META-DIFFERENCING: AN INFRASTRUCTURE FOR SOURCE CODE DIFFERENCE ANALYSIS

A dissertation submitted

to Kent State University in partial

fulfillment of the requirements for the

degree of Doctor of Philosophy

by

Michael L. Collard

August, 2004

Dissertation written by

Michael L. Collard

M.Sc., Kent State University, USA, 1993

B.Sc., Kent State University, USA, 1984

Approved by

_____ , Chair, Doctoral Dissertation Committee

_____ , Members, Doctoral Dissertation Committee

_____

_____

_____

Accepted by

_____ , Chair, Department of Computer Science

_____ , Dean, College of Arts and Sciences

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I am very pleased with the quality and experience of my Dissertation Committee. I want to thank my committee members Dr. Javed Khan, Professor Paul Wang, and Professor Gregory Shreve for their time and effort.

I would also like to thank some of my peers. Important basic parts of the presented infrastructure were the result of collaboration with Huzefa Kagdi, who also provided feedback on the rest of my work. I follow in the footsteps of Andrian Marcus, who set an example of what it means to be a doctoral student. Dale Haverstock was always available to listen to my ideas and offer perceptive comments. Walter Pechenuk graciously proofread this entire document and offered useful writing advice.

I would especially like to thank my advisor and mentor, Jonathan Maletic. Without his support I would not have made this achievement. He not only taught me what research is, he turned me into a researcher. He even taught me a little about teaching. Even when I thought he was wrong, he wasn't. I can't imagine a better advisor.

Michael L. Collard

July 2004, Kent, Ohio

DEDICATION

To my wife, Joan.

***

To my parents, Michael and Mary Ann Collard.

***

To Harold, Pancho and Charlie.

CHAPTER 1

Introduction

This dissertation proposes, realizes, and validates a novel approach for the representation and analysis of differences between source-code documents. The approach is termed *meta-differencing* as it supports the derivation of high-level facts about differences from differences. By using meta-differencing we can mechanically answer questions such as "What types of changes were made to a program?" or "Was a specific program entity changed?". We can also answer questions about the context of those changes. Importantly, it allows these types of questions to be answered from a syntactic, structural, or documentary perspective. This is almost impossible to do with current methods and is typically done by manual inspection. Additionally, the answers to these questions are in a form suitable for further processing of the source code and the differences. This research directly supports engineers in a wide range of software-evolution tasks such as change management, analysis of version histories, identification of possible errors, refactoring, impact analysis, etc.

1.1 Problem Description and Motivation

Current mechanisms for source code versioning do not easily take advantage of structural and syntactic information in the source code. This hinders the analysis and manipulation of difference information in a version history to support complex

development and maintenance tasks. The reason for these limitations is that popular differencing algorithms and tools, e.g., UNIX utilities *diff* and *patch*, take a character-based document view of source code files. Unfortunately, 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, but no syntactical structure information is explicitly stored or used.

An alternative approach is a more compiler-centric or data view of the source code (i.e., post parsing). In this view, all syntactical information is stored in an Abstract Syntax Tree[1] (AST) and differences are operations on that tree (or graph). In this process lexical information is lost and results in a complete lack of traceability back to the original source-code document. In addition, only source code that can be compiled is usable, which may be problematic during maintenance and evolution.

The research presented here develops a fine-grained syntactic-level differencing approach. This approach directly supports the analysis of source-code differences. We term this meta-differencing, as additional knowledge of the differences can be automatically derived via simple queries. Meta-differencing allows software engineers to ask complex questions about the differences between two versions of software. For example, one can automatically determine that an *if* statement was added in a given software change/update. Specifically, by using this approach we can easily determine

---

[1] More often it is actually stored as a graph (ASG)

what type of syntactic construct or program entity that was modified. More importantly, we can track the morphological changes that occur within the source.

In order to realize meta-differencing, a sophisticated underlying infrastructure is necessary. As such, the approach is built on top of an XML representation of the source code developed to support program-analysis tasks, namely srcML[2] [Collard, Kagdi, Maletic 2003; Collard, Maletic, Marcus 2002; Maletic, Collard, Marcus 2002]. This representation explicitly embeds high-level syntactic information within the source code in such a way that it does not interfere with the textual/documentary context of the source code. The representation is unique in that it preserves the programmer's view of the source code while at the same time explicitly adding parts of the abstract syntax to the source. srcML directly supports such tasks as lightweight fact extraction, source-code transformations, and the integration of source models (e.g., call graphs) using XML tools and standards. It also supports the embedding of meta-information into the source (e.g., hyperlinks).

Meta-differencing is implemented using both *diff* and srcML; integrating the line difference information into a srcML-based format. This combines the efficiency and robustness of the character-based approach with a source-code representation that supports both document and data views of source code. This syntactic view of the source code and the differences can then be opportunistically combined with XML tools to support program analysis and development tasks.

---

[2] Pronounced "source M L"

The process starts at the textual layer with the source code and the differences between them generated by the *diff* utility. This is the standard view of the source code and version changes used in popular versioning systems. The textual level of the differences is then raised to a syntactic level just as it was done for the source code. A syntactical XML representation of the source-code differences combines the srcML versions of the original and modified documents with the textual differences between them. This new format, called *srcDiff*[3], includes both versions of the document and the required difference information.

The addition of version information allows source-code queries on srcML to be extended to queries on the differences between source-code documents. Information can be extracted that summarizes the number or types of changes, e.g., how many program elements were deleted, added, or replaced and what types of program elements are changed. Also supported are queries involving the location of changes with respect to program elements, i.e., the number of changes located in a particular program element (e.g., method).

Validation of the meta-differencing approach is presented as a two-step process. First, a case study is presented that validates the robustness and accuracy of the presented source-code representation (srcML), as well as the approach of using common XML tools to perform queries on and conduct fact extraction on source code. This study is done using an established fact-extraction benchmark and allows comparison to other

---

[3] Pronounced "source diff"

approaches. This case study is further extended by demonstrations of using the srcML format and XML tools to perform non-intrusive source-code transformations including refactoring. Second, a case study is presented that validates the creation of the source-difference format (srcDiff) with the approach of performing meta-differencing by applying common XML tools to the source-difference format. A well-known open-source system and its associated version history are the basis for this study. This case study is further extended by demonstrations of difference transformations.

Because the source-difference format (srcDiff) is an extension of the source-code format (srcML), the combination of these two case studies, and the further work on transformations, validates the overall meta-differencing approach.

1.2 Research Hypothesis and Questions

The general hypothesis of this research is that in order to realize meta-differencing we need a holistic view and representation of source code that allows interoperability with various methods and tools. That is, we must use an underlying (new) representation that allows us to easily develop more efficient methods. This is the classic trade-off between representation complexity and algorithmic complexity found in many computer-science problems. The approach taken here is to spend effort up front on the representation to ease the higher-level algorithmic tasks later in the process. As such the decision has been made to utilize XML and opportunistically leverage the mature set of XML tools and technologies to perform meta-differencing of source code in a flexible, transparent, and reasonably efficient manner.

The basic question that this research addresses is: "Can we extract and represent differences between source-code documents in a reasonably efficient manner in order to support automated analysis of these differences?" Numerous ancillary issues arose while conducting this work. Some are answered within this dissertation, while others are left for future work. General questions include the following:

- How closely related is difference analysis to source-code analysis?
- What are the categories or types of differences of interest?
- Can meta-differencing be used as a general infrastructure for supporting a broad range of software-evolution tasks?

Specific questions that address the usability and interoperability of this approach include the following:

- How can an XML infrastructure be effectively leveraged to support the analysis, querying, and transformation of source code?
- What are the characteristics of a format that takes advantage of this existing infrastructure?

1.3 Research Contributions

The results of this research differ in several ways from related approaches. Meta-differencing allows querying that is not possible with either textual or semantic differencing. It extends the limited work in syntactic differencing to a broader range of software-engineering tasks. It provides for a representation of these differences that opens research possibilities into the further analysis and transformation of differences.

Most importantly, this work presents a novel approach to the representation of source code and source-code differences.

Additionally, other specific contributions of this work include:

- Development and definition of a document-oriented XML representation for source code, namely srcML, which allows the full use of XML technologies to be robustly and transparently applied to realistic (production-quality) source code.
- Extension and major improvement of the robustness and speed of the translator from source code into srcML.
- Definition of an expressive addressing language for source code.
- Demonstration of how an addressing language for source code can be extended to provide source-code querying.
- Demonstration of how non-intrusive source-code transformations can be performed using common XML transformation languages.
- Performance of a case study that applied srcML and common XML tools to a popular source-code fact-extraction benchmark that resulted in very reasonable results compared to heavier-weight, full-parser based approaches.
- Definition and development of srcDiff, an XML representation of multiple versions of a source-code document, and creation of a method and implementation that created srcDiff from source-code and line differences with a complexity no greater than that of line differencing.

- Performance of a case study that applied meta-differencing to a complete, medium-sized open-source application and answered practical questions about changes to that system.

1.4 Organization of the Dissertation

The next chapter presents a detailed overview of the approach and discusses other methods for differencing. Afterwards CHAPTER 3 will provide a background discussion of XML and XML technologies. This chapter is included to make the dissertation more self-contained and may be skipped by the reader familiar with this topic.

The representation/querying of differences involve representation/querying of source code, and as a result the next chapter, CHAPTER 4, deals specifically with XML representations of source code. After discussing the general issues of these representations, it then presents srcML, the source code representation that the meta-difference format is based on. Following that is CHAPTER 5, in which full details of the srcML representation, including an explanation of all elements, are presented. In CHAPTER 6 the applicability of srcML to the general problems of addressing, querying, transformation, refactoring, and integration of higher-level abstractions into source code are presented. The chapter concludes with a discussion of other researchers' use of srcML.

Meta-differencing is, in some sense, fact extraction on a difference format, and the difference format is an extension of srcML. As such, CHAPTER 7 presents the results of a case study of leveraging standard XML tools with srcML to solve problems in

a standard C++ fact-extraction (i.e., querying) benchmark. It demonstrates that this approach compares very favorably to heavier-weight parser-based approaches.

In CHAPTER 8 the source-code difference representation, srcDiff, is presented based on srcML. Included is a discussion of the current work on source differencing, requirements for the srcDiff format, and how the format is generated using srcML versions of the document and the output of the utility *diff*.

CHAPTER 9 presents a case study of applying meta-differencing to a complete application. The study demonstrates that the srcDiff format is practical for a complete real-world application. It also demonstrates that meta-differencing can be used to answer fine-grained differencing questions about source code. After the case study, CHAPTER 10 demonstrates some additional tasks that meta-differencing can support including transformation, pattern detection, and constraints on differences.

The last chapter, CHAPTER 11, presents the main results, discusses the conclusions of the work, and indicates some of the future directions of the research. The appendices present schemas for srcML and srcDiff, and full code for XML transformation examples.

Related work is placed throughout the dissertation in the chapter most closely in context. Related work on XML source code representations is in CHAPTER 4, Section 4.3. Related work on C++ Fact Extraction is in the case study on fact extraction in CHAPTER 7, Subsection 7.1.3. Related work on differencing, including source code and XML, is in CHAPTER 8, Section 8.1.

1.5 Bibliographical Notes

Segments of this dissertation are extended versions of previously published papers. Portions of the chapters on the srcML representation in CHAPTER 4 and 5 are based on and reflected in the work presented in [Collard, Maletic, Marcus 2002; Maletic, Collard, Marcus 2002], and [Maletic, Collard, Kagdi 2004]. The fact-extraction benchmark in CHAPTER 6 is adapted and extended from [Collard, Kagdi, Maletic 2003]. ChaptersCHAPTER 8, 9 and 10 on meta-differencing including the meta-differencing case study is greatly extended from [Collard 2003] and [Maletic, Collard 2004].

CHAPTER 2

Overview of Approach

This chapter presents an overview of the approach used for realizing meta-differencing. First the current state of differencing is examined to determine the requirements for meta-differencing. This also sets the stage for the current difficulties in achieving difference analysis. Then an overall approach is given that meets these requirements.

2.1 Current State of Differencing

Differencing mainly falls into the two categories of textual and semantic differencing. Textual differencing is a relatively efficient, commonly used approach that takes a character-based view of text files. In textual differencing two files are different if they have different characters at a given point. Semantic differencing finds differences between the meaning (behavior) of a program and takes an AST (Abstract Syntax Tree) view of the source code. In semantic differencing two programs are different if they have different semantic meaning to the statements.

The rest of this section will look at the specific limitations of these approaches as well as other considerations that motivate the need for another approach.

2.1.1 Limitations of Textual Differencing

For the extraction and application of differences the UNIX utilities *diff* and *patch,* or some variation, are widely used. These utilities are limited to a line-level granularity. This means that the difference algorithms find line-based differences, the differences are represented as changes between lines and the differences are applied to individual lines in the patched, i.e., modified, file. Extraction is limited to differences between multiple versions of the same file and the application of a patch is on the entire file.

While *diff* and *patch* do have these limitations, they have many advantages. The algorithms are relatively efficient, and because they perform a character-based comparison, they are flexible and can be used on any text file. They ignore the underlying syntax of the text and as such are very robust in regard to the contents or state of the text. The character-based view also allows these utilities to preserve all of the original lexical information in the source code, which is a key reason why these utilities are so widely used.

However, being character-based, these tools are often at odds with the syntactic structure of the source code, which crosscuts the structure that the developer understands and hinders the analysis, manipulation, and tool construction based on source-code differences. During the extraction of differences, this character-based view prevents the support of more abstract extraction mechanisms, and extraction of syntactic-level elements is particularly difficult.

Also problematic in the character-based view are such things as expressing the difference between two versions in a form that a developer easily understands, e.g., as a

change to a specific program entity. Specifically, the character-based representation of differences prevents:

- **Expression using Syntactic and Documentary Structure.** Expressing the difference between two versions in a form that a developer (without manual inspection) can understand, e.g., a specific function changed, or a header comment has changed.
- **Characterization and Categorization.** Determining the characteristics of a set of changes and placing them into categories, e.g., changes only occurred to white space and comments; therefore, only documentary changes were made and no changes were made to the interface.
- **Searching and Querying.** 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.
- **Generation of Metrics.** Using the information on differences to produce metrics about the system as a whole or for individual syntactic elements, e.g., metric of difference stability for a set of source-code documents or for an individual class (which may be in multiple files).
- **Programmatic Use.** Creating applications that manipulate differences is difficult and relies on regular expressions as the only API (Application Programmer Interface).

With regard to patching, i.e., the application of difference information to form the modified version of the document, the character-based view inhibits the application of

differences to individual syntactic elements in the source code. Specifically, the character-based representation of differences prevents:

- **Patching at the Syntactic-Level.** Applying differences to individual syntactic elements in the source code, e.g., patch only comment changes.
- **Filtering Patches based on Categorization or Location.** 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 to a specific function.
- **Splitting Large Patches into a Series of Patches.** Generating 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.
- **Rearranging the Patch Order.** Making a patch less dependent on the original source file so that a series of patches might be able to be applied out-of-order, e.g., apply the second patch before the first patch if possible.

Other limitations of textual differences concern both the given location and content of a change, i.e., where a change occurred and what changed. Both limitations cause problems for the analysis of source-code changes. For the location of a change, a general-purpose textual difference has no choice but to refer to the physical line location of where a change occurred. First, this gives no context of the change with regard to the syntax, and it can only be utilized in the context of the original document. Second, a line number is not robust in the case of further changes, e.g., the line address of a difference

will change when other differences are made. Third, what is a single syntactic change to the programmer may be represented as two individual line changes (e.g., moving all of the statements inside an *if*-statement block by deleting the starting and ending lines for the block). For the contents of the change we are given the line, and, without knowing the specific context of the change, we are unable to easily determine exactly what was changed. Only by close manual examination of the source code can we determine what syntactic elements in the code were changed.

2.1.2 Limitations of Semantic Differencing

At the opposite end from textual differencing, both in abstraction level and the notion of what a change is, is semantic differencing. Semantic differencing is concerned with changes in the programming-level semantics of an entire program. It ignores lexical, syntactical, and documentary changes and focuses only on changes that may affect the behavior in the execution of a program.

Semantic changes are program wide. A change in one part of a single file can cause semantic changes across the entire project, e.g., the change of the type in a variable declaration changes the type of expressions that use the variable. Because the changes are program wide the differencing must be applied to the complete, entire program.

Semantic differencing represents the entire program in an AST and can be seen as working on the output of the compiler. Lexical, documentary, and unneeded syntactic information is lost due to the abstraction of the AST ("That's why it's called an Abstract Syntax Tree" [Van De Vanter 2002]). These large ASTs (actually graphs) have to be

compared using expensive graph comparisons. Heuristics using a top-down approach are often used for any practical applications.

In addition to issues of practical application to real-world sized projects, the other limitation of semantic differencing is the abstraction level. Programmers often use the abstractions of a programming language to hide type and other information, e.g., *typedef*, *macros* in the C/C++ languages. A simple textual change to a type or to an inheritance hierarchy is not a change that should be seen cascading through the entire program.

While semantic differencing can support some analysis tasks, especially for the calculation of metrics, it is often the wrong approach to differencing from a programmer's perspective.

2.1.3 Realistic State of Source Code

The source code that a programmer deals with is typically a collection of source-code files and associated include files. In the best case the collection forms a complete, compilable system. However, there are practical situations which prevent this:

- During editing and transformation code is often in a non-compilable state. The intermediate steps of the editing and transformation process include many points where this is true. This is especially the case during refactoring, where the intermediate steps produce code that is not compilable.
- Due to poor source management of a project, or changes to the specific libraries used, associated include files may be missing or unavailable for use. This can also occur when changing from one version of a library to another.

- The source code may be in a dialect of the original language which only compiles in a specific compiler. The source code may only compile with an older version of the compiler due to changes in language standards or stricter adherence to those standards.
- During editing, transformation, or analysis the particular source code of interest often consists of a code fragment and not a complete program.

These situations reflect a realistic state of source code during the entire software life cycle. This is especially the case for re-engineering tasks such as API changes and for many types of transformations. A source-code view that requires a complete, compilable system is not realistic will have limited uses in actual practice.

2.1.4 Preservation of Programmer's View

When source code is presented to the programmer, the original layout of the program must be preserved. Not doing so will mean the rejection of the system as it has been shown with multiple projects and tools in the real world [Cordy 2003; Van De Vanter 2002]. In [Cordy 2003] examples are given on large projects where any potential changes to the system had to be presented to the programmers in the exact same view of the source code that they were familiar with. If not, the proposed changes were rejected.

In [Van De Vanter 2002] the concept of the *documentary structure* of source code, whose elements include all white space and comments, is presented. It is described as what a programmer places in the source code for the sole purpose of assisting whoever is reading the program. Examples given show that at times white space, such as line

breaks and indentation can be more important than comments and that the notion of a single comment is not well defined.

This documentary structure is often at odds with the linguistic structure of the program. Unfortunately for parse-tree-based approaches, this documentary structure is often not preserved, and if an attempt is made to do so, the documentary structure cannot easily be integrated into the representation. This applies to any tool used during software development that, for any reason, displays or transforms source code.

In contrast to these requirements, software development tools typically take a compiler-oriented approach of representing the source code based on a syntax tree according to the formal linguistic structure of the program. It has been observed that these compiler-oriented approaches are not a good match to the problems that they are trying to solve [Klint 2003; Van De Vanter 2002].

Any differencing approach must preserve documentary structure and must allow it to be a first-class part of the document. A successful analysis and transformation system will allow a combination of changes to any of the text, whether it is part of the formal syntactic structure or not. The textual difference tools described earlier (i.e., *diff* and *patch*) meets these criteria.

2.1.5 Requirements for Meta-Differencing

The current popular use of differencing gives a model for what is required to successfully enhance the analysis of differencing. In order to represent the differences between source-code documents, the following are characteristics that the source-code representation must support:

- **Preservation of all textual information.** Differences between two source-code documents can include any type of textual information and must be able to present the programmer's view as she originally wrote it. An identity difference (i.e., a difference with no changes) must present textually, not just semantically, equivalent versions of the documents.
- **Ability to access all textual information at the same level.** All textual information, be it white space, statements, function names, etc. should be equally accessible. This promotes the equivalence of documentary structure with the syntactic structure of statements themselves.
- **Preservation of document order.** Preservation of the original document order allows straightforward querying and transformation. A change in the document order requires following links even for simple queries and does not reflect the programmer's view of the structure of the source code.
- **Locality of change.** A textual change should only potentially change the elements that contain it. A simple change to the type of a variable should not ripple through the entire program.
- **Ability to handle code fragments and non-compilable code.** The representation should handle source code in any state in order to be fully utilized.
- **Conversion to/from existing textual differencing.** Programmers are familiar with and have used systems (e.g., versioning systems *CVS* [Cederqvist 2004], *Subversion* [Collins-Sussman, Fitzpatrick, Pilato 2004]) based on textual differences. A straightforward conversion to and from textual differencing is

needed so that the representation can be fully integrated with existing versioning systems.

By following these requirements differencing approaches can extend the capabilities of current differencing approaches while simultaneously allowing integration with existing systems.

2.2 The Approach

To realize meta-differencing, an underlying XML representation (srcML) for source code that explicitly embeds syntactic information with the source is utilized. srcML is a synergistic representation that preserves the textual context of the source code while adding the required abstract syntactic context. Additionally, srcML can represent source code at all stages of development and evolution, e.g., non-compilable code and code fragments.

Figure 2.1 presents an overview of how the source code, srcML, and meta-differencing tie together. The bottom textual layer is the source code, and the results of *diff* represented as simple text files. The next layer consists of documents in srcML. The srcML translator used to translate C++ to srcML forms the infrastructure for much of this research. Once the source-code file is represented in srcML we can utilize a wide array of XML tools (e.g., XPath, XQuery, etc.) to integrate higher-level models. At the top layer, XML tools can again be leveraged to construct applications. These applications take the form of fact extractors, pretty printers (reformatting), or other program-analysis tools. These applications work on both the srcML and the higher-level source models.

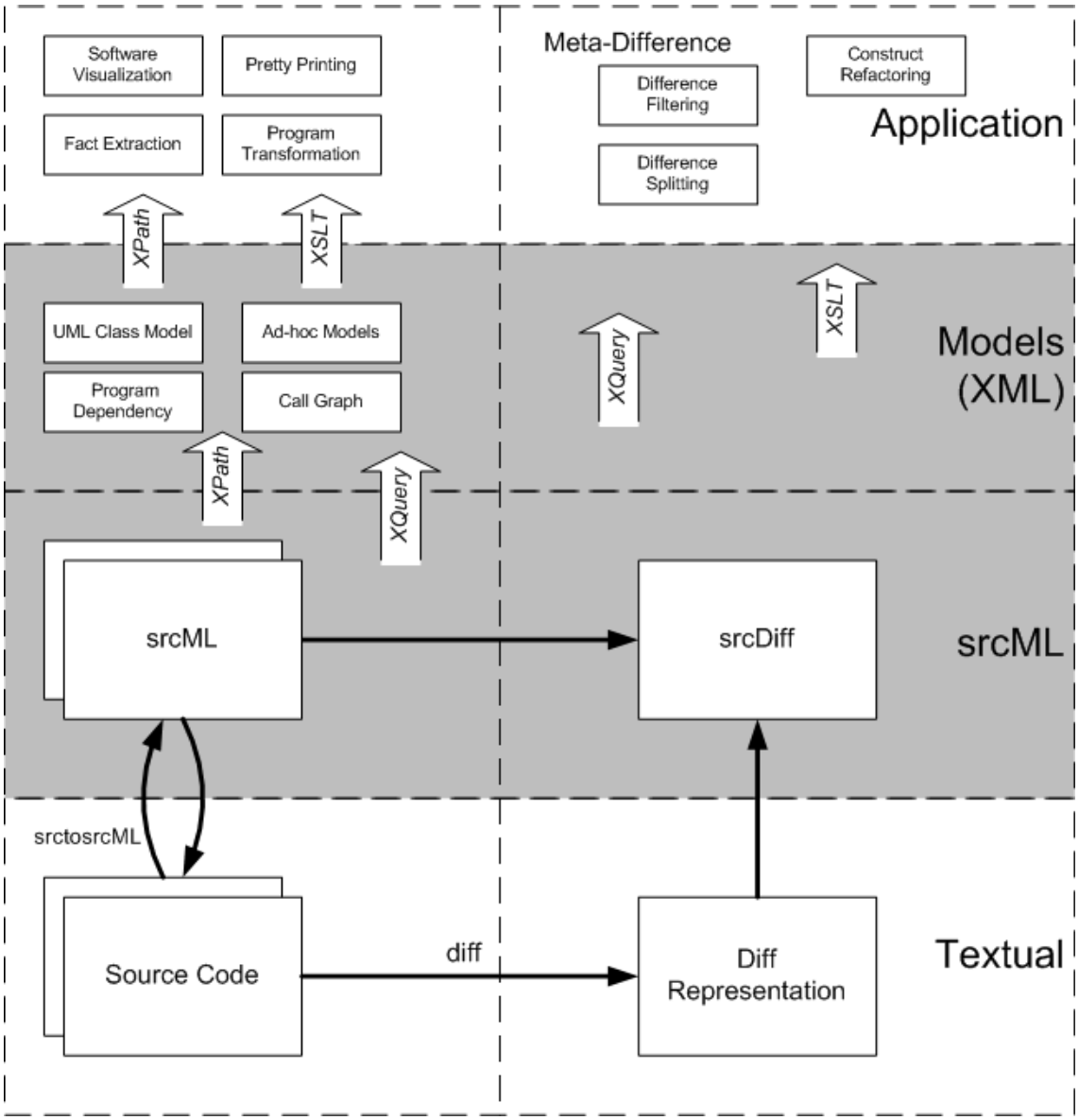


Figure 2.1. The meta-differencing architecture is based on a srcML view of source code files and XML technologies. The bottom layer represents plain text files. These are translated into XML representations. At that point XML tools can be leveraged to generate source models or carry out fact extraction. The Application layer uses the source models as input to other program analysis tasks. The srcDiff is generated from a combination of srcML and the results from the diff utility. Analysis of two different versions of source can then be done.

Meta-differencing is implemented using a combination of the utility *diff* and srcML by translating line differences into a srcML-based format. This combines many advantages (i.e., efficiency and robustness) of the character-based approach with a representation that supports both document and data views of source code.

This syntactic view of the source code and the differences, with higher-level source models, can then be opportunistically combined with XML tools to support analysis and development tasks.

2.2.1 Representation of Difference Information

The difference representation is stored using an *intensional* [Mens 2002] format. Intensional formats store the differences, with the original and modified source code, in a single document. The source code from both versions is stored in srcML, and the areas of changes are marked using specific difference elements. This is similar to way that the version control system *SCCS* stores multiple versions of documents.

By combining both versions of the source code and the differences into a related document querying and transformation of differences becomes an extension of the querying and transformation of source code. Unlike in extensional formats, where the differences are stored separately, this allows both the original and modified document to be viewed, queried, and transformed with the preservation of the document ordering of both versions.

2.2.2 Extraction of Difference Information

The process starts at the textual layer with the source code and the differences between them generated by *diff*. This is the view of the source code and the version changes as used in popular versioning systems (i.e., *CVS*). We must now raise the textual level of the differences to a syntactic level just as we did for the source code.

The difference format is constructed from srcML versions of the source code. These versions are marked with changes to the textual items from the output of the utility *diff* performed on the original source-code files. Once marked the two srcML documents are interleaved into a single document that contains the complete contents of both the original and modified documents. The resulting srcDiff document is a well-formed XML document, with the source code represented in srcML, and the location of differences marked with difference elements. The document is formed with a complexity no greater than that of the line difference algorithm used.

2.2.3 Leveraging XML Technologies

As a result of using an XML representation, locations in the source code can be referenced by using XPath [W3C 1999a], the XML language for addressing locations inside XML documents. These addresses can be used to extract particular source-code elements [Collard, Kagdi, Maletic 2003] and to form links to them. Once in a syntactic-level XML format, standard XML tools, APIs and programming languages (e.g., DOM, SAX, XPath, XSLT) can be used to locate, query, and transform a combination of the source code and the differences. This allows for a wide variety of analysis, transformation, and practical development tools to be constructed on this base.

The next chapter presents background information on XML and the associated tools and technologies with respect to how it is used for this research.

CHAPTER 3

XML Background

The approach and the research presented herewith to a great extent leverages XML for software-engineering tasks in a manner different from that of other approaches. The XML infrastructure is a collection of related standards and technologies that work together to support the format. It is necessary to understand what the XML infrastructure provides in order to understand the presented work.

This chapter provides background information on the parts of XML specifically used or compared to in this research. The chapter presents a full range of XML including the origins and basic requirements, addressing of XML documents using XPath, APIs for processing, and a discussion of transformation and schema languages. The chapter concludes with a general discussion on the requirements for leveraging the entire group of XML technologies. The reader familiar with XML may skip this chapter, or refer to it when necessary without loss of continuity.

3.1 Introduction to XML

The eXtensible Markup Language (XML) [W3C 2000b] is a standard for marking text documents. It is based on SGML (Standard Generalized Markup Language) [ISO 1986], and like SGML, it is a *meta* language that allows the creation of markup languages for specific applications. By stating requirements for the textual data of a document and

of its markup, XML is able to provide a starting point for other standards, tools, and technologies.

A document in XML consists of text data with text markup surrounding the data. The markup forms elements that define a segment in the document. The element is typically delimited in the document by a start tag and an end tag. For example, the following forms the element *item*: *<item>.....</item>*. Elements may contain text, other elements, a combination of text and other elements, or nothing at all. An empty element, i.e., an element with no content between the tags, may also be delimited by a single tag of the form: *<item/>*.

Besides marking segments of the document, elements may also contain attributes inside the start tag as a set of name-value pairs. For example, the following forms the element *item* with the attribute *name,* and the value of the attribute is "value":

<item name="value"> .....</item>.

Attributes have specific limitations. Each attribute name in an element is unique. Any white space in the attribute value may be normalized, i.e., starting and ending white space removed, multiple white space reduced to a single white-space character, and end-of-line characters removed. The data in an attribute is text only and cannot contain elements.

Character data (i.e., text not in a tag) has certain restrictions to allow for distinction between it and markup. First the ampersand ('&') and left angle bracket ('<') characters must be escaped. This is typically done using the entity references *&* and *<* respectively. In addition right angle-bracket characters ('>') may also be escaped

and this is typically done by using the entity reference *>*. Certain other character may not occur at all (even as an entity reference) including the form feed character (i.e., **).

In text documents the end-of-line character is represented using difference byte sequences on different computer platforms, e.g., carriage return and line feed for DOS, line feed for UNIX and carriage return for Mac. In XML documents all of these different line endings are replaced a single line-ending sequence. Once converted into XML, documents that differ only in the end-of-line character from these different platforms are identical.

White space handling in XML can be a confusing issue. The XML standard specifically mentions the preservation of significant white space (i.e., spaces, tabs and blank lines). The standard states that it is common for some document formats to have significant white space with a specific mention of poetry and source code. The special attribute *xml:space* with the value "preserve" may be used to make this preservation more explicit.

3.1.1 Well-Formed and Valid XML

In order to promote interoperability all XML must be *well-formed*. This minimal set of requirements for the elements in the document requires that:

- All elements have a start and end tag (or a single empty element)
- The elements must nest properly, i.e., for a given element the start tag and end tag must be in the content of the same element.

The structure thus formed is that of a tree with the root element forming the root of the tree, and all elements forming subtrees. The rules for well-formed XML do not constrain the particular names of the element and attributes used, or specify rules regarding the specific content of an element or attribute.

Constraints on what elements are in the document and the composition of their contents are the realm of validity. A schema is a set of rules stating these constraints. Validity is measured by whether the XML obeys the constraints of the schema. There may even be a collection of different definitions to promote validity. The purpose of validity is to verify that a document is following the rules of a given XML format.

So for a given XML document all that is required is that the document is well-formed. Whether it is valid is up to a schema that may or may not exist, and may be as flexible or strict as necessary.

3.1.2 XML Format Design

The specific requirements of XML are quite general. It is up to the designer of the XML application (i.e., a specific XML format) to decide on the set of elements, the attributes that these elements have (if any), and how they interrelate to the textual data being marked.

One basic question is what information should be put into elements and what should be put into attributes. The flexibility of elements and the limitations of attributes, e.g., multiplicity of one, can make this decision straightforward. In other cases this is a difficult decision. One guideline often given is that attributes should be reserved for

meta-data, i.e., data about the element data [Harold 2004]. The difficulty may be in deciding what part of the data is meta-data in a given application.

3.1.3 XML as a Document and Data Format

Although XML began as a document format based on SGML (i.e., XML 1.0) [W3C 2000b] it has also developed into a data format. This produces two views of XML, one as a document format, and another as a data format. Popular books on XML are often split into separate sections for narrative-centric documents and data-centric documents [Harold, Means 2001]. Document formats, such as DocBook [Walsh 2002], TEI [Sperberg-McQueen, Burnard 2002], 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 with text strictly structured.

XML document formats are designed for the original generation of content in an XML format. The user edits the text directly in the XML format, or uses a tool that stores the text in an 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 is for export to a non-XML format, or possibly in importing into the XML format. There is little interest in an XML document format to support the viewing or editing of a document that is stored in a 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 document. Written documents are not typically stored in an XML data format, only an abstraction of them.

Document formats are a combination of highly structured format at the organizational level of chapter, section, etc. (e.g., elements *Chapter*, *Section* in DocBook, elements *div*, *body* in TEI) and attributes storing meta-data. In the narrative sections, document formats tend to be semi-structured with markup inserted around the text (e.g., elements *emphasis*, *acronym* in DocBook, element *name* in TEI). These sections often contain mixed content (i.e., XML elements that contain both other XML elements and text). A more familiar example of this is XHTML (an XML version of the HTML web page language) where text is structured into paragraphs with parts of the text marked with style tags (e.g., element *strong*).

Data formats are often fully structured with no mixed content (i.e., XML elements containing either other XML elements or text, but not both simultaneously). There are no narrative sections, but only storage of strictly-structured data.

Another difference between document and data formats is the associated data model, or infoset. White space may be significant in a document format, while in a data format it is typically insignificant. Comments in an XML document are not considered part of the documents character data, and an XML processor does not have to permit the retrieval of comments [W3C 2001a]. However the XSLT model [W3C 1999b], primarily thought of as a document-processing language more than a data-processing language, includes comments. For document-view applications, we might want access to the comments and for a data-view application we might not.

A more general difference is the intent of the format. Document formats mark the text to give it structure or to add additional information, not necessarily to form an abstract model. The complete contents of the text are preserved, and the loss of the markup does not necessarily mean a loss of the original text. On the other hand data formats represent an abstract model of the original information. Loss of the markup loses essential information that is not necessarily recoverable from the text alone.

These separate views of XML effect which parts of the XML technologies are used, e.g., the schema language used. The decision of which direction to lean, document or data, is based on the need to preserve or interact with the format of the original data, as well as the applications used.

3.2 Addressing Locations in XML Documents

The tree structure of XML allows data to be organized in a rich format. In order to link or process internal parts of this tree, an addressing language is needed. XPath [W3C 1999a] is a standard language for addressing parts of an XML document.

An XPath expression is the address of a subset of the XML document. It describes a path to get to this subset. The path may refer to a single item or a group of items. In addition to describing a path into the XML document, XPath expressions can also include predicates and string manipulation. XPath is normally used inside another tool, such as the transformation languages XSLT or STX, or is used with an API to extract parts of the XML document for further processing. XPath is a subset of XQuery, an XML query language.

3.3 XML APIs

XML defines formats that can be used for a wide variety of applications. These applications often require that the XML be processed to produce data in a non-XML format (e.g., calculate metrics on an XML document) or into another XML document (e.g., transform into XHTML). The variety has led to multiple ways for accessing the contents of an XML document in a programming language via an API (Application Programmer's Interface).

One characterization of the APIs for XML is whether access to the XML document is sequential or random. With random access a complete tree representing the XML document is available and the client program is free to randomly move to any position in the tree. With sequential access the client program accesses the XML document as a stream of nodes. The client program cannot access previously seen nodes and cannot jump ahead in the stream without accessing all nodes in between.

Another characterization of the APIs for XML is where the control of the process is located, whether in the API or in the client program. This is described as whether the API is "push" or "pull". With a "push" API the client program declares what processing will occur when a particular node is reached. This event specification is handed over to the API that controls the parsing and calls the appropriate client-program code when the event occurs. With a "pull" API, the client program specifically controls access to the XML document.

The most flexible access to an XML document is the DOM (Document Object Model) [W3C 1998], an API providing tree access to an XML document. Client

programs can use the API for both sequential and random access to the original XML document. The DOM is defined using a generic API with standard bindings to programming languages such as Java, C, C++ and Python.

The random access permitted by the DOM allows for the direct implementation of client programs with no restrictions as to access order of the tree. However, the DOM requires that the entire XML document be parsed and loaded into memory before any access is allowed. The overhead of construction and storage of the DOM tree in memory can be costly with large source-code documents or a large number of source-code documents.

A less flexible but much more efficient API is SAX (Simple API for XML) [SAX 2001], an event-driven "push" API for XML documents. Parsing events, such as element start tags, text, element end tags, are delivered in sequential order for the user program to process. Officially defined for Java, unofficial bindings to other programming languages such as Python do exist.

For XML programming SAX is the most efficient. Only the current node of the tree is stored. The disadvantage is that the client program is responsible for storing any needed information between events. It is also difficult with an event-driven interface to deal with multiple XML documents simultaneously. In addition, the lack of an official standard in any language except Java creates portability concerns with other programming languages.

Another alternative to the DOM is the TextReader API [Veillard], which supports "pull" access to an XML document. Its stream access to an XML document makes it

straightforward to navigate through multiple XML documents simultaneously. Like SAX, it does not store the entire document in memory at one time, which makes it more efficient for large documents. In a TextReader application, the client program explicitly "reads" the nodes in the XML tree one at a time. At each node the API provides full access to the XML node at that point, e.g., starting tag, attribute, etc. It also allows for mixing tree and XPath operations into the streaming interface. The TextReader API does not have an official standard however much information is available. The API supported by libxml2 [Veillard] provides both C and Python bindings and is based on the *XmlTextReader* and *XmlReader* classes of the C# language.

3.4 XML Transformation Languages

The previous section described some of the XML APIs that allow access to an XML document from a traditional programming language (e.g., C++, Java). However, XML transformation may involve the addition of many XML elements generated by the client program. If so, these client programs are often composed of a large number of XML output statements surrounded by small amounts of logic. Another approach is to mix the program logic into the XML elements. One way to do this is to insert logic as XML elements around the XML elements of the output. This approach is taken by some of the programming languages specifically designed for transforming XML. Programs written in these languages are often themselves written in XML.

The most well-known is XSLT (eXtensible StyLesheeT language) [W3C 1999b], a programming language specifically designed for XML transformation. An extension of XPath is used to match and process parts of the XML document tree. A program

requiring random (versus sequential) access to the XML document can be written in XSLT. This provides more support than using a general-purpose programming language with the DOM. However, the program has many of the same memory and time requirements of the DOM since XSLT processors construct a DOM-like tree internally.

The XSLT 1.0 standard is at the level of W3C Recommendation and is supported by processors including *Xalan* [Apache], *Saxon* [Kay 2004], and *xsltproc* [Veillard]. The XSLT 2.0 [W3C 2003] standard is currently a Working Draft with the only support in the *Saxon* processor.

A less well known XML transformation language is STX (Streaming Transformations for XML) [Cimprich 2002]. It is very similar to XSLT with the difference that STX works directly from a SAX interface. Only a subset of XPath expressions are supported since the entire XML document tree is not stored, and only the direct ancestors can be accessed. A program requiring only sequential access to the XML document can be written in STX. This provides more programming support than using a general-purpose language with SAX. The STX standard is only at the level of Working Draft and implementation support is limited. Current processors include a Java implementation *Joost* [Becker 2004] and a Perl implementation *XML::STX* [Alliance].

3.5 XML Query Languages

It is possible to use an XML API and write a program to process a stream of tree nodes, or directly walk the tree of a XML document extracting the required information for a particular problem. However, the programmer must know the details of the API, and a new program must be written for each new query. As with SQL (Standard Query

Language) for relational databases, specific query languages have been developed for XML. These query languages not only allow a client to use a general-purpose tool to extract information from an XML document, but also to refer to locations in the XML document.

As previously discussed the standard way of addressing inside of XML documents is XPath [W3C 1999a]. XPath has a compact, non-XML syntax that can be used as a path notation in an XML document where the XML is represented as a tree of nodes. Since XPath describes a path to the given location evaluation of an XPath expression is the evaluation of a query on the XML document. This path can be executed by an XPath-aware API or a command-line tool (e.g., the program *xpath* which uses the Perl module *XML:XPath* [Sergeant 2000]) or inside a transformation language, such as XSLT, and it can be used as a query language.

A more complete query language analogous to SQL is the XML Query language XQuery [W3C 2002]. It uses an extension of XPath to address locations in XML documents. Unlike XPath, it provides ways for structuring the returned data with new elements. It allows for many of the same results as those obtained by using XPath in XSLT programs.

3.6 XML Schema Languages

The purpose of an XML schema language is to put further constraints on the contents of an XML document beyond that of being well-formed. There is a variety of ways of defining a schema, some of which are still under development, and the choice among them is still under debate.

The oldest schema language for XML is DTD (Document Type Definition) included in the XML specification [W3C 2000b]. DTD is a non-XML format that allows the expression of constraints on the names of elements and attributes and on where they can be placed. For elements that contain text, DTD specifies very little about the actual content. For mixed content (i.e., elements containing both text and other elements), DTD can only specify which elements appear but cannot constrain the multiplicity of an element. In general, DTD is from the document history of XML. It has no capabilities to easily handle XML namespaces, and that as well as an inability to constrain the textual contents and element count makes it difficult to use for XML data applications which require validity constraints of the textual data.

In order to correct some of the problems with DTDs and to provide better support for data-oriented applications, XML Schema [W3C 2001c] was created. XML Schema provides a way to constrain not only the elements but also the types of data allowed in the text of the document. XML Schema is large and complex and is more suited for data-oriented XML rather than document-oriented XML.

A lighter-weight approach to schema definition and validation is RELAX NG, a schema language for XML based on TREX (Tree Regular Expressions for XML) and RELAX (Regular Language Description for XML). For this research, the main feature is that it has unrestricted support for mixed content (i.e., semi-structured data) and is based on patterns rather than types.

3.7 Infrastructure of XML

The combination of the many parts of XML provides an infrastructure for the handling of documents and data. Multiple languages for addressing and querying as well as for constraints and transformations provide a rich toolkit to draw from, with several choices at each step to consider.

Once an XML application, i.e., an XML format, is created, a developer can mix and match which XML technologies to apply. XPath can be used to address locations in the document within transformation languages (e.g., XSLT, STX), or within APIs (e.g., DOM), or with standalone tools (e.g., *xpath*). Within these tools, the developer can choose not to validate the XML, or the XML can be validated using a DTD an XML Schema, or RELAX NG constraints. The power of XML comes from the interoperability of this set of tools.

CHAPTER 4

An XML Representation of Source Code

Representation of the differences between versions of source code requires representation of the source code. The chosen source-code representation has a direct impact on the capability to query and transform source-code differences. XML's original application to document representation and later adaptation to data representation makes it a clear choice for the representation of source code.

In this chapter we describe an XML application, srcML (SouRce Code Markup Language) [Collard, Kagdi, Maletic 2003; Collard, Maletic, Marcus 2002; Maletic, Collard, Marcus 2002], that supports both document and data views of source code. The format adds structural information to 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*), is 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 it allows a data view of the document without the loss of document information. The reverse is also allowed with document-view information, e.g., white space, comments, etc., used in the data view for searches or queries.

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

The feature of srcML that differentiates it from other related approaches is its ability to preserve all semantic information from the source code. This chapter motivates the document view of source code that srcML uniquely has as well as the use of XML to represent source code.

4.1 Document and Data Views 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 they effectively replace plain-text files. The programmer view is a document view of the source, while the compiler view is a data view of the source.

The current document view (i.e., the one that a programmer sees and works with) consists of a simple text file with a sequence of characters arranged in lines. Typical programmer application tools reflect this fundamental view of source code, such as the already discussed *diff* utility. Regular-expression tools, (e.g., *grep*) search text files to find lines containing a match to a given regular expression again reinforcing 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 form. They are powerful enough to locate the beginning of some syntactical elements, such as *if* statements; however, these applications require very complex regular expressions that are difficult for a programmer to create.

This type of matching is also used in context-sensitive editing tools; for example, *emacs* font-lock modes provide highlighting and other features based on regular-expression matching. However, these regular expressions are difficult to create and are of limited use for programmer or task-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.

Parsing and lexical analysis for a typical programming language generate an Abstract Syntax Tree (AST) and a symbol table. This is an abstract model of the source

code in which the format and contents 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 the transition from Figure 2.1 to Figure 4.2. 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 preprocessor, is used to remove the comments before the parser itself works on the document. While some work on ASTs does keep comments, the latter are often stored as attributes of a given syntactical element of the language. Although it may be easy to decide that a comment inside a syntactical element, such as a block, belongs to the block, it is not easy to decide to which of the comments a sibling syntactical element belongs.

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. In addition to this, it puts them into the form of line and column numbers that are difficult to integrate with the syntactic information.

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.

```
/*
      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 4.1. The source code consists of multiple files with the definition file (bottom) and the associated declaration (i.e., include) file (top).

```
void rotate(int&, int&, int&);

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

Figure 4.2. The AST view of source code is a single view of the two files with all comments, preprocessor directives, grouping of statements, and group declaration of variables eliminated

Since the AST 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.

4.2 Requirements for a Source Code Representation

In Table 4.1 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 management of documents. They also include tasks such as querying and linking to other documents. Software-engineering tasks reflect concepts such as version control, debugging, and analysis.

Table 4.1. Degree of support each representation lends to different tasks. There are two groups of tasks, the document aspects and the software aspects.

	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

For these groups of tasks, we have three possible source code formats: plain-text that the programmer typically works with, 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 of 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 for querying to extract subparts. As such, this format is very awkward for these types of tasks.

The AST and symbol-table format provides more support for most of these tasks, but for different reasons. This is due to the loss of textual information in the source code. For tasks in which source code is a final result, such as viewing, editing, and software visualization, the resultant view differs extremely from the source-code view. For document linking, it limits what we can link to in the source code. The only task that this

mapping does not hinder is with regard to 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. In addition, the AST format is, at best, limited to syntactically-correct code fragments. In general it requires a complete parsing precludes any application 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.

4.3 Related Work on XML Source Code Representations

Nowadays, any type of new data representation or manipulation is either using, considering, or currently being compared to XML. This is especially the case for representing source-code information where a number of options currently exist.

The data-oriented XML approach is taken by formats that currently exist for representing source-code information (e.g., AST or ASG) namely, GXL [Holt, Winter, Schürr 2000], CppML [Mammas, Kontogiannis 2000], ATerms [van den Brand, Sellink, Verhoef 1998], GCC-XML, and Harmonia [Boshernitsan, Graham 2000]. In these formats, the AST (actually an abstract syntax graph) of the source code, as output from a compiler intended for code generation, is stored in an XML data format. 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.

These representations are constructed as data-exchange languages or for displaying program structural information. They form an abstract model of the source code whose abstraction is not designed to represent the programmer's view. None of these representations directly supports the representation of comments or formatting information. The most widely used of these, GXL [Holt, Winter, Schürr 2000], is an XML-based exchange format for graph-like structures based on GraX (Graph eXchange format) [Ebert, Kullbach, Winter 1999], and RSF (Rigi Standard Format) [Wong 1998]. 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 for easy transformation.

More closely related to srcML is Badros' work on JavaML [Badros 2000], 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 associates them to elements of the program. Therefore, the location of comments is

not preserved. The association of comments with language 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 should be dealt with separately. In addition, even though the JavaML format claims to be able to produce the original source code, all formatting information is lost in JavaML and the original source-code document cannot be regenerated from JavaML representations.

In the same vein, the Harmonia framework [Boshernitsan, Graham 2000] and cppML/JavaML developed at the University of Waterloo [Mammas, Kontogiannis 2000] are closely related approaches as 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 meta-data, cppML only uses tags and records the additional information as attributes in the tags. The differences mentioned above for Badros' work stand for these approaches as well.

The XML data view of source code, since it is based on the AST, is a "heavyweight" format. It requires full parsing of the original document and generation of the complete AST. Many comprehension activities do not require a total parse tree or AST of the source code. Atkinson [Atkinson, Griswold 1996] argues that generating the entire AST is often impractical and that performing incremental parsing or "as needed" parsing is a better approach for many analysis tasks.

Although these source-code formats have specific uses and permit the use of the XML technologies, they don't address all of the requirements dictated for source-code

representation. Source-code documents are constructed by programmers that use the documentary structure of the language (i.e., comments, white space, etc.), interspersed with the syntactical constructs of the language (e.g., function, class) to express their ideas [Van De Vanter 2002] . A format that handles the narrative portions of the document is a better fit to the representation problem.

Other work on source-code interchange formats includes the work by Malton, et al. [Malton et al. 2001]. 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.

Similar to our approach, the MultiText Analytical Repository System (MARS) [Cox, Clarke, Sim 1999] stores source code in a structured-text repository. The source code is annotated with supplementary text that is viewed using HTML tags; however, the supplemental text is stored separately from the source code. Additional meta-data can also be associated with the source code such as the output of analysis tools. Supplementary text can include named links between positions in the token sequence and permit representation of graphs. This allows a software repository that is independent of the source model that is being represented. The query language of MultiText, GCL, can be used to extract source code based on the supplemental text. The difference with our work on source-code representations is that XML is not used and that the tags are not explicitly stored with the text.

Related to the general problem of storing documentary information (white space, comments, etc.) in a representation, it is argued in [Van De Vanter 2002] that such

representations must capture the right information so that the documentary information can be placed back into the source code after manipulation. The first approach listed is to attach documentary structure to the AST. This approach is rejected because of the difficulty of determining which documentary structure belongs to which syntactic item. The second approach is to store all documentary information. This approach is rejected due to the large storage requirements, and while it stores the documentary information, it does not store the documentary structure. The third approach is to only record the information needed for that particular transformation. The srcML format takes the second approach of storing all documentary information, but does so without excessive storage costs.

4.4 srcML: An XML Document/Data View of Source Code

The previous sections motivate the development of srcML and why the decision was made to simply superimpose 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. Representation of source code as structured documents directly supports the following:

- Representation of multiple levels of granularity within the AST;
- Multiple levels 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., and
- Preservation of the: location of constructs, text formatting information, comments and their location, file names and structure, and 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 document approach that srcML takes forms a model of the source code for the purposes of addressing locations and not to form an abstraction to get rid of "unessential" details. The resulting model is very robust since any incorrect or missing markup does not change the source code; it only changes the ability to address those elements.

In the design and construction of srcML, there are a number of factors that have an effect on the adoption of this type of approach. In short, selecting a lightweight philosophy is behind many of these issues. We discuss each issue that we saw as the requirements for our format:

- **Selection of document view over data view.** As previously discussed, XML applications (e.g., XML formats) and processing fall into the categories of a document view (e.g., DocType, etc.) or of a data view (e.g., SOAP). These two views influence the layout, semantics, and tools for the format. Source code, especially in an unprocessed state, is closer to a document than to data. That is, we program by writing documents, not by specifying parse trees. When required to make a decision between the two views, the document view is the clearest

choice. This does not lead to any loss, since a data view can simultaneously be supported with careful handling during processing and analysis.

- **Preservation of the original data (i.e., source code).** Allowing all of the original data to remain in any kind of transformation process is an accepted data-manipulation design principle. This is true even if the data is not currently needed. In srcML, the preservation of all original text allows for the restoration of the complete original document. The format does not try to reorganize or change the text. It only augments the text with markup that extends its capabilities, while still allowing the added information to be easily extracted.
- **Markup only what is of interest.** In srcML the markup stops at the expression level (i.e., expressions and the identifiers contained in the expressions are given markup). The full AST of the expression is not given markup because this does not meet the requirements of the uses that we saw. While marking full expressions down to the operator and parentheses level may be useful for expression rewriting and other compiler-oriented tasks, it is not useful for the identified tasks.
- **Tag Names based on programmers view.** Programmers know the basic syntax of the language that they are programming. They also know the syntactic name of many of the program entities, i.e., block, function, type, etc. They typically do not know the exact distinction between the finer points of syntax naming. For example, in the function declaration: *const int foo();* programmers will, for the most part, identify the return type of the function as *const int*. In srcML the type

const int is marked using the element *type*. Technically, however, the type is *int*, and *const* is a type specifier. Most programmers only care about this distinction when reading syntax diagrams.

- **Minimization of meta-data.** srcML representation contains few attributes. Markup in the source code is to provide navigation and access to the content and not to store meta-data. Information that may be derived from other parts of the program, e.g., the type of a variable used in an expression, can be directly derived or stored externally, as opposed to storing this information as attributes.
- **Easy conversion to and from the original format.** Source code text is extracted directly from srcML by removing markup and minor output un-escaping. This can be done by a variety of XML tools or even by simple Perl or Python programs.
- **Lightweight schema.** There is a tendency to produce a source code format that is very stringent, i.e., any documents produced in the representation can only represent compilable C/C++ programs. The philosophy of srcML is that this is unnecessary since there are compilers to test if a program can compile. Limiting the source code that can be represented to compilable code is a hindrance, since the source code may be in a state that cannot be compiled (e.g., it is under maintenance or subject to refactoring). This implies that we can do analysis on incomplete programs, programs with missing libraries, and source code that is under construction.

- **Format efficiency.** One of the criticisms of XML formats is their size. Data represented in an XML format may expand rapidly as levels of markup and attributes are used. This may result in an increased size hundreds of times over the original document and is especially problematic to DOM approaches (including XSLT), which store the entire tree in memory before processing is started. In practice, a srcML document is on the average five times larger than the original source-code document. Full AST markup in XML can result in the file being increased by hundreds of times over the original size of the file [Power, Malloy 2002].

4.4.1 Document View of srcML: 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 edited by the developer). Generating the original source document is a simple task of output of all the text nodes. 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 XML attributes, however, is normalized to a single space and is not preserved. Thus, all text that was in the original source code is stored as text within the elements. Only meta-information about the code is stored 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 srcML root 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 done in JavaML, this would lead to the loss of what the white space was composed. Also, the amount of storage seems to be about the same as storing these two attributes for every element. Furthermore, 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. They also allow processing to switch from an element view to a text view at any point in the processing.

Program comments are stored in their own element treating them as first-class syntactic elements with all formatting and location preserved. The user of the format can decide 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.

4.4.2 Data View of srcML: 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., an .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.

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. The srcML representation encodes data extracted from a partial derivation of the syntax tree. Parsing the source to a specific granularity level (e.g., expression) 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).

4.4.3 Preprocessor Constructs

A practical interest here is the difficulty dealing with macros, templates, and other preprocessor constructs in languages such as C/C++. The srcML format 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., elements *cpp:**) and not run through the preprocessor. The source is not completely parsed for translation into srcML. A partial derivation is used, stopping at 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. The format can also represent a syntactically-incorrect code fragment. The issue of syntactical correctness is a compiler problem and is not of supreme importance to the representation.

4.4.4 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, such as in the source code fragment:

int f(a);.

In order to solve the ambiguity of whether *f* is a function or variable identifier, we must know whether the identifier *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. If the identifier *f* was declared in an include file we would have to process all include files that it included.

If we didn't want to process the include file, or if we have a code fragment where this cannot be done, we can choose the assumption that identifier *a* is a type. 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 tool 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.

4.5 The srcML Translator

Translating source code to srcML is a multi-stage process where the core unit is the srcML translator. The translator-generator program ANTLR [Parr, Quong 1994] is used to construct the srcML translator from a pred-LL(k) grammar specification and a context stack to maintain context information. Actions, both pre and post, are attached to the grammar specification to markup the source code with XML start and end tags of the appropriate syntactic structure.

Upon identifying the beginning of a syntactic structure, an XML start tag is inserted into the token stream, and a transition occurs in the context stack to reflect the state of the construct being parsed. When the terminating token of a statement or block token is encountered, the context information from the context stack is used to insert end XML tags for the appropriate closing structures. This approach to parsing is motivated by the island-grammar concept, particularly by the idea of an *island with lakes* [Moonen 2001]. Moonen advocates the application of island grammar in the generation of robust parsers for source-model extraction and applications that do not need the complete parse tree. Much more information about the srcML translator and the island-grammar approach can be found in [Kagdi 2003].

4.5.1 Translator Requirements

In addition to the common requirements of an application, such as accuracy, reliability, and speed, the srcML translator has the additional requirement that it be able to be fully integrated with XML technologies. The list of additional requirements includes:

- **Responsiveness.** The decision to use event-parsing allows for output as soon as a statement is detected. Since output is immediately available, the translator can be integrated into a stream XML processing (i.e., SAX, STX, TextReader), allowing for the efficiencies of memory that these approaches allow. Although not as important for tree XML processing (i.e., DOM, XSLT) it does allow for simultaneous translation and tree building.
- **Flexibility.** It is important to support all possible forms for using any tool. Programmers still often use CLI (Command Line Interface) tools (e.g., *grep*), because they are fast, portable, and easy to use once learned. They also are easily scriptable, leading to low-level productivity tools. Also, GUI (Graphical User Interface) tools can be built on top of the existing CLI tool.
- **Scalability.** The event-driven translation approach allows for its use on large source-code files and large collections of source-code files.
- **Extensibility.** The srcML translator's CFG view of the source code and output in XML provides a base for further source-code processing. This processing may more accurately markup the source code based on external information (i.e., a symbol table), or transform it for another usage entirely.
- **Portability.** The srcML translator is a CLI program that can be used on both MS Windows and Linux.
- **Robustness.** The translator must generate a well-formed srcML document, regardless what the state of the input source code is. This is related to the lightweight view of schema conformance.

4.5.2 Translator Architecture

Translation proceeds with higher-level entities being parsed and marked appropriately before going into the details of constituent or lower-level entities. The subsequent levels are processed in later passes. This hierarchal approach enables us to control translation at the desired level of interest. Additionally, source-code irregularity in syntax present at one level does not impact parsing at other levels. Multi-pass parsing, with our partial grammar specification approach, supports an event-driven interface to the source code.

The srcML translator takes into account only a CFG view of the non-CFG C++ grammar. This transparent view of the translator introduces problems of misidentification of constructs that are syntactically identical at the context-free level. This kind of ambiguity complicates the production of parsers and fact-extraction tools for C++. Additional stages may be added to refine translation by using a refinement filter that is an XML transformation program.

4.5.3 Enhancements to Support Meta-Differencing

The original srcML translator as described in [Kagdi 2003] is referred to as the alpha version. It was used for the C++ Fact-Extraction Benchmark case study in CHAPTER 7. For the purposes of the work on meta-differencing, and to better support other research groups use of srcML, a new beta version was developed. The beta version of the srcML translator was significantly re-engineered and included significant changes over the previous alpha version with respect to robustness, speed, and maintainability. In this section we will describe some of these changes.

First, an extensive test suite was created for the srcML translator. This, with some automated testing tools, allowed for regression testing as changes were made to the translator. The test suite consists of a set of over 500 core srcML test cases. The automatic testing program extracts the source code from the srcML, runs the srcML translator and then compares the generated srcML to the original. Errors are automatically noted. In addition to the core test cases, there is a much larger set of generated test cases, which number over 1300. The generated test cases are all created by XML transformations using XSLT. The transformations included *if* to *while* conversion, definition to declaration conversion for functions, classes, etc., and parameter list creation in constructors. In addition transformations were written to automatically insert comments inside of every span of white space to verify that comments were handled correctly.

Internally the srcML translator underwent a major re-engineering. The alpha srcML translator used individual flags to store the state of the translator between inputs which caused a problem with some nested statements, e.g., a nested *switch* statement. In addition, much of the code was dependent on individual elements creating consistency problems for situations such as ending a statement with a semi-colon.

The flags and element-dependent code was replaced with a stack of modes. Each mode had its own set of flags and its own stack of open elements. When a mode was ended all open elements in that mode were automatically ended. This insured that the elements would be well formed, and allowed grouping of common actions. For example, all statements terminated at a semi-colon were in a mode with a MODE_STATEMENT

flag. When a semi-colon was reached if it was in a mode with this flag then the mode was terminated. This allowed for greater consistency of element handling.

The alpha version of the translator assumed that the source code was in C++ or in a C subset of C++. In most cases, this did not affect translation. However, some keywords in C++ are not keywords in C (e.g., *try*) and can be used as identifier names (e.g., *try* used as a label). In order to solve this problem, language modes were added to the translator. The current languages allowed are C and C++, with room for other languages (e.g., Java) to be added in the future.

The new version contains major speed improvements, translating at ~7500 lines/second (3 GHz Pentium 4, Linux version, single file) in C++ mode, and over 8000 lines/second in C mode. This compares to the ~100 lines/second that the alpha version carries out on the same file.

Changes were made to the srcML markup and incorporated into the srcML format stated in CHAPTER 5. The default on the new translator is the new markup, however a compatibility flag (-c) was added that allows the output of the old srcML. The compatibility mode does not make a noticeable difference in speed.

Other changes were made to the srcML markup. Where previously simple names inside of complex names were not marked:

<name>a::b</name>.

They are now marked to allow for access to the entire name, as well as with respect to names nested inside:

<name><name>a</name>::<name>b</name></name>.

This also affects array references where the entire array reference is considered a name, and the name of the array is a nested name:

<name><name>a</name>[]</name>.

Inside of types, names were not being marked:

<type>int</type>.

They are now marked to allow for easier addressing of individual parts of the type:

<type><name>int</name></type>.

Several element name changes were made to increase the consistency and provide more intuitive names. The element *using_directive* was changed to the element *using* in the style of other directives. In the area of function definitions, declarations, and calls, the element *formal_params* was replaced with the element *parameter_list*, and the element *actual_params* was replaced with the element *argument_list*. The element *param* inside an argument list was replaced with the more accurate element *argument* to prevent confusion with the element *param* in a parameter list.

Smaller changes include changing template arguments and parameters to the same marking as function arguments and parameters. Throw specifiers in function headers were previously unmarked and are now marked with the element *throw* allowing for direct addressing. The new element *decl* was added for declarations and is nested inside the element *decl_stmt* in a similar way that the element *expr* is nested inside the element *expr_stmt*.

CHAPTER 5

The srcML Format

The previous chapter covered the background and general characteristics of srcML. This chapter describes in detail the elements and the constructs of the srcML format. Information about each element and their attributes are given with explanations of the choices made. In addition, a full DTD for srcML is given in APPENDIX A. A RELAX NG was generated from the DTD.

The srcML format matches the syntactic structure of the source code as closely as possible. Therefore, this chapter is organized by statement and program entity. The only exceptions are the first two sections on the root element and documentary-structure elements.

5.1 Root Element

A source-code document is stored inside the root srcML element *unit*:

```
<unit
  xmlns="http://www.sdml.info/srcML/src"
  xmlns:cpp=http://www.sdml.info/srcML/cpp
  dir="src"
  filename=”rotate.h”>
</unit>
```

The srcML namespace is http*://www.sdml.info/srcML/src* and the optional *cpp* namespace is *http://www.sdml.info/srcML/cpp*. The optional attributes *dir* and *filename* contain the name of the directory and the actual filename, respectively.

The element *unit* contains any of the top-level srcML elements including documentary structure, statements, and declarations. In addition, *unit* elements may be nested to allow for several source-code documents to be represented in a single srcML document. This is especially useful for querying and transformation that simultaneously involve more than one document at a time.

5.2 Documentary Structure

The documentary structure is formed by elements that may appear at any point in the srcML document. They are not part of the syntactic part of the programming language and are often not considered part of the "program". They are usually eliminated by the preprocessor before actual compilation takes place. However, they are part of source-code documents and are completely preserved in srcML. Since they often crosscut the linguistic structure of the language, they are allowed to occur at any point in the other elements.

5.2.1 White Space

All white space (i.e., spaces, tabs, and blank lines) from the source-code document is preserved in srcML. As with all text in srcML, the source-code document ordering of white space is maintained. No explicit element is used for white space (or any other text) because text in XML forms its own default element (e.g., the function

text() in XPath). When there are concerns about the fragility of white space during processing the special XML attribute *xml:space,* with the value of "preserve", may be used to make the preservation more explicit. More information about white space handling in XML can be found in Section 3.1.

The end of line character is normalized to a single character, as per the XML standard. When converting srcML back to source code, the translation tool may use the end of line character that is the default for the target platform. Text source-code documents containing more than one type of end of line character (e.g., DOS, UNIX) will lose this information as part of the import into srcML.

5.2.2 Comments

Source-code comments are contained in the element *comment*.

```
<comment type="line">// counter
</comment>
<comment type="block">/* */</comment>
```

Line comments (i.e., starting with the string "//" and ending at the end of the line) are distinguished from block comments (i.e., sequence starting with the characters "/*" and ending with the characters "*/") by the attribute *type*. The line-comment element begins immediately to the left of the double slashes that mark them and end on the very beginning of the next line. The starting slashes and end-of-line character are included in the element. Block comments begin immediately to the left of the character sequence "/*" and ends immediately to the right of the character sequence "*/".

Comments typically contain text, but may include any other markup. This permits for representing code which has "commented out", i.e., previously non-comment

source code later place inside a comment. Differentiating between documentary text and statements in a comment is a difficult problem, but may be clear when multiple versions of a document, before and after commenting, are available.

5.2.3 Macro Calls

Macro calls, like comments, may occur inside any element. The element *macro* encloses the complete macro call:

<macro>D<argument_list>(a;)</argument_list>;</macro>

The macro call may contain an argument list enclosed in the element *argument_list*.

In general it is difficult to differentiate macros from function calls. The basic rule is that a *macro* element is used when what is being marked cannot possibly be a function call, e.g., it contains a statement. The handling of macro calls in unprocessed code is a difficult issue with macro detection likely to become more refined in the future.

5.3 Block

The block construct is part of the switch statement, array initialization, and may form a statement by itself. In srcML no distinction is made between these different uses. A block is marked using the element *block*:

<block>{

}</block>

The block delimiters are explicitly preserved. The block forms a top-most element (similar to the element *unit*) and can contain any of the other elements that the *unit* element can contain, except for a nested unit.

5.4 Selection Statements

5.4.1 If Statement

The element *if* encloses the complete *if* statement including the enclosed then statement:

```
<if>if <condition>(<expr><name>a</name></expr>)</condition><then>
;</then><else>else <block>{
}</block></else></if>
```

The condition is enclosed in a *condition* element. The programming language defines a condition as occurring inside the parentheses and does not consider the parentheses part of the condition. However, for ease of addressing the element *condition* does include the surrounding parentheses, and a separate *expr* element is used to contain the expression inside the condition.

The then statement is enclosed in the element *then* and the optional else statement is enclosed in the element *else*. The element *else* includes the keyword *else*. There is no equivalent keyword to mark the start of the then statement in a *if* statement, but the element *then* is introduced for consistency with the else statement.

5.4.2 Switch Statement

The switch statement is enclosed in the element *switch* including the block:

```
<switch>switch <condition>(<expr><name>a</name></expr>)</condition>
<block>{
 <case>case <expr>1</expr>:
   <break>break;</break>
 </case><case>case <expr>2</expr>:
   <break>break;</break>
</case>}</block></switch>
```

The switch condition is enclosed in a *condition* element, and the body of the statement is enclosed in a *block* element. The same markup rules for the *condition* element in the *if* statement apply.

Inside the block the case and default regions are contained in the elements *case* and *default*, respectively. These regions start at the appropriate keyword (i.e., *case* and *default* respectively) and extend all the way to the start of the next region or the end of the block, whichever comes first. All white space following the first region in the switch block (i.e., keyword *case* or *default*) is enclosed in a region. The *case* element includes an element *expr* to mark the expression. This expression is optional.

5.5 Iteration Statements

5.5.1 While Statement

The while statement is enclosed in the element *while*, including the nested statement:

```
<while>while <condition>(<expr>1</expr>)</condition> <block>{
}</block></while>
```

The condition is enclosed in a *condition* element. The rules for the markup of the condition are the same as in the *if* statement.

5.5.2 For Statement

The for statement is enclosed in the element *for* including the nested statement:

```
<for>for (<init><expr><type><name>int</name></type> <name>i</name>
=<init> <expr>0</expr></init></expr>;</init>
<condition><expr><name>i</name> &lt; 5</expr>;</condition>
<incr><expr>++<name>i</name></expr>,
<expr>++<name>s</name></expr></incr>)
  <block>{

}</block></for>
```

The header of the for statement includes the elements *init*, *condition*, and *incr*. The element *init* contains the first item in the header, the initialization statement that is executed once before the first iteration. The element *condition* contains the second item in the header, the condition that is evaluated to determine if the loop is executed. And the element *incr* contains the third item in the header, the statement that is executed after executing the body of the loop.

5.5.3 Do While Statement

The do while statement is enclosed in the element *do* including the nested statement:

```
<do>do <block>{
}</block> while <condition>(<expr><name>a</name></expr>)</condition>;</do>
```

The condition is enclosed in a *condition* element. The rules for the markup of the condition are the same as in the *if* statement.

5.6 Basic Statements

5.6.1 Break Statement

The element *break* encloses the complete break statement including the terminating semicolon:

<break>break ;</break>

5.6.2 Continue Statement

The element *continue* encloses the complete continue statement including the terminating semicolon:

<continue>continue ;</continue>

5.6.3 Return Statement

The element *return* encloses the complete return statement including the nested expression and the terminating semicolon:

<return>return <expr><name>a</name></expr> ;</return>

The expression is contained in a nested *expr* element and is optional. The expression starts after the keyword and the following white space and end before the white space right before the terminating semicolon.

5.6.4 Goto Statement

The element *goto* encloses the complete goto statement including the terminating semicolon:

<goto>goto <name>error</name>;</goto>

The name of the label is itself is marked with the element *name*. There is no verification that the name is valid, i.e., exists as a label.

5.6.5 Label Statement

The element *label* encloses the complete label statement including the terminating semicolon:

<label><name>error</name>:</label>

The name of the label is itself is marked with the element *name*.

5.6.6 Expression Statement

The entire expression statement is enclosed in the element *expr_stmt* including the terminating semicolon:

<expr_stmt><expr><name>a</name> + <name>b</name></expr>;</expr_stmt>

The expression itself is in the element *expr* which is used for all expressions. The only elements contained in an expression are the element *name* for the names of identifiers, and the element *call* for function calls. Operators are not explicitly marked.

5.7 Basic Declarations

5.7.1 Variable Declaration

The entire variable declaration is enclosed in the element *decl_stmt*, including the terminating semicolon:

<decl_stmt><decl><type><name>const</name> <name>int</name></type>
<name>a</name></decl>;</decl_stmt>

The actual declaration is enclosed in the element *decl*, which does not include the terminating statement semicolon. The type of the declaration is enclosed in the element *type*. The definition of type is very loosely defined and can include type specifiers, compiler options, etc. No distinction is made between these subparts of the type, except

to mark the individual names. In general, the type of a variable declaration is everything that occurs before the name of the (first) variable.

5.7.2 Typedef Declaration

The entire typedef is enclosed in the element *typedef*:

```
<typedef>typedef <type><name>const</name> <name>int</name> *</type>
<name>a</name>;</typedef>

<typedef>typedef  <function_decl><type><name>int</name></type>
(*<name>f</name>)
<parameter_list>()</parameter_list>;</function_decl></typedef>
```

The type of the declaration is enclosed in the element *type*. The term type is often loosely used and may include (or not include) type specifiers, compiler options, etc. No distinction is made the markup between these subparts of the type. Function declarations can also be part of a typedef using the same elements.

5.7.3 Enum Declaration

An enumerated type is contained in the element *enum:*

```
<enum>enum <name>a</name> <block>{
   <expr><name>b</name></expr>,
   <expr><name>c</name></expr>
}</block>;</enum>
```

The enum includes an optional embedded *name* element and a block. The block contains a series of expressions separated by semi-colons. The semi-colons are part of the *block* element.

5.7.4 Extern Statement

The extern declaration is contained in the element *extern*:

<extern>extern "C" <block>{}</block></extern>

The body of the extern is also enclosed. Currently the type of the extern is not marked separately.

5.7.5 Asm Statement

The asm statement is contained in the element *asm*:

<asm>asm ("rep stosl");</asm>

At this point there is no further markup of an *asm* statement. The format is very liberal as to the contents. Because the contents are assembly-language statements any further markup would be specific to that assembly language.

5.7.6 Struct

The struct definition is contained in the element *struct*:

<struct>struct <name>A</name> <block>{<public type="default"/>

}</block>;</struct>

In C++ mode the struct is extended and follows the class *element* closely. The only difference is that a *struct* element may contain a default section specifier of element *public* with the *type* attribute value of “default”.

A struct declaration is contained in the element *struct_decl*:

<struct_decl>struct <name>A</name> ;</struct_decl>

5.7.7 Union

The union definition is contained in the element *union*:

<union>union <name>A</name> <block>{

}</block>;</union>

A union declaration is contained in the element *union_decl*:

<union_decl>uniont <name>A</name> ;</union_decl>

5.8 Functions

The largest amount of markup occurs in a function definition or declaration header. All of the separate parts of the header, i.e., type, name, parameters, etc., are of interest so all are individually marked. The marked parts of a function include the the parts of the header and block for function definition. A throw list is optional. A complete function definition is enclosed in the element *function*:

<function><type><name>int</name></type>
<name>f</name><parameter_list>()</parameter_list>
<block>{}</block></function>

A function declaration is similar except for the missing block and is enclosed in the element *function_defn* including the terminating semicolon:

<function_decl><type><name>int</name></type>
<name>f</name><parameter_list>()</parameter_list>;</function_decl>

The element *type* encloses the entire type of the function. Similar to a variable declaration, the element includes all items in the function header before the function name. Individual names inside the type are marked accordingly. No differentiation is made between standard type names, e.g., *int*, and user-defined type names allowing standard type extensions to the base language, e.g., a dialect, without change in the markup. The name of the function is enclosed in the element *name*. Complex names are further subdivided.

The function parameter list is enclosed in the element *parameter_list*:

```
<parameter_list>(<param><decl><type><name>int</name></type>
<name>a</name></decl></param>)</parameter_list>
```

The parameter list starts before the left parentheses after the function name and ends at the matching right parentheses. Individual parameters are enclosed in the element *param* with the commas separating them as part of the enclosing *parameter list* element.

Each *param* element includes a declaration enclosed in the element *decl* which is also used in variable declaration. The element *decl* includes a type and a parameter name. If there is only one name in the parameter then it is consider part of the type and no *name* element will occur. More information on the markup of declarations can be found in the section on variable declarations.

5.9 Classes

Classes add elements for classes, constructors and destructors. In addition it introduces sub-elements for inheritance, access region, and member initialization list.

5.9.1 Class

The class definition, i.e., the declaration of the member functions and data members of a class, are contained in the element *class*:

```
<class>class <name>A</name> <super>: <specifier>virtual public</specifier>
<name>B</name></super>
<block>{<private type="default"></private>

}</block>;</class>
```

¨The name of the class is contained in the element *name*. All optional inheritance information is contained in the element *super* which begins right before the character ‘:’

and ends before the white space that starts the block. It may contain a series of specifiers (e.g., *virtual*, *public*, etc.) and a *name* element with the name of a base class.

Inside the declaration block the access regions are marked with elements *public*, *private* and *protected*. These elements begin right before the keyword that marks them and extend all the way to the next access region or the end of the block, whichever comes first. The only exception is the default region that occurs at the start of the block. The default region occurs whenever the body of the class does not begin (including white space) with an explicit access region. This is given an attribute *type* with the value "default" to allow differentiation between implicit and explicit access regions.

Data members inside the class definition are marked as variable declarations and member functions are marked as functions. Addressing the members exclusively requires using the context of the *class* element.

5.9.2 Constructor

A constructor definition is enclosed in the element *constructor*:

```
<constructor><name>A</name><parameter_list>()</parameter_list>
  <member_list>: </member_list><block>{

}</block></constructor>
```

Typically the first contained element is the constructor name in a *name* element, however, an optional type is allowed (e.g., *inline*). This is followed by the complete parameter list enclosed in a *parameter_list* element and by an optional member initialization list enclosed in a *member_list* element. Individual member initializations

are marked as calls using the *call* element. Last, the body of the constructor is enclosed in a *block* element.

A constructor declaration is distinguished from a definition and is enclosed in the element *constructor_decl*:

```
<constructor_decl><name>A</name><parameter_list>()</parameter_list>
;</constructor_decl>
```

Typically the first contained element is the constructor name in a *name* element; however, an optional type is allowed as in the constructor definition. This is followed by the complete parameter list enclosed in a *parameter_list* element. The *constructor_decl* element ends immediately to the right of the terminating semicolon.

5.9.3 Destructor

A destructor definition is enclosed in the element *destructor*:

```
<destructor><name>A</name> <parameter_list>()</parameter_list> <block>{
}</block></destructor>
```

Typically the first contained element is the destructor name in a *name* element; however, an optional type is allowed (e.g., *virtual*). This is followed by the empty parameter list enclosed in a *parameter_list* element. Last, the body of the destructor is enclosed in a *block* element.

A destructor declaration is distinguished from a definition is enclosed in the element *destructor_decl*:

```
<destructor_decl><name>A</name><parameter_list>()</parameter_list>
;</destructor_decl>
```

Typically the first contained element is the destructor name in a *name* element; however, an optional type is allowed as in the destructor definition. This is followed by the empty parameter list enclosed in a *parameter_list* element. The *destructor_decl* element ends immediately to the right of the terminating semicolon.

5.10 Templates

Templates are placed around functions and classes. The element *template* encloses the entire template definition, including the templates class/function:

```
<template>template
<parameter_list>&lt;<param><type><name>int</name></type>
<name>n</name></param>&gt;</parameter_list>
<function_decl><type><name>int</name></type> <name>foo</name>
<parameter_list>()</parameter_list>;</function_decl></template>
```

The parameters for the template are enclosed in the element *parameter_list*. These include the template parameter list characters "<" and ">" which are escaped by using the entity references *<* and *>*. Inside the parameter list is a series of parameters marked by individual *param* elements. The parameter has at least a type, enclosed in the element *type*, and an optional name, enclosed in the element *name*.

5.11 Namespaces

A namespace region is enclosed in the namespace element:

```
<namespace>namespace <name>a</name> <block>{

}</block></namespace>
```

The optional name of the namespace is contained in the element name. An unnamed namespace has no name element at all:

<namespace>namespace <block>{

}</block></namespace>

A namespace alias has two names and uses the same element namespace:

<namespace>namespace <name>a</name> = <name>b</name>;</namespace>

The using statement is enclosed in the element *using*:

<using>using <name><name>a</name>::<name>b</name></name>;</using>

5.12 Exception Handling

A *try* statement is enclosed by the element *try*:

<try>try <block>{

}</block></try>
<catch>catch (<param><decl><type><name>a</name></type>
<name>b</name></decl></param>)
<block>{

}</block></catch>

The block is considered part of the *try* element. The *catch* statement is enclosed by the element *catch* and is not enclosed in the element *try*. It includes a single parameter enclosed in the element *param*.

The *throw* statement is enclosed by the element *throw*:

<throw>throw <expr><name>d</name></expr>;</throw>

The element *throw* is also used for the throw clause in a function.

5.13 Preprocessor Directives

The elements for the C-Preprocessor directives are in a separate namespace, *http://www.sdml.info/srcML/cpp*, for easier addressing and transformation as a group. This also allows for distinction between statements in the base language and preprocessor

directive, e.g., the C keyword *if* and the preprocessor directive *#if*. They are given the namespace alias *cpp* in the rest of this section. The *cpp* elements use some of the elements of srcML, including *name* and *expr*. They also contain some new internal elements.

Because directive keywords (e.g., *if*, *include*) do not start the preprocessor directive (i.e., there is a leading '#'), they are marked using the element *cpp:directive*:

<cpp:directive>include</cpp:directive>

5.13.1 Include

A preprocessor *#include* is marked using the element *cpp:include*:

<cpp:include>#<cpp:directive>include</cpp:directive><cpp:file>"a.h"</cpp:file></cpp:include>

The directive *include* is explicitly marked with a *cpp:directive* element. An optional *cpp:file* element contains the complete filename.

5.13.2 Define

A preprocessor *#define* is marked using the element *cpp:define*:

<cpp:define>#<cpp:directive>define</cpp:directive> <name>A</name> <expr>1</expr></cpp:define>

The directive *define* is explicitly marked with a *cpp:directive* element. An optional *name* element contains the preprocessor identifier. The value of the preprocessor identifier is contained in an *expr* element.

5.13.3 Undef

A preprocessor *#undef* is marked using the element *cpp:undef*:

<cpp:undef>#<cpp:directive>undef</cpp:directive> <name>A</name></cpp:undef>

The directive *undef* is explicitly marked with a *cpp:directive* element. An optional *name* element contains the preprocessor identifier.

5.13.4 If

A preprocessor *#if* is marked using the element *cpp:if*:

<cpp:if>#<cpp:directive>if</cpp:directive><expr><name>a</name></expr></cpp:if>

The directive *if* is explicitly marked with a *cpp:directive* element. The condition is contained in an *expr* element.

5.13.5 Else

A preprocessor *#else* is marked using the element *cpp:else*:

<cpp:elsef>#<cpp:directive>else</cpp:directive></cpp:else>

The directive *endif* is explicitly marked with a *cpp:directive* element.

5.13.6 Endif

A preprocessor *#endif* is marked using the element *cpp:endif*:

<cpp:endif>#<cpp:directive>endif</cpp:directive></cpp:endif>

The directive *endif* is explicitly marked with a *cpp:directive* element.

5.13.7 Elif

A preprocessor *#elif* is marked using the element *cpp:elif*:

<cpp:elif>#<cpp:directive>elif</cpp:directive><expr><name>a</name></expr></cpp:elif>

The directive *elif* is explicitly marked with a *cpp:directive* element. The condition is contained in an *expr* element.

5.13.8 Ifdef

A preprocessor *#ifdef* is marked using the element *cpp:ifdef*:

<cpp:ifdef>#<cpp:directive>ifdef</cpp:directive> <name>a</name></cpp:ifdef>

The directive *ifdef* is explicitly marked with a *cpp:directive* element. An optional *name* element contains the preprocessor identifier that is being checked for definition.

5.13.9 Ifndef

A preprocessor *#ifndef* is marked using the element *cpp:ifndef*:

<cpp:ifndef>#<cpp:directive>ifndef</cpp:directive>
<name>a</name></cpp:ifndef>

The directive *ifndef* is explicitly marked with a *cpp:directive* element. An optional *name* element contains the preprocessor identifier that is being checked for definition.

CHAPTER 6

Application of srcML

The transparent, srcML view of source code allows the direct use of XML in source-code documents for addressing, querying, and transformation. The general process is shown in Figure 6.1, where the source code is raised to an equivalent srcML document. Any work done at the XML level can be directly related to the textual source-code document.

This chapter will explain in detail how these tasks can be performed. It will begin by showing how source code can be addressed, will extend addressing into querying, and will demonstrate non-intrusive transformations including refactoring. It continues by demonstrating how higher-level abstractions can be integrated into these problems. This chapter finishes by discussing other software-engineering projects that are applying srcML to solve practical problems.

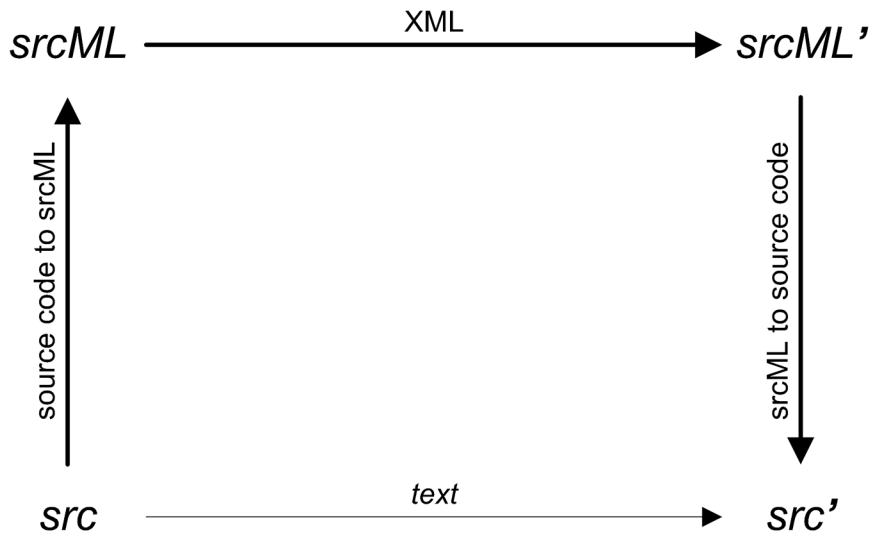


Figure 6.1. With srcML, XML tools and technologies can be applied to source-code documents. The source code is translated into srcML, XML is applied to the srcML form, as well as any results directly mapped or transformed back to the source-code document.

6.1 Addressing and Querying Source Code

Once source code is in a srcML format, it is addressable by XPath, the standard language for addressing locations in XML documents. XPath expressions using srcML elements can be used to specify particular points in the source code. In this way, XPath provides a very rich set of ways to express a location in source code.

A specific location in a source-code document can be expressed in multiple ways in XPath. Any specific source-code entity can be found by location in the document relative to any other source-code entity; the XPath expression */unit/function[1]* is the address of the first function definition (in relation to document ordering) at the top level of the document. Function definitions can occur at any level in the XML document including inside namespaces; the XPath expression *//function[1]* finds the first function definition that occurs in the entire document, not just at the top level.

For source code-entities that are named, e.g., classes, functions, and namespaces, the name can be used in the address, and correspondingly as a name in an XPath predicate. For example, the XPath expression *//function[name='convert']* locates the function definition with a name *convert*.

To locate an entity in the context of other entities requires a path in that context. The XPath expression *//function//if[1]* locates the first *if* statement (at any level) inside a function definition. The XPath expression *//function/block/if[1]* locates the first *if* statement inside a function that is not nested inside another statement (i.e., at the top level inside the block of the function).

The ability of XPath to use the same syntax to refer to a specific item as it does to a group is an extremely useful construct. Using the previous examples, */unit/function* refers to all functions at the top level and *//function* locates all function definitions in the entire document. Even the XPath expression *//function[name='convert']* actually locates all function definitions with the name *convert*. There may be more than one function with the same name due to function overloading, or even a temporary (un-compilable) transition to rewrite the function. In context, the expression *//function/block/if* locates all *if* statements inside any function. We can get more specific and find all *if* statements inside of the function with the name *convert* by using the XPath expression *//function[name='convert']//if*. XPath expressions of this type can be used with any of the entities in srcML including functions, classes, statements, types, comments, preprocessor directives, etc. This provides an extremely rich method for expressing the location of source-code elements.

Source code can also be queried for the purposes of extraction or transformation by using XPath addresses. All that is required is to evaluate the expressions inside an XML tool that supports XPath (e.g., XSLT, TextReader) or with a standalone command-line tool. In this case, source-code querying is just an extension of source-code addressing.

6.2 Transformation of Source Code

With srcML, XML transformations can be used to perform non-intrusive transformations on source code. Transformation can be performed on any selected element of the syntactic or documentary structure, while simultaneously preserving all

other elements. The process consists of translating source-code documents into a srcML document, the srcML document is then transformed using any XML transformation approach (e.g., SAX, DOM, XSLT) into another srcML document, and the transformed srcML document can be easily translated into a transformed source-code document.

The transformation process is shown in Figure 6.2. The source-code document, *src*, is converted by the srcML translator into an equivalent srcML document, *srcML*. XML transformations of the srcML document convert *srcML* to *srcML'*. Afterwards, the transformed srcML document, *srcML'*, is translated back to a source-code document, *src'*. Depending on exact XML transformation used, the transformation process is non-intrusive. All elements of the source-code document, including documentary structure, such as white space, preprocessor directives, etc., can be passed unaltered through the transformation. An identity XML transformation at the srcML level, i.e., *srcML* is identical to *srcML'*, is equivalent to an identity transformation at the textual level, i.e., *src* is identical to *src'* with the possible exception of a change in end of line character. Only the source code elements that require changes are altered in any way.

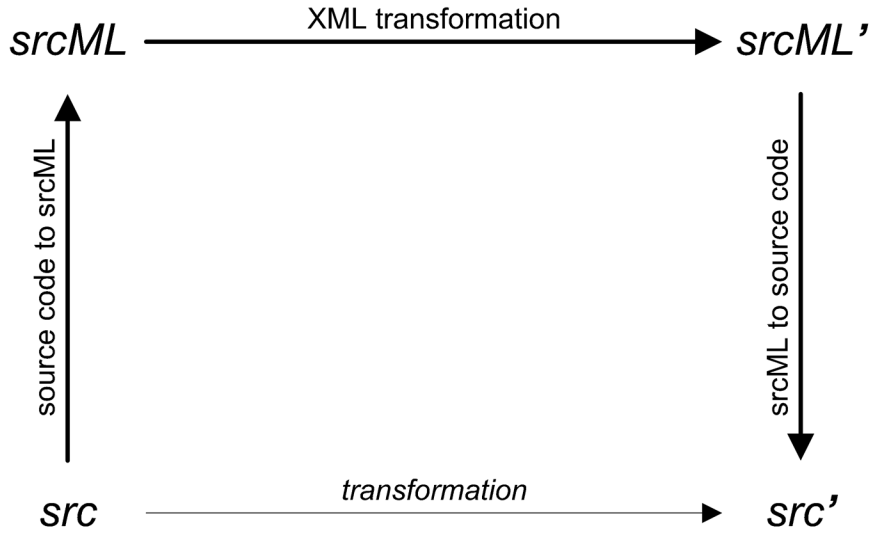


Figure 6.2. With srcML source-code transformations are raised to the level of XML transformations. The source code is translated into equivalent srcML, the srcML undergoes XML transformation, and the resulting srcML is translated back to source code.

```
#include <iostream>

int readsum() {

    // total elements
    int total = 0;
    int n;
    if (std::cin >> n) //
sums total
        total += n;

   return n;
}
```

```
#include <iostream>

int readsum() {

    // total elements
    int total = 0;
    int n;
   while (std::cin >> n) //
sums total
        total += n;

   return n;
}
```

Figure 6.3. Transformations on source code are performed that preserve all documentary information with changes only to specifically chosen elements. The conditional (if) shown on the left was converted to an iterative (while) statement shown on the right with the preservation of all other original information.

One characteristic of srcML requires careful attention. As a document-oriented format srcML stores not only the elements for a statement, but also the keyword for that element. In a data-oriented format, these keywords would not be stored, and a transformation would only pay attention to the elements.

In Figure 6.3 a source code transformation is shown that converts the if statement to a while statement. All other elements remain unchanged. Figure 6.4 shows the important templates of an XSLT program that performed this transformation. The first template matches any *if* statement that does not have an *else*, and replaces the *if* element with a *while* element. The second template removes the *then* element without changing its contents. The last template replaces the keyword *if* with the keyword *while*.

Care must be taken in the translation to translate the element (i.e., from element *if* to element *while*) and the keyword (i.e., from keyword *if* to keyword *while*), and to filter

out the element *then*. Not shown in the program is the default copy template that preserves all items not matched by the shown templates.

Of special note is the preservation of white space. Some XML processors consider all white space in elements insignificant, and the processors normalize all white space. However, our experience has found that careful use of these tools allows the process to preserve these elements.

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet
        xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
        xmlns:src="http://www.sdml.info/srcML/src"
        xmlns:str="http://exslt.org/strings"
        extension-element-prefixes="str"
        version="1.0">

<!-- change the if element to a while element -->
<xsl:template match="src:if[not(.//src:else)]">
    <src:while>
          <xsl:apply-templates/>
    </src:while>
</xsl:template>

<!-- filter the then element itself (tags), but not the
contents of the then -->
<xsl:template match="src:if/src:then">
    <xsl:apply-templates/>
</xsl:template>

<!-- change the if keyword to a while keyword -->
<xsl:template match="src:if/text()[1]">
    <xsl:value-of select="str:replace(., 'if', 'while')"/>
</xsl:template>

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

</xsl:stylesheet>
```

Figure 6.4. A non-intrusive source code transformation in XSLT. The templates match *if* statements that do not have *else*, the *then* element of the *if* statement, and the text with the keyword if. The templates replace the *if* element with a *while* element, remove the *then* element while preserving its contents, and replaces the keyword *if* with the keyword *while*, respectively.

6.3 Refactoring Source Code

A specific transformation of current interest is *refactoring,* a source-to-source transformation that preserves program behavior [Opdyke 1992]. The purpose of refactoring is to improve the overall structure of the code so that it can be more easily extended, repaired, and to increase comprehension.

Fowler [Fowler 1999] has created a catalog of refactorings for Java which has become a de-facto standard. One of the refactorings is titled "Replace Nested Conditional with Guard Clauses". This refactoring applies to methods where a conditional statement is wrapped around normal (i.e., non-error) processing to prevent execution when there is an error in the input or in the state of the method. During normal processing the condition is always true. The condition is false only with an error.

Fowler argues for the removal of the nested conditional because it hinders the comprehension of the normal execution path. Instead a *guard clause* is used to handle abnormal situations. A guard clause is a conditional statement placed before normal processing that checks for an error, and if it occurs returns from the method before the code for normal processing is executed.

Preservation of documentary structure is extremely important during refactoring. The output of a refactoring is a modified source-code document that is returned to the programmer to continue editing/viewing. This output must preserve the comments, white space, etc. that was present before the transformation.

```
int factorial(int n) {

  // factorial value
  int product = -1;

  // check for proper
values
  if (is_non_negative(n)) {

    // calculate factorial
    product = 1;
    int i = 1;
    while (i <= n) {

      // update the product
      product *= i;

      // next value
      ++i;
    }
  }

  //now the factorial
  return product;
}
```

```
int factorial(int n) {

  // factorial value
  int product = -1;

  // check for proper
values
  if (!is_non_negative(n))
    return -1;

  // calculate factorial
  product = 1;
  int i = 1;
  while (i <= n) {

    // update the product
    product *= i;

    // next value
    ++i;
  }


  //now the factorial
  return product;
}
```

Figure 6.5. An example of the refactoring "Replace Nested Conditional with Guard Clause". The original function on the left uses a nested conditional to check the status of the input parameter. The refactored function on the right uses a guard clause and preserves the normal processing.

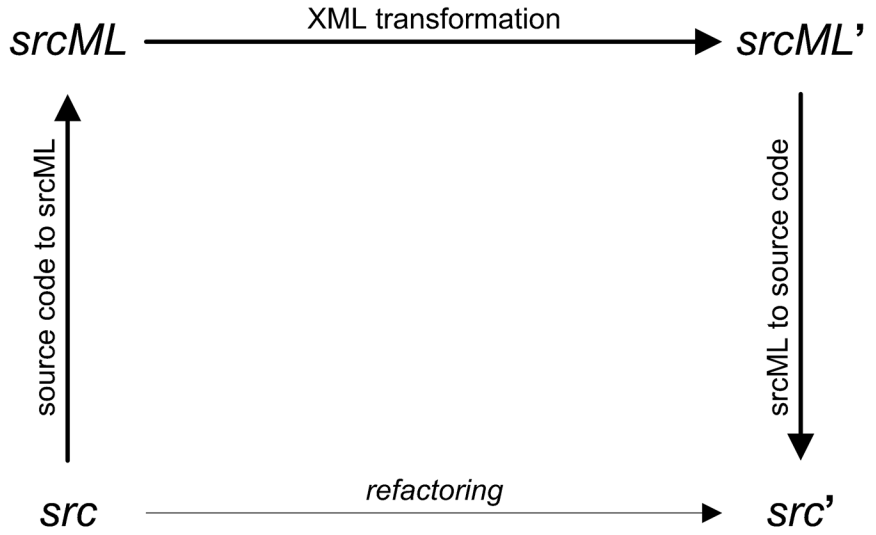


Figure 6.6. With srcML source-code refactoring can be performed as an XML transformation.

Current refactoring tools have difficulty with the preservation of these elements and in languages such as C++ only a limited number of refactorings from the catalog are available. In addition, tools that can perform refactorings are very carefully constructed for specific refactorings. The tools cannot be easily enhanced or extended, and they do not provide a general purpose format and/or language that can be used by a programmer to adapt them to their own use, or to write additional transformations [Vanter 2002].

The previous transformation using srcML provides a model for how refactorings can be implemented. Figure 6.6 shows how the XML transformation can be used for refactoring. The remainder of this section will demonstrate a non-intrusive refactoring of source code via using an XML transformation. The example shown is the refactoring already described.

Portions of an XSLT program that can perform a non-intrusive source-code refactoring are shown in Figure 6.7. As in the transformation in Section 6.2, a default copy template is used so that, unless otherwise specified, the elements and text will go through the transformation unaltered. The main template matches the *if* statement that forms the nested conditional using the XPath expression:

src:if[src:condition/src:expr//src:name=$cname]

```
<!-- match the if with a condition containing a call to the function
$cname -->
<xsl:template
match="src:if[src:condition/src:expr//src:name=$cname]"
xml:space="strip">

  <!-- create the guard -->
  <xsl:apply-templates select="." mode="guard"/>

  <!-- calculate indent on the if -->
  <xsl:variable name="text before" select="preceding-
sibling::text()[1]"/>
  <xsl:variable name="if indent" select="string-
length($text before)"/>

  <!-- process the block -->
  <xsl:apply-templates select="src:then/src:block" mode="unblock">
    <xsl:with-param name="unindent length" select="$if indent"/>
  </xsl:apply-templates>

</xsl:template>

<!-- create guard clause -->
<xsl:template match="src:if" mode="guard">
if (!<xsl:value-of select="src:condition/src:expr"/>)
    return -1;
</xsl:template>

<!-- unblock statements -->
<xsl:template match="src:block" xml:space="strip" mode="unblock">
  <xsl:param name="unindent length" select="2"/>

  <!-- strip off the starting and ending braces -->
  <xsl:variable name="contents" select="substring(., 2, string-
length(.) - 2)"/>

  <!-- output the rest of the contents with indentation fixed -->
  <xsl:call-template name="unindent">
    <xsl:with-param name="text"   select="$contents"/>
    <xsl:with-param name="length" select="$unindent length"/>
  </xsl:call-template>

</xsl:template>
```

Figure 6.7. A non-intrusive source code refactoring in XSLT. The nested conditional contains an if statement that addressed by an XPath expression using the if statement elements. A new guard clause is created by mixing plain text and XSLT calls for the if statement. The contents of the nested conditional block are processed as a single string with the indentation fixed. The un-indenting string code is not shown.

This expression matches all *src:if* elements whose condition has the name contained in the parameter *$cname*. This parameter contains the name of a function call that indicates this is a nested conditional.

After matching the nested conditional the main template generates the guard clause and converts the former contents of the guard clause so that they are at the top level of the function. In addition, it fixes the indentation on these contents so that they line up with the rest of the program.

The main template uses another template, with the mode attribute "guard", to construct the guard clause. A text *if* statement that embeds the condition from the nested conditional and inserts a *return* statement is formed in the template:

```
if (!<xsl:value-of select="src:condition/src:expr"/>)
   return -1;
```

This demonstrates a big advantage of the transparency of the srcML representation. It is not necessary to markup all new, generated code in srcML. Explicit source code text can be used in the transformation program interspersed with XML programming statements. If full srcML markup is needed, this partial srcML document can be converted to source code, and then processed by the srcML translator.

After the creation of the guard clause the main template calculates the previous indentation of the *if* keyword using the XPath expression:

preceding-sibling::text()[1]

The remainder of the main template handles the contents of the nested conditional and "un-nests" the contents. The contents of the block are converted to a

string. The braces at the start and end of the block are removed by extracting a substring of the block that removes the first and last characters:

substring(., 2, string-length(.) - 2)

This string is passed to a template that removes these characters from the beginning of each line. The template used for this is not shown, but is a recursive template that splits the strings at the new line and checks for needed trimming of the white space. It is very advantageous to find locations in the source code using a rich addressing language, and when needed process an entire segment of the XML document as a string.

The current form of the refactoring has some limitations. First, the identification of a nested conditional (versus normal processing) is limited to a special function name. The user would have to indicate, perhaps by location, where the nested conditional is. Second, the *return* statement has a fixed return value. The value could be indicated by the user, e.g., via a parameter, or deduced from the context. Improvements can also be made to the XSLT program to make it simpler and more straightforward by defining special XPath functions for the string handling, e.g., indentation.

This example demonstrated a flexible, non-intrusive approach to source-code refactoring. By using a source-code representation appropriate to the problem, all levels of source-code information were available for transformation. Although XSLT was used in the example, any XML transformation language or API could have been used instead.

6.4 Integration of Higher-Level Abstractions

One of the major design decisions of srcML was to limit the abstraction to that of the source code. This limitation creates a problem when use of a higher-level abstraction is desired. These abstractions may be derived directly from the source code (e.g., symbol table) or may involve information external to the source code (e.g., design-level information). Alone these higher-level abstractions are source models of the source code. Typically source-model abstractions are generated with little traceability to the source code. Any information not directly stored in the source model is difficult to obtain. Integration of the source model and the source code is difficult.

This section will demonstrate that source models can be integrated into the source code by showing how a particular source model (i.e., a call graph) can be represented and used in queries that integrate both the source model and the source code. Both higher-level abstractions and a low-level source code abstraction are possible.

6.4.1 Source Model Representation Using XLink and srcML

The srcML format was used to further extend the lightweight fact-extraction concept, along with the use of XML technologies, and 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 APIs, 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 call-graph information with the source code. 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 include 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 allows the linking of entities that are typically discarded during pre-processing such as comments, preprocessor directives, and macros. In detail we show how XPath can be used to locate a function in the source code and used in queries integrating both the source code and the call graph.

This section demonstrates that higher-level source models can be represented and integrated with srcML and that the XML linking language XLink can be used with srcML. Additionally, we show that XPath can be used to address a particular function definition at various levels of specificity.

6.4.2 Call Graphs

Call graphs, connected graphs that represent function calls within each function (or module), are widely used during the reverse-engineering process. They are the basis of a number of useful metrics including fan in, fan out, coupling, and to a lesser extent cohesion.

Construction of a call graph for a given software system is quite straightforward via the symbol table and other artifacts produced by a compiler. Representation of a call

graph can also be equally straightforward (e.g., a matrix). However accurate and straight forward this seems, it does suffer from lack of flexibility and interoperability. For one thing you must be able to compile the source code and this may be problematic due to current state of the system, lack of include files, platform change, etc.

A number of researchers [Cox, Clarke 2000; Moonen 2002; Murphy, Notkin 1996] have been developing lightweight approaches to the general problem of constructing source models [Murphy, Notkin 1996; Murphy et al. 1998]. Source models are used to explicitly represent the higher-level relationships within source code than is evident by the source code. Common source models include call graphs, abstract syntax trees, inheritance hierarchies, UML class diagrams, etc. and are a model of the interrelationships between program entities. Lightweight approaches are typically based on a task-specific extraction of the model using lexical techniques that are tolerant of source code irregularities. This is in contrast to full-parser-based approaches that require regular, compilable code and that often store (and extract) much more than is needed.

The work presented further extends the lightweight fact extraction concept, with the use of XML technologies to represent, manipulate, and navigate links (relationships) within the source code. The links are represented using the XML Linking Language (XLink). The combination, or blurring, of the source code and the source model as an XML format is leveraged via the APIs, tools, and technologies of XML. These tools are then used to query the source code and call graph. The call graph is 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 next section describes the issues related to representing call graphs. Afterwards the source model is detailed with a description of our source-code representation, srcML, and a C++ to srcML translator. Lastly, how XLink is used to represent relationships, such as call-graph information is given, with how the call graph and the source code can be used together in queries.

For a given set of source-code documents there are potentially multiple different call graphs based on the set of documents chosen, the general approach of the construction (static analysis versus dynamic analysis), the decisions used during the implementation of the algorithm, and the needs of the particular task being performed. In addition, there are many choices involving the amount of coupling between the call graph and the source code.

Call graphs are constructed from static analysis of the source code or dynamic analysis of a program's run-time behavior. For statically constructed call graphs the particular algorithm used and the implementation of the algorithm can lead to differences in the generated call graph [Murphy et al. 1998]. These may produce many different call graphs for the same source code.

In addition to this, call graphs can be constructed on an ad-hoc basis to support a particular analysis approach or task. These ad-hoc approaches may not fit well with a representation if the representation has a strict schema specifying exactly what information will be represented. At the same time, producing totally unique representation is problematic due to the lack of existing tool support that is necessary to handle many common tasks. Our work addresses this issue by utilizing a flexible

representation that can be easily adapted to the current task, while at the same time providing a rich tool-set.

Any nontrivial software project consists of a large number of source-code documents. If the project is multi-platform, there is often different builds of the software typically controlled by preprocessor definitions. As such, construction and representation of a call graph can depend on which particular source-code documents are being used for the analysis. Therefore, this leads to the possibility of several call graphs for the same set of source-code documents, each call graph mapping to a particular subset.

Even with a fixed set of source-code documents there may exist multiple versions of the call graph as the source code changes. A developer involved in a re-engineering task, i.e., conversion from one API to another, can use various versions of a call graph corresponding to changes in the source code. Each version would be used for a different task. The initial call graph of the original system would show an overall picture of the entire re-engineering task, while the current call graph of the system would show the current status of the conversion task.

6.4.3 Relationship to Source Code

To be especially useful for program comprehension, the relationship between the call graph and the source code is important. A call graph that does not include a representation of (or handle to) the original source code requires the developer to go from the call graph to a separate source-code document which limits the possibilities of fully integrating the call graph into the problem-solving process. It also precludes using the

source code to query/filter the information in the call graph or using the call graph information in a query on the source code.

What is potentially more useful is to use the call graph information in the query of the source code. Starting at a particular function in the call graph, source code that meets certain criteria is then extracted, with respect to both in the immediate function and in all functions that are called, directly and indirectly from that point. For example, the expressions containing a specific call, invoked directly or indirectly from a given function, can be extracted.

When presenting results involving source code queries, it is very important that the resulting source code be presented in the context that it originally existed [Van De Vanter 2002]. For queries integrating call graphs and source code, it must be possible to map from the call graph to some representation of the original source code.

Queries involving both call graphs and source code require locating functions and function calls in the source code. A function definition is a source-code construct represented in srcML with the element *function*. Even though the full AST of expressions is not represented in srcML, function calls are represented with the element *call*. A source-code example and equivalent srcML representation is given in Figure 6.9.

```
<call
  xlink:type = "simple"
  xlink:href = "…URI function g…">
</call>
```

Figure 6.8. Simple link is embedded in srcML. Only relevant srcML elements are shown.

6.4.4 Representing a Call Graph with XLink

The XLink [W3C 2001b] specification describes how attributes may be inserted into XML documents to represent links between resources in XML documents. A *resource* is an XML element or portion of an XML element (a resource may also refer to a non-XML element). A *link* is a relationship between resources and is represented by a *linking element,* an XML element conforming to the XLink specification and indicating that a link exists.

Linking is represented by the use of XLink attributes including *type*, *href*, the semantic attributes *role*, *arcrole*, *title*, the traversal attributes *label*, *from*, *to*, and the behavior attributes *show* and *actuate*. To permit integration with pre-existing set of XML elements, these attributes are in their own XLink namespace. The namespace prefix *xlink* is commonly declared in a namespace using the XML attribute *xmlns*. The XLink examples in the following sections will use the namespace *xlink* without showing the declaration.

In XLink, a simple link represents a unidirectional link from a single linking element to another single resource. The linking element serves as both the source of the

```
int f(int n) {
    return g(n);
}

<function>int <name>f</name>(int n) <block>{
       return <expr><call><name>g</name>(n)</call> </expr>;
}</block></function>
```

Figure 6.9. Function and function call (top) and equivalent srcML representation (bottom). Only relevant srcML tags are shown.

```
<function xlink:type = ”extended”>
    …
</function>
```

Figure 6.10. Extended link embedded in srcML function. Only relevant srcML elements shown.

link and the container for the link information. Any XML element can be a simple link as long as it contains the value of ‘simple’ for the attribute *type* and the attribute *href* with a URI (Uniform Resource Identifier) of the destination resource. A simple link representing a function call in function *f* to function *g* can be represented directly by adding attributes to the srcML element call as shown in Figure 6.8. The attribute *href* contains the source code location of the function *g*. The attribute *type* is given the value ‘simple’. If only simple links are used then the attribute *type* can be given a default value and left out of the linking element.

Simple links have less functionality than extended links and are more useful for existing linking mechanisms, such as in HTML.

Extended Links

The full capabilities of XLink are available with *extended links*, including bi-directional links, third-party links, and the ability to store the links separately from the document that contains the resources. All of these capabilities are applicable for representation of a call graph. Extended link elements are indicated by the value ‘extended’ for the attribute *type*, as shown in Figure 6.10. In the representation of a call graph, the srcML *function* element is used for the extended link because extended links may contain multiple internal links. This allows for the representation of the source of a call-graph link as a function, instead of from a *call* element in the function

```
<function xlink:type = "extended">
      …
  <call
    xlink:type = "resource"
    xlink:label = "call-g">…</call>
      …
</function>
```

Figure 6.11. Extended link embedded in srcML function with local call-element resources.

Any resource contained inside the extended link is a *local resource*. Each *call* element internal to a particular *function* element is a local resource of that function and is indicated by the value 'resource' for the attribute *type* as shown in Figure 6.11. Resources contain an attribute *label* used inside of the extended link to refer to the resource.

A resource that is not contained in the extended link element is a remote resource. The location of the remote resource is stored in a locator element. A *locator element* is any element with the value 'locator' for the attribute *type*. In the case of a call graph, the function being called is the remote resource. Because there is no existing source code element within the function that can be used to store this information, a new element, locator, is added as shown in Figure 6.12.

A locator element contains an attribute *href* with the value of the URI of the remote resource, i.e., the function definition being called. Each individual function that is being called would be represented by a single locator element. The attribute *label* contains the name of the remote resource to be used inside of the extended link.

Representation of the relationship between a function call and the called function definition requires connecting the local resource (the *call* element containing an attribute *type* with the value 'resource') to the remote resource (the *locator* element). An *arc*

```
<function xlink:type="extended">
  <locator
    xlink:type    = "locator"
    xlink:label   = "function-g"
    xlink:href    = "…URI function g…"/>
    …
  <call
    xlink:type    = "resource"
    xlink:label   = "call-g">…</call>
      …
  <arc
    xlink:type    = "arc"
    xlink:arcrole = "call"
    xlink:from    = "call-g"
    xlink:to      = "function-g"/>
</function>
```

Figure 6.12. Extended link embedded in srcML with arc connecting call to function.

element expresses the connection as shown in Figure 6.12. The representation includes elements for remote resources of functions that are called, local resources of calls, links between the calls, and the called function. The attribute *arcrole* on the *arc* element describes the meaning of the arc and is useful if multiple types of arcs are to be represented.

This representation of the call graph is embedded into the srcML representation of the source code. The representation consists of adding *xlink* attributes and the *locator* element to the srcML representation. The integration of the call graph representation into the source code representation limits the flexibility of the format. Separating the extended links from the srcML document and storing them in their own document would increase the flexibility of the representation.

External Linking

In third-party links the links themselves (the call graph or source model) are separated from the resources that they represent relationships about (the source code).

```
<function xlink:type="extended">
  <!-- Remote resources
       called functions -->
  <function
    xlink:type  = "locator"
    xlink:label = "function-g"
    xlink:href  = "…URI function g…"/>

  <!-- Remote resources of calls -->
  <call
    xlink:type  = "locator"
    xlink:label = "call-g"
    xlink:href  = "…URI call to g…"/>

  <!-- Link between call and
       function definitions -->
  <arc
    xlink:type  = "arc"
    xlink:from  = "call-g"
    xlink:to    = "function-g"/>
</function>
```

Figure 6.13. Third-party links in separate document with all resources remote, i.e., not embedded in srcML.

.

Instead of embedding the link information in the srcML we store the link information separately. The representation now must include elements for remote resources of functions that are called, remote resources of calls, and links between the calls and the called function.

All resources are now in the srcML document and are separate from the extended links, and therefore resources are now remote resources. All the *type* attributes with the value 'resource' now have the value 'locator' as shown in Figure 6.13. A document with link information can be separated into two documents: the original content and the extended links; the separate documents can be combined into a single document. A call graph can be extracted from the source code, or combined with the source code.

Multiple Third-Party Links

The abstraction of the source code that a call graph typically uses does not contain information about the actual call itself but only indicates which functions are called by a particular function. The relationship is between the functions and not the individual calls in the function.

This causes two changes in the representation. First, the function containing the call becomes a remote resource. Second, the calls themselves no longer have to be resources. The representation now includes elements for remote resources of calling functions, remote resources of called functions, and links between the calling and the called functions. A representation for multiple functions is shown in Figure 6.14.

Instead of the *arc* element, the *call* element is now a remote resource that represents the connection. The function that contains the call changes from the extended

```
<callgraph xlink:type="extended">
  <!-- Remote resources
       calling functions -->
  <function
    xlink:type  = "locator"
    xlink:label = "function-f"
    xlink:href  = "…URI function f…"/>

  <!-- Remote resources
       called functions -->
  <function
    xlink:type = "locator"
    xlink:label= "function-g"
    xlink:href = "…URI function g…"/>

  <!-- Links between calling and
       called functions -->
  <call
    xlink:type = "arc"
    xlink:from = "function-f"
    xlink:to   = "function-g"/>
</callgraph>
```

Figure 6.14. Multiple third-party links in separate linkbase document.

link to a remote resource. Every function that is used, either as a container of a call or as a called function, has a locator element.

A collection of related extended links can be stored in a *linkbase*. The only specification requirement for a linkbase is that it be an XML document consisting of extended links. As such, the call graph of a single source-code document or set of source-code documents can be stored in a single linkbase and the source code is in a separate srcML document. The call graph contains locator elements for the remote resources, i.e., functions, in the srcML document.

6.4.5 Mapping Call Graph to Source Code

All elements representing remote resources contain an attribute *href* whose value is a URI (Uniform Resource Identifier). The first part of the URI is the location of the XML document, e.g., the srcML document. The second part is the location within the XML document.

In this section the representation of the link from the call graph to the source code will be discussed including the representation of the location of the source-code documents, a brief overview of XPath, and the location of the particular function inside of the source-code document.

The first part of the URI is the location of the source-code document. In the simple case, this would be the name of the local srcML file. In a more complex case, this could be a remote URL of the document.

Function Location using XPath

The second part of the URI is the actual source code construct internal to the above named file, i.e., the function. The format of this part of the URI follows the XPointer specification, which is an extension of XPath.

Since XPointer is an extension of XPath, full XPath expressions can be used for this part of the URI. The extension of XPointer to XPath allows for addressing parts of element data, which is not needed in this case.

Specific locations in source code can be specified with an XPath expression that uses srcML elements [Collard, Kagdi, Maletic 2003]. To find all function definitions at the top level of a source-code document, the XPath expression */unit/function* can be used since the root element is *unit*. To extract function definitions at any level, including inside namespaces, conditional compilation sections, etc., the XPath expression *//function* does so by looking for the element *function* starting at the document root and looking through the entire XML document.

To refer to a particular function, we can use the predicate features of XPath and include the name as in the XPath expression *//function[name='g']*, which locates the function definition with a name *g*.

Specificity of Function Location

The exact XPath expression to locate a particular function will depend on what the call graph construction algorithm found. If the algorithm was lexically based, then parameter type information may not be available, i.e., all the algorithm knows is that a call to a function named *g* occurred. Since function names can be overloaded in C++,

this does not necessarily refer to a unique function definition. A lexically-based approach may also include a count of the number of parameters. If the algorithm was based on a full parse, then the type of each parameter may be known.

This wide range of situations does not cause a problem with this representation. Any degree of specificity that a construction algorithm can find can be represented. The XPath expression can include name, number of parameters, type of each parameter, and return type.

6.4.6 Integrated Source Code and Source Model Queries

The advantage of using XLink to represent source models, such as call graphs, includes using standard XML tools and technologies to query, transform, and even create the source model itself. Because both the call graph and the source code are in a related format, queries can involve both the call graph and the source code. The query may include only the call graph, use the source code to query the call graph, and employ the call graph to query the source code. The rest of this section will explore each combination of these queries.

For queries involving following links XPath alone is not sufficient. In these cases XSLT can be used. The following XSLT programs demonstrate the application of the query, but the focus is on the XPath expressions used. The XPath expressions shown can be used with most XML languages and tools.

Call Graph Queries

Many call graph queries are in a form that starts at a given function in the call graph and traverses the call graph finding what other functions can be reached, either directly or indirectly. These queries can be either direct or indirect calls from a function or to a function.

Extracting direct calls from a function, where the label of the starting function is *$start*, can be done by using the XPath expression:

/callgraph/call[@xlink:from=$start]/@xlink:to

The beginning of the expression, */callgraph/call*, matches all calls in the call graph. The predicate *[@xlink:from=$start]* selects only those calls that originate from the starting function *$start*. Once only the calls that start in this function have been isolated, the attribute *xlink:to* contain the labels of the functions that are called. The result of this XPath expression is the labels of all the functions that are called from the starting function.

Extracting direct calls to a function is accomplished by the XPath expression:

/callgraph/call[@xlink:to=$function]/@xlink:from

The uses of the *call* element attributes are reversed. The attribute *xlink:to* is used in the predicate and the attribute *xlink:from* in the rest of the path. The result of this XPath expression is the labels of all the functions that are called from the starting function.

Indirect calls may involve multiple traversals of the call graph. This requires repeated applications of the direct XPath expression on the results of the previous

```
<!-- recursive call -->
<xsl:template match="call[not(@xlink:to=preceding::call/@xlink:to)]">

  <!-- action performed at each function:  output the current function label
-->
  <xsl:copy-of select="@xlink:to"/>

  <!-- variable of called function label -->
  <xsl:variable name=”$called_function” select="@xlink:to"/>

  <!-- recursively call -->
  <xsl:apply-templates
select="/callgraph/call[@xlink:from=$called_function]"/>

</xsl:template>
```

Figure 6.15. Parts of a recursive XSLT program whose output is all functions called directly and indirectly starting at a particular point in the call graph.

application of the expression. Another tool, such as XSLT, can be used to apply the XPath expression recursively. A portion of an XSLT program showing a recursive application of the XPath expression for direct calls is shown in Figure 6.15. The program produces a list of the labels of all functions that are directly called by the starting function.

The XSLT template matches all *call* elements. The predicate *[not(@xlink:to=preceding::call/@xlink:to)]* only allows *call* elements with unique values for the attribute *xlink:to*. At each unique call the XSLT template can perform an action. In this case the action outputs the label of the function called, but it can easily be specialized to a particular task. After the action on the node is performed, the recursive call is made by using the XSLT *xsl:apply-templates* element on all the calls this time using the attribute *xlink:to*. This starts the next traversal on the call graph.

Either changing the action that is performed when the call is reached or by controlling under what circumstances the recursive call is made can modify this XSLT

program. Both modifications may involve the current *call* element, the *function* elements (both the source and the destination of the call), any other parts of the call graph, or related XML documents such as the source code in srcML.

Source Code in Call Graph Queries

The source code can be used in call-graph queries to extract a partial call graph where only the functions of interest are shown. Multiple views based on the source code can be produced from the same call graph. Each may be for a specific task where the views can be generated iteratively or on an ad-hoc basis.

The process of using source code in call-graph queries requires a series of steps to get from the label for a function that is contained in the call to the actual source code element. The steps will be discussed separately.

Function label to locator element: Starting with the function label *$label* we need to find the locator element with this label using the XPath expression:

/callgraph/function[@xlink:label=$label]

All the locator elements whose attribute *label* is equal to the function label are selected.

Locator element to source code elements: The attribute *xlink:href* of the function element (the locator) contains an XPath expression that points to the remote resource, i.e., the actual function definition in the source code. With the variable *$locator* as the locator element (the result of the previous step) the value of this attribute can be used in the XPath expression:

dyn:evaluate($locator/@xlink:href).

The attribute *href* is taken from the locator element to get the source code elements. The XSLT extension function *dyn:evaluate* dynamically evaluates an XPath expression and returns the elements that it matches.

This evaluation may return multiple function elements from the source code, depending on its specificity. This is not a problem, because any further evaluation will apply to all source code *function* elements that match.

Application of Source Code Element: The contents of the source code (a srcML *function* element) can be used as a filter by applying it as a test to the action part of the traversal program. The following XPath expression returns any *while* elements in the srcML element *function*:

$function//while.

The result of the source code XPath expression can be used to control the action in the traversal program by using the results as a test condition. This allows the control of the action based on the actual contents of the source code as represented in srcML. Since srcML has complete traceability to the original source code, the output of the call graph traversal can be dependent on the actual contents of the source code.

The above steps can be combined into one XPath expression. The following XPath expression will let us control the call graph query based on whether the function contains a comment with the string "logging" in it:

dyn:evaluate(/callgraph/function[@xlink:label=$label/@xlink:href])

//comment[contains(.,'logging')]

Any content of the function definition may be used to filter the call graph, including statements, white space, preprocessor directives, etc. In addition the source-code environment in which the function definition finds itself , i.e., containment by a preprocessor directive, namespace and location in terms of other code, can also be used. The source code can also be used to control whether the traversal of the call graph continues. An extension function to XPath can be used to hide the complexity of the linking from the call graph to the source code.

Call Graphs in Source Code Queries

The call graph extends the reach of source-code queries into the definitions of functions both directly and indirectly called. Source-code queries can involve not only the source code of the immediate function but all source code that may be executed, directly or indirectly, by calling this function.

In order to use the call graph in a source code query, we must find the locator of our source-code function in the call graph. The attribute *xlink:href* of all call graph *function* elements would have to be evaluated and used to filter out only the ones of interest, as in the following XPath expression:

/callgraph/function[set:has-same-node(evaluate(@xlink:href), $function)]

which finds the call graph function elements where the dynamic XPath evaluation of the attribute *href* is compared to the source code function element *$function*. The expression uses the extension function *set:has-same-node* which indicates if the source code function is reachable from the call-graph function locator. An XPath extension function

can be written that hides the complexity of the linking of the source code to the call graph.

At this point, we are in the call graph and can perform call-graph queries as described in the previous sections. The contents of all the functions called from this function are now available for querying. Any content of the function definition may be extracted. Examples include finding all expressions that contain a call to a particular function and finding the first expression that contains a call to a particular function.

6.4.7 Conclusions

The representation of a call graph in the XLink allows for the use of standard XML tools for performing queries on call graphs. The combination of this call-graph representation with an XML format, i.e., srcML, for the source-code document allows for the integration of call-graph information with the contents of the actual source-code documents.

The call-graph representation presented allows for representing multiple call graphs in a single or separate document. These can be applied to the same source-code document or any subset of the source-code document. Querying across all of the representations is allowed.

The representation presented may be easily extended to allow for the support of other source models by using other srcML elements (besides functions) as remote resources including class, namespace, declaration, etc.

6.5 Adoption of srcML

The motivation for srcML was to assist the greater software-engineering community in working with source code, including analysis, transformation, and editing. The format is currently being used by researchers outside of the original srcML developers. This section will describe some of their uses.

6.5.1 Aspect Weaving

A research group at the Automatic Control Laboratory of ETH-Zurich in Switzerland under the direction of Professor Schaufelberger used srcML to implement XWeaver, an aspect weaver for C++ [ETH 2004]. The overall architecture of XWeaver is shown in Figure 6.15 Element names of srcML are used to specify join points (points in the original source code where aspects can be woven into) and constraints on the join points. The aspects are woven into the source code using an XSLT transformation providing non-intrusive aspect weaving.

The XWeaver research group used the beta version of the srcML translator. Many improvements to the translator were made from their use on a commercial source-code base.

6.5.2 Reengineering UML from Source Code

The OMF (Object Modeling Framework) is a Master's Thesis project of Andrew Sutton under the direction of Dr. Jonathan Maletic. It is an API for the underlying meta-model of UML (Unified Modeling Language). The OMF allows a programmer to construct and manipulate the meta-model using the Python programming language.

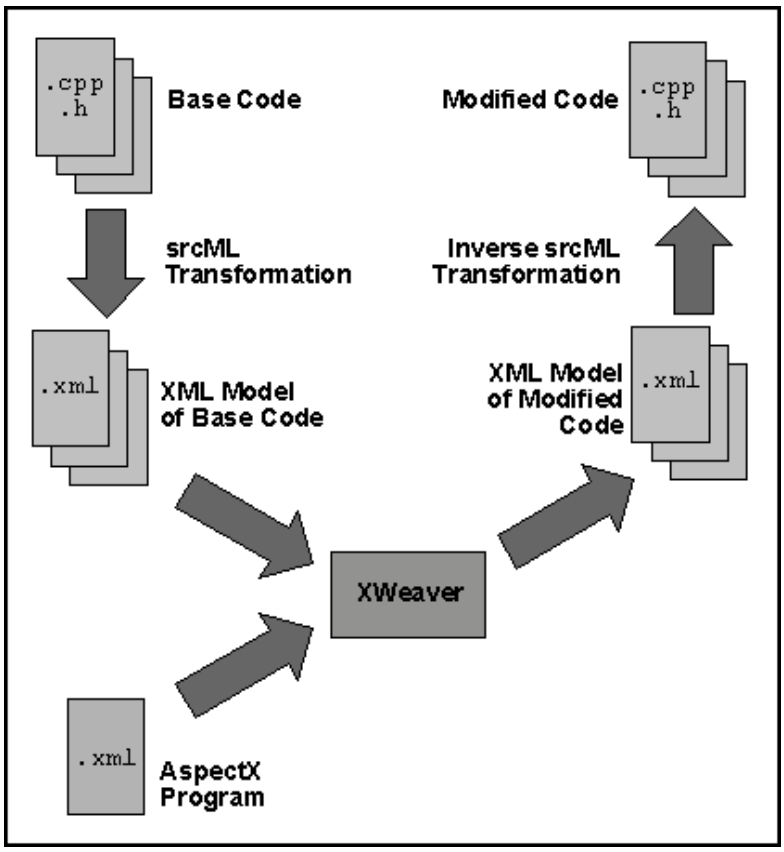


Figure 6.17 The XWeaver process uses the srcML translator to convert the base code into an XML model. The XML model is woven with the AspectX program into a modified XML model of the source code. This XML model of the source code is translated to modified source code. Diagram taken from [ETH 2004]

As an example of a re-engineering task that the OMF can support a program was written by Sutton that extracts an UML class diagram from source code. The program reads the srcML information into a Python program using a DOM API. The srcML information in the program is used to call the OMF API. A case study for this task was performed on the same HippoDraw application used for the meta-differencing case study in CHAPTER 9.

6.5.3 Supporting Source Code Editing

Two students at Wayne State University, Qun Xu and Yi Xie, under the direction of Dr. Andrian Marcus, are working on a source-code editor that takes advantage of the srcML format. The user works with the source code in the same manner that she is familiar with. Behind the scenes the source code is translated into srcML and used to enhance editing through the use of style sheets and other XML technologies. Currently srcML has been used in a component to .NET so the user can open a srcML file and the corresponding C++ code editor will display the text. The user can view and edit both the srcML and the C++ source code.

CHAPTER 7

Case Study: Fact Extraction

A case study was performed to validate whether srcML, a format with a definite document-oriented approach, could be applied to data-oriented tasks. The task selected was fact extraction, i.e., the querying of programs and source code. For this task a previously existing benchmark for C++ fact extraction was used.

A lightweight fact extractor was created that used 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-compilable 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. The work presented in this section supports this proposal by demonstrating:

- A translator exists that converts from C/C++ source code to srcML

- The translator can handle irregular source code
- A Context-Free Grammar view of source code can be represented in srcML
- The srcML format and translator form an infrastructure to support further processing and refinement
- XPath can be used as a query language for source code
- Common XML tools and technologies can be successfully used with srcML
- The srcML file format has a reasonable file size
- Source-code addressing can be converted in both directions from line number to XPath expression.

7.1 Fact Extractors

Source-code fact extraction is the process of extracting facts, entities, and the relationships, from source code given a specific query. It involves processing (e.g., parsing and/or searching) the source code to extract the particular facts, expressing a desired query, and formatting the output of the query.

Fact extractors are a vital tool for reverse engineering, re-engineering, maintenance, testing, and general development of software systems. They are used to help developers comprehend software by uncovering relationships between classes, modules, units, functions, etc. Fact extractors can be of great benefit in locating possible errors in source code, as well as identify concerns of interest across a system.

In the approach presented here, we convert C++ source code into an XML representation, namely srcML [Collard, Maletic, Marcus 2002; Maletic, Collard, Marcus 2002]. This underlying representation is then leveraged via the APIs, tools, and

technologies of XML to give us a lightweight, robust, and tolerant C++ fact extractor. We use the term *lightweight* to highlight the fact that only lightweight parsing is done and a number of very low-level type facts can not be directly derived from the data source (i.e., srcML markup of the C++ source).

Our method allows the extraction of high-level entities such as functions, classes, namespaces, and templates, as well as middle-level entities such as individual statements (*if*, *while*, etc.), declarations and expressions. Lower-level entities such as variables and function calls can also be queried. Additionally, it allows the extraction of entities that are typically discarded during pre-processing, such as comments, preprocessor directives, and macros. The entities are extracted with full lexical information, such as white space and all original source code information.

The following section will address some of the problems encountered during fact extraction and address the related work in the field of fact extraction. We then describe srcML and our C++ to srcML translator. Additionally, we briefly address related XML source code representations. Our approach to using XML technologies to support fact extraction is then detailed and lastly the results of applying our method to a fact extraction benchmark [Sim, Holt, Easterbrook 2002] are given.

7.1.1 Extracting Facts from C++ Code

A number of challenging, well known, technical problems exist for building fact extractors for C++ [Dean, Malton, Holt 2001; Ferenc et al. 2001; Sim, Holt, Easterbrook 2002]. Additionally, the work done by Sim et al [Sim, Holt, Easterbrook 2002] and the benchmark they developed uncovered a number of other problems relating to the types of

questions and perspectives of the users of fact extractors. The results of researchers applying tools to this benchmark revealed that there are often many correct answers to the same question. The correctness depends on the perspective of the user and their particular software engineering task. Different tasks require different levels of detail about the system to support the particular type of comprehension necessary to complete the task. For example, a user may be interested in variable, type, or comment information while trying to understand a group of modules for reverse engineering. Another user may need the possible level of function call nesting and dynamic typing for fault localization.

This gives credence to fact extractors with very different capabilities and complexities. While many fact extraction tools rely on a complete parsing of the entire system, we have chosen another avenue that, we believe, augments those approaches.

One of the most important issues of fact extraction is the input itself (i.e., the source code). It is typically a single source code file and associated include files. In the best case, you have a complete, compilable system. In other cases there may be code fragments, compilation problems, or missing associated include files. The source code can be in a dialect of the original language(s), or it can be code that will compile under one version of a compiler but not another. Of particular interest here is that our approach allows fact extraction in most of these later situations. This can be a distinct advantage in many situations (e.g., platform change, library change).

The C++ language, in particular, is a challenging language to parse and extract facts from. This is due, in part, to the preprocessor and the numerous macro constructs

that are used in conjunction with the language. Also, there are a great many versions and variations of C++ in wide use. However, the biggest problem facing someone wanting to construct a fact extractor is that C++ is defined by a non-Context Free Grammar. This makes full parsing of the language difficult, and lexical analyzers may produce incorrect results.

The other critical part of fact extraction is with respect to the query. The input to the fact extraction process is the specification of the desired fact. This may be in the form of a query language, or it may require a specialized program to extract the answer. A simple specification input may be limited to what facts can be extracted. However, both of these approaches require a learning process on the part of the user.

Another issue for the specification is what level of understanding the tool has for the language being processed. For example, does the tool already know how new types are introduced into the language or is it a part of the specification to list the language constructs that can form a new type. A very flexible tool may require configuration. The amount of configuration and the configuration language affect both the usefulness and the difficulty of using the tool.

Finally, we must consider the output of the fact extractor. If the extractor is being used as part of a larger process, such as for source model generation, then the output format of the fact is important to the ease of using the result. If the extracted fact, such as a section of source code, is to undergo further processing, then the tool should be able to output the extracted facts in its original format.

7.1.2 Characterization of Fact Extractors

Murphy [Murphy, Notkin 1996] describes an approach to source-model extraction using the terms lightweight, flexible, and tolerant. Here, *lightweight* means that extracting a new fact requires a relatively short specification. With respect to process, this includes not only the search specification but the configuration of the tool. *Flexible* refers to what information can be extracted from the original source code, such as comments, macros, etc. This also includes using this information to select the extraction, such as function extraction with regard to some particular content of their comments. *Tolerant* refers to how complete are the source-code documents. There could be missing include files, the code may not compile, or a dialect of C++, such as from a particular version of the compiler, may have been used.

For XML to be used in the context of a lightweight fact extractor, the XML must also be processed in a lightweight form. This requirement is for both the XML markup language used and the time/memory requirements.

Our representation, srcML, and the translator we have developed use a lexically-based approach allowing the translator to be used on incomplete, non-compilable code, and code fragments. Since this can generate incorrect results in certain cases, it has been constructed to allow for additional refinement of the translated result. These refinements include information from associated source-code files and heuristics from the user.

7.1.3 Related Work on Fact Extraction

Parser-based fact extractors include cppX [CPPX 2001], Acacia [Acacia 2001], and Columbus/CAN [Ference et al. 2002]. LSME (Lightweight Source Model

Extraction), described in [Murphy, Notkin 1996], is a tool for generating high-level source models using a specification language based on regular-expressions. The user can specify what she wants to match in the source code or other system artifacts and the actions that she wants performed. The system will produce a scanner that generates the system model. The specification is small and only has to be written for the needs of the particular source model that is being generated. There are no restrictions on the artifacts that the scanner can be applied to, and there are few constraints placed on the condition of the artifacts.

In [Murphy, Notkin 1996] the comparison is made between lexical- and parser-based approaches to source-model extraction. The parser-based approaches are described as heavyweight when an extractor for a new language needs to be generated, and, as such, these approaches are typically inflexible with regard to the constraints placed on the kinds of artifacts and are not tolerant of the (poor) condition of the source code.

The lightweight characteristic applies to the creation of a new extractor. A distinction is not made between creating a new extractor for a new source-code language and creating a new extractor for the extraction of a new system model.

In [Cox, Clarke 2000] the categorization of [Perry 1987] is used to show the LSME (Lexical Source Model Extraction) lexically extracted unit-level models but not syntactic-level models. In [Cox, Clarke 2000] this work is extended by using both lexical and parsing techniques and by comparing the results. The extraction is of individual entities, such as function definitions, calls, statements, expressions, etc. Their

comparison shows that lexically-based approaches can produce useful results of the entity extraction that is performed.

7.2 Utilizing srcML and XML for Fact Extraction

A combination of tools was used for benchmark fact extraction. To convert to the srcML format the srcML translator, *src2srcml*, was used. To execute XPath expressions on the srcML file, the command line tool *xpath* was used. However, any XPath enabled tool can execute the XPath expression. The tool *xpath* was chosen for its simplicity and easy integration to the fact-extraction process.

The typical output of an XPath query is an XML document fragment. Fact-extraction queries often return part of the source code or the line number of where the source code is located. Because of the direct traceability of srcML, the document fragment can be directly translated back to the original source-code fragment. The tool used to do this is *srcml2src*. It converts from srcML back to the original source code. This is a simple script using *stripsgml*, which is a part of the perlSGML package.

XPath statements refer to specific points in the srcML document. Fact extraction questions, including those in the benchmark, often ask for a line number or line count as the answer. The XPath statement can be directly translated into the line number in a particular document. The tool that was used is *srcpath2line*. It translates from an XPath statement to a line number in a source file and is a simple program written in the event-driven XML transformation language, STX.

Any conversion from srcML to another format, such as the original source code, line number, etc., becomes the last stage of the fact-extraction process.

7.3 C++ Fact Extraction Benchmark Results

In order to determine the needs of a fact extractor, the CppETS 1.1 [Sim 2002] benchmark for C++ fact extractors was used as a test bed. This benchmark has been applied to many of the parser-based fact extractors previously discussed and is a good choice since it helped to define exactly what was meant by fact extraction.

The benchmark consists of 19 test buckets in the category of accuracy and 10 in the robustness category. There are a total of 99 questions. The file sizes ranged from 46B to 47KB, and the corresponding srcML representation ranged from 851B to 63.2KB with a ratio ranging from 1.251 to 7.586.

The srcML translator and the XML tools for extraction described in the last section were applied to this benchmark. The remainder of this section describes these results.

7.3.1 Format of the answer

The benchmark had a variety of ways to format the output. In some cases, such as for a statement, the requested output should be the actual code. In other cases the line number, range of line numbers, and number of bytes is requested.

The form of the output did not affect whether the fact is extractable or not. The XPath expression to extract the answer is applied to the *xpath* tool when the actual code is requested and to the *srcpath2line* tool when the line number is requested.

There were a small number of questions that requested a byte count. We do not have a specific tool to calculate this type of value.

7.3.2 Entities in isolation

Many of the questions were concerned with the direct extraction of entities with no information regarding their context in the source code. Since these are directly marked with tags in srcML simple XPath expressions, with the *xpath* tool, they were able to directly extract these entities using XPath in the manner described in the last section. For example, to extract the named namespaces, the XPath expression *//namespace* was used. The entities extracted in this way include variable declarations and uses, function declarations and definitions, preprocessor directives, namespaces, and templates. These were primarily questions that referred to entities that the programmer defines.

7.3.3 Entities in context

For other extractions, the context of the entity was important. For example, the same element names are used in srcML for any type of variable declaration for global variables, local variables, and class data members. To find these specific categories the context of the element must be used, e.g., if a class data member was requested, then the XPath expression *//class//decl* was used. This is part of the tradeoff over using specific elements for specific uses versus general elements for general usage. In some cases the context must be used.

7.3.4 Scope & Type

The issue of scope was too complex in all but the simplest cases to solve in a straightforward manner with the tools that we used. An example is with a C++ *friend* function, linking the use of a variable to the variable declaration in the class.

A similar problem occurs with questions involving the type. Simple type questions are relatively easy to solve by extracting the type from the declaration. But given a use of a variable in any statement, it is too complex to determine its type. This would have required a (partial) symbol table to be built requiring a further processing stage.

7.3.5 Entities extracted using string matching

There are examples of programming constructs that do not directly relate to a keyword or special symbol. For example, pure virtual functions have the "= 0" at the end of their function declarations.

Since all the original text is preserved, it is possible to detect this very specific textual pattern for a pure-virtual function by comparing the text at the end of the virtual-function declaration. XPath does include string matching and may include regular-expression matching in a future version. However, this was not included in the results since it seemed to be a hack around something missing in srcML.

7.3.6 Extraction with missing files

The robustness test buckets contained examples of missing files, including missing include files and libraries. With the tools that are used, the missing files did not affect the result of answering the questions. A difference would have been seen had an additional tool that applied XPath expressions across files were to be used. This would have increased the number of questions that this approach could have answered in the other buckets.

7.3.7 Preprocessor

The preprocessor directives are straightforward to extract. However, it is beyond the scope of the tools used to attempt to extract the source code that is to use given arbitrary values for the preprocessor symbols.

This is one area where a specialized tool could be built that worked on the srcML representation. The tool could go through all preprocessor directives and keep track of their current values. This new tool could have answered all of the preprocessor directive questions fully.

Table 7.12. Summary of C++ fact extraction benchmark results compared to previous results presented at IWPC'02

	Fact Extractor	Full Answer	Partial Answer	No Answer
Previous Benchmark Results	Acacia	32%	16%	52%
	Columbus	19%	11%	70%
	Cppx	45%	19%	35%
	TkSee/SN	28%	18%	54%
	srcML Translator	44%	8%	48%

7.3.8 Dialects

The robust section included different dialects of C++ and the results were mixed. For the test bucket with MS Visual C++ extensions, the extraction was successful. For the g++ extension test bucket the extraction was not successful.

The difference is the nature of the language extension. The MS Visual C++ extension includes the addition of a type keyword. Since srcML has a generic type and specifier element it handled the added keyword easily. The g++ language extension is not an added type or specifier keyword, and it did not fit into the existing grammar as easily.

The extractor failed the IBM Visual Age test bucket not due to a language extension, but to the use of a macro.

7.4 Conclusions

The results of applying our lightweight fact extractor to the benchmark are quite reasonable in comparison to the published results of the other types of tools. The other tools (Acacia, Columbus, Cppx, and TkSee/SN) were able to answer approximately the same number of questions. However, the application of the tools to the benchmark resulted in [Kienle 2002] large improvements for a number[4] of the tools.

Since the output format is in srcML, it is easily converted to other required formats, such as another XML format and even back to source code. Additional processing can take multiple fact extractor results and combine them to form higher-level source models. The output can also be converted to the required input format of tools that can use these results, i.e., visualization tools, source model generators, etc.

---

[4] As presented during a working session at IWPC 2002.

Given the lightweight approach used here, this is a reasonable and simple method for many tasks of fact extraction. Our results also indicate that the approach will scale well due to the reasonable srcML file sizes and the capability of event-driven translation.

CHAPTER 8

An XML Representation of Source Differences

The most important part of realizing meta-differencing is the representation of source-code differences. This chapter presents srcDiff, a new representation of source-code differences.

The description of srcDiff begins by placing the representation in context with a related work section on differencing, including both source code and XML differencing. Details of the srcDiff format are presented, and then followed by a method for creating the format itself.

8.1 Related Work on Differencing

There are a number of purposes for differencing including comprehension, patching, merging, and automatic change detection. The varying purposes affect what is considered a difference, and what format is used for the resulting representation.

The related work on differencing includes the work on source-code differencing, and XML differencing. The following subsections will look at each of these separately.

8.1.1 Source Code Differencing

Algorithms for merging are categorized [Hunt, Tichy 2002] into line based, process based, structure based, and semantic based. The close alliance between the format for representing differences and patching/merging makes these categories relevant

to this work. Line-based differencing is equivalent to character-based differencing. In process-based differencing a special editor is used to carefully record all changes to code. All changes are special operations such as “rename”. Since very little code is developed using editors of this type, the approach has limited applications. In structure-based differencing the two versions of the source code are parsed into parse trees, and the difference is performed on the parse tree. However, the algorithm typically knows very little language-specific information, and much of the lexical structure is lost. The semantic-based algorithms are concerned with what the program computes, not how it is written. However, in most cases of using difference information, what we are concerned with is how the program is written.

Current mechanisms for source-code differencing center on computing textual differences between two files or comparing the parse trees generated for the source code. In order to find the changes that occurred between two existing source-code documents, either textual, syntactic, or semantic differencing can be used [Conradi, Westfechtel 1998]. Most commercial tools use textual deltas with syntactic and semantic deltas used for operations such as merging [Mens 2002]. The most popular differencing algorithms and tools, i.e., UNIX utilities *diff* [Hunt, McIllroy 1976] and *patch*, take a character-based document view of source code files. In the case of the extraction of textual differences using *diff* [Hunt, McIllroy 1976] (or a variation), line differences between two files are found with added/deleted/changed lines recorded by their line number in the original file and by changed content. The lines in the file are compared using the LCS (Longest Common Subsequence) algorithm [Hunt, Szymanski 1977], and is computed relatively

efficiently. Because the comparison is character-based, the algorithm can be applied to any text file. Since the algorithm ignores the underlying syntax of the source code, it is very robust and tolerant of source-code irregularities.

The UNIX utility *patch* can be used to generate a modified version of the document, using the original version and the output of *diff*. In the extraction and application of differences the character-based view used in both the difference extraction by *diff* and the application by *patch* allows these utilities to preserve all of the original textual information and context of the source code.

Differences can also be found at the syntactic level. In general, this is a difference comparison of two ASTs. Syntactic differencing has been primarily used for improving merging of differences [Mens 2002]. The main drawback is that the algorithm must know the syntax of the source code, and the difference comparison is typically not as efficient as the LCS algorithm.

Of particular interest is the work on LTDIFF [Hunt 2001], where the LCS algorithm is used on sequences in the parse tree combining the advantages of syntactic information with the efficiency and results of the LCS algorithm. In its worst case the resulting difference algorithm is as efficient as the LCS algorithm. This approach has been applied to merging in ELAM (Extensible Language Aware Merging) for syntactic merging. Hunt identifies that full support of non-language constructs, such as comments and preprocessor directives, are necessary for full practical application of syntactic differencing [Hunt 2004].

In semantic differencing code segments are considered equivalent if they perform the same computation. The approach used by Horowitz [Horwitz, Reps 1992] was limited to a simplified language without any procedures. The Semantic Diff tool [Jackson, Ladd 1994] worked at the procedure level and only considers a comparision between the input and output to the procedure. Any changes in syntactic structure are not detectable. The general case is undecidable, therefore, limited heuristics have been developed [Hunt, Tichy 2002].

Of particular interest is very recent extension of semantic differencing to object-oriented programs. The work in [Apiwattanapong, Orso, Harold 2004] augments a CFG (Control Flow Graph) to represent object-oriented constructs such as inheritance hierarchies. For example, at any point in the program if an object calls a method, and the class of the object is in an inheritance hierarchy that was changed, then the program considers the method call to be a difference. While useful for higher-level views of a program, such as determining code coverage, this is not useful for a programmer view of the source code. This view is at odds with the purpose of object-oriented programming in general. The programmer created an inheritance hierarchy to separate this concern from the source code at the location of a method call. In addition, this technique takes a top-down approach to what has been changed by comparing lists of names of classes, methods, etc. from the multiple versions. Renames of these entities are seen as adds and deletes.

In [Magnusson, Asklund, Minor 1993], fine-grained version control is used in a collaborative software environment. Individual elements inside source-code documents

are selected for version control. The elements used are then set for the entire project. In the Fluid project [Wagner, Graham 1997], a fine-grained version control infrastructure for trees, such as ASTs, is provided. Both of these cases are complete source code management systems and are not directly compatible with other versioning systems.

Differences are also of interest in the area of mobile code systems where the need to be able to update code easily is necessary. The model CodeWeave has been proposed for fine-grained mobility of code [Mascolo, Picco, Roman 2005] and is implemented using XML for a simple programming language [Emmerich, Mascolo, Finkelstein 2000].

8.1.2 XML Differencing

All XML documents are in a textual format and it is possible to apply the *diff* utility to them. However, the results are not appropriate to many XML applications. There has been a great deal of interest in a new set of difference utilities that can be more appropriate and take more advantage of XML.

The *XML TreeDiff* [IBM 1998] project at IBM was a set of JavaBeans that generated a difference in the XUL (XML Update Language) format. XPath was used to refer to positions in the document, but the XPath expressions used the lowest-level numeric references to nodes. The project is officially retired.

The tool *xmldiff* [Microsoft 2002a] is a proprietary Microsoft product that generates a file in the proprietary format XDL (XML Diff Language) [Microsoft 2002b]. The XDL format records any changes needed to the original document. It goal appears to be use with the included patch tool. The algorithm used is not described.

Another *xmldiff* [Logilab 2003] applies the EZS algorithm and is implemented in Python. It produces output in the XUpdate [Laux, Martin 2000] format. The format was a project the XML:DB XML Database Initiative [XML:DB 2002] but work seems to have stopped on the format and the web site (www.xmldb.org) no longer exists. XUpdate was designed as an edit script language and although it used XPath provided no context information. In addition, it was not possible to mark changes to parts of a text node, which creates a difficulty for differences in comments.

The algorithm X-Diff [Wang, DeWitt, Cai 2003] takes an unordered tree-model approach to XML differencing where siblings are not considered. It produces a tree-to-tree correction edit script with a time complexity of $O(n^2)$. This approach may be useful to source-code differences for program constructs that are explicitly named (e.g., functions, classes) but not for fine-grained differencing of statements where order does matter.

The DUL (Delta Update Language) [Mouat 2002] is the output format of the tool *diffxml* [Mouat 2004]. The DUL language is a separate edit script description that can be applied to the XML document. The tool relies on the algorithm *mkdiff* with a time complexity of $O(n^2)$.

The company DeltaXML [DeltaXML 2004a] built its entire product line around its difference product. The tool is called *DeltaXML* [DeltaXML 2004c] and produces its own format [DeltaXML 2004b]. This format is of special interest. Difference information is added as attributes on existing elements indicating if the element is unchanged, deleted, or added. No attribute is available for moved data. However,

because of the use of attributes, every element in the difference file must have an attribute, even if it is nested inside an element of the same difference category. Also, white space is ignored, which must be due to the absence of an explicit element to place an attribute on. The comparison can be applied in an ordered or unordered manner.

The *diffmk* utility [Walsh] was originally written by Norman Walsh in Perl. It has recently been rewritten in Java. The utility applies the LCS (Longest Common Subsequence) algorithm to a flattened tree. It produces a document that shows the differences in context of the original documents that is designed for viewing.

8.2 srcDiff Representation

The srcDiff format is a single multi-version source-code document with additional elements to mark version differences. The original and modified versions of the srcML representation of the source code are integrated with version differences. Differences are marked by XML elements inserted into the multi-version source-code document around the syntactical elements that were added or deleted.

Figure 8.2 shows a side-by-side example of a modified source-code document with the original on the left and the modified on the right. In the original version, deleted source code is displayed with a strikethrough. In the modified version, added source code is shown in bold. The example shows the change in the filename of the preprocessor directive *include* and a change in the block comment of the year from 2003 to 2004. In the function, the enclosing if statement is deleted but not the contents of the block. In addition, two statements involving the variables *total* and *product* are inserted.

Figure 8.1 shows the srcDiff equivalent. Only the relevant srcML markup is shown for brevity.

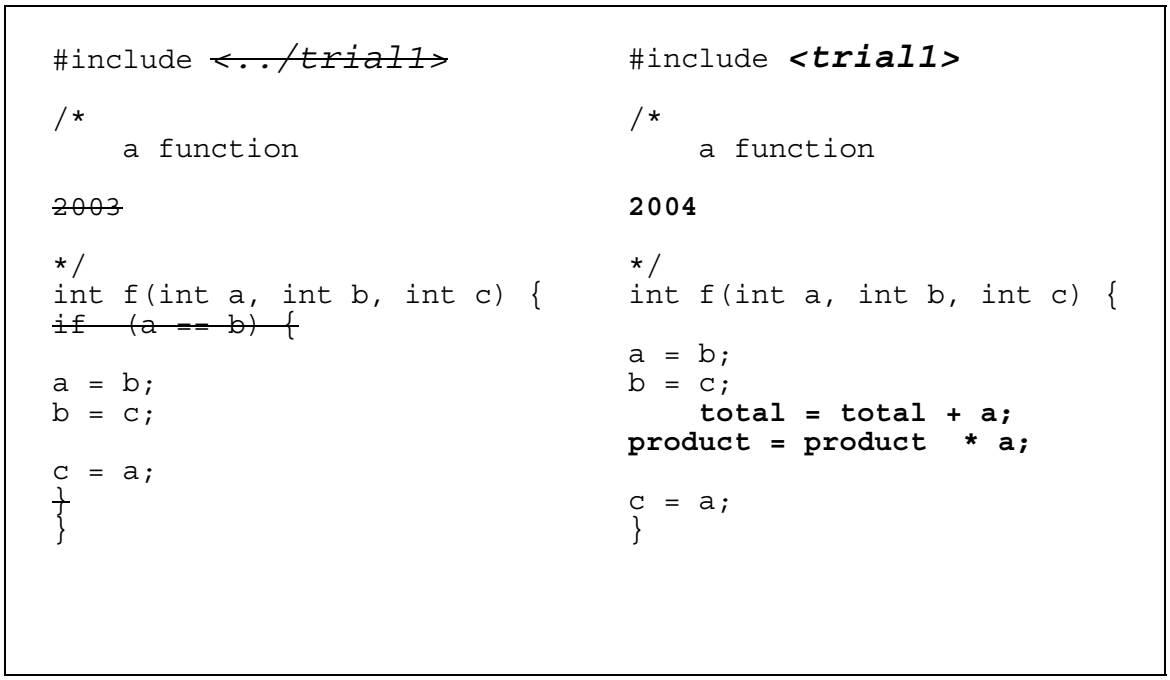


Figure 8.1. Different versions of the same source-code file. Textual deletions are shown in strikethrough; textual additions are shown in bold.

```
<diff:common>
<diff:old><cpp:include># include &lt;../trial1&gt;</cpp:include>
</diff:old><diff:new><cpp:include># include &lt;trial1&gt;</cpp:include>
</diff:new>

<comment type="block">/*
    a function

<diff:old>2003</diff:old>
<diff:new>2004</diff:new>

*/</comment>
<function>int f(int a, int b, int c) <block>{
<diff:old><if>if (a == b) <block>{
<diff:common>
        a = b;
        b = c;
<diff:new>    total = total + a;
        product = product * a;
</diff:new>
        c = a;
</diff:common>}</block></if>
</diff:old>}</block></function>
</diff:common>
```

Figure 8.2. A srcDiff program fragment with selected srcML markup. Textual deletions are shown in strikethrough; textual additions are shown in bold.

In order to construct this single document, we must address the merging of the versions and at the same time deal with marking up the differences. The results of these steps must produce a well-formed XML document. The merging of srcML versions must produce well-formed srcML, and the difference elements must be inserted while preserving the well-formed property.

8.2.1 Difference Elements

In order to mark where changes occurred in the srcDiff document, difference elements are inserted. The difference elements are in their own namespace to allow for easy detection, apart from the srcML elements of the source code. In the following examples, the namespace alias *diff* is used. The namespace *diff* is composed of three elements are summarized in Table 8.1.

Table 8.1. Elements added to srcML documents to mark common, deleted, and added elements and text.

Difference Elements	Original Version	Modified Version
diff:common	Yes	Yes
diff:old	Yes	No
diff:new	No	Yes

Sections containing text and markup that are in both versions of the document are placed inside of *diff:common* elements. Sections containing text and markup that has been deleted are enclosed in *diff:old* elements. And sections containing text and markup that has been added are enclosed in *diff:new* elements.

It is often necessary to nest these elements for well-formed XML. For example, if we delete an *if* statement but not the statements inside the *if* statement block, then we require an element *diff:old* around the if statement itself with a nested element *diff:common* around the statements inside the block.

The difference elements, i.e., *diff:**, form a difference axis of the document. We can determine which difference elements contain any point in the document. Interesting parts of the document can be extracted by looking at the parent of an element along the difference axis. Elements with a parent of element *diff:common* along the difference axis are the elements that exist in both documents, elements with a parent of element *diff:old* along the difference axis are the elements that only exist in the original version of the

document, and elements with a parent of *diff:new* along the difference axis are the elements that only exist in the modified version of the document.

Both the original and modified srcML versions of the document can be extracted from the srcDiff multi-version form of the document. The original document is the text and markup with a parent of element *diff:common* or element *diff:old* along the difference axis. The new version of the document is the text and markup with a parent of element *diff:common* or element *diff:new* along the difference axis.

8.2.2 Mapping Changes to Difference Elements

A textual difference, such as a deleted line or range of lines, maps to a range in the srcDiff document. If the contents of the srcML elements (i.e., all elements and text not in the difference axis) in this range is well formed, then the textual difference can be directly marked using the appropriate difference element, i.e., elements *diff:**. However a single textual difference may cross-cut the syntactic structure of the program, e.g., a line containing the start of a block is deleted while the contents of the block remain. We cannot directly mark this range of the document within a difference element, because it will lead to a document that is not well-formed and thus not in XML. A single textual difference (range) and a single srcML element interact in the following ways:

well-formed - The srcML element is totally contained in the range, i.e., both the start and end tags of the element occur between the two points,

start tag only - The srcML element starts in the range (the start tag is contained in the range) but does not end inside the range (the end tag for this element is not in the range), and

end tag only - The srcML element ends in the range (the end tag is contained in the range) but does not start inside the range (the start tag for this element is not in the range).

If only one of the tags for a single srcML element is not inside of a range of a textual difference, then the other tag must occur in a textual difference earlier/later in the document. If not then the version of the document related to this difference (the original version and element *diff:old* or modified version and element *diff:new*) will not be well-formed. When cross-cutting occurs, combinations of textual differences must be correctly matched to each other so that the appropriate difference element can be wrapped around the section.

8.3 Extracting srcDiff

The srcDiff format is constructed using the original and modified version of the document in srcML. The output of the utility *diff* between the original textual versions of the documents is used to control the combination of the two versions and to mark where changes occurred.

The srcDiff format is generated with a Python program using the libxml2 library for XML processing. Because the simplest way to write the program is as a stream, the DOM (Document Object Model) was not used. The SAX (Simple API for XML) was considered, however it was not used since working with multiple input streams is difficult to write using the event-driven model of SAX. Instead of these traditional XML APIs the "pull" API TextReader was utilized. TextReader allows stream access to the document, which makes it straightforward to navigate through multiple srcML documents

simultaneously. Like SAX it does not store the entire document in memory at one time which makes it more efficient for large documents.

Both srcML documents are processed in sections common to both versions, they are taken from the original srcML document in sections of deleted text, as well as from the modified srcML document in sections of added text. The decision to change sections is controlled by the textual *diff*, while the question whether to embed the new section or return from the current is based on the current markup. These section changes are marked with the difference elements while the srcML output is from the appropriate input srcML document stream.

In general, the textual differences control which section the srcDiff translator processes. These textual changes match well to the srcML markup since it closely follows the text. However, since the textual differences do not take into account the srcML markup some special cases have to be handled - for example, the end tag of a line comment.

8.3.1 Requirements for Difference Sections

There are multiple ways that the differences between the source-code documents can be combined while still preserving the original document ordering of both documents. The guidelines include:

Common Sections - Since we are most interested in documents that are primarily similar, common sections (i.e., sections with content that is in both versions) should be as large as possible. This includes the document as a whole and any common sections nested inside deleted or added sections;

Pure Deleted Sections - In the line difference format contiguous sequences of deleted lines form a single difference. Therefore in sequences of purely-deleted elements the section should be as large as possible; and

Pure Added Sections - Analogous to purely-deleted sections.

These sections are straightforward to maintain when each separate statement is either in a common, purely-deleted or purely-added section. However, source code differences often do not occur in such clean sections as do line differences. Therefore, a statement view of the differences must also be taken into account.

8.3.2 Line Difference Format

The srcDiff translation process uses the output of the utility *diff* for line differences. The utility *diff* is run on the original and modified text source-code documents. In this section the format of the line difference information will be explained.

Line differences are in three parts. The first and third parts are line sequences in which the changes occur. The first part is the sequence of lines from the original document, and the third part is the sequence of lines from the modified document. The second part is the type of change: addition, deletion, or change. These types are encoded in a single character in the following manner:

d – Lines of text deleted from the document, i.e., lines of text that occur in the original document only;

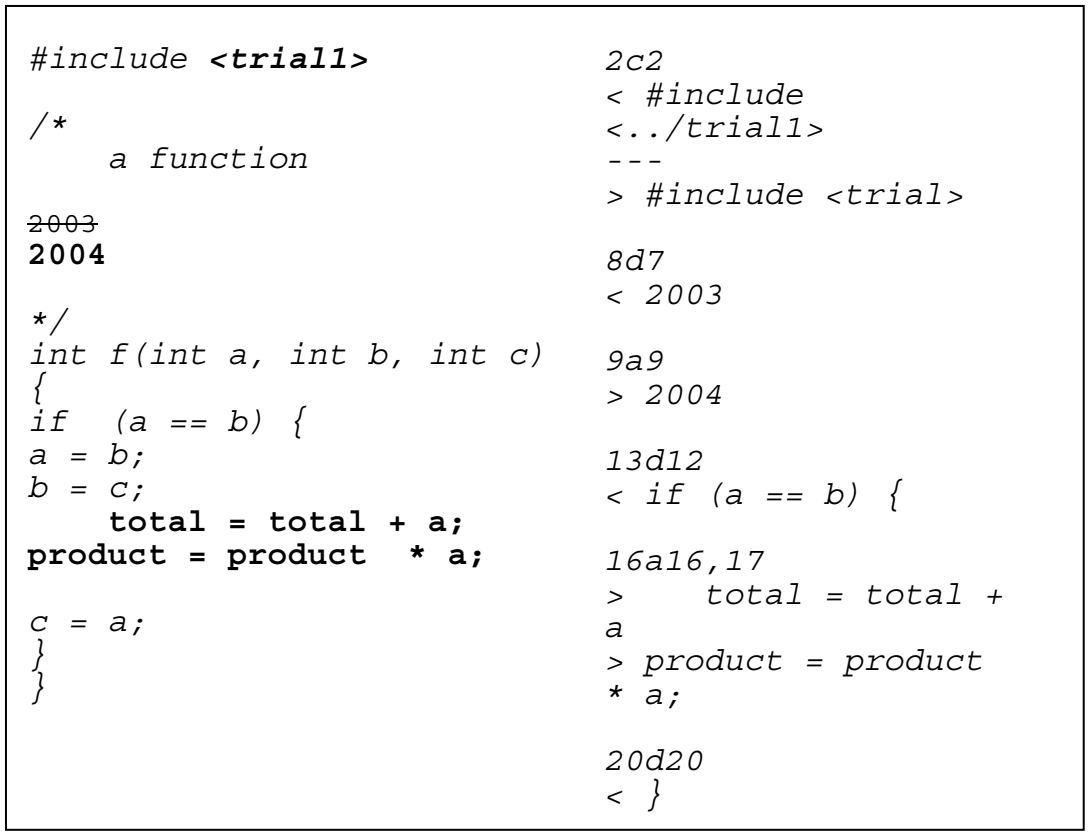
a – Lines of text added to the document, i.e., lines of text that occur in the modified document only; and

c – Lines of text that were replaced in the document, i.e., lines of text that occur in the original document and were replaced in the modified version.

The *diff* utility can also show the lines of text that were changed, however this information is redundant to us (since we have both versions of the document) and is ignored.

8.3.3 Extraction of srcDiff from Source-Code Documents

The srcDiff algorithm requires three input sources: the original version of the source-code document in srcML, the modified version of the source-code document in srcML and the textual differences between the text source-code documents. The algorithm for converting source code into the srcDiff representation is a pipeline process with three main sections: annotation of srcML input streams using *diff* information, the interleaving of these multiple srcML streams into a single XML stream, and the insertion of difference markup. The multiple versions of the source-code document are produced using the srcML translator. The textual difference is a result of applying the utility diff to the original source-code documents.



```
#include <trial1>

/*
    a function

~~2003~~
2004

*/
int f(int a, int b, int c)
{
if  (a == b) {
a = b;
b = c;
    total = total + a;
product = product  * a;

c = a;
}
}
```

```
2c2
< #include
<../trial1>
---
> #include <trial>

8d7
< 2003

9a9
> 2004

13d12
< if (a == b) {

16a16,17
>    total = total +
a
> product = product
* a;

20d20
< }
```

Figure 8.3 Combined view of the original and modified source code on the left with the corresponding output of the utility *diff* on the right. Textual deletions are shown in strikethrough; textual additions are shown in bold.

8.4 Processing of Separate srcML Streams

The line-difference information can be applied to the original and modified versions separately. For the original version this information indicates what remains and what was deleted. For the modified version this information indicates what remains and what was added. Therefore, the line differences can be mapped to the two versions separately before the versions are combined. Accomplishing this is best done as a multi-stage process.

8.4.1 Single Line Nodes

In order to simplify further processing, it is desirable that a particular node in the stream exit only on a single line. However, the text nodes provided by most XML APIs (including the TextReader API used) may include multiple lines, e.g., the new line at the end of an element combined with the text at the start of the next line.

During the processing of the srcML streams, text nodes that exist on multiple lines are split into multiple text nodes each containing a single line of text. This allows the rest of the processing to assume that each node is only on one line.

8.4.2 Node Annotation with Line Difference Information

Line-difference information is added to the nodes in the srcML streams from both versions of the document. This information is based on the text of the source code. This may be directly mapped to the srcML start elements since they start on the same line that the text they enclose begins on.

The srcML stream for each version of the document is annotated separately. For the original version, the deleted and changed (i.e., difference type of *d* and *c*) line-difference information is used. For the modified version, the added and changed (i.e., difference type of *a* and *c*) line-difference information is used. The appropriate line numbers in the tuple are used, i.e., the first line sequence for the original version and the second line sequence for the modified version.

8.4.3 Adjusting for the Well-Formed Property

The srcDiff format contains all of the elements of both the original and modified version of the source code simultaneously. Elements that are common to both, deleted, or added are combined in the same document. This must be done in such a way as to preserve the well-formed properties of the resulting XML. A srcML element from the original document interacts with a srcML element from the modified version of the document in a number of ways.

First, the elements may not overlap at all. One element starts and ends before the other begins. This occurs when a complete element is deleted or added to the document. For example when a complete statement is added or deleted.

<if>...</if>...<while>...</while>

Or, one element may be completely nested inside the other element:

<e1>...<e2> ... </e2>...</e1>

This occurs when an element is inserted inside an existing element, or an element inside an existing element is deleted, e.g., a statement is inserted or deleted from the inside of a block.

<if> <expr_stmt> </expr_stmt> </if>

Or, two elements share a common ending point:

<e1> ... <e1> ... </e1>

This situation occurs when the start of a statement is deleted in one document and added in another, e.g., a *while* statement is replaced with an *if* statement but the contents of the statement (the contents of the block) remained the same:

```
<if><block> <comment></comment>
<while> ... <block>

</block></while></if>
```

The ending element of the block is shared between both and cannot be easily extracted because of the comment after it.

For the well-formed cases, i.e., no overlap and nested cases, integration of the two versions is straightforward. The problem case of shared elements must be detected and handled appropriately. A solution to the shared element is simply to duplicate the ending block:

```
<if><block><comment></comment>
<while><block>
                ...
</block></while></block></if>
```

The adjustment for shared elements is handled in the individual srcML streams by making sure that the difference annotation on srcML elements is well formed.

8.5 Interleaving srcML Streams

The individual, difference-annotated srcML streams are interleaved to form the srcDiff document. This is done by taking a statement view of the srcML stream. The individual documents are viewed as statements and content around statements.

The term *version state* is used to indicate whether that particular content is in common, is added, or is deleted.

The general rules for integrating the streams are:

- Keep common, added, and deleted sections as large as possible.
- Complete statements in a single version state should be contiguous.
- Header of nesting statements (e.g., if statement) in a single version state should be contiguous
- Nesting statements with a header in one version state

8.5.1 Documentary Structure

The documentary elements of white space and comments are treated differently depending on where they are placed. This subsection will explain how they are handled between statement parts. This will be further explained in a later section, but for the purposes of this section assume that we are discussing documentary structure at the top level of the document (i.e., not inside of a statement).

The documentary elements are processed in a natural ordering. Whatever line differences are found have been made well formed by the previous stages and can be handled in the following manner:

1. Output all common documentary structure

2. Output all deleted documentary structure
3. Output all added documentary structure

The difference element section wrapping that takes part at the final stage will make sure that these parts are placed in correct order.

8.5.2 Simple Statements

The *simple statements* are the statements that do not contain any nested blocks or other statements or declarations. They specifically include expression statements, break statements, typedef declarations, function declarations, variable declarations, etc. They specifically do not include function definitions, class definitions, conditional statements, iterative statements, etc.

Simple statements are placed in a single version state, i.e., they are completely in common, added, or deleted. This corresponds to what line differences typically find, and is necessary for statements such as an expression statement.

8.5.3 Complex Statements

The *complex statements* are the statements, declarations, and definitions that have nested content, i.e., function definition, class definition, conditional statements, etc. The complex statements are handled separately in order to allow for changes to their structure (or content) without unnecessarily marking changes to their content (or structure).

Figure 8.4 shows examples of situations where the processing of complex statements must be treated carefully. Conditional statements can be wrapped around existing statements (i.e., existing statements are placed into a block of a new conditional

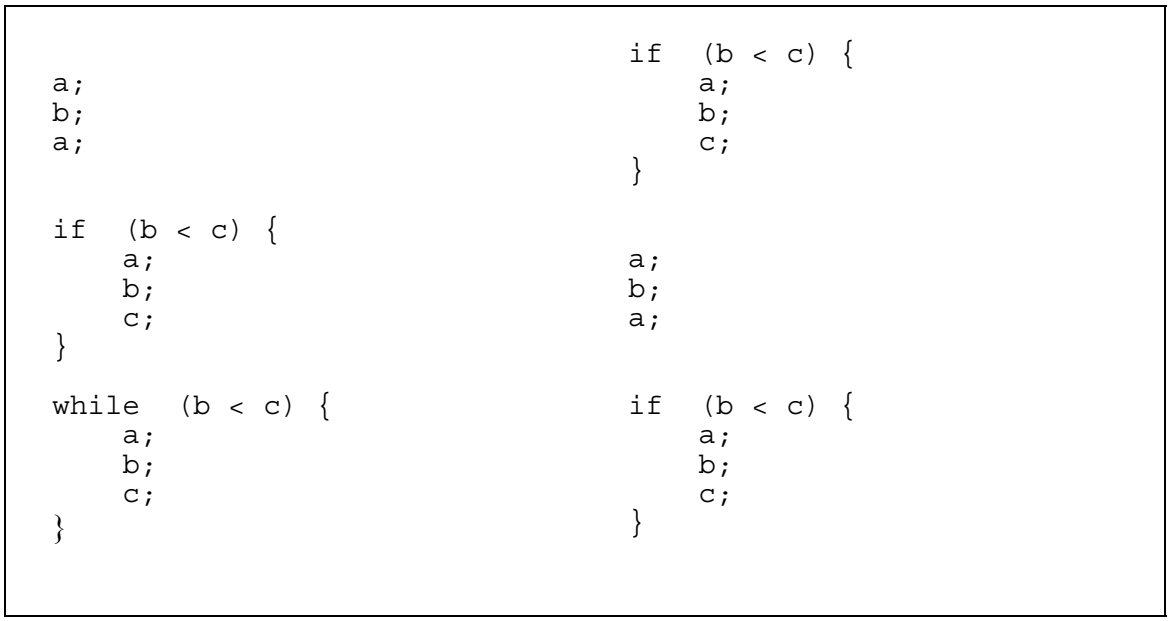


Figure 8.4. Examples of changes in complex statements with preservation of nested data. Old code is on the left; new code is on the right. The top example shows an if statement wrapped around existing group of statements. The middle example shows the removal of the nesting statement. The bottom example shows a while statement replaced with an if statement.

statement). The wrapping conditional statement may be removed without altering the contents (i.e., an if statement is removed, but the contents of the nested block remain the same). And the wrapping conditional statement may be replaced with another nesting statement.

In order to handle the nested contents of these complex statements separately the header of a complex statement is processed separately from the contents. The header is treated the same as a simple statement, and the body is processed as if they were not nested. This allows for added/deleted wrapping of complex statements to be handled separately from the nested content, and the nested content to be dealt with separately.

8.5.4 Wrapping Comments

The textual differences match almost directly to the srcML elements. For an individual line of source code, if the text is in common (i.e., unchanged) then the srcML is also in common. The general rules are:

- If text on a line is unchanged, then elements that are completely contained on that line are also unchanged and
- If an element changes, its textual content must change.

This is due to localized parsing of translation to srcML. The markup depends on the immediate context, not on distant changes. This is not the case for semantic differencing or any other approach that stores meta-data in the elements. The meta-data is derived from not only the immediate context but from data gathered at potentially distant parts of the program. A common example would be type information at the site of variable usage.

The only exception where common text does not imply common elements is where lines of code have been wrapped in a comment, i.e., “commented-out”. These are sections of statements, declarations, etc. which in one version were marked accordingly,

while in the other version were wrapped in a comment and became plain text as far as the translator could detect.

In order to preserve the markup of statements in commented-out code, when in common sections of comments any srcML markup in either stream is placed in the output stream, even if it only appears in one stream.

8.6 Complexity

The input, annotation, and correction to make the elements well-formed of the srcML streams are linear. A stack is used to keep track of open elements. The statement interleaving does require that a statement (or the header of a statement for a nesting statement) be completely read before processing. This uses a queue and never stores more than a single statement (statement header) at a time.

The entire srcDiff process with srcML streams and the output of the *diff* utility is linear. Since the srcML translator is of linear complexity, the entire srcDiff process from text source code to srcDiff format is no more than the complexity of the *diff* utility.

CHAPTER 9

Case Study: Meta-Differencing HippoDraw

Meta-differencing is the extraction of information about differences between versions of source code and the context which they are in. It includes the number of changes, the number of changed entities, which entities were changed, and where did the changes occur. In other words, it is fact extraction applied to differences.

In order to validate the meta-differencing approach it was applied to a complete application. This chapter describes the application of meta-differencing to an open-source C++ application HippoDraw. Meta-differencing questions were asked that are not easily possible with current approaches to difference analysis.

9.1 Background of HippoDraw

HippoDraw [HippoDraw 2004] is an open-source application providing a data-analysis environment. The most recent version contains approximately 60 KLOC of source code in over 400 C++ files.

For the case study, versions 1.4.0 (posted January 10, 2004) and 1.5.1 (posted February 2, 2004) were used. They represent major releases of the source code, and at the time of the experiment were the two latest versions. HippoDraw is an active project and since the time of the case study versions 1.6.1 through 1.9.5 (posted June 22, 2004) have been released.

9.2 A srcDiff Representation for HippoDraw

In order to perform meta-differencing of HippoDraw, all of the source code had to be converted into the srcDiff representation. The case study was performed on a 3GHz PC running Linux. The study started with the initial source-code versions.

The source code for both versions was first converted to a srcML representation using the updated srcML translator. The translation was applied to each file separately. The total time for translation of all 422 files from version 1.4.0 and all 423 files from version 1.5.1 was under two minutes.

The utility *diff* was then individually applied to each original file from version 1.4.0 and its matching modified file from version 1.5.1. Files were matched based on the identical directory and file names. The total time to perform the textual differencing was under 2 minutes for all pairs of 422 files.

After the srcML versions and textual-difference information was obtained the srcDiff format could be generated. Of the original 422 pairs of files, 144 files had actual changes. The srcDiff translator, src2srcDiff, was applied to each of these 144 files. APPENDIX C contains a complete list of these files, including the sizes of the original source code file and the equivalent srcML document for each version, and the size of the resulting srcDiff files. The data in the table shows that there was only a modest increase in srcDiff file sizes over that of the srcML representation of the Version 1.4.0 source-code document with the largest increase a factor of 1.43.

The total time to translate from the srcML versions of the files to the srcDiff representation was under 10 minutes for all 144 files with changes. As a result, the

overall time to convert the entire HippoDraw application from text source code to the srcDiff format was under 14 minutes.

9.3 Addressing and Querying Source Code Differences

Because srcDiff is directly based on srcML meta-differencing can be performed using the same approach as in source-code fact extraction, i.e., we can apply XML tools to perform the various tasks. Identifying changes in srcDiff documents is an extension of querying of srcML documents. XPath statements to find the program items of interest mixed with difference elements allow us to automatically determine the kind of changes that occurred. In the following sections we will explore how we can query source-code differences.

When addressing or querying source code alone, all elements can be treated equally. Distinction between different source-code elements is based on the specific problem being solved. However, with the addition of the difference elements it is natural to consider the source code and the difference elements as two distinct groups and treat them that way in addressing and querying.

As with the call-graph querying in Section 6.4, we have to look at the interaction between the source code and the difference elements. In most cases we will want to query the source code using the difference information. First, we can take the source code and determine what specific changes it includes, i.e., additions and deletions. Second, we can take the source code and determine what specific changes it is included in.

In the following sub-sections we will examine the queries from both perspectives. We will let *$source* refer to a srcML XPath expression (i.e., the location of *$source* in the source code).

9.3.1 Source Code Containing Specific Changes

We can determine if specific source-code locations have contents in specific versions, i.e., do the source code contents contain any deletions or additions? To find the source code that is in a specific version, we look at the context of the source code in terms of the difference elements. An XPath expression can be used to find source-code entities with deleted text:

$source[.//diff:old]

The path *.//diff:old* refers to any difference element with deleted items at any descendant of the current location. By using it in a predicate, we are selecting parts of the code in *$source* that include deletions. To select parts that include additions, the element *diff:new* can be used in the predicate, and to select parts of source code that include text in both versions, we can use the element *diff:common* in the predicate

9.3.2 Source Code Contained in Specific Versions

We can also determine if specific source-code locations are contained in specific versions. To find the source location *$source* that is in a specific version, we look at the context of the source code in terms of the difference elements. An XPath expression can be used to find deleted source-code entities:

$source[ancestor::diff:[1] = ancestor::diff:old[1]]*

The predicate is examining the difference axis, given by the expression *ancestor::diff:**. The difference axis is a subset of the ancestor elements. Specifically, it is the ancestor elements that are difference elements. The index of 1 refers to the direct parent along the difference axis (XPath starts indexing at 1). The deleted ancestor axis is given by *ancestor::diff:old*. The comparison is checking whether the direct parent along the difference axis is a deleted section. Added source code can be found in a similar manner by comparison to *diff:new*.

9.4 Case Study Questions

The case study attempted to find answers to representative questions regarding changes to a source-code document. Each question is only a starting point for additional queries, and it is important to judge not only whether the question can be answered. The broader issues for each question are whether it can be answered at all; can more precise questions be answered, and can the approach be expanded to other syntactical elements. The questions were:

Q1: Does the change only affect comments?

This answers the basic question of how the differencing approach (and representation) handles comments. Can the approach detect comment changes? Can these comment changes be put in context of syntactical elements e.g., function definitions? And, can we match the comment changes back to their original place in the document?

Q2: Are new methods added to an existing class?

This answers the basic question of how the differencing approach (and representation) understands the context of unchanged and changed elements. Can the approach detect syntactical elements that are unchanged? Can the approach detect changed elements in this context? And, can we match both levels of elements back to the original place in the document?

Q3: Are there changes to preprocessor directives?

This answers the basic question of how the differencing approach (and representation) handles preprocessor statements and the multiple views of the program that they represent. Can the approach detect changes in preprocessor directives? Can the approach detect what context the preprocessor statements are in? And, can we match these preprocessor statements back to their place in the source code?

The following sections will answer these questions separately. For each question the query and form of the result will be given and discussed.

9.5 Detecting Documentary Changes

Current differencing approaches are only able to answer this question in a limited form. Textual differencing does not directly understand the syntactic context of a line change and cannot directly determine if all line changes are in or only contain comments without visual inspection. Semantic differencing does not store or find changes in comments because they are not concerned with document information, which does not change the semantic meaning of the program.

There is a limited, indirect approach to answering this specific question for textual differences. To do so requires that the files be processed in the following ways:

1. Generate the modified file by applying the textual differences
2. Remove comments from both the original and modified source code file using the preprocessor or a simple script
3. Compare using a textual difference program.

If the files, with the comments removed, are identical then the changes were only made to the comments; if not, then the changes were also made to other syntactical elements. This approach is based on the ease of removing comments from a program due to their simple syntax. Therefore, it is not extensible to other similar questions, e.g., questions about other entities in the program. Even with the same entity, i.e., comments, the query cannot be refined to focus on individual parts of the source code, e.g., the changes to comments inside functions.

Identifying changes in srcDiff documents is an extension of querying of srcML documents. XPath expressions to find the program items of interest mixed with difference elements allow us to automatically determine the kind of changes that occurred. XPath statements can be used to find comments with deleted text:

//comment[.//diff:old].

Added text is found by comparison to *diff:new*. To find comments that are in a changed section, we need to look at the context of the comment in terms of the difference elements. XPath expressions can be used to find deleted comments:

//comment[ancestor::diff:[1] = ancestor::diff:old[1]]* .

The index of 1 refers to the parent along the difference axis, since XPath starts indexing at 1. Added comments can be found in a similar manner by comparison to

diff:new. To determine if a change contains anything other than comments, the following expression is used:

//[not(comment)][ancestor::diff:*[1] != ancestor::diff:common[1]]* .

It is more direct to use XPath within an XSLT program to filter comment changes and to see what remains. By applying an XSLT filtering program based on these XPath statements to the srcDiff files for HippoDraw, we can automatically determine that the change from version 1.4.0 to 1.5.1 for the file ColorPlot.h only includes comment changes.

9.6 Detection of New Method

This question tests whether the differencing approach can differentiate changes in declarations. Current differencing approaches have a limited ability to answer this question and expand upon it. Textual differencing does not directly understand the syntactic context of a line change and cannot directly determine if all line changes are in a method, let alone a class. Semantic differencing is able to answer this question, but cannot expand upon it or provide a format for further processing..

With meta-differencing in order to find new methods, we first determine what classes exist in the original version and have not been deleted. We use the following XPath expression to extract the existing classes by finding all classes and then detecting the version that they are in:

//class[ancestor::diff:[1] = ancestor::diff:common[1]]* .

Once we have the classes an XPath statement is then applied to each class to determine the names of any added methods:

//function_decl[ancestor::diff:[1] = ancestor::diff:new[1]]/name.*

The utility *xpath* allows us to execute the first XPath statement to find the class and then to apply the second XPath statement to the result. By applying the above XPath statement to the srcDiff files for HippoDraw, we can automatically determine that the change from version 1.4.0 to 1.5.1 added the methods *setZoomPan* and *isZoomPan* to the class CutController in the file CutController.h.

9.7 Detection of Preprocessor Statement Changes

The fundamental question for this type of change is "Are there changes to preprocessor directives?" Semantic differencing is not able to answer this question due to practical issues and philosophy of what a change is. Preprocessor statements often form a multi-program view where the specific view depends on the value given to preprocessor variables at the command line or in the included files, e.g., preprocessor values identifying compilers such as *GCC* or *MSVC* in include files. Semantic differencing views the source code after preprocessing, i.e., after the preprocessor directives have been removed and a single view of the program has been produced. The change of definition of a preprocessor directive can change the code that is used and even the type of a preprocessor symbol. A single change in a preprocessor directive can have many changes in the semantic meaning of a program, including the type, etc. of a variable.

With textual differencing it is not difficult to use regular expressions to differentiate between changes that are to preprocessor directives, and those that are not. Preprocessor statements are (usually) all on one line and textual differencing does not

remove any text before processing. But this approach does not extend to other program entities, and cannot be used with other program entities to focus where the change occurred in the context of the syntactic elements.

This query can be applied to any srcML element e.g., function, class, etc. The source code location can be the result of any srcML query. In srcML, preprocessor directives are in a separate namespace from the other language elements allowing them to be more easily identified as a group. All preprocessor directives are in elements of the form *cpp:** when the namespace alias *cpp* is used. Since preprocess directives are on their own lines, they can be directly extracted. All deleted preprocessor statements can be matched using the XPath statement:

//cpp:[ancestor::diff:*[1] = ancestor::diff:old[1]]*

All added preprocessor statements can be matched using the XPath statement:

//cpp:[ancestor::diff:*[1] = ancestor::diff:new[1]]*

Specific directives, e.g., *include* directives, can be found and queried separately using an XPath expression of the form:

//cpp:include[ancestor::diff:[1] = ancestor::diff:new[1]]*

By applying the above XPath statement to the srcDiff files for HippoDraw we can automatically determine that the change from version 1.4.0 to 1.5.1 included a change to a preprocessor statement, specifically a change in an include file. The file FunctionController.h additionally includes the file *axes/AxesType.h*.

9.8 Conclusions

The approach leveraged XML tools, and because of this, the work is very adoption-oriented. It can be easily utilized by the open-source community. In general it also benefits the field of Software Engineering by supporting both the analysis of individual source-code differences and version histories. This supports the automatic identification of the specific syntactic nature of a change (e.g., an *if* statement was changed), the creation of software metrics based on a change, and they assist in impact analysis based on changes. Developers/researchers can readily analyze what is actually happening during the practice of software development.

The case study shows that the meta-differencing approach can be performed fairly efficiently. The majority of the time was taken by creation of the srcDiff representation from the srcML representation (and line-difference information). This should have taken much less time since the entire process is linear. The time could be because the translator was implemented in Python. The libxml2 TextReader API, which the meta-difference translator used, is also available for C/C++ and the srcDiff creation program could be implemented in C++, which can lead to a major speedup.

Since the work is based on srcML, it is limited to C/C++ source code. However, C/C++ source code, with its non-CFG (Context Free Grammar) and use of a preprocessor, provides a particularly difficult case compared to other languages. The results that are generated by the application to C/C++ will be applicable to other languages, particularly procedural and object-oriented languages. In most cases these other languages will prove to be easier.

The other limitations to the research are identical to other non-parser based approaches. The information gathered is often only approximate without a symbol table and the existence of all necessary source-code files used to produce a running program. However, the architecture of the srcML translator supports additional steps in the translation where this missing information can be collected and integrated into the format. It is also possible to perform further analysis and integrate this information into the srcML representation.

CHAPTER 10

Application of Meta-Differencing

The case study on meta-differencing demonstrated that queries/fact extraction can be performed on source-code differences while still supporting a complete document view of source code. However, the case study did not take full advantage of difference information in a document view.

The result of a meta-difference query can be at any point in a wide range of abstraction levels, from the lowest level of the actual source code to whatever abstract view is required for the particular task. This preservation of the lowest abstraction level allows a meta-difference query to be easily integrated into source code/difference transformation, pattern matching, and constraints.

This chapter will give examples for some of the tasks that meta-differencing can be applied to. It uses the querying and transformation capabilities of meta-differencing for use with specific tasks. The flexibility given by meta-differencing and the document-oriented approach allows for each task a variety of approaches that can be used. Because of this, each task is discussed with an example of a particular way of accomplishing the task.

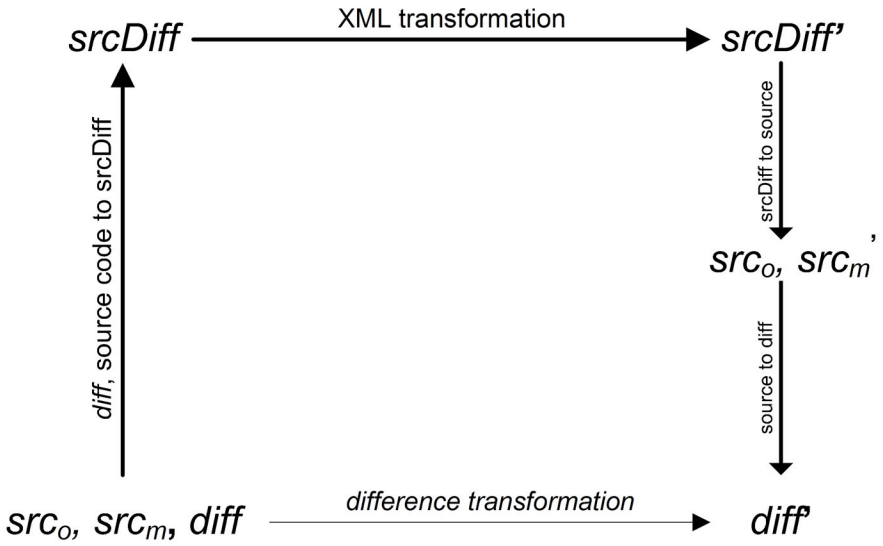


Figure 10.1. Overview of difference transformation presents how XML transformations can be used to perform transformations on textual differences. The textual difference is converted into a srcDiff representation, transformed using XML and then converted back to a textual difference.

10.1 Transformation of Source-Code Differences

Transformation of source-code differences is an extension of transformation of source code in srcML. Figure 10.1 shows an overview of the difference transformation. First, the original textual difference, *diff*, and the original, src_o, and modified, src_m, versions of the source code are converted into an equivalent XML representation, *srcDiff*. Then, XML transformations are applied to *srcDiff* to create the transformed difference, *srcDiff'*. After the transformation, a new modified source-code document, src_m', is extracted from the transformed difference, *srcDiff'*. At this point the new version of the modified source code can be directly used. It can also be combined with the original version of the source code to create a new textual difference, *diff'*.

Because of the transparency of srcML transformations on source code the difference transformation is also transparent. No original textual information is lost. If the XML transformation is an identity transformation, i.e., *srcDiff* is identical to *srcDiff',* then src_m is identical to src_m'.

This transparency of the difference transformation allows for a variety of ways to use the results of the transformation. The following sections will discuss some of the applications and the approaches used.

10.2 Dividing Differences

Differences between two versions of source code are typically generated at intervals or times manually decided by the developer. In a version-control system, the programmer decides when to enter in, e.g., *commit* in CVS terms, a modified version.

This may lead to a change whose version granularity is too large, i.e., the developer did not enter an update into the versioning system at a frequent enough interval or at the (in hindsight) appropriate times.

When another developer is given these differences they are limited to a file granularity. They can either apply or not apply the entire difference. However, a single difference may contain changes to multiple places in the source-code document, and many types of changes. This can create difficulties in integrating the difference.

One difficulty of a difference with too large a granularity is that it may be difficult to test. Assume a previous version worked correctly (i.e., test suite executed successfully). Then, the new changes are applied. If the modified version does not work correctly (i.e., the test suite executes unsuccessfully) then it is difficult to know what part of the changes caused the problem. If the difference was partitioned, i.e., it was a set of changes instead of just one, then the individual changes could be applied, tested and verified individually.

Another problem is unnecessary coupling between the changes. In this case the differences serve more than one purpose, e.g., differences that both fix a bug and correct a spelling error in an unrelated comment. Both of these changes are for different purposes and belong in separate changes. Not only might we want to apply these changes separately, we may also want to describe and classify them separately for purpose of analysis.

In this section we will demonstrate how a single difference can be divided into two individual differences that, when performed sequentially, performs the same series of changes.

10.2.1 Dividing Process

An overview of the dividing process is given in Figure 10.2. The process starts with the initial text difference. This difference (as well as the srcML versions of the source code) is lifted to the srcDiff representation. The dividing of a difference is now an XML transformation on the srcDiff format.

Like the queries themselves, the division of differences can be based on source-code entities or locations. Source-code entities can be all comments or all changes of a specific syntax item, e.g., all names. Locations can be any area in the source code, e.g., a particular function, a particular class, or a range location of statements.

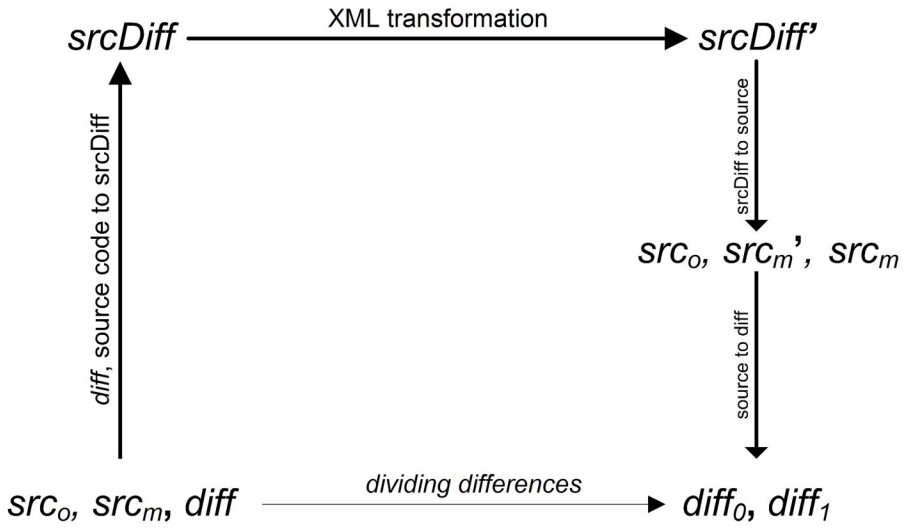


Figure 10.2. Using srcDiff XML transformations on differences can be used to divide a difference. First, the difference is converted to the srcDiff format. Second, the srcDiff undergoes an XML transformation. This modified srcDiff can be used to extract a new version of the source code. This new version has only the changes that remain in the new srcDiff. In addition, the intermediate version of the source code can be used to generate divided differences.

10.2.2 Dividing by Syntactical Element

The first example will be dividing a difference by a particular syntactic element. The syntactic element for the example is the *comment* element. Comment changes, both deletions and additions, will be allowed, but other deletions/additions will not be. We want to separate comment changes and only allow them to go through. The XSLT program will be based, as were the other transformation examples, on a default copy that preserves all text and elements in the transformation, except for the ones that we want to change.

To preserve only the comment changes, we need to filter out all non-comment changes. The XPath expression that will match all non-comment nodes in inside of the *diff:old* section is:

diff:old/node()[name()!='comment']

The expression *node()* matches all elements and text nodes. The predicate compares the name to make sure we are not in a *comment* element. The XPath expression is put as the match on a null-template which filters out these elements. The following is the XSLT template that must be added to the identity XSLT program in order to filter the deletion of all non-comments:

<xsl:template match=" diff:old/node()[name()!='comment']"/>

Allowing only comments to be added can be accomplished by introducing a null template with *diff:old* replaced by *diff:new*.

10.2.3 Dividing by Location

Another way of dividing differences is by location, e.g. a particular function. To split by location, we can select a particular location and allow only changes, i.e., difference elements *diff:old* and *diff:new*, in that location. The XPath expression that will filter all deletions outside of a function with the name *sort* is:

node()[ancestor::diff:[1]=ancestor::diff:old[1] and*

not(ancestor::function[name='sort'])]

This expression matches all nodes which are directly in a deleted section and are not inside of the function with the name *sort*. Filtering all additions can be done in a similar manner, and the exact context can be easily changed.

10.3 Difference Pattern Detection

It would be useful to be able to detect patterns of differences. This would allow you to take a code difference and discover if it meets certain criteria. As with the difference transformations, it is possible to detect patterns both by syntactical element or location. Examples include detection of differences to a given set of functions, to only documentary structure, or to name changes only inside of a particular function definition.

It is possible to construct queries directly on the srcDiff documents. However, the transformation of differences provides a somewhat easier approach. The basic steps are:

1. Construct a srcDiff document from the differences
2. Transform the srcDiff document allowing the changes in the pattern
3. Generate a textual difference from the srcDiff document
4. Apply the textual difference

5. Compare the results to the previous difference.

In this way the transformations can be used to detect specific patterns of differences.

10.4 Difference Constraints

A difference constraint is a rule that the change must follow. Any of the difference patterns could be part of a constraint. Constraints form a standard for the validity of the document, in this case the validity of the changes to a source-code file. As with pattern detection, queries can be used to state constraints on a document. In addition pattern detection can be used as a constraint, and the procedure to follow is similar.

A more interesting approach is to use a schema. APPENDIX B includes the schema for the srcDiff representation in the form of a DTD. The srcDiff DTD is not a strict schema in that it extends the srcML schema to allow the use of the difference elements at any point in a srcDiff document. For particular purposes it might be useful to have a more constrained schema. For example, the schema might only allow difference elements to contain srcML comment elements, or only allow difference elements inside the body of function definitions, i.e., no changes allowed to the header of the function.

Forming a stricter schema for srcDiff in a DTD would be particularly difficult. It would require that the base srcML DTD were more easily extensible than it is and would require a DTD written in the manner of the XHTML Modules [W3C 2000a]. However, a RELAX NG schema for srcML is available and RELAX NG is much easier to extend and to specify patterns.

10.5 Conclusions

The combination of srcML marking locations of program entities, and srcDiff marking where changes occur reduces dividing differences to the process of selecting the correct XPath expression to serve as a filter. This section has only started to explore the possibilities for dividing differences.

CHAPTER 11
Conclusions and Future Research

This work presents meta-differencing, an infrastructure for conducting analysis of source-code differences. XML technologies and a document-oriented source-code representation provide the underlying platform for analysis of differences for a wide range of software-evolution tasks.

11.1 Main Results

The research defines a document-oriented XML representation for source code, srcML. This format supports a full document and data view of source code including addressing, querying, and transformation. The application of srcML to a standard C++ fact-extraction benchmark showed that by leveraging XML, a robust, light-weight approach to source-code querying can produce results comparable to less tolerant heavier-weight (i.e., parser-based) approaches. Improvements in the robustness and speed to the srcML translator showed that the approach can be easily applied to industry-sized applications.

The research defines an XML representation of source-code differences, srcDiff. This format extends the srcML representation to multiple versions of a source-code document. A translator is presented that uses the source-code documents, along with the

textual differences between them, to form the srcDiff representation with a complexity no greater than that of textual differences.

The research presents meta-differencing as a method for performing analysis on source-code differences. By extending the addressing and querying of source code on the srcML representation, meta-differencing is able to query source-code differences and automatically answer questions that typically must be done manually. By performing a case study on a medium-sized application, the research shows that useful questions can be automatically answered through the meta-differencing approach.

As a whole, the research validated an approach that takes a programmer-centric view. In this view, source code is a document that can be viewed from a lexical, syntactic, structural, or documentary perspective. By preserving all of the information in the document and carefully annotating the syntactic elements, all of these perspectives can be supported by appropriately leveraging XML.

11.2 Future Research Directions

The infrastructure presented in this research is based on the srcML format. The applicability of the approach described to a source-code task is entirely dependent on the robustness and tolerance of the srcML translator. Extending the translator's ability to handle source code in a realistic state extends the source-code base that srcML features can be applied to.

Of particular interest is the handling of macros. Currently the srcML translator handles macro calls, even ones that include statements. However, the macro facility of C/C++ is often used for strange purposes and new uses continue to be invented

(unfortunately). The macro issue is the main problem with using the srcML translator on many code bases, and is the reason that only 90% of the Linux kernel can be translated at this point.

Editing also creates particular problems for the state of source code. Intermediate editing states of code are often not only non-compilable, but not even well-formed, e.g., unmatched parentheses, blocks, etc. Extending the flexibility of the translator is necessary for handling the realistic state of source code that often arises in editing.

Extension of srcML to other languages, e.g., Java and Python, will bring the benefits of the approach to those languages. Both of these languages should be easier than C/C++ due to their Context Free Grammar. In addition, Python uses dynamic typing so lightweight approaches that do not have a symbol table are not at a disadvantage to full, parser-based, heavy-weight approaches.

11.2.1 Lightweight Source Models

The work presented on call-graph models demonstrated that higher-level source models can easily be represented as associations between source-code elements and these models then used for source-code querying and fact extraction. Taking this a step further, it is possible to do the same thing with any kind of association between any source-code entities.

In any software project there can be a great number of ad-hoc, informal associations between parts of source code. There are associations between methods in a class, e.g., a get-set pair of methods, associations between statements, e.g., statements

added to fix a bug, and associations between statements and comments. At this point there is no easy way to store, query, or process these associations.

Using an XML linking language makes it possible to build lightweight source models, ad-hoc collections of associations between source-code entities. Lightweight source models do not form an external abstraction of the source code, but are a collection of named links into the code. They form internal abstractions for the purpose of supporting querying and addressing of the code, and would allow for the full integration of source models with source code

11.2.2 Linking Source Code

As shown, the srcML format provides a language for the addressing of source-code elements. The transparency of the format, i.e., the no-loss translation to and from source code, and the speed of the translator, allows the srcML language to be directly applied to source code.

The ability to link to source code allows other information in an XML format, e.g., design documents, requirements, test cases, and documents, to link to locations in the code. Design documents, such as in UML (Unified Modeling Language), can refer to the locations in source code where they are implemented. Requirement documents can have links to the source code that implements the requirements. Test cases can be mapped to the source code that they are involved in. Project documents of all kinds can refer to the source code, from education and clarification to simple “ToDo” lists.

Linking with design documents would allow for true “round-trip” engineering. In round-trip engineering the design documents can be used to generate source code and

source code can be used to generate the design documents. More importantly, this allows for changes in one to be automatically reflected in the other, e.g., a change in a source code reflected in the UML design.

11.2.3 Extension of the srcDiff Format to a Complete Version History

Software Engineering is concerned with the process of developing software systems. Strangely, it is very limited when it comes to studying the most fundamental process of all; that of a programmer editing a source-code document. The large-scale processes covered by development methodologies are heavily researched, but the small-scale process of implementing features, fixing bugs, and, in general, determining how a piece of software is changed at the source-code level has only had limited study. In order for the field of software engineering to fully support, enhance, and increase comprehension of the act of programming a representation for a series of individual, small changes to a source-code document is needed.

The difference format presented in this research is the start of such a representation. It has the robustness to be used at any time in the development process, and full capability for querying and transformation. Currently, the srcDiff format is limited to two versions of source code at a time. Extension of the format to more than two versions would support queries over an entire version history.

Multiple-version differences would require changes in the srcDiff elements. One possibility is to replace the individual elements for each part, i.e., common, deleted and added, with a single difference element, and use the attribute to record which version(s)

its contents were in. A bigger challenge would be how to translate from the individual file differences into this multiple-version format.

11.2.4 Partitioning into Version Histories

The demonstration of the division of differences split a single difference into two differences. Repeated application of this same approach would allow the creation of a complete series of sequential differences. This would allow the partitioning, reorganization, and combination of version histories. A single change could be partitioned in multiple ways allowing the study of a change from many different perspectives.

11.2.5 XML Native Database

In this work, queries were performed separately on individual files. Queries that cannot be performed on a single file one at a time require either multiple passes or the combining of individual files into one large file. Using multiple passes requires storing intermediate information, and using one large file slows down the processing..

One solution is to use a native XML database. Native XML databases allow the entry of XML data following any desired schema. The individual source-code documents in srcML and the difference information in srcDiff could be put into the database. Also, any other project information in an XML format could be entered, including meta-data (e.g., type information), extracted source model information (e.g., call graph), and design information (e.g., class diagram). Then, meta-differencing queries could be performed on the higher-level source model views in the database.

11.2.6 A New View of Validity

Other approaches to the use of XML for source-code representation have attempted to construct schemas that prevent the representation of non-compilable source code. In their view the measure of validity is whether or not the source-code representation allows for the encoding of source code that is compilable. This is a difficult thing to do, especially with type requirements.

However, this "problem" has already been solved. To determine if an XML representation of a source code document is compilable the source code can be extracted and the source code compiled. And has been shown, it is often more interesting to deal with segments of source code that are not compilable.

For validity it is important to make sure that the correct elements are used. It might additionally be of interest to constrain the sub-elements or the multiplicity, e.g., a single *condition* element in an *if* element. It is less important to verify that a method name used on an object has been declared in the class of that object.

A better use of validity is to apply it to the structure of the source code itself. Extending a base schema to only allow for certain code combinations would be especially useful. Examples of this use of validity include restricting the use of triple-nested loops, or requiring that if particular functions are used, e.g., an *open()*, a related function must be used in the same block, e.g., a *close()*.

The srcML representation allows the creation of special-purpose schemas to determine the validity of a source-code document based on constraints on any syntactic and documentary elements. It allows these schemas to constrain both categories of

elements, i.e., syntactic and documentary, at the same time allowing for a rich language to express source-code validity.

APPENDIX A

srcML DTD

This appendix contains the DTD (Document Type Definition) of srcML. A DTD is a standard format for defining the schema of an XML format.

```
<!--
    srcML.dtd

    srcML is an XML application that adds information about syntactic
    structure to source code using XML tags.  Currently it is being
    used with C++ source code.

    Any compilable C++ program can be represented in srcML.  The dtd
    does not try to match the grammar of C++ but to contain it,
    i.e., all valid C++ programs can be represented in srcML, but not
    all valid srcML documents are valid C++ programs.

    srcML also stores all information in the source code file before
    the preprocessor is run.  This includes preprocessor directives,
    comments, macros, etc.

    srcML is a project of the Software DevelopMent Laboratory (SDML).
    For more information see the SDML website at:
        www.sdml.info
-->

<!-- Common elements -->
<!ENTITY % common.extra "">
<!ENTITY % common "comment | macro %common.extra;">
```

```
<!-- Basic elements -->
<!ENTITY % basic_elements "%common; | block">

<!-- Basic statements -->
<!ENTITY % basic_statements "if | while | for | do | switch | break |
goto | label | expr_stmt">

<!-- Basic declarations -->
<!ENTITY % basic_declarations "decl_stmt | typedef | enum | extern |
struct | struct_decl | union | union_decl | asm">

<!-- Base function -->
<!ENTITY % function_elements "function | function_decl | return">

<!-- Class elements -->
<!ENTITY % class_elements "class | class_decl | constructor |
constructor_decl | destructor | destructor_decl">

<!-- Template elements -->
<!ENTITY % template_elements "template">

<!-- Namespace elements -->
<!ENTITY % namespace_elements "namespace | using">

<!-- Exception handling elements -->
<!ENTITY % exception_elements "try | throw | catch">

<!-- Preprocessor directives-->
<!ENTITY % cpp "cpp:include | cpp:define | cpp:undef | cpp:if |
cpp:then | cpp:else | cpp:endif | cpp:elif | cpp:ifdef | cpp:ifndef |
cpp:line">

<!-- Elements -->
<!ENTITY % statement "%basic_elements; | %basic_statements; |
%basic_declarations; | %function_elements; |
%class_elements; | %namespace_elements; | %template_elements; |
%exception_elements; | %cpp;">
<!--
```

```
  main element
  Element unit can contain any statement or other units
-->
<!ELEMENT unit ( #PCDATA | unit | %statement; )* >
<!ATTLIST unit
    xmlns     CDATA #IMPLIED
    xmlns:cpp CDATA #IMPLIED
    filename  CDATA #IMPLIED
    dir       CDATA #IMPLIED
>

<!--
  comment
  Comments are of two types distinguished by an attribute type.
  Block comments are of the form: /*  */
  Line comments are of the form:  //
-->
<!ELEMENT comment      ANY>
<!ATTLIST comment type ( block | line ) #REQUIRED >

<!-- block, compound statement; -->
<!ENTITY % access_specifier "public | private | protected" >
<!ENTITY % switch_section "case | default" >
<!ELEMENT block ( #PCDATA | %statement; | %access_specifier; |
%switch_section; | expr )* >

<!-- expressions -->
<!ELEMENT expr | #PCDATA | name | call | type | init | block | %common;
)* >

<!-- expression statements -->
<!ELEMENT expr_stmt        ( #PCDATA | expr | %common; )* >

<!-- declaration -->
<!ELEMENT decl ( #PCDATA | type | name | init | argument_list |
%common; )* >
```

```
<!ELEMENT type  ( #PCDATA | name | enum | class | struct | union |
%common; )* >
<!ELEMENT name            ( #PCDATA | name | argument_list | expr |
%common; )* >
<!ELEMENT init            ( #PCDATA | block | expr | %common; )* >

<!-- declaration statement; -->
<!ELEMENT decl_stmt        ( #PCDATA | decl | %common; )* >

<!-- typedef statement; -->
<!ELEMENT typedef          ( #PCDATA | type | name | function_decl |
%common; )* >

<!-- label statement; -->
<!ELEMENT label            ( #PCDATA | name | %common; )* >

<!-- goto statement; -->
<!ELEMENT goto             ( #PCDATA | name | %common; )* >

<!-- asm statement; -->
<!ELEMENT asm              ( #PCDATA | %common; )* >

<!-- enum statement; -->
<!ELEMENT enum             ( #PCDATA | name | block | %common; )* >

<!-- if statement; -->
<!ELEMENT if               ( #PCDATA | condition | then | else |
%common; )* >
<!ELEMENT then      ( #PCDATA | %statement; )* >
<!ELEMENT else      ( #PCDATA | %statement; )* >

<!-- loops -->
<!-- while statement; -->
<!ELEMENT while     ( #PCDATA | condition | %statement; )* >

<!-- do..while statement; -->
<!ELEMENT do        ( #PCDATA | %statement; | condition )* >
```

```
<!-- for statement; -->
<!ELEMENT for    ( #PCDATA | init | condition | incr | %statement; )* >
<!ELEMENT incr      ( #PCDATA | expr | %common; )* >
<!ELEMENT condition ( #PCDATA | expr | %common;)* >

<!-- switch statement; -->
<!ELEMENT switch    ( #PCDATA | condition | block | %common; )* >
<!ELEMENT case      ( #PCDATA | expr | %statement; )* >
<!ELEMENT default   ( #PCDATA | %statement; )* >
<!ELEMENT break     ( #PCDATA | %common; )* >

<!-- function call -->
<!ELEMENT call      ( #PCDATA | name | argument_list | %common; )* >
<!ELEMENT argument_list    ( #PCDATA | argument | %common; )* >
<!ELEMENT argument         ( #PCDATA | name | expr | %common; )* >

<!-- functions -->
<!ENTITY % function_prototype "type | name | parameter_list | throw |
specifier | block | %common;">
<!ELEMENT function         ( #PCDATA | %function_prototype; )* >
<!ELEMENT function_decl    ( #PCDATA | %function_prototype; )* >
<!ELEMENT parameter_list   ( #PCDATA | param | %common; )* >
<!ELEMENT param            ( #PCDATA | decl | name | init | type |
function_decl | %common; )* >
<!ELEMENT specifier        ( #PCDATA | %common; )* >
<!ELEMENT return           ( #PCDATA | expr | %common; )* >

<!-- class -->
<!ENTITY % class_contents "name | super | block | %common;">
<!ELEMENT class            ( #PCDATA | %class_contents; )*>
<!ELEMENT class_decl       ( #PCDATA | %class_contents; )*>
<!ELEMENT struct           ( #PCDATA | %class_contents; )*>
<!ELEMENT struct_decl      ( #PCDATA | %class_contents; )*>
<!ELEMENT union            ( #PCDATA | %class_contents; )*>
<!ELEMENT union_decl       ( #PCDATA | %class_contents; )*>
```

```
<!-- methods -->
<!ENTITY % method_prototype "specifier | name | parameter_list | throw
| %common;">
<!ELEMENT constructor      ( #PCDATA | %method_prototype; | member_list
| block )*>
<!ELEMENT member_list      ( #PCDATA | call | %common; )*>
<!ELEMENT constructor_decl ( #PCDATA | %method_prototype; )*>
<!ELEMENT destructor       ( #PCDATA | %method_prototype; | block )*>
<!ELEMENT destructor_decl  ( #PCDATA | %method_prototype; )*>
<!-- member function category -->
<!ENTITY % member_function "function_decl | function | constructor |
constructor_decl | destructor | destructor_decl | decl_stmt | %common;"
>
<!ELEMENT public           ( #PCDATA | %member_function; )* >
<!ATTLIST public
    type     CDATA #IMPLIED
>
<!ELEMENT protected        ( #PCDATA | %member_function; )* >
<!ELEMENT private          ( #PCDATA | %member_function; )* >
<!ATTLIST private
    type     CDATA #IMPLIED
>
<!ELEMENT super            ( #PCDATA | specifier | name | %common; )* >

<!-- exception handling -->
<!ELEMENT try              ( #PCDATA | block | %common; )* >
<!ELEMENT throw        ( #PCDATA | expr | argument | %common; )* >
<!ELEMENT catch        ( #PCDATA | param | block | %common; )* >

<!-- template -->
<!ELEMENT template     ( #PCDATA | parameter_list | class | function |
function_decl | %common;)* >

<!-- namespace -->
<!ELEMENT namespace    ( #PCDATA | name | block | %common; )* >
<!ELEMENT using        ( #PCDATA | name | %common; )* >
```

```
<!-- extern -->
<!ELEMENT extern       ( #PCDATA | block | %common; )* >

<!-- macro -->
<!ELEMENT macro        ( #PCDATA | name | argument_list | %common; )* >

<!-- cpp - C-PreProcessor -->
<!ELEMENT cpp:directive ( #PCDATA ) >
<!ELEMENT cpp:file     ( #PCDATA ) >
<!ELEMENT cpp:include  ( #PCDATA | cpp:directive | cpp:file | %common;
)* >
<!ELEMENT cpp:define   ( #PCDATA | cpp:directive | name | expr |
%common; )* >
<!ELEMENT cpp:undef    ( #PCDATA | cpp:directive | name | %common; )* >
<!ELEMENT cpp:if       ( #PCDATA | cpp:directive | expr | %common; )* >
<!ELEMENT cpp:then     ( #PCDATA | cpp:directive | %common; )* >
<!ELEMENT cpp:else     ( #PCDATA | cpp:directive | %common; )* >
<!ELEMENT cpp:endif    ( #PCDATA | cpp:directive | %common; )* >
<!ELEMENT cpp:elif     ( #PCDATA | cpp:directive | expr | %common; )* >
<!ELEMENT cpp:ifdef    ( #PCDATA | cpp:directive | name | %common; )* >
<!ELEMENT cpp:ifndef   ( #PCDATA | cpp:directive | name | %common; )* >
<!ELEMENT cpp:line     ( #PCDATA | cpp:directive | file | %common; )* >
```

APPENDIX B

srcDiff DTD

This appendix contains the DTD (Document Type Definition) of srcDiff. A DTD is a standard format for defining the schema of an XML format. The srcDiff DTD is a direct extension of the srcML DTD.

```
<!--
    srcdiff.dtd

    srcDiff is a difference format for source code.  It represents
    differences between two source code files.

    srcML is an XML application that adds information about syntactic
    structure to source code using XML elements.  It is defined in
    the DTD srcml.dtd.

    srcML is a project of the Software DevelopMent Laboratory (SDML).
    For more information see the SDML website at:
        www.sdml.info
-->

<!-- Define the difference elements with no restrictions on content -->
<!ELEMENT diff:common          ANY >
<!ELEMENT diff:old             ANY >
<!ELEMENT diff:new             ANY >

<!-- All difference elements -->
<!ENTITY % diff.elements "| diff:common | diff:old | diff:new">
```

```
<!-- Override the common extra elements to include the difference
elements -->
<!ENTITY % common.extra "%diff.elements;">


<!-- Add an additional attribute to the root element unit for the
difference namespace -->
<!ATTLIST unit
    xmlns:diff CDATA #IMPLIED
>


<!-- srcML is used for source code markup -->
<!ENTITY % srcML SYSTEM "srcml.dtd">
%srcML;
```

.

APPENDIX C

HippoDraw srcDiff Files

This appendix contains a table with a list of all the files of HippoDraw used in the case study. The table is organized by directory and filename. For each file the size of the original source code, the size of the equivalent srcML representation, and the ratio between the original source code and the srcML document is given. This information is for Version 1.4.0 and Version 1.5.1 individually. The second-to-last column contains the size of the srcDiff document. The final column contains the ratio between the size of the Version 1.4.0 srcML document and the srcDiff document is given. The figures are summarized at the end of the table.

		V 1.4.0			V 1.5.1			
Filename	**src**	**srcML**	**Ratio**	**src**	**srcML**	**Ratio**	**srcDiff**	**Ratio**
binners/Bins1DBase.cxx	3,984	16,317	4.10	4,181	16,922	4.05		
binners/Bins1DBase.h	3,928	9,284	2.36	4,082	9,490	2.32	12,262	1.32
binners/Bins1DHist.cxx	3,005	12,966	4.31	3,045	13,054	4.29	13,680	1.06
binners/Bins1DProfile.cxx	3,217	13,823	4.30	3,257	13,911	4.27	14,477	1.05
binners/Bins2DBase.cxx	7,115	28,367	3.99	7,417	28,923	3.90		
binners/Bins2DBase.h	3,994	9,909	2.48	4,144	10,111	2.44	12,884	1.30
binners/Bins2DHist.cxx	4,310	18,305	4.25	4,412	18,483	4.19	21,214	1.16
binners/Bins2DHist.h	1,764	4,404	2.50	1,778	4,436	2.49	4,890	1.11
binners/Bins2DProfile.cxx	4,343	18,527	4.27	4,464	18,724	4.19	22,002	1.19
binners/Bins2DProfile.h	1,863	4,632	2.49	1,877	4,664	2.48	5,108	1.10
binners/BinsBase.h	5,514	12,697	2.30	5,661	12,926	2.28	16,105	1.27
binners/BinsFunction.cxx	5,225	17,964	3.44	5,307	18,243	3.44		
binners/BinsFunction.h	3,286	6,982	2.12	3,327	7,098	2.13	8,135	1.17
controllers/CutController.cxx	8,453	30,584	3.62	8,934	32,645	3.65	31,187	1.02
controllers/CutController.h	6,327	12,412	1.96	6,810	13,394	1.97	13,902	1.12
controllers/DisplayController.cxx	28,208	106,915	3.79	28,342	106,896	3.77	116,819	1.09
controllers/FunctionController.cxx	23,048	86,091	3.74	24,912	94,238	3.78	6,664	0.08
controllers/FunctionController.h	13,660	25,719	1.88	14,190	26,758	1.89	28,232	1.10
datareps/ColorPlot.h	1,476	2,736	1.85	1,476	2,736	1.85	2,987	1.09
datareps/DataRep.cxx	4,641	18,806	4.05	5,360	21,238	3.96	24,124	1.28
datareps/DataRep.h	9,269	17,238	1.86	9,830	18,376	1.87	20,206	1.17
datareps/DyHistogram.cxx	1,140	3,754	3.29	1,174	3,836	3.27	4,260	1.13
datareps/DyHistogram.h	2,117	3,653	1.73	2,205	3,741	1.70	3,849	1.05
datareps/Profile2D.h	1,358	2,604	1.92	1,353	2,599	1.92	2,786	1.07
datareps/ProfileHist.cxx	1,019	3,316	3.25	1,025	3,322	3.24	3,649	1.10
datareps/TextDataRep.cxx	1,683	5,794	3.44	1,640	5,235	3.19		
datareps/XYPlot.cxx	2,710	9,482	3.50	2,746	9,518	3.47	10,982	1.16
datareps/XYZPlot.h	1,161	2,393	2.06	1,167	2,399	2.06	2,633	1.10
datareps/ZPlot.h	1,286	2,615	2.03	1,157	2,486	2.15	2,811	1.07
datasrcs/CircularBuffer.cxx	1,839	7,119	3.87	1,613	6,273	3.89	7,234	1.02
datasrcs/CircularBuffer.h	2,817	5,419	1.92	2,768	5,259	1.90	5,886	1.09
datasrcs/DataPointTuple.h	1,498	2,620	1.75	1,577	2,908	1.84	3,029	1.16
datasrcs/NTuple.cxx	12,856	54,988	4.28	13,632	57,448	4.21	61,261	1.11
datasrcs/NTuple.h	13,753	24,504	1.78	14,355	25,929	1.81	27,467	1.12
datasrcs/NTupleController.cxx	9,566	37,195	3.89	9,722	37,880	3.90	38,079	1.02
datasrcs/NTupleController.h	6,835	12,942	1.89	7,094	13,438	1.89	13,812	1.07
functions/Gaussian.cxx	3,233	12,534	3.88	3,237	12,489	3.86	12,948	1.03
hippoplot/AxisModelLog.h	2,449	5,181	2.12	2,637	5,401	2.05	5,743	1.11
hippoplot/ViewBase.h	12,194	23,893	1.96	12,202	23,896	1.96	24,352	1.02
minimizers/BFGSFitter.cxx	14,994	49,123	3.28	15,202	49,171	3.23	51,208	1.04
minimizers/BFGSFitter.h	5,888	11,526	1.96	5,655	11,265	1.99	11,633	1.01
minimizers/LMFitter.cxx	6,667	22,935	3.44	6,702	23,018	3.43	23,210	1.01
minimizers/NTupleChiSqFCN.cxx	1,068	3,399	3.18	1,136	3,626	3.19	3,500	1.03
minimizers/NTupleChiSqFCN.h	1,608	3,135	1.95	1,655	3,293	1.99	3,532	1.13
minimizers/NTupleLikeliHoodFCN.cxx	2,426	6,618	2.73	2,734	7,576	2.77		
minimizers/NTupleLikeliHoodFCN.h	1,509	3,270	2.17	1,555	3,427	2.20	3,673	1.12
minimizers/NumLinAlg.h	4,309	12,340	2.86	4,577	12,804	2.80	14,174	1.15
minimizers/StatedFCN.h	3,190	6,682	2.09	3,496	7,154	2.05	7,354	1.10
plotters/CompositePlotter.cxx	18,340	74,565	4.07	19,763	79,544	4.02		
plotters/CompositePlotter.h	9,029	18,120	2.01	10,029	20,083	2.00	23,005	1.27
plotters/CutPlotter.cxx	6,919	28,068	4.06	6,977	28,174	4.04	29,851	1.06
plotters/CutPlotter.h	4,824	10,350	2.15	5,011	10,537	2.10	10,728	1.04
plotters/PlotterBase.cxx	14,349	54,580	3.80	14,466	54,018	3.73	66,126	1.21
plotters/PlotterBase.h	19,750	38,038	1.93	19,949	38,144	1.91	43,992	1.16

plotters/TextPlotter.cxx	3,021	10,701	3.54	3,107	10,805	3.48	13,244	1.24
plotters/TextPlotter.h	3,055	7,525	2.46	3,112	7,608	2.44	8,788	1.17
plotters/XMultiYPlotter.cxx	14,992	58,082	3.87	15,166	58,332	3.85	62,662	1.08
plotters/XMultiYPlotter.h	3,270	7,477	2.29	3,289	7,509	2.28	7,923	1.06
plotters/XYColorPlotter.cxx	6,035	23,756	3.94	6,051	23,571	3.90	26,036	1.10
plotters/XYPlotter.cxx	5,098	19,289	3.78	5,089	19,079	3.75	20,517	1.06
projectors/BinningProjector.cxx	5,081	19,222	3.78	5,182	19,384	3.74	26,143	1.36
projectors/BinningProjector.h	3,592	8,254	2.30	3,696	8,374	2.27	10,724	1.30
projectors/DyHist1DProjector.cxx	4,964	17,622	3.55	4,969	17,487	3.52	20,119	1.14
projectors/DyHist1DProjector.h	3,305	6,003	1.82	2,991	5,531	1.85	6,905	1.15
projectors/DyHist2DProjector.cxx	4,341	15,827	3.65	4,349	15,604	3.59		
projectors/DyHist2DProjector.h	2,824	5,803	2.05	2,706	5,550	2.05	6,470	1.11
projectors/FunctionProjector.cxx	7,362	28,093	3.82	7,340	28,012	3.82	31,584	1.12
projectors/FunctionProjector.h	6,236	13,568	2.18	6,519	13,859	2.13	15,375	1.13
projectors/Hist1DProjImp.cxx	3,580	13,895	3.88	3,655	14,051	3.84	16,049	1.16
projectors/Hist1DProjImp.h	2,027	4,402	2.17	2,059	4,460	2.17	5,068	1.15
projectors/Hist2DProjImp.cxx	4,276	17,073	3.99	4,433	17,367	3.92	21,967	1.29
projectors/Hist2DProjImp.h	2,930	6,252	2.13	2,985	6,374	2.14	7,551	1.21
projectors/Map1Projector.cxx	5,222	21,044	4.03	5,273	20,983	3.98	24,129	1.15
projectors/Map1Projector.h	3,054	6,366	2.08	3,105	6,444	2.08	7,594	1.19
projectors/Map2Projector.cxx	5,549	21,496	3.87	6,408	23,562	3.68	26,502	1.23
projectors/Map2Projector.h	2,586	5,384	2.08	2,646	5,514	2.08	6,376	1.18
projectors/Map3Projector.cxx	4,813	18,209	3.78	4,975	18,654	3.75		
projectors/Map3Projector.h	2,423	5,465	2.26	2,444	5,525	2.26	6,467	1.18
projectors/MapMatrixProjector.cxx	8,673	34,986	4.03	9,403	36,831	3.92	44,336	1.27
projectors/MapMatrixProjector.h	7,050	15,428	2.19	7,435	15,942	2.14	18,635	1.21
projectors/NTupleProjector.cxx	11,415	45,663	4.00	11,418	45,428	3.98	46,787	1.02
projectors/NTupleProjector.h	8,828	16,926	1.92	8,639	16,602	1.92	17,620	1.04
projectors/Profile2DProjector.cxx	7,194	27,785	3.86	7,422	27,997	3.77	35,325	1.27
projectors/Profile2DProjector.h	3,683	7,600	2.06	3,604	7,440	2.06	9,082	1.20
projectors/ProfileProjector.cxx	5,239	20,382	3.89	5,247	20,273	3.86	23,513	1.15
projectors/ProfileProjector.h	2,480	5,170	2.08	2,222	4,712	2.12	6,299	1.22
projectors/ProjectorBase.cxx	5,010	17,978	3.59	5,093	18,054	3.54		
projectors/ProjectorBase.h	14,085	24,103	1.71	14,052	24,104	1.72	29,301	1.22
projectors/StHist1DProjector.cxx	2,823	9,821	3.48	2,918	10,009	3.43		
projectors/StHist1DProjector.h	2,924	6,119	2.09	2,969	6,149	2.07	7,020	1.15
projectors/StHist2DProjector.cxx	4,182	15,508	3.71	4,309	15,698	3.64	18,452	1.19
projectors/StHist2DProjector.h	3,177	6,688	2.11	3,222	6,718	2.09	7,589	1.13
projectors/StripChartProjector.cxx	2,484	9,511	3.83	2,575	9,610	3.73	10,301	1.08
projectors/StripChartProjector.h	1,413	2,763	1.96	1,575	2,987	1.90	3,442	1.25
qt/AxisWidget.cxx	14,781	61,394	4.15	15,096	62,509	4.14	67,779	1.10
qt/AxisWidget.h	5,534	13,303	2.40	5,778	13,818	2.39	16,078	1.21
qt/AxisWidgetPlugin.cxx	1,553	4,356	2.80	1,556	4,359	2.80	4,721	1.08
qt/CanvasView.cxx	19,217	77,167	4.02	19,887	79,523	4.00	83,557	1.08
qt/CanvasView.h	7,695	16,793	2.18	7,889	17,268	2.19	17,542	1.04
qt/CanvasWindow.cxx	22,290	79,601	3.57	22,379	79,581	3.56	83,452	1.05
qt/CanvasWindow.h	12,901	25,794	2.00	12,972	25,946	2.00	26,140	1.01
qt/FigureEditor.cxx	23,194	103,895	4.48	24,386	109,243	4.48	114,981	1.11
qt/FigureEditor.h	7,575	15,196	2.01	7,865	15,743	2.00	16,292	1.07
qt/Inspector.cxx	33,703	129,508	3.84	51,444	200,129	3.89		
qt/Inspector.h	9,989	17,228	1.72	11,884	22,322	1.88	24,414	1.42
qt/PickTable.cxx	4,227	17,807	4.21	4,227	17,807	4.21	18,156	1.02
qt/PickTable.h	2,258	4,385	1.94	2,069	4,196	2.03	4,491	1.02
qt/PlotTable.cxx	1,798	7,145	3.97	1,836	7,231	3.94	8,743	1.22
qt/QtView.cxx	5,763	25,130	4.36	5,924	25,816	4.36	26,031	1.04
qt/QtView.h	4,944	10,898	2.20	5,030	10,984	2.18	11,196	1.03

qt/main.cxx	865	2,686	3.11	983	2,804	2.85	2,907	1.08
reps/AverageTextRep.cxx	1,605	5,710	3.56	1,682	5,900	3.51		
reps/AverageTextRep.h	1,044	2,568	2.46	1,065	2,627	2.47	3,214	1.25
reps/AxisRepBase.h	6,162	12,456	2.02	6,247	12,541	2.01	12,655	1.02
reps/ColorBoxPointRep.cxx	3,677	15,764	4.29	3,123	12,810	4.10	16,257	1.03
reps/ColorSymbolPointRep.cxx	3,418	13,427	3.93	3,424	13,433	3.92	13,803	1.03
reps/ColumnPointRep.cxx	3,271	12,608	3.85	3,385	13,212	3.90	14,636	1.16
reps/ContourPointRep.cxx	13,569	53,922	3.97	13,613	54,014	3.97	55,032	1.02
reps/SymbolPointRep.cxx	6,781	26,998	3.98	6,821	27,038	3.96	28,841	1.07
reps/SymbolPointRep.h	4,526	9,594	2.12	4,655	10,030	2.15	10,219	1.07
reps/SymbolType.h	1,028	1,848	1.80	1,075	1,927	1.79	2,033	1.10
reps/TextRepBase.h	1,274	2,724	2.14	1,279	2,729	2.13	2,911	1.07
reps/TextRepFactory.cxx	1,027	3,733	3.63	1,068	3,822	3.58	4,562	1.22
transforms/BinaryTransform.h	5,127	12,421	2.42	5,135	12,476	2.43	12,865	1.04
transforms/HammerAito.cxx	9,571	34,504	3.61	9,648	34,687	3.60		
transforms/HammerAito.h	3,854	7,888	2.05	3,936	8,032	2.04	8,830	1.12
transforms/Lambert.cxx	9,136	33,856	3.71	9,125	33,862	3.71	34,563	1.02
transforms/Lambert.h	3,064	7,496	2.45	3,076	7,552	2.46	7,786	1.04
transforms/LogTransform.cxx	11,772	42,893	3.64	11,911	43,064	3.62	43,058	1.00
transforms/PeriodicBinaryTransform.cxx	3,621	12,809	3.54	3,683	13,101	3.56		
transforms/PeriodicBinaryTransform.h	3,070	7,653	2.49	3,096	7,721	2.49	8,558	1.12
transforms/XYTransform.cxx	3,128	13,273	4.24	3,132	13,215	4.22	14,399	1.08
transforms/XYTransform.h	2,669	6,996	2.62	2,668	7,071	2.65		
xml/AxisModelXML.cxx	2,211	9,251	4.18	2,235	9,225	4.13		
xml/AxisModelXML.h	2,174	4,789	2.20	2,171	4,844	2.23	5,628	1.18
xml/BinsBaseXML.cxx	2,218	8,882	4.00	2,263	8,962	3.96	9,931	1.12
xml/MapMatrixProjectorXML.cxx	2,302	9,668	4.20	2,391	9,805	4.10	13,092	1.35
xml/PlotterBaseXML.cxx	10,030	38,740	3.86	10,112	38,911	3.85	41,744	1.08
xml/PlotterBaseXML.h	3,893	8,559	2.20	3,886	8,628	2.22	9,390	1.10
Minimum	865	1,848	1.71	983	1,927	1.70	2,033	0.08
Maximum	33,703	129,508	4.48	51,444	200,129	4.48	116,819	1.42
Average	6,078	19,410	3.01	6,358	20,301	2.99	19,951	1.12
Median	4,205	12,653	3.14	4,268	12,990	3.02	13,680	1.11

REFERENCES

[Acacia 2001] Acacia, (2001), "Acacia - the C++ Information Abstraction System", AT&T, Web page, Date Accessed: 11/01/2001, http://www.research.att.com/sw/tools/Acacia/.

[Alliance] Alliance, G., "XML::STX", Date Accessed: May 15, http://www.gingerall.org/charlie/ga/xml/p_stx.xml.

[Apache 2004] Apache, (2004), "Xalan-J", Date Accessed: Jun 11, http://xml.apache.org/xalan-j.

[Apiwattanapong, Orso, Harold 2004] Apiwattanapong, T., Orso, A., and Harold, M. J., (2004), "A Differencing Algorithm for Object-Oriented Programs", in Proceedings of 19th IEEE International Conference on Automated Software Engineering (ASE '04), Linz, Austria, September 20-25.

[Atkinson, Griswold 1996] Atkinson, D. C. and Griswold, W. G., (1996), "The design of whole-program analysis tools", in Proceedings of 18th International Conference on Software Engineering (ICSE'96), Berlin, Germany, March 25-30, pp. 16-27.

[Badros 2000] Badros, G. J., (2000), "JavaML: A Markup Language for Java Source Code", in Proceedings of 9th International World Wide Web Conference (WWW9), Asterdam, The Netherlands, May 13-15.

[Becker 2004] Becker, O., (2004), "Joost", Date Accessed: May 15, http://joost.sourceforge.net.

[Boshernitsan, Graham 2000] 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), Brisbane, Australia, November 23-25, pp. 287-289.

[Cederqvist 2004] Cederqvist, P., (2004), "CVS--Concurrent Versions System v1.12.8", Date Accessed: May 16, https://www.cvshome.org/docs/manual/cvs-1.12.8/cvs.html.

[Cimprich 2002] Cimprich, P., (2002), "Streaming Transformations for XML (STX) Version 1.0 Working Draft", http://stx.sourceforge.net/, Web page, Date Accessed: 11/15/2002, http://stx.sourceforge.net/documents/spec-stx-20021101.html.

[Collard 2003] Collard, M. L., (2003), "An Infrastructure to Support Meta-Differencing and Refactoring of Source Code", in Proceedings of 18th IEEE International Conference on Automated Software Engineering (ASE '03), October 6-10, pp. 377-380.

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

[Collard, Maletic, Marcus 2002] 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, November 8-9, pp. 34-41.

[Collins-Sussman, Fitzpatrick, Pilato 2004] Collins-Sussman, B., Fitzpatrick, B. W., and Pilato, C. M., (2004), "Version Control with Subversion", Date Accessed: May 16, http://svnbook.red-bean.com/svnbook/book.html.

[Conradi, Westfechtel 1998] Conradi, R. and Westfechtel, B., (1998), "Version Models for Software Configuration Management", *ACM Computing Surveys*, vol. 30, no. 2, June, pp. 232 - 282.

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

[Cox, Clarke 2000] Cox, A. and Clarke, C., (2000), "A Comparative Evaluation of Techniques for Syntactic Level Source Code Analysis", in Proceedings of 7th Asia-Pacific Software Engineering Conference (APSEC'00), Singapore, pp. 282-291.

[Cox, Clarke, Sim 1999] Cox, A., Clarke, C., and Sim, S., (1999), "A Model Independent Source Code Repository", in Proceedings of IBM Center for Advanced Studies Conference (CASCON'99), November 8-11, pp. 381-390.

[CPPX 2001] CPPX, (2001), "CPPX - Open Source C++ Fact Extractor", Web page, http://swag.uwaterloo.ca/~cppx/.

[Dean, Malton, Holt 2001] Dean, T. R., Malton, A. J., and Holt, R. C., (2001), "Union Schemas as a Basis for a C++ Extractor", in Proceedings of Eighth Working

Conference on Reverse Engineering (WCRE'01), Stuttgart, Germany, October 2-5, pp. 59-70.

[DeltaXML 2004a] DeltaXML, (2004a), "DeltaXML.com Change Control for XML in XML", Date Accessed: May 16, http://www.deltaxml.com.

[DeltaXML 2004b] DeltaXML, (2004b), "How DeltaXML Markup Represents Changes to XML Files", Date Accessed: May 16, http://www.deltaxml.com/core/deltaxml-changes-markup.html.

[DeltaXML 2004c] DeltaXML, (2004c), "Introduction to DeltaXML", Date Accessed: May 16, http://www.deltaxml.com/pdf/deltaxml-intro.pdf.

[Ebert, Kullbach, Winter 1999] 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, October 6-8, pp. 89 - 100.

[Emmerich, Mascolo, Finkelstein 2000] Emmerich, W., Mascolo, C., and Finkelstein, A., (2000), "Implementing Incremental Code Migration with XML", in Proceedings of 22nd International Conference on Software Engineering (ICSE'00), Limerick, Ireland, June 4-11, pp. 397 - 406.

[ETH 2004] ETH, (2004), "XWeaver", Date Accessed: March 16, http://control.ee.ethz.ch/~ceg/XWeaver/.

[Ferenc et al. 2001] Ferenc, R., Gyimóthy, T., Sim, S. E., Holt, R. C., and Koschke, R., (2001), "Towards a Standard Schema for C/C++", in Proceedings of Eighth

Working Conference on Reverse Engineering (WCRE'01), Stuttgart, Germany, October 2-5, pp. 49-58.

[Ference et al. 2002] Ference, R., Magyar, F., Beszedes, A., Kiss, A., and Tarkiainen, M., (2002), "Columbus – Tool for Reverse Engineering Large Object Oriented Software Systems", in Proceedings of In Proceedings of SPLST 2001, June 2001, pp. 16-27.

[Fowler 1999] Fowler, M.,(1999),*Refactoring: Improving the Design of Existing Code*, Addison-Wesley.

[Harold 2004] Harold, E. R.,(2004),*Effective XML: 50 Specific ways to improve your XML*, Boston, MA, Addison Wesley.

[Harold, Means 2001] Harold, E. R. and Means, W. S.,(2001),*XML In A Nutshell*, Sebastopod, CA, O'Reilly & Associates.

[HippoDraw 2004] HippoDraw, (2004), "HippoDraw: Main Page", Date Accessed: 03/10, http://www.slac.stanford.edu/grp/ek/hippodraw/index.html.

[Holt, Winter, Schürr 2000] 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, Queensland, Australia, November, 23 - 25, pp. 162-171.

[Horwitz, Reps 1992] Horwitz, S. and Reps, T. W., (1992), "The Use of Program Dependence Graphs in Software Engineering", in Proceedings of International Conference on Software Engineering (ICSE), Melbourne, Australia, May 11 - 15, pp. 392 - 411.

[Hunt 2001] Hunt, J. J., (2001), "Extensible Language-Aware Differencing and Merging", in *PhD thesis*: Universitt Karlsruhe.

[Hunt 2004] Hunt, J. J., (2004), "Fast Semi-Semantic Differencing and Merging", Web page, Date Accessed: 02/01/2004, http://wwwswt.fzi.de/cocoon/mount/swt/mitarbeiter/jjh/.

[Hunt, Tichy 2002] Hunt, J. J. and Tichy, W. F., (2002), "Extensible Language-Aware Merging", in Proceedings of IEEE International Conference on Software Maintenance (ICSM'02), Montreal, Canada, October 3-6, pp. 511-520.

[Hunt, McIllroy 1976] Hunt, J. W. and McIllroy, M. D., (1976), "An Algorithm for Differential File Comparision", in *Technical Report 41*: AT&T Bell Laboratories Inc.

[Hunt, Szymanski 1977] Hunt, J. W. and Szymanski, T. G., (1977), "A Fast Algorithm for Computing Longest Common Subsequences", *CACM*, vol. 20, no. 5, May, pp. 350 - 353.

[IBM 1998] IBM, (1998), "XML TreeDiff", Date Accessed: May 16, http://alphaworks.ibm.com/tech/xmltreediff.

[ISO 1986] ISO, (1986), "Standard Generalized Markup Language (SGML)".

[Jackson, Ladd 1994] Jackson, D. and Ladd, D. A., (1994), "Semantic Diff: A Tool for Summarizing the Effects of Modifications", in Proceedings of IEEE International Conference on Software Maintenance (ICSM'94), Victoria, British Columbia, Canada, September 19-23, pp. 243-252.

[Kagdi 2003] Kagdi, H., (2003), *Using An Island Grammar Approach for Lightweight Parsing : A C++ To SrcML Translator*, Kent State University, Kent, Masters Thesis Thesis.

[Kay 2004] Kay, M., (2004), "Saxon The XSLT and XQuery Processor", Date Accessed: June 20, http://saxon.sourceforge.net.

[Kienle 2002] Kienle, H., (2002), "A Benchmark for C++ Fact Extractors: Results and Observations", IWPC, Web Page, Date Accessed: 11/15/2002, http://cedar.csc.uvic.ca/kienle/view/IWPC2002/Workshop.

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

[Laux, Martin 2000] Laux, A. and Martin, L., (2000), "XUpdate Working Draft", http://exist-db.org/xmldb/xupdate/xupdate-wd.html.

[Logilab 2003] Logilab, (2003), "xmldiff", Date Accessed: May 16, http://www.logilab.org/projects/xmldiff.

[Magnusson, Asklund, Minor 1993] Magnusson, B., Asklund, U., and Minor, S., (1993), "Fine-grained revision control for collaborative software development", in Proceedings of 1st ACM Symposium on Foundations of Software Engineering (FSE'93), pp. 33-41.

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

[Maletic, Collard, Kagdi 2004] Maletic, J. I., Collard, M. L., and Kagdi, H. H., (2004), "Leveraging XML Technologies in Developing Program Analysis Tools", in Proceedings of 4th International Workshop on Adoption-Centric Software Engineering (ACSE'04), Edinburgh, Scotland, May 25, pp. 80-85.

[Maletic, Collard, Marcus 2002] 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, France, June 27-29, pp. 289-292.

[Malton et al. 2001] 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, Kontogiannis 2000] 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.

[Mascolo, Picco, Roman 2005] Mascolo, C., Picco, G. P., and Roman, G.-C., (2005), "CodeWeave: Exploring Fine-Grained Mobility of Code", *Journal of Automated Software Engineering*, pp. (to appear).

[Mens 2002] Mens, T., (2002), "A State-of-the-Art Survey on Software Merging", *IEEE Transactions on Software Engineering*, vol. 28, no. 5, May, pp. 449 - 462.

[Microsoft 2002a] Microsoft, (2002a), "Microsoft XML Diff and Patch 1.0", Date Accessed: January 19, http://apps.gotdotnet.com/xmltools/xmldiff.

[Microsoft 2002b] Microsoft, (2002b), "Microsoft XML Diff Language v.1.0 Beta", Date Accessed: January 19, http://apps.gotdotnet.com/xmltools/xmldiff.

[Moonen 2001] Moonen, L., (2001), "Generating Robust Parsers using Island Grammars", in Proceedings of 8th IEEE Working Conference on Reverse Engineering (WCRE'01), Suttgart, Germany, October 2-5, pp. 13-24.

[Moonen 2002] Moonen, L., (2002), "Lightweight Impact Analysis using Island Grammars", in Proceedings of Proceedings of the 10 th International Workshop on Program Comprehension (IWPC'02), Paris, France, pp. 219-228.

[Mouat 2002] Mouat, A., (2002), *XML Diff and Patch Utilities*, Heriot-Watt University, Edinburgh, Scotland, Senior Project Thesis.

[Mouat 2004] Mouat, A., (2004), "diffxml", Date Accessed: May 16, http://diffxml.sourceforge.net.

[Murphy, Notkin 1996] Murphy, G. C. and Notkin, D., (1996), "Lightweight Lexical Source Model Extraction", *ACM Transactions on Software Engineering and Methodology*, vol. 5, no. 3, July, pp. 262-292.

[Murphy et al. 1998] Murphy, G. C., Notkin, D., Griswold, W. G., and Lan, E. S., (1998), "An Empirical Study of Static Call Graph Extractors", *ACM Transactions on Software Engineering and Methodology*, vol. 7, no. 2, April, pp. 158-191.

[Opdyke 1992] Opdyke, W. F., (1992), *Refactoring Object-Oriented Frameworks*, University of Illinois at Urbana-Champaign, Dissertation Thesis.

[Parr, Quong 1994] Parr, T. J. and Quong, R. W., (1994), "Adding Semantic and Syntatic Predicates To LL(k): pred-LL(k)", in Proceedings of International Conference on Compiler Construction (to appear).

[Perry 1987] Perry, D., (1987), "Software Interconnection Models", in Proceedings of Internation Conference on Software Engineering, March, pp. 61-69.

[Power, Malloy 2002] Power, J. F. and Malloy, B. A., (2002), "Program Annotation in XML: a Parse-tree Based Approach", in Proceedings of 9th Working Conference on Reverse Engineering, Richmond, Virginia, October 2002, pp. 190-198.

[SAX 2001] SAX, (2001), "Simple API for XML (SAX)", SAX, Web page, Date Accessed: 01/20/2002, http://www.saxproject.org/.

[Sergeant 2000] Sergeant, M., (2000), "XML::XPath", Date Accessed: May 16, http://xml.sergeant.org/xpath/.

[Sim 2002] Sim, S. E., (2002), "CppETS Benchmark", http://cedar.csc.uvic.ca/kienle/view/IWPC2002/Benchmark, Tar Zipped File, http://cedar.csc.uvic.ca/kienle/view/IWPC2002/Benchmark.

[Sim, Holt, Easterbrook 2002] Sim, S. E., Holt, R. C., and Easterbrook, S., (2002), "On Using a Benchmark to Evaluate C++ Extractors", in Proceedings of 10th International Workshop on Program Comprehension, Paris, France, pp. 114-123.

[Sperberg-McQueen, Burnard 2002] Sperberg-McQueen, C. M. and Burnard, L., (2002), "TEI P4: Guidelines for Electronic Text Encoding and Interchange": Text Encoding Initiative Consortium.

[Van De Vanter 2002] Van De Vanter, M. L., (2002), "The Documentary Structure of Source Code", *Information and Software Technology*, vol. 44, no. 13, October 1, pp. 767-782.

[van den Brand, Sellink, Verhoef 1998] 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.

[Vanter 2002] Vanter, M. L. V. D., (2002), "The Documentary Structure of Source Code", *Information and Software Technology*, vol. 44, no. 13, October 1, pp. 767-782.

[Veillard 2004a] Veillard, D., (2004a), "Libxml2 XmlTextReader Interface tutorial", Date Accessed: May 16, http://www.xmlsoft.org/xmlreader.html.

[Veillard 2004b] Veillard, D., (2004b), "The XML C parser and toolkit of Gnome", Date Accessed: May 16, http://www.xmlsoft.org/.

[Veillard 2004c] Veillard, D., (2004c), "The XSLT C Library for Gnome libxslt", Date Accessed: May 16, http://xmlsoft.org/XSLT/.

[W3C 1998] W3C, (1998), "Document Object Model (DOM) Level 1 Specification", W3C, Web page, Date Accessed: 01/20/2002, http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/.

[W3C 1999a] W3C, (1999a), "XML Path Language (XPath) Version 1.0 W3C Recommendation", W3C, Web page, Date Accessed: 01/20/2002, http://www.w3.org/TR/1999/REC-xpath-19991116.

[W3C 1999b] W3C, (1999b), "XSL Transformations (XSLT) Version 1.0", W3C, Web page, Date Accessed: 01/20/2002, http://www.w3.org/TR/xslt.

[W3C 2000a] W3C, (2000a), "Building XHTML Modules", W3C, Web page, Date Accessed: May 15, http://www.w3.org/TR/xhtml-building.

[W3C 2000b] W3C, (2000b), "Extensible Markup Language (XML) 1.0", W3C, Web page, Date Accessed: 01/20/2002, http://www.w3.org/TR/2000/REC-xml-20001006.

[W3C 2001a] W3C, (2001a), "XML Information Set Version 1.0", Webpage, Date Accessed: 6/2002, http://www.w3.org/TR/2001/REC-xml-infoset-20011024.

[W3C 2001b] W3C, (2001b), "XML Linking Language (XLink) Version 1.0 W3C Recommendation", W3C, Web page, Date Accessed: 05/15/2003, http://www.w3.org/TR/2001/REC-xlink-20010627/.

[W3C 2001c] W3C, (2001c), "XML Schema Part 1: Structures W3C Recommendation", Date Accessed: May 16, http://www.w3.org/TR/xmlschema-1/.

[W3C 2002] W3C, (2002), "XQuery 1.0: An XML Query Language W3C Working Draft", W3C, Web page, Date Accessed: 01/20/2002, http://www.w3.org/TR/2002/WD-xquery-20021115/.

[W3C 2003] W3C, (2003), "XSL Transformations (XSLT) Version 2.0 W3C Working Draft", W3C, Web page, Date Accessed: Feb 3, http://www.w3.org/TR/xslt20.

[Wagner, Graham 1997] 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.

[Walsh] Walsh, N., "diffmk", Date Accessed: May 11, http://sourceforge.net/projects/diffmk/.

[Walsh 2002] Walsh, N.,(2002),*Docbook: The Definitive Guide*, Reilly & Associates, Inc.

[Wang, DeWitt, Cai 2003] Wang, Y., DeWitt, D. J., and Cai, J.-Y., (2003), "X-Diff: An Effective Change Detection Algorithm for XML Documents", in Proceedings of 19th International Conference on Data Engineering (ICDE'03), Bangalore, India, pp. 519-530.

[Wong 1998] Wong, K., (1998), "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.

[XML:DB 2002] XML:DB, (2002), "XML:DB Initiative for XML Databases", Date Accessed: April 11, http://www.xmldb.org.