

An Infrastructure to Support Meta-Differencing and Refactoring of Source Code

Michael L. Collard

*Department of Computer Science
Kent State University
Kent Ohio 44242
330 672 9039
collard@cs.kent.edu*

1 Introduction

The proposed research aims to construct an underlying infrastructure to support (semi) automated construction of refactorings and system wide transformation via a fine grained syntax level differencing approach. We term this differencing approach meta-differencing as it has additional knowledge of the types of entities being differenced.

The general approach is built on top of an XML representation of the source code, specifically srcML¹ [Maletic'02]. This representation explicitly embeds high level syntactic information within the source code in such a way as to not interfere with program development and maintenance. This representation allows us to perform version differencing at any granularity level within the source code (i.e., not just a file or line). Specifically, we can very easily determine what type of syntactical construct or program entity has been modified. More importantly, we can track the morphological changes that occur within the source and construct a transformation to duplicate these changes.

Because both the source code and the difference are represented in XML, the transformational language, XSLT, can be used to model these changes. We propose to develop an environment (development/maintenance) that automatically generates XSLT programs based on changes to a program. The developer can then reapply these transformations to similar maintenance and development tasks (e.g., refactorings).

2 Current Problems and Solutions

Current mechanisms for source code versioning do not easily take advantage of structural and syntactic information in the source code. This hinders the analysis, manipulation, and tool construction of the difference information in a version history to support complex development and maintenance tasks, especially in the area of refactoring.

The reason for these limitations is that popular differencing algorithms and tools, i.e., UNIX utilities *diff* and *patch*, take a character-based document view of source code files. Obviously, this is not the programmer-centric view of the source code as originally entered into the file. In the *diff* view all lexical information in the file is preserved, differences are changes to characters and no syntactical structure information is stored.

An alternative is a more compiler-centric or data view of the source code after parsing. In this view all of the syntactical information is stored in an abstract syntax tree (AST) and differences are operations on that tree. In the process lexical information is lost and there is a lack of traceability back to the original source code document. In addition, only source code that can produce an AST can be used and during maintenance this may be problematic.

2.1 Current State of Differencing

For the extraction and application of differences the UNIX utilities *diff* and *patch*, or some variation, are widely used. Their representation is limited to a line level granularity. That is, the difference algorithms find line-based differences, represent the differences as changes between lines, and apply the differences to individual lines in the patched file. Extraction is limited to differences between multiple versions of the same file and the application of a patch is on an entire file.

While *diff* and *patch* do have limitations, there are advantages. The algorithms are relatively efficient and because they perform a character-based comparison are flexible and can be used on any text file. They ignore the underlying syntax of the text and as such are very robust.

The character-based view also allows these utilities to preserve all of the original lexical information in the source code and as such is a key reason they are so widely used. Limitations of Character-Based Differences

However being character-based these tools are often at odds with the syntactic structure of the source code crosscutting the structure that the developer understands and hindering the analysis, manipulation, and tool construction based on source code differences. During

¹ Pronounced "source M L"

the extraction of differences this character-based view hinders the support of more abstract extraction mechanisms. Extraction of syntactic level elements is particularly difficult. For example, a developer may be interested in extracting all the changes to a specific function or only comment changes.

Also problematic in the character-based view is such things as expressing the difference between two versions in a form that a developer easily understands, e.g., a name change to a function. Determining the characteristics of a set of changes and placing them into categories is also difficult. This is useful when changes only occurred to white space and comments so only documentary changes were made and there were no changes to the interface. Searching the differences for particular code patterns and queries on the differences themselves, e.g., detection of the version in which a particular function last changed.

With regard to patching, the character-based view inhibits applying differences to individual syntactic elements in the source code, e.g., patch only comment changes. Also, applying only parts of a patch based on the category that the patch is in, or based on the location where it is going to be applied, e.g., allow only documentary changes and changes a specific function.

It is difficult to generate a series of patches from a large patch based on categorization or location, e.g., a large patch for an API change is split into a series of patches where each patch only changes small groups of functions at a time. And making a patch less dependent on the original source file so that a series of patches can be applied out-of-order, e.g., apply the second patch before the first patch.

3 Proposed Approach

If the character-based differencing view is replaced by a syntax-based view, this will allow direct support of refactoring tasks such as:

- **Separation of Transformation and Location of a Refactoring Step** Splitting the “what was changed” (the transformation) from the “where was it changed” (the location)
- **Generalization of Refactorings** Taking a series of refactoring steps that applies to an individual syntactic element and generalizing it so that it can be applied to other syntactic elements of the same type
- **Refactoring by Example** Extraction of a refactoring technique using analysis differences from a specific refactoring example.

In the approach taken here we continue to use the current utilities to perform the extraction of line differences. The line differences are translated into a

representation that understands the syntax of source code language (in particular C++). This combines many advantages of the low-level document view, i.e., efficiency and robustness, with a source code representation that supports both document and data views of source code.

The source code representation that we propose to use is srcML, an XML application that is used to add explicit structural information to raw source code text files. It can represent source code at all stages of development and evolution, e.g., non-compiling code and code fragments, and matches the robustness of the diff utilities when it comes to C++ source code.

As a result of using an XML representation, locations in the source code can be referenced using XPath, the XML language for addressing inside XML documents. This can be used both to extract particular source code elements [Collard'03], and to use as links to them (work submitted to WCRE'03). Work remains with regard to using XPath for this application and in particular we must determine a canonical form for XPath expressions for the purpose of representing locations in source code.

The differences are represented in XSLT (extensible StyLesheet Language), a programming language specifically designed for transformations of XML documents. In XSLT programs an XML document is matched against XPath expressions of templates. This maps to the requirements for representing a difference with the XPath of the template containing the address of the syntax element that is being changed, and the content of the template containing the changed code.

Instead of creating an application for a specific task, we will leverage the utility of existing XML tools to create an infrastructure for support of difference analysis and tools. In this way we can opportunistically combine the tools and technologies of XML.

The main task that we intend to address is the extraction of refactorings from a version history. This would allow the generation of a refactoring from an example. Then we intend to investigate how the refactoring can be generalized.

4 Related Work

In [Magnusson'93] fine-grained version control was used in a collaborative software environment. Individual elements inside source code documents were selected for version control. The elements used were set for the entire project. In the Fluid project [Wagner'97] a fine-grained version control infrastructure for trees, such as AST's is provided. Both of these cases are complete source code management systems and are not directly compatible with other versioning systems.

A number of options currently exist for representing source code information (e.g., AST or ASG) in a XML

data format namely, GXL [Holt'00], CppML [Mammas'00], ATerms [van den Brand'98], GCC-XML, and Harmonia [Boshernitsan'00]. 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. These XML data views of source code, since they are based on the AST, are a “heavyweight” format that requires complete parsing of the original document and generation of the complete AST. In JavaML and GCC-XML the AST is mapped to the nested structure of XML. In GXL 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 [Holt'00] is an XML-based exchange format for graph-like structures based on GraX (Graph eXchange format) [Ebert'99], and RSF (Rigi Standard Format) [Wong'98]. 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 [Badros'00], 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. In the same realm, the Harmonia framework [Boshernitsan'00] and cppML/JavaML developed at the University of Waterloo [Mammas'00] 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.

Other work on source code interchange formats includes the work by Malton et al [Malton'01]. 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.

5 Current Status

Our current work involves the development of the source code representation srcML. In addition to the development of the format itself a C++ source code translator has been developed. The fitness of the format and the translator has been shown by their application as a source code fact extractor and for the representation of source models, such as a call graph. In both cases srcML is used in a framework that opportunistically combined XML tools and technologies.

5.1 Source Code Representation in srcML

srcML (SouRce Code Markup Language) [Collard'03, Collard'02, Maletic'02] is an XML application that supports both document and data views of source code. The format adds structural information raw source code files. The document view of source code is supported by the preservation of all original lexical information including comments, white space, preprocessor directives, etc. from the original source code file. This permits transformation equality between the representation in srcML and the related source code document.

A lightweight data view of source code is supported by the addition of XML elements to represent syntactic structures such as functions, classes, statements, and entire expressions. Other structural information including macros, templates, and compiler directives (e.g., #include), are also represented. The data view stops at the expression level with only function calls and identifier names marked inside of expressions thereby allowing reasonable srcML file sizes.

The data view allows for a search-able and query-able representation. This can be mixed with a document view to permit multiple levels of abstraction (or views), and allows a data view of the document without losing any of the document information. The reverse is also allowed with document information, e.g., white space, comments, etc., used in the data view for searches or queries.

A srcML document can represent source code at any stage of the development or maintenance process. It has been designed to represent code fragments, non-compiling source code, and source code with missing associated files, i.e., include files. A C++ source code to srcML translator has been develop and takes advantage of the flexibility of the srcML representation. In addition, it is able to translate incomplete and non-compiling source.

5.2 srcML Based Fact Extraction

A lightweight fact extractor was created that utilizes XML tools, such as XPath and XSLT, to extract static

information from C++ source code programs. The source code was first converted into an XML representation, srcML, to facilitate the use of a wide variety of XML tools. The method is deemed lightweight because only a partial parsing of the source is done. Additionally, the technique is quite robust and can be applied to incomplete and non-compile-able source code. The trade off to this approach is that queries on some low level details cannot be directly addressed. This approach is applied to a fact extractor benchmark as comparison with other; abet heavier weight, fact extractors.

In detail the work showed that a lightweight and robust source code representation could be queried for fact extraction using XPath. The translator used a multi-stage approach to allowing for further refinement of the analysis.

5.3 Source Model Representation

The srcML format was used to further extend the lightweight fact extraction concept, along with the use of XML technologies, to represent, manipulate, and navigate links (relationships) within the source code. The links were represented using the XML Linking Language (XLink). The combination, or blurring, of the source code and the source model as an XML format was leveraged via the API's, tools, and technologies of XML. These tools were then used to query the source code and call graph. The call graph was stored in a linkbase, a separate XML document consisting of XLinks allowing the representation and use of multiple source models on the same set of source code documents.

The representation allows for the integration of the call graph information with the source code itself. In effect, we actually allowed one to imbed any type of meta-information into the source (not just call graphs). The types of source code elements that can be linked included high-level entities such as functions, classes, namespaces, and templates, as well as middle-level entities such as individual statements (e.g., if, while, etc.), declarations, and expressions. Additionally, it allowed the linking of entities that are typically discarded during pre-processing such as comments, pre-processor directives, and macros. In detail we showed how XPath could be used for the location of a function in the source code and used in queries integrating both the source code and the call graph.

6 References

- [Badros'00] Badros, G. J., (2000), "JavaML: A Markup Language for Java Source Code", in Proceedings of 9th International World Wide Web Conference (WWW9), Amsterdam, The Netherlands, May 13-15.
- [Boshernitsan'00] Boshernitsan, M. and Graham, S. L., (2000), "Designing an XML-Based Exchange Format for Harmonia", in Proceedings of Seventh Working Conference on Reverse Engineering (WCRE'00), Nov. 23-25, pp. 287-289.
- [Collard'03] Collard, M. L., Kagdi, H. H., and Maletic, J. I., (2003), "An XML-Based Lightweight C++ Fact Extractor", in Proceedings of IEEE Workshop on Program Comprehension (IWPC'03), Portland, OR, May 10-11, pp. 134-143.
- [Collard'02] Collard, M. L., Maletic, J. I., and Marcus, A., (2002), "Supporting Document and Data Views of Source Code", in Proceedings of ACM Symposium on Document Engineering (DocEng'02), McLean VA, Nov. 8-9, pp. 34-41.
- [Ebert'99] Ebert, J., Kullbach, B., and Winter, A., (1999), "GraX — An Interchange Format for Reengineering Tools", in Proceedings of Sixth Working Conference on Reverse Engineering (WCRE'96), Atlanta, GA, Oct. 6-8, pp. 89 - 100.
- [Holt'00] Holt, R. C., Winter, A., and Schürr, A., (2000), "GXL: Toward a Standard Exchange Format", in Proceedings of 7th Working Conference on Reverse Engineering (WCRE '00), Brisbane, Australia, Nov., 23 - 25, pp. 162-171.
- [Magnusson'93] Magnusson, B., Askund, U., and Minor, S., (1993), "Fine-grained revision control for collaborative software development", in Proceedings of ACM Symposium on Foundations of Software Engineering, pp. 33-41.
- [Maletic'02] Maletic, J. I., Collard, M. L., and Marcus, A., (2002), "Source Code Files as Structured Documents", in Proceedings of 10th IEEE International Workshop on Program Comprehension (IWPC'02), Paris, June 27-29, pp. 289-292.
- [Malton'01] Malton, A. J., Cordy, J. R., Cousineau, D., Schneider, K. A., Dean, T. R., and Reynolds, J., (2001), "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, pp. 127-134.
- [Mammas'00] Mammas, E. and Kontogiannis, C., (2000), "Towards Portable Source Code Representations using XML", in Proceedings of 7th Working Conference on Reverse Engineering (WCRE '00), Brisbane, Queensland, Australia, November, 23 - 25, pp. 172-182.
- [van den Brand'98] van den Brand, M., Sellink, A., and Verhoef, C., (1998), "Current Parsing Techniques in Software Renovation Considered Harmful", in Proceedings of 6th International Workshop on Program Comprehension (IWPC'98), Ischia, Italy, June 24-26, pp. 108 - 117.
- [Wagner'97] Wagner, T. A. and Graham, S. L., (1997), "Incremental Analysis of Real Programming Languages", in Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 31-43.
- [Wong'98] Wong, K., (1998), "The Rigi User's Manual - Version 5.4.4." The Rigi Group, Date Accessed: 01/20, <http://ftp.rigi.esc.uvic.ca/pub/rigi/doc/rigi-5.4.4-manual.pdf>.