

Supporting Source Code Difference Analysis

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

Abstract

The paper describes an approach to easily conduct analysis of source-code differences. The approach is termed meta-differencing to reflect the fact that additional knowledge of the differences can be automatically derived. Meta-differencing is supported by an underlying source-code representation developed by the authors. The representation, srcML, is an XML format that explicitly embeds abstract syntax within the source code while preserving the documentary structure as dictated by the developer. XML tools are leveraged together with standard differencing utilities (i.e., `diff`) to generate a meta-difference. The meta-difference is also represented in an XML format called srcDiff. The meta-difference contains specific syntactic information regarding the source-code changes. In turn this can be queried and searched with XML tools for the purpose of extracting information about the specifics of the changes. A case study of using the meta-differencing approach on an open-source system is presented to demonstrate its usefulness and validity.

1. Introduction

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 the 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, i.e., UNIX utilities `diff` and `patch`, take a character-based document view of source-code files. Obviously, this is not the programmer-centric view of the source code as originally entered into the file. In the `diff` view, all lexical information in the file is preserved, differences are changes to characters, but no information about the syntactic-structure is explicitly stored.

An alternative approach is a compiler-centric or data view of the source code (i.e., post parsing). In this view

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

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 through simple queries. For example, one can automatically determine that a condition in an if-statement was added in a given change. Meta-differencing allows software engineers to easily ask complex questions about the differences between two versions of software. In order to realize meta-differencing, an underlying infrastructure with a new representation of source code is necessary.

Meta-differencing is built on top of an XML representation of the source code, specifically srcML¹ [4, 5, 19]. This representation explicitly embeds high-level syntactic information within the source code in such a way as to 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 things as lightweight fact extraction and construction of source models (e.g., call graphs) using standard XML tools such as XQuery and XPath. It also supports the embedding of meta-information into the source (e.g., hyperlinks) and source-code transformations.

By using this approach we can easily determine what type of syntactic construct or program entity was modified. More importantly, we can track the morphological changes that occur within the source. Because both the source code and the difference are

¹ Pronounced "source ML"

represented in XML, the transformational language, XSLT, can be used to model these changes.

The next section (2) describes the current state of affairs with regards to conducting analysis of source-code differences. Related research on differencing approaches is examined in perspective to our work and their limitations to this problem. Section 3 presents an overview of the underlying infrastructure (i.e., srcML) to address our problem. We also summarize supporting evidence that srcML is a sound basis for difference analysis. Section 4 describes our approach and introduces the concept and details of meta-differencing.

In an attempt to validate our approach we present a small case study in section 5. We demonstrate our approach by examining HippoDraw, a well known open-source application framework. We study the differences between two versions of this system by asking a number of questions regarding the nature of the changes.

2. Current Problems and Solutions

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 [6]. Most commercial tools use textual deltas with syntactic and semantic deltas used for operations such as merging [23]. The most popular differencing algorithms and tools, i.e., UNIX utilities `diff` [16] and `patch`, take a character-based document view of source code files. In the case of the extraction of textual differences using `diff` [16] (or a variation), line differences between two files are found with added/deleted/changed lines recorded by their line numbers in the files and by changed content. The lines in the file are compared using the LCS (Longest Common Subsequence) algorithm [17] which 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 then be used to generate a modified version of the document given 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 the original textual information and context of the source code.

Differences can also be found at the syntactical level. In general this is a difference comparison of two AST's. Syntactical differencing has been primarily used for improving merging of differences [23]. The main

drawback is that the algorithm must understand the syntax of the source code, and the difference comparison is typically not as efficient as LCS.

Of particular interest is the work by Hunt, on LTDIFF [13] 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 [13]. 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) [13] 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 [14].

In semantic differencing, such as in [12], code segments are considered equivalent if they perform the same computation. The general case is undecidable therefore limited heuristics have been developed [15].

In the work by Magnusson et al. [18], 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 [30] a fine-grained version-control infrastructure for trees, such as AST's is provided. Both of these cases are complete source-code management systems and are not directly compatible with other versioning systems.

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 [22] and is implemented using XML for a simple programming language [8].

2.1. Limitations of Character-Based Solutions

Textual-differencing approaches are limited to a line-level granularity. That is, the difference algorithms find line-based differences, represent the differences as changes between lines, and apply the differences to individual lines in the patched file. Extraction is limited to differences between multiple versions of the same file and the application of a patch is on an entire file.

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 regards to the programming-language syntax and it can only be utilized in the context of the original document. Secondly, 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. Thirdly, 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 of 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.

These textually-based tools are often at odds with the syntactic structure of the source code crosscutting the structure that the developer understands and hindering the analysis, manipulation, and tool construction based on source-code differences. During the extraction of differences this character-based view hinders the support of more abstract extraction mechanisms. Extraction of syntactic-level elements is particularly difficult. For example, a developer may be interested in extracting all the changes to a specific function or only comment changes.

Also problematic in the character-based view is such things as expressing the difference between two versions in a form that a developer easily understands. Determining the characteristics of a set of changes and placing them into categories is also difficult. This would be useful for detecting when only documentary changes occur, i.e., changes only to white space and comments, with no changes to the interface or logic.

In general there is no practical means to search the differences for particular code patterns or construct queries on the differences, e.g., detection of the version in which a particular function last changed.

With regard to patching, the character-based view inhibits applying fine-grained differences. The differences cannot be applied to individual syntactic elements, specific locations in the source code, or based on a category that the patch is in. A large patch (such as for an API change) cannot be split into a series of patches, where each patch only changes small groups of functions at a time.

3. Supporting Difference Analysis

To realize meta-differencing we utilize an underlying XML representation, namely srcML, for the source code that explicitly embeds syntactic information with the source. 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-compileable code and code fragments.

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 representation must preserve all textual information in its original order, and allow access to all textual information at the same level. It must also be able to handle source code in a realistic state, i.e., code fragments and un-compileable code

3.1. Overview of Approach

Figure 1 presents an overview of how the source code, srcML, and meta-differencing tie together. At the bottom layer is the source code and the results of `diff` represented as simple text files. The next layer consists of documents in srcML. We currently have a fairly robust prototype to translate C++ to srcML that forms the basis for much of this research. Once the source-code file is represented in srcML one can utilize a wide array of XML tools (e.g., XPath, XQuery, etc.) to construct higher-level models. For example a static call graph model can very easily be constructed from a given srcML document. 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.

Meta-differencing is implemented using a combination of `diff` and srcML by translating line differences into srcML. This combines many advantages

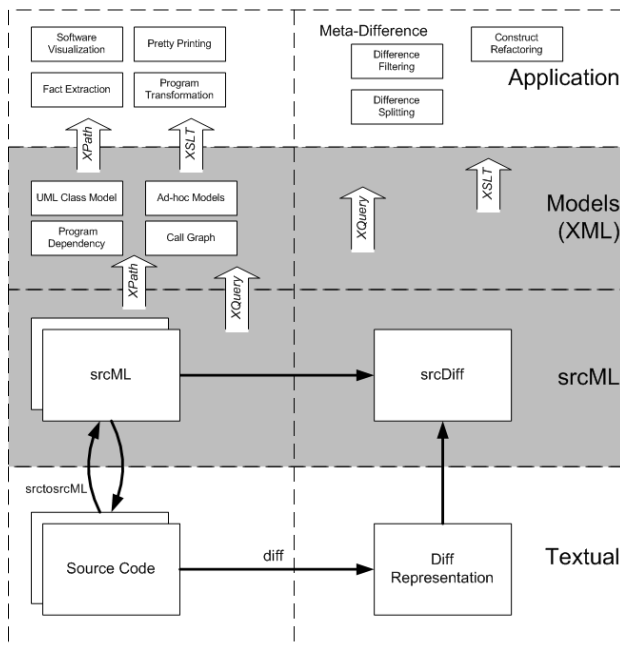


Figure 1. Realizing meta-differencing by leveraging XML representations, XML tools, and diff.

(i.e., 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, along with higher-level source models, can then be opportunistically combined with XML tools to support analysis and development tasks.

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. We must now raise the textual level of the differences to a syntactic level just as we did for the source code. We use a syntactical XML representation of the source-code differences combining the srcML versions of the original and modified documents along with their textual differences. This format, called *srcDiff*, includes both versions of the document and the required difference information.

As a result of using an XML representation, locations in the source code can be referenced using XPath, the XML language for addressing locations inside XML documents [29]. These addresses can be used to extract particular source-code elements [4] and to form links to an element. Once in a syntactic-level XML format standard XML tools, API's 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 section describes srcML in more detail, followed by a short description of how XML tools integrate with the representation. Then an example of using srcML as the basis for a lightweight C++ fact extractor is presented along with how abstract source models can be tied into this representation. A description of how meta-differencing will be realized is then given with a discussion on some limitations of this approach.

3.2. Source Code Representation in srcML

srcML (SouRce Code Markup Language) [4, 5, 19] is an XML application 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 lexical information including comments, white space, preprocessor directives, etc. from the original source-code file. This permits transformation equality between

the representation in srcML and the related source-code document.

A lightweight data view of source code is supported by the addition of XML elements to represent syntactic structures such as functions, classes, statements, and entire expressions. Other structural information including macros, templates, and compiler directives (e.g., `#include`), are also represented. The data view stops at the expression level with only function calls and identifier names marked inside of expressions thus allowing reasonable srcML file sizes. The srcML for the simple program below is given in Figure 2.

```
#include <iostream>

// A function
void
f(int x)
{
    std::cout << x + 10;
}
```

```
<unit xmlns="http://www.sdml.info/srcML/src" xmlns:cpp="http://www.sdml.info/srcML/cpp">
<cpp:include>#<cpp:directive>include</cpp:directive><cpp:file>&lt;iostream&gt;</cpp:file></cpp:include>

<comment type="line">// A function
</comment><function><type>void</type>
<name>f</name><formal_params>(<param><type>int</type> <name>x</name></param></formal_params>
<block>{
  <expr_stmt><expr><name>std::cout</name> &lt;&lt; <name>x</name> + 10</expr></expr_stmt>
}</block></function>
</unit>
```

Figure 2. Example of srcML.

In this example we see that all of the original text is present, including the preprocessor directive `include` and all original comments. Some of the original text that are meta-characters in XML, e.g., `'&'`, have been encoded but all text is preserved. The documentary structure of the original text including spacing and lines are also preserved. The inserted XML tags allow for the addressing and location of textual elements according to their location in the XML document.

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

Once the document is in srcML locations in the srcML document (and corresponding locations in the textual source-code document) can be referred to using the XML addressing language XPath. For example, to refer to the second if-statement inside of a function named `foo` we can use the following XPath:

```
//function[name="foo"]/ block/if[2].
```

XPath can also be used to represent groups of elements, such as all functions. The capability to use XPath addresses is built into most XML tools and is used extensively in XML transformation languages such as XSLT. The XPath standard forms the base of the XML query language XQuery.

Unlike the physical representation of an address that a line number references, XPath addresses describe one of many possible paths to a location. The XML elements serve as reference points along the path. This makes XPath addresses much more resilient to changes in other parts of the document, unless they change the nested XML elements.

A C++ source code to srcML translator has been developed that takes advantage of the flexibility of the srcML representation for robust and lightweight conversion of C++ to srcML. Unprocessed source code (source code before the C preprocessor is run) is converted into the srcML format.

The translator uses a partial parsing approach based on island grammars [24, 25] allowing for the insertion of tokens around only the textual elements of interest, such as the beginning and ending of a statement or an identifier name. Textual items of little interest, such as the operators in an expression, are not marked up. This allows for very flexible parsing of well-formed source code, i.e., source code with structural elements such as semicolons and curly braces in the right place. This allows it to be able to translate incomplete and non-compilable source.

The translator is written using a recursive-descent parser permitting on-the-fly translation. Start elements marking the beginning of a statement are issued before the entire statement is parsed. This allows for the integration into memory-efficient stream-oriented XML processing (such as SAX) which is needed for scalability and for integration into development environments.

A number of options currently exist for representing source-code information (e.g., AST or ASG) in a XML data format namely, GXL [11], CppML [21], ATerms [28], GCC-XML, and Harmonia [3]. In these formats the AST (actually an ASG) of the source code, as output from a compiler intended for code generation, is stored in a data XML data format. These XML data views of source code, since they are based on the AST, are a “heavyweight” format that requires complete parsing of the original document and generation of the complete AST. Closely related to srcML is Badros’ work on JavaML [2], which is an XML application that provides an alternative representation of Java source code. Other work on source-code interchange formats includes the work by Malton et al [20]. While this work is not an XML application it has many of the same features as srcML and the other formats described previously.

3.3. srcML Based Fact Extraction

In order to demonstrate the capabilities of the srcML format, a lightweight fact extractor was created that utilizes XML tools, such as XPath and XSLT, to extract static information from C++ source code programs. The srcML translator was used with source-code files from a C++ Fact-Extractor Benchmark [26, 27] and the results of those experiments are presented in [4].

The benchmark consisted of a number of test cases and a number of different questions. Cases included incomplete and non-compilable code. The benchmark had previously been applied to a selection of C++ fact-extraction tools such as Acacia [1], Columbus [9], and Cppx [7].

Overall the lightweight approach of using XPath to query srcML documents with common XML tools was quite reasonable in comparison to the published results of the other parser-based tools. Our lightweight approach performed as well as many of the heavyweight parsing approaches. In addition the extraction of information in XML allows for integration into other tools and for a multitude of purposes. The work demonstrates that a lightweight and robust source-code representation can be queried for fact extraction using XPath.

4. Meta-Differencing

The format used to represent differences between versions of a srcML document is called *srcDiff*. *srcDiff* is an intensional version format [23] that embeds versioning information as tags inside of a srcML document. The difference documents contains both versions of the source-code documents simultaneously permitting querying and transformation that is an extension of querying and transformation of srcML documents.

Textual differences, as from the utility *diff*, are mapped to locations in the srcML document. The specific line changes are then mapped to source-code elements. Because the line changes at the textual level can crosscut the corresponding syntactic elements (e.g., a deleted line may include the keyword “if” and the condition of an if-statement) the syntactic elements must be matched to the line change. The deletion of a syntactic element may correspond to separate line changes that occur far from one other, e.g., an if-statement with a block.

These differences are marked in the *srcDiff* format with the addition of new tags to indicate which srcML elements are added to, or deleted from, the source code along with which srcML elements are in both versions.

The addition of the version information allows for queries on the differences between source-code documents to be an extension of source-code queries on

srcML. Information can be extracted that summarizes the number or types of changes, that is how many program elements were deleted, added, or replaced and what kind of program elements are changed. Also supported are queries involving the location of changes with respect to program elements, e.g., how many changes are located in a particular program element such as a particular method.

```

<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>
</diff:common>}</block></if>
</diff:old>}</block></function>
</diff:common>

```

Figure 3. A srcDiff program fragment with selected srcML markup. Textual Deletions are shown in strikethrough; textual additions are shown in bold.

4.1. srcDiff Format

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 3 presents a fragment of a srcDiff document. Textually deleted source code (and srcML) is shown in strikethrough, added code is shown in bold. Critical srcML elements are shown while a number are left out for brevity. Notice the additional tags to support the difference (i.e., `diff:common`, `diff:old`, `diff:new`) these will be detailed in section 4.3.

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

4.2. Merging Document Versions

The srcDiff format contains all the elements from both the original and modified versions of the source code simultaneously. Elements that are common to both deleted and 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. The elements may not overlap, that is one element must start and end before the other starts. This occurs when a complete element is deleted or added to the document, e.g., a complete statement is added or deleted:

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

One element may be completely nested inside another element. This occurs when an element is inserted inside of 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 in an if-statement:

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

Finally, two elements may share a common ending point. This occurs when the start of a statement is deleted in one version and similar statement is added. For example, 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 tag of the block is shared between both statements and the starting tag cannot be shared because of the preceding comment.

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>
```

Now that the different versions of the srcML documents are integrated into a single document the location of difference elements must be determined. In the next section we examine the insertion of difference elements into the document to mark these areas of change.

4.3. 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 examples the namespace alias `diff` is used. The namespace `diff` is composed of three elements:

- `<diff:common>` - Used to mark sections containing srcML elements and source code that are common to both versions of the document.
- `<diff:old>` - Used to mark sections containing srcML elements and source code that are present in the original version of the document but deleted in the modified version.
- `<diff:new>` - Used to mark sections containing srcML elements and source code that are not present in the original version of the document but added to the modified version.

An explicit element for replacement changes is not used here because they can be detected by a `<diff:old>` section followed immediately by a `<diff:new>` section.

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 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. A given location in the source code may exist in more than one difference element since the difference elements can be nested.

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 type `<diff:common>` along the difference axis are those that exist in both documents. Elements with a parent of type

`<diff:old>` along the difference axis are those 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.

The different srcML versions of the document, i.e., original and modified version, can be extracted from the srcDiff multi-version form of the document. The original document is the text and markup with a parent element of `<diff:common>` or `<diff:old>` along the difference axis. The modified version of the document is the text and markup with a parent element of `<diff:common>` or `<diff:new>` along the difference axis.

4.4. 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. 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).
- *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 single 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 `<diff:old>` or modified version and `<diff:new>`) was not well formed. When crosscutting occurs, combinations of textual differences must be correctly matched to each other so that the appropriate difference element can be wrapped around the section.

4.5. Generating 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 API's the "pull" API `TextReader` is 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, and taken from the modified srcML document in sections of added text. The decision to change sections is controlled by the textual-difference information, while whether to return from a nested section or embed a new section is based on the current markup. The section changes are marked with the difference elements, while the srcML output is from the appropriate input 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.

5. Case Study

In an attempt to validate our approach we performed a case study regarding changes to a source-code document. Current approaches to change analysis cannot easily address these fairly simple questions:

- Q1. Does the change only affect comments?*
- Q2. Are new methods added to an existing class?*
- Q3. Are there changes to pre-processor directives?*

Our approach is applied to an open source application, HippoDraw [10], to demonstrate that these questions can be addressed. HippoDraw is used to build data analysis applications and it consists of both a library and an application. The application contains approximately 60

KLOC of source code in over 400 C++ files for each version. The most recent versions at the time of the study, versions 1.4.0 and 1.5.1 are used in the case study. The following steps were applied on a 3 GHz PC running Linux:

- 1) The source code for both versions is first converted to srcML using the srcML translator. This took under two minutes for all 422 files from version 1.4.0 and all 423 files from version 1.5.1.
- 2) The utility `diff` is then individually applied to each file in the older version and the matching file in the newer version. This took 2 minutes for the 422 files.
- 3) For each file in the older version, the srcML of the original file, srcML of the newer version, and the output of the utility `diff` was used to build a srcDiff file. This was only applied to files that had differences between them. This took less than 10 minutes for the 144 files with changes.

5.1. Only Comment Change

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. XPath statements can be used to find comments that include 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 statements 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 statement 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 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.

5.2. New Method Added

In order to find new methods we first determine which classes exist in both versions. We use the following to extract the existing classes:

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

An XPath statement is then be applied to these classes to determine the names of new methods:

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

The command-line utility *xpath* allows us to execute the first XPath statement to find the class and then 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`.

5.3. Preprocessor Change

In srcML, preprocessor directives are in a separate namespace from the other language elements allowing for easier processing as a group. All preprocessor directives are in elements of the form `<cpp:*>` when the namespace alias `cpp` is used. Since preprocessor directives are on their own lines they can be directly extracted. All deleted preprocessor directives can be matched using the XPath statement:

```
//diff:old//cpp:*
```

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

```
//diff:new//cpp:*
```

Specific directives, e.g., `include`, can be found and specific information, e.g., included file name, can be extracted using an XPath of the form:

```
//diff:old//cpp:include/cpp:file
```

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 to file `FunctionController.h` additionally includes the file `axes/AxesType.h`.

6. Limitations

Because the srcDiff translator uses textual differences from the utility `diff` it has a line-based granularity of differences. This leads to more changes than is actually necessary, e.g., two lines of expressions with a single variable difference. Further processing of individual differences, such as comparing beginning and ending of lines, could reduce this to a finer granularity level with no increase in complexity. However, this can still miss common sequences within the line. Only the application of the LCS on the elements in the lines will permit true fine-grained differencing. We are currently working on how to integrate this into the srcDiff translator.

One of the goals of meta-differencing is to support a variety of applications in the program analysis and

development areas. The use of the general infrastructure of XML along with independent tools to convert from the textual representations of both source-code documents and the differences allows the use of meta-differencing for a customizable variety of applications in a wide variety of tools.

In terms of the analysis of differences, the meta-difference format allows the exploration of the changes that developers make to a specific project. This analysis can be used for such things as difference categorization, identification of program areas where changes are most prevalent, and calculations of metrics.

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

7. Conclusions

Based on our initial investigations meta-differencing proves to be a very useful, reasonably efficient, and robust approach to supporting the analysis of source-code differences. The lightweight XML approach is easy to use and integrates well with other tools.

In general, the approach benefits the greater software-engineering community by supporting both the analysis of individual source-code differences and version histories. This will support the automatic identification of the specific syntactic nature of a change (e.g., a conditional added to an if-statement), the creation of software metrics based on a change, and assist in impact analysis based on changes. Developers/researchers can readily analyze what is actually happening during the practice of software development and we hope our tools will support the efforts to empirically analyze version histories in a systematic fashion.

8. Acknowledgements

We would like to thank the program committee for their excellent comments. This work was supported in part by a grant from the National Science Foundation (CCR-02-04175).

9. References

[1] Acacia, "Acacia - the C++ Information Abstraction System", Date Accessed: 11/01/2001, Online at <http://www.research.att.com/sw/tools/Acacia/>, 2001.

- [2] Badros, G. J., "JavaML: A Markup Language for Java Source Code", in Proceedings 9th International World Wide Web Conference (WWW9), Amsterdam, May 13-15 2000
- [3] Boshernitsan, M. and Graham, S. L., "Designing an XML-Based Exchange Format for Harmonia", in Proceedings Seventh Working Conference on Reverse Engineering (WCRE'00), November 23-25 2000, pp. 287-289.
- [4] Collard, M. L., Kagdi, H. H., and Maletic, J. I., "An XML-Based Lightweight C++ Fact Extractor", in Proceedings 11th IEEE International Workshop on Program Comprehension (IWPC'03), Portland, OR, May 10-11 2003, pp. 134-143.
- [5] Collard, M. L., Maletic, J. I., and Marcus, A., "Supporting Document and Data Views of Source Code", in Proceedings ACM Symposium on Document Engineering (DocEng'02), McLean VA, November 8-9 2002, pp. 34-41.
- [6] Conradi, R. and Westfechtel, B., "Version Models for Software Configuration Management", ACM Computing Surveys, 30, 2, June 1998, pp. 232 - 282.
- [7] CPPX, "CPPX - Open Source C++ Fact Extractor", Online at <http://swag.uwaterloo.ca/~cppx/>, 2001.
- [8] Emmerich, W., Mascolo, C., and Finkelstein, A., "Implementing Incremental Code Migration with XML", in Proceedings 22nd International Conference on Software Engineering (ICSE'00), June 4-11 2000, pp. 397 - 406.
- [9] Ferenc, R., Magyar, F., Beszedes, A., Kiss, A., and Tarkaiainen, M., "Columbus – Tool for Reverse Engineering Large Object Oriented Software Systems", in Proceedings SPLST 2001, June 2001 2002, pp. 16-27.
- [10] HippoDraw, "HippoDraw: Main Page", Online at www.slac.stanford.edu/grp/ek/hippodraw/index.html, 2004.
- [11] Holt, R. C., Winter, A., and Schürr, A., "GXL: Toward a Standard Exchange Format", in Proceedings 7th Working Conference on Reverse Engineering (WCRE '00), Brisbane, Queensland, Australia, November, 23 - 25 2000, pp. 162-171.
- [12] Horwitz, S. and Reps, T. W., "The Use of Program Dependence Graphs in Software Engineering", in Proceedings International Conference on Software Engineering (ICSE), Melbourne, Australia, May 11 - 15 1992, pp. 392 - 411.
- [13] Hunt, J. J., "Extensible Language-Aware Differencing and Merging", in PhD thesis: Universitt Karlsruhe, 2001.
- [14] Hunt, J. J., "Fast Semi-Semantic Differencing and Merging", Date Accessed: 02/01/2004, Online at <http://wwwswt.fzi.de/cocoon/mount/swt/mitarbeiter/jjh/>, 2004.
- [15] Hunt, J. J. and Tichy, W. F., "Extensible Language-Aware Merging", in Proceedings IEEE International Conference on Software Maintenance (ICSM'02), Montreal, Canada, October 3-6 2002, pp. 511-520.
- [16] Hunt, J. W. and McIlroy, M. D., "An Algorithm for Differential File Comparison", AT&T Bell Labs Inc. 1976.
- [17] Hunt, J. W. and Szymanski, T. G., "A Fast Algorithm for Computing Longest Common Subsequences", CACM, 20, 5, May 1977, pp. 350 - 353.
- [18] Magnusson, B., Asklund, U., and Minor, S., "Fine-grained revision control for collaborative software development", in Proceedings 1st ACM Symposium on Foundations of Software Engineering (FSE'93), 1993, pp. 33-41.
- [19] Maletic, J. I., Collard, M. L., and Marcus, A., "Source Code Files as Structured Documents", in Proceedings 10th IEEE International Workshop on Program Comprehension (IWPC'02), Paris, France, June 27-29 2002, pp. 289-292.
- [20] Malton, A. J., Cordy, J. R., Cousineau, D., Schneider, K. A., Dean, T. R., and Reynolds, J., "Processing Software Source Text in Automated Design Recovery and Transformation", in Proceedings IEEE 9th International Workshop on Program Comprehension (IWPC'01), May 12-13 2001, pp. 127-134.
- [21] Mammas, E. and Kontogiannis, C., "Towards Portable Source Code Representations using XML", in Proceedings 7th Working Conference on Reverse Engineering (WCRE '00), November, 23 - 25 2000, pp. 172-182.
- [22] Mascolo, C., Picco, G. P., and Roman, G.-C., "CodeWeave: Exploring Fine-Grained Mobility of Code", Journal of Automated Soft Engineering, 2005, pp. (to appear).
- [23] Mens, T., "A State-of-the-Art Survey on Software Merging", IEEE Transactions on Software Engineering, 28, 5, May 2002, pp. 449 - 462.
- [24] Moonen, L., "Generating Robust Parsers using Island Grammars", in Proceedings 8th IEEE Working Conference on Reverse Engineering (WCRE'01), October 2-5 2001, pp. 13-24.
- [25] Moonen, L., "Lightweight Impact Analysis using Island Grammars", in Proceedings of the 10th International Workshop on Program Comprehension (IWPC'02), Paris, France, 2002, pp. 219-228.
- [26] Sim, S. E., Holt, R. C., and Easterbrook, S., "On Using a Benchmark to Evaluate C++ Extractors", in Proceedings 10th International Workshop on Program Comprehension, Paris, France, 2002, pp. 114-123.
- [27] Sim, S. E. and Kienle, H. M., "Concordance for CppETS 1.1 Test Buckets", Date Accessed: 11/15/2002, Online at <http://cedar.csc.uvic.ca/twiki/kienle/pub/IWPC2002/Benchmark/concordance11.htm>, 2002.
- [28] van den Brand, M., Sellink, A., and Verhoef, C., "Current Parsing Techniques in Software Renovation Considered Harmful", in Proceedings 6th International Workshop on Program Comprehension (IWPC'98), Ischia, Italy, June 24-26 1998, pp. 108 - 117.
- [29] W3C, "XML Path Language (XPath) Version 1.0 W3C Recommendation", Date Accessed: 01/20/2002, Online at <http://www.w3.org/TR/1999/REC-xpath-19991116>, 1999.
- [30] Wagner, T. A. and Graham, S. L., "Incremental Analysis of Real Programming Languages", in Proceedings ACM SIGPLAN Conference on Programming Language Design and Implementation, 1997, pp. 31-43.