g., *grep*, *diff*, editors, etc. The standard APIs at this level are based on regular expressions which provide a rich approach to finding patterns in the source code.

However, when it comes to assisting programmer comprehension of source code the textual representation is limited. While programmers may write source code at a textual level, they simultaneously view source code at the structural (e.g., block), syntactic (e.g., statement) and documentary (e.g., comment, white space) levels.

This creates problems for supporting comprehension in three ways: addressing, context, and integration. Addressing is how we specify a location in the source code, context is what surrounds a location in the source code, and integration is how we combine the source code with higher-level information.

Plain text only allows physical addressing, i.e., by line and column number. A physical address does not give any additional information of what is in the document at that location and is only useful to tasks and tools that work at a purely textual level.

The problem of context is related to the problem of addressing: Given a location, how can we determine the useful context? Instead of showing a fixed number of surrounding lines of code it would be more useful to show context based on the syntactic structure, e.g., the current statement or function.

The problem of integration is how to connect information that is at a higher level of abstraction, e.g., source or design models. Plain text doesn't allow the insertion of this easily, except in comments. Also, higher-level information typically requires addressing of the syntactic elements in the source code.

A level of special interest is the documentary structure e.g., comments. These elements are particularly difficult to represent because their structure may crosscut the statements syntactic and structural organization [4]. The contents of comments and their relationship to the syntactic structure is becoming more structured in tools such as Javadoc and Doxygen where the semantics and their semantics is often based on positional relationships to other elements. In addition, there is little experience in representing documentary structure because they are typically removed before compilation.

The rest of this paper will present experiences with srcML, a representation of source code that attempts to solve the basic problem of addressing locations in source code. An overview of the srcML approach will be given including design decisions and how srcML overcomes the limitations of the plain-text representation. Following that are examples of applications using srcML and how these design decisions have affected them. Last there is some discussion of practical issues and a conclusion.

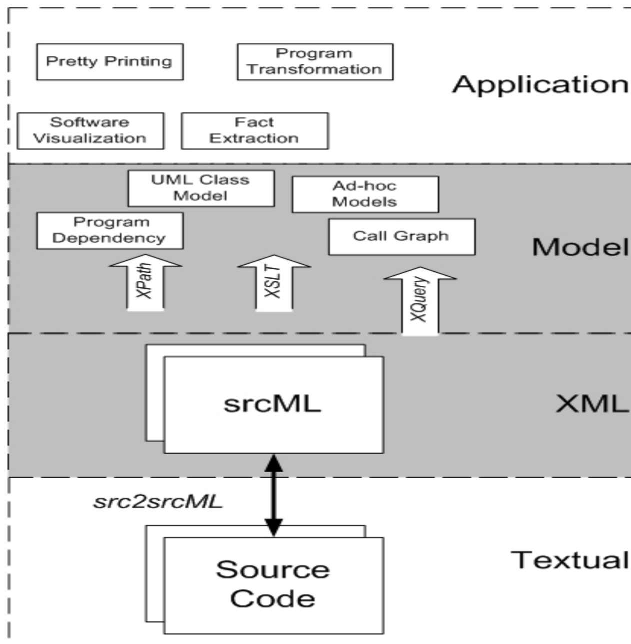

Figure 1. The approach starts with an XML representation, srcML , over the original Textual level. Applications can use Model, XML, and the Textual layers simultaneously.

2. srcML Approach

The approach taken by srcML to the problems of addressing source code is shown in Figure 1. The set of source-code documents is converted to an XML representation, srcML, which allows access to all levels of information in a source-code document, i.e., lexical, structural, syntactic and documentary.

This section will first describe the srcML format. Then the issue of how srcML solves the problems of addressing, context, and integration of higher-level information will be discussed. More information on srcML, including its relation to other XML representations of source code, can be found in the references.

2.1. srcML Design Decisions

The source code representation srcML [1, 2] adds syntactic information as XML elements into the source code text. In the creation of srcML many design decisions were made regarding the relationship of the original source code text to the text in the srcML representation:

- All original text is preserved (including keywords) with any necessary escaping of XML meta-characters.
- The source code text is not stored as attributes but as text inside of element, i.e., between start and end tags.
- The source code is not processed. Preprocessor directives and macros are represented as they appear in the text document.
- Comments and white space are preserved intact where they originally appear.

As a result, the representation provides a direct mapping to the original source code. A source code file can be converted to the srcML representation and back with no loss of textual information. An identity transformation at the srcML level is equivalent to an identity transformation at the textual level.

In srcML, XML elements mark the locations of syntactic elements (e.g., statements, classes, functions, etc.), documentary elements (e.g., comments), and preprocessor directives (e.g., *#include*). The elements surround the text that they describe.

2.2. srcML Addressing of Source Code

With the text of the source code enclosed in XML elements, an XML address at the srcML level becomes an address to the original source code. The XML addressing language XPath can refer to locations in the source-code document. The XPath address of the function with the name *convert* could be found with: *//function[name='convert']*. The XPath address of the first comment inside this function could be found with: *//function[name='convert']//comment[1]*. We can even locate the comment based on its contents.

The previous XPath expression indicates that these comments are in the function *convert*. The XPath expression directly provides context by describing a path to the entity. Once given a location additional queries using XPath can also be used to discover the context.

2.3. Integration of Higher-Level Information

The markup used by srcML purposefully remains at a low level of abstraction. The format does not include information that would require further analysis, e.g., such as type information for a use of a variable. The reasons for this include:

- Representation of code fragments where the necessary type information may not be present.
- Include files that may contain necessary declarations are not inserted because preprocessing is not done.
- Limited effect from non-local changes. If type information was stored, then changing a declaration of a variable could change the markup of every use of that variable.

Actually, there are many types of information from higher-level source models that we may also want to include, e.g., call graphs, dependency graphs, design-level information etc. In addition, there is information from ad-hoc models that we may wish to use. The approach taken by srcML is to store this higher-level information separately at the Model level as in Figure 1. Applications would then use this in conjunction with the XML level. This allows the Model level to include multiple models of different or the same type.

While this approach provides the most flexibility, it does complicate processing. The srcML format was designed to be extensible. The minimal use of attributes allows room for storage of true meta-data (e.g., type information) directly in the format. This is especially the case if the use of srcML is self-contained in the application.

3. Experience with Applications

The ability to address source code using XPath has been applied to several applications. We will briefly discuss how these applications were affected by the design decisions of srcML.

As reported previously at IWPC [1], basic fact extraction can be performed on source code using XML XPath-aware tools. A query of the srcML representation can be directly mapped to a query of the original source code document. In addition, querying can involve anything in the original source code document.

The disadvantages of the approach are due to the lack of access to higher-level information. Queries involving type require a search to find a declaration, which may not be in the same document, e.g., a data member of a class used in a method defined outside the class. This is also true for information from other models. Allowing the query to also access information at the Model level would fix this problem.

As can be expected for a format that so carefully preserves all original text, srcML is very useful for transformations whose result is returned to the programmer, e.g., refactorings. XPath expressions can be used to identify the locations where changes should occur, and to extract needed information from the current document. Once identified, the source code entity can be transformed at the XML or text level. This makes it easy to do changes such as re-indenting a block.

An example of an extension of srcML can be found by looking at the srcDiff format. The srcML format was extended to represent multiple versions of a source-code document in the format srcDiff [3]. From a srcDiff document the entire contents of either the original or modified version of a document can be extracted. More importantly, fact extraction on srcML was extended to allows fact extraction on syntactic differences.

The creation of the srcDiff format is computed efficiently by using the combined textual and syntactic view that srcML provides. The original and modified srcML of a document are interleaved using information from a textual difference (i.e., output of *diff*) of the original text documents. Additional XML elements are used to mark sections of the document that belong to the old, new and both versions of the document. The interleaving is a linear process, allowing for the efficient creation of the srcDiff format.

4. Practical Issues

While XML representations can be quite large as compared to the original text format, in practice srcML is of reasonable sizes with srcML. For example the source code in the Linux kernel (2.6.10) takes 161M to store in text and 536M in the srcML, giving a ratio of text to srcML of less than 3.5. Interestingly, when compressed using the utility *gzip* the text takes 38M and the srcML 54M, giving a ratio of compressed text to compressed srcML of less than 1.5.

A translator from source code to srcML, src2srcml [1] exists which takes a top-down parsing approach. Much work has been done on increasing the robustness and speed of the translator. Currently the srcML translator can translate at a speed of 11,000 lines per second.

5. Conclusions

The srcML format was designed to allow the straightforward access of information in source-code documents. It provides a way to address and manipulate the actual contents of a source code file using an XML infrastructure in an efficient and flexible way.

6. References

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

[2] Collard, M. L., Maletic, J. I., and Marcus, A., "Supporting Document and Data Views of Source Code", in Proceedings ACM Symposium on Document Engineering (DocEng'02), McLean VA, November 8-9 2002, pp. 34-41.

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

[4] Van De Vanter, M. L., "The Documentary Structure of Source Code", Information and Software Technology, 44, 13, October 1 2002, pp. 767-782.