STRUCTURAL ANALYSIS OF SOURCE-CODE CHANGES IN LARGE SOFTWARE
THROUGH SRCDIFF AND DIFFPATH

A Thesis

Presented to

The Graduate Faculty of The University of Akron

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

Michael J. Decker

August, 2012
STRUCTURAL ANALYSIS OF SOURCE-CODE CHANGES IN LARGE SOFTWARE
THROUGH SRCDIFF AND DIFFPATH

Michael J. Decker

Thesis

Approved: 

Advisor
Dr. Michael L. Collard

Faculty Reader
Dr. Kathy J. Liszka

Faculty Reader
Dr. Chien-Chung Chan

Accepted: 

Dean of the College
Dr. Chand Midha

Dean of the Graduate School
Dr. George R. Newkome

Department Chair
Dr. Yingcai Xiao

Date
ABSTRACT

Comprehension of changes in large software is a major challenge in software development. The srcDiff format supports the analysis of differences in large software systems, however, previous tools had were not complete. A new tool is presented that generates the srcDiff format from two versions of a software system. Furthermore, a new format, diffPath, is presented which is an XPath expression that uniquely identifies a source code change. A tool to generate diffPath expressions has also been created. Both tools are evaluated on releases of the GNU GCC compiler.
DEDICATION

I dedicate this thesis to all the professors who helped me throughout my undergraduate and graduate curriculum.

I also dedicate this thesis to my family, my mother, sister, and my father now deceased.

Most importantly, I dedicated this thesis to God, whose inspiration in my life has given me strength to overcome the obstacles in my path.
TABLE OF CONTENTS

<p>| LIST OF TABLES | vii |
| LIST OF FIGURES | viii |
| CHAPTER | |
| I. INTRODUCTION | 1 |
| II. SRCML FORMAT | 4 |
| III. SRCDIFF FORMAT | 8 |
| IV. DIFFPATH | 14 |
| V. SRCDIFF IMPLEMENTATION | 20 |
| General Algorithm | 21 |
| Main Differencing Algorithm | 23 |
| Shortest Edit Script | 24 |
| Shortest Edit Script Structure | 24 |
| Move Detection | 26 |
| Handling Differences | 27 |
| Output of Common Node Sets | 28 |
| Output of Pure Operations | 28 |
| Output of Change Operations | 29 |
| One-to-One Change | 31 |
| Many-to-Many Change | 41 |</p>
<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. A list of tags in srcML ordered by category, then alphabetically</td>
<td>6</td>
</tr>
<tr>
<td>2. The namespace and tags added by srcDiff</td>
<td>8</td>
</tr>
<tr>
<td>3. The attributes available on <code>diff:delete</code> and <code>diff:insert</code> tags</td>
<td>11</td>
</tr>
<tr>
<td>4. Summary of srcML/srcDiff elements with additional information in</td>
<td>18</td>
</tr>
<tr>
<td>complex diffPath expressions</td>
<td></td>
</tr>
<tr>
<td>5. List of nest elements and what is possibly to nest</td>
<td>36</td>
</tr>
<tr>
<td>6. List of versions of GCC used for the srcDiff and diffPath case</td>
<td>56</td>
</tr>
<tr>
<td>study</td>
<td></td>
</tr>
<tr>
<td>7. Language to Extensions used by src2srcml and srcdiff</td>
<td>57</td>
</tr>
<tr>
<td>8. Results of applying srcML and srcDiff to GCC</td>
<td>59</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figures</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Source-code of a function that computes the sum of integers in an array</td>
<td>4</td>
</tr>
<tr>
<td>2. The same source code in Figure 1 has been converted into srcML. The original text is bold for clarity</td>
<td>5</td>
</tr>
<tr>
<td>3. An example of a srcML archive containing a header file and implementation file</td>
<td>7</td>
</tr>
<tr>
<td>4. srcML of the <em>sum</em> function to sum a double array. The original text is bold for clarity</td>
<td>9</td>
</tr>
<tr>
<td>5. srcDiff document containing the srcML of Figure 2 and Figure 4. The original text is bold for clarity and the srcDiff is in italics</td>
<td>10</td>
</tr>
<tr>
<td>6. A srcDiff archive with three files: one same, one deleted, and one inserted</td>
<td>12</td>
</tr>
<tr>
<td>7. Two simple diffPath expressions of the first two changes in Figure 5. The steps/elements are bold for clarity. Together they represent the change of a return type</td>
<td>14</td>
</tr>
<tr>
<td>8. An added if statement with the variable c in the then changed to d. The original text is bold and the srcDiff is in italics. srcDiff common is in non italics</td>
<td>16</td>
</tr>
<tr>
<td>9. The three diffPath expressions formed from Figure 8. The steps are highlighted in bold and the rest is left non bold</td>
<td>16</td>
</tr>
<tr>
<td>10. A more complex diffPath of the first diffPath in Figure 7. The steps are bold for clarity, and the decorated information, predicates are left non bold</td>
<td>17</td>
</tr>
<tr>
<td>11. Two ways of marking up the predicate paths. Shows what a function would look like with and without a complex name for both ways. The single step is bold, and the remaining is left normal</td>
<td>19</td>
</tr>
<tr>
<td>12. General algorithm to compute the srcDiff of two files. Handles the conversion to srcML, converting into an internal format, and the start of the differencing process</td>
<td>22</td>
</tr>
<tr>
<td>13. Main differencing algorithm that is recursively applied on matched node sets</td>
<td>25</td>
</tr>
</tbody>
</table>
14. Two versions of a piece of source code where a move occurred. The statement ‘a;’ has been moved from the beginning to the end .................................................................26

15. Two versions of a piece of source code where a move occurred. The statement ‘a;’ has been moved from the beginning to the end .................................................................27

16. The output and trimming of white space in a pure operation .................................30

17. Two versions of a piece of source code where the single statement ‘c;’ is inserted ..30

18. srcDiff showing the inserted code from Figure 17 ....................................................30

19. One-to-one change handling for a syntax match. Shows handling of white space, output of common sequences, and the recursive call .........................................................32

20. Source-code where the statement ‘a;’ is wrapped with an if .................................35

21. srcDiff of the source-code that has been nested in Figure 20 .................................35

22. Output of a nested node sets, including the handling for white space, the superset and the recursive call ........................................................................................................38

23. Output of a change, including the handling of white space .......................................40

24. Example source code where a many to many occurs .............................................41

25. An incorrect mapping of statements where order is not preserved .......................42

26. Main dynamic programming algorithm to pair node sets ........................................44

27. Pseudo-code for checking in the x-axis direction .....................................................46

28. Pseudo code for checking in the y axis direction .....................................................48

29. Pseudo code for checking on the diagonal .............................................................49

30. Scatter plot of the time to srcDiff plotted against the number of non white space changes .........................................................................................................................61

31. Scatter plot of time to create diffPaths from srcDiff using srcDiff lines of code as the independent variable ..........................................................................................63
CHAPTER I
INTRODUCTION

Lehman states that a software system must undergo change to maintain its usefulness, if it does not the users will grow more dissatisfied. He also states that as a system is changed, the system will continually grow more and more complex unless something is done to correct the complexity [1-3]. From this one can gather that change is necessary for the success of software. However, changes to a system make the system more complex and more difficult to manage or understand. At some point it may become too complex to apply changes to the system. This can lead to a large amount of effort to reduce the complexity. Changes can also introduce new features that need to be tested and adequate regression testing must be provided to maintain previously added functionality and to test that no errors were introduced. Successfully managing the changes to a system, knowing the impact that changes will have, and what is affected is also necessary.

In small systems, change is a small problem, and may be able to be surmised with only a developer or a few developers, however, as the size of software increases, so does the size of the problem. These words are closely related to the sentiments expressed by Edsger Dijkstra in “The humble programmer” [4]. That is, when there were no
computers, programming was not a problem, however, now that capability of computers has become much greater, so has the difficulty in programming.

In meaningful systems, software development requires many developers, not all of which are in the same location or the same country. Describing the change contributed by so many developers and providing the necessary testing and fault detection needed to maintain a system’s health is a full-time task that can require much more than a few developers, and it can be too much to understand for the many. Analysis of change, especially in large systems, such as describing the changes to a system or analyzing the overall affect, change impact, on the system may help to reduce the cost of maintaining the system, and it can also help to reduce the amount of effort in maintenance tasks.

In this thesis, two formats will be discussed that provide for the analysis of software change. The first is the srcDiff format [5, 6], an XML format based off of srcML [7, 8]. srcML is an XML format that annotates source-code text with information from the Abstract Syntax Tree (AST). srcDiff extends srcML by including additional XML tags for the marking of changes in source code. Previous tools have been implemented to convert two versions of source code into the srcDiff format, usually trying to apply line differences to the syntax based srcML. However, these tools were incomplete. As part of the contribution for this thesis, a new srcDiff tool is implemented that efficiently and accurately converts two versions of source code into the srcDiff format.

A new format, diffPath, is introduced that describes the address of a change in the form of an XPath expression. Also introduced is a tool to create diffPath expressions
from the srcDiff format that is both accurate, scaleable, and can be used to support the evolution of source code. Both implementations and formats are evaluated on an extensive portion of the releases of the GNU GCC compiler [9].

CHAPTER II begins with a description of the srcML format. Then, the srcDiff extensions and the srcDiff format are described in CHAPTER III. In CHAPTER IV the diffPath format is presented. CHAPTER V describes the tool that creates srcDiff documents from two versions of source code. The details of the tool to create diffPath expressions are presented in CHAPTER VI, and in CHAPTER VII a case study is presented on the GNU GCC compiler. CHAPTER VIII provides the related work and in CHAPTER IX, the conclusion and future work is provided.
CHAPTER II

SRCML FORMAT

srcML is an XML format for the representation of source code. srcML wraps lexical tokens in the source code with XML elements that correspond to information from the Abstract Syntax Tree (AST). srcML also seeks to preserve the programmers view of source-code. This means that the original source-code text, including any preprocessor instruction, is completely preserved in the srcML format. The only exception is the possible normalization of line endings. With srcML it is possible to convert the original source code of a software system into the srcML format and then convert it back to the original-source code by removing the tags [7, 8, 10]. Figure 1 shows the source-code of a function that computes the sum of the integers in an array. Figure 2 shows the same source-code in the resulting srcML format.

```c
int sum(int integer_array[], int length) {
    int sum = 0;
    for(int i = 0; i < length; ++i) {
        sum += integer_array[i];
    }
    return sum;
}
```

Figure 1. Source-code of a function that computes the sum of integers in an array.
Figure 2. The same source code in Figure 1 has been converted into srcML. The original text is bold for clarity.

As shown in Figure 1 and Figure 2, the srcML tags form syntactic constructs that tightly wrap themselves around the tokens. For instance, function is wrapped with a `function` tag as soon as the function prototype begins and ends as soon as the function block ends. Similarly, the `for` tag begins before the `for` keyword and ends after the end of the for’s block. White space is only included in tags where it must be, and otherwise is...
left unmarked and untouched with the exception of the possible normalization of newlines. Figure 2 shows only a small sampling of the tags that are currently included in the srcML format. Table 1. shows the complete list of the current srcML tags [10].

srcML has a wide variety of tags. Some, such as `krparameter_list` and `package`, are language specific. Others, such as those in the ‘Extra Markup’ category, are extra markup options for more fine-grained markup.

Table 1. A list of tags in srcML ordered by category, then alphabetically.

<table>
<thead>
<tr>
<th>Category</th>
<th>srcML Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>File/Project</td>
<td>unit</td>
</tr>
<tr>
<td>Statement</td>
<td>asm, block, break, case, comment, condition, constraint, continue, decl, decl_stmt, default, do else empty_stmt, enum, expr, expr_stmt, extern, for, goto, if, incr, index, init, label, macro, name, namespace, range, requires, switch, template, then, type, typedef, using, while</td>
</tr>
<tr>
<td>Function/Method</td>
<td>argument, argument_list, call, function, function_decl, param, parameter_list, return, specifier</td>
</tr>
<tr>
<td>Class</td>
<td>class, class_decl, constructor, constructor_decl, destructor, destructor_decl, friend, member_list, private, protected, public, super</td>
</tr>
<tr>
<td>Struct and Union</td>
<td>struct, struct_decl, union, union_decl</td>
</tr>
<tr>
<td>Exception</td>
<td>catch, throw, throws, try</td>
</tr>
<tr>
<td>K&amp;R C</td>
<td><code>krparam</code>, <code>krparameter_list</code></td>
</tr>
<tr>
<td>Java</td>
<td>extends, finally, implements, import, package</td>
</tr>
<tr>
<td>Extra Markup</td>
<td>lit:literal, type:modifier, op:operator</td>
</tr>
<tr>
<td>Debug</td>
<td>marker, mode, parse</td>
</tr>
<tr>
<td>Misc</td>
<td>escape</td>
</tr>
</tbody>
</table>

The `unit` tag is the root element of a srcML document. It represents both the root for an entire software system as well as an individual file. When representing an entire software system, the root `unit` tag will contain another `unit` tag nested within itself. There will be one nested `unit` tag for each source-code file marked up in srcML.
document that contains nested \textit{unit} elements is collectively known as a srcML archive \cite{10}. It is also possible to have a srcML archive with only one nested \textit{unit} tag. Figure 3 shows a sample srcML archive consisting of the file from Figure 1 and an appropriate header file. The contents of the files are left out for brevity.

\begin{verbatim}
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<unit xmlns="http://www.sdml.info/srcML/src">
  ...
  ...
</unit>
\end{verbatim}

Figure 3. An example of a srcML archive containing a header file and implementation file.

Since any srcML document is an instance of XML, any commonly known XML technologies, such as XPath and XSLT, can be used to collect information and transform the XML and underlying source-code. With an entire project/system in a single srcML archive, srcML is a powerful format for fact extraction and source-code transformation. For more information of using srcML for fact extraction and source-code transformation see \cite{10, 11}. 


CHAPTER III

SRCDIFF FORMAT

The srcDiff format is an extension to the srcML format. It includes the addition of a new namespace and three tags [5]. Table 2 summarizes the namespace and tag additions

Table 2. The namespace and tags added by srcDiff.

<table>
<thead>
<tr>
<th>Prefix:Namespace</th>
<th>xmlns:diff=&quot;<a href="http://www.sdml.info/srcDiff">http://www.sdml.info/srcDiff</a>&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elements</td>
<td>diff:common, diff:delete, diff:insert</td>
</tr>
</tbody>
</table>

In essence, the srcDiff format is a structure to represent multiple versions of a source-code file or system in a single XML document. A srcDiff document not only contains the delta between the two files or systems, such as would be given by the `diff` utility [12], but it also contains the unchanged lines marked up in srcML. A srcDiff document with versions of multiple files or an entire system is a srcDiff archive, and it is similar to a srcML archive. In fact, a srcDiff archive can be seen as the merger of two srcML archives containing both the set of changes and the unchanged srcML. A version of a software system represented as one of the completely preserved srcML archives can then be extracted from the srcDiff archive. Since srcML preserves the complete unaltered text and srcDiff preserves the srcML documents, then srcDiff preserves the original
source-code as well. The only exception is the possible change of line endings as described in CHAPTER 2 [5]. Figure 4 shows the srcML of a different version of the \textit{sum} function that returns the sum of doubles in an array. Figure 5 shows a srcDiff document containing both versions of the \textit{sum} function.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
filename="sum2.cpp"><cpp:include>
#<cpp:directive>include</cpp:directive>
<cpp:file>"sum.hpp"</cpp:file></cpp:include>

<function><type><name>double</name></type><name>sum</name><parameter_list>(
<param><decl><type><name>double</name></type><name>double_array</name><index>[]</index></name><index></name><name><name>int</name><index></name><name>length</name></index></index></param>)<parameter_list></function>
<decl_stmt><decl><type><name>double</name></type><name>sum</name><init><expr>0</expr></init></decl>;<decl_stmt>
<for><init><decl><type><name>int</name></type><name>i</name><init><expr>0</expr></init></decl>;<init>condition><expr><name>i</name><init>&lt;</init><name>length</name><init></expr></condition><incr><expr><name>i</name><init>++</init></expr></incr>)<block>{

<expr_stmt><expr><name>sum</name><init><expr><name>sum</name><init><expr><name>double_array</name><index>[]</index></name><index></name><expr></expr></expr><init><expr><name>i</name><init></expr></init></expr><init><expr></expr></init></init></expr>;<expr_stmt>

}</block></for>

<return><name>sum</name></name></return>

</block></function>

</unit>
```

Figure 4. srcML of the \textit{sum} function to sum a double array. The original text is bold for clarity.
As seen in Figure 5, the tags \texttt{diff:insert} and \texttt{diff:delete} correspond to the insertion and deletion of pieces of source-code. Although not shown in Figure 5, the \texttt{diff:common} tag is used to mark up an area as common source code/srcML.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
xmlns:diff="http://www.sdml.info/srcDiff" language="C++"
filename="sum.cpp|sum2.cpp"><cpp:include>
#include</cpp:include><cpp:file">
#include "sum.hpp"
</cpp:file></cpp:include>

<function><type><name>
  int<diff:delete type="change">
  <diff:insert type="change">double</diff:insert></name>
</type><name>sum</name><parameter_list>(
  <param><decl><type><name>
  int</name><diff:delete type="change"><diff:insert type="change">
  double</diff:insert></name></type><name>
  integer_array</name><index>[ ]</index></name>,
  <param><decl><type><name>
  int</name></type><name>
  length</name></name></param>)</parameter_list>
  <block>{
    <decl_stmt><decl><type><name>
    int</name><diff:delete type="change"><diff:insert type="change">double</diff:insert></name></type><name>
    sum</name>=<init><expr>0</expr></init></decl>
  </block>
  <for><init><decl><type><name>
    int</name></type><name>
    i</name></name>=<init><expr>0</expr></init>;</init>
  <condition><expr><name>
    i</name></expr><condition><expr><name>
    length</name></expr></condition>;<incr><expr><name>
    i</name></expr><incr></incr>)</for>
  <block>{
    <expr_stmt><expr><name>
    sum</name></expr><expr><name>
    integer_array</name><index>[ ]</index>+=<name><name>
    double_array</name><index>[ ]</index></name></expr"><expr>
  </block></for>
  <return><return><expr><name>
    sum</name></expr></return>
  </block></function>
```

Figure 5. srcDiff document containing the srcML of Figure 2 and Figure 4. The original text is bold for clarity and the srcDiff is in italics.
The unit tag by default creates an area of commonality. So, the \textit{diff:common} tag is not needed except in special situations, such as within a difference. These situations occur because of the tree structure of XML, i.e., added source code may not result in a leaf node in the XML document [5]. For example, the insertion deletion of an if statement that wraps a statement is marked with a \textit{diff:insert}, however, the statement in the if is in common to both versions and requires a \textit{diff:common} surrounding it to mark the statement as unchanged. More information on this subject will be presented when discussing the implementation of srcDiff in \textsc{CHAPTER V}.

Also able to be seen in Figure 5, attributes are defined on the \textit{diff:delete} and \textit{diff:insert} tags. These indicate a special kind of difference, and in the case of Figure 5, they indicate that the delete/insert sequence is a change. Table 3 provides a summary and description of the different types of attributes that are currently part of srcDiff.

Table 3. The attributes available on \textit{diff:delete} and \textit{diff:insert} tags.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>change</td>
<td>Indicates the type of difference is a change, i.e., a delete followed immediately by an insert</td>
</tr>
<tr>
<td>type</td>
<td>whitespace</td>
<td>Indicates that the insert or delete contains only whitespace.</td>
</tr>
<tr>
<td>move</td>
<td>unique id</td>
<td>Indicates that the same exact piece of code occurs somewhere else in the document, i.e., the same code was deleted in one place and inserted in another. The unique id indicates the matching inserted/deleted pair.</td>
</tr>
</tbody>
</table>

In a srcDiff archive, a file can be added or removed such that the file existed in one version, however, the file does not exist in the other. Figure 6 illustrates a srcDiff archive where a file is in common to both versions, one is deleted, and a third is added.
The units of deleted/inserted files are not wrapped with a `diff:delete` or `diff:insert`.
Instead, the filename attribute on the nested `unit` tag indicates the type of operation.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<unit xmlns="http://www.sdml.info/srcML/src" xmlns:diff="http://www.sdml.info/srcDiff" dir="old|new">
  <unit xmlns:cpp="http://www.sdml.info/srcML/cpp" language="C++" filename="a.cpp"/>
  <unit xmlns:cpp="http://www.sdml.info/srcML/cpp" language="C++" filename="b.cpp|"/>
  <unit xmlns:cpp="http://www.sdml.info/srcML/cpp" language="C++" filename="|c.cpp"/>
</unit>
```

Figure 6. A srcDiff archive with three files: one same, one deleted, and one inserted.

In the `dir` and `filename` attributes, the root directory and filename respectively, if the names, differ, they are separated with a ‘|’. For instance on the root `unit` tag, the attribute `dir` contains the two directories ‘old’ and ‘new’ separated by a ‘|’. Any name or path that is to the left of the ‘|’ is what existed in the previous version and what is on the right of the ‘|’ is the name or path that exists in the newer version. An empty name on either side means that the name or path did not exist in that version. For this example, this indicated that the root directory of the old version is ‘old’ and the root directory of the new version is ‘new’. In the first nested `unit`, the file exists in both versions with the same name. So, the filename is simply ‘a.cpp’. The second `unit` filename attribute is ‘b.cpp|’. This indicates that the old version filename was ‘b.cpp’, and there is no associated filename in the new version (a deleted file). Likewise, the third `unit` tag has no
previous version, but the new filename is ‘c.cpp’ (an inserted file). Although the actual contents are left out for brevity, the contents of the deleted and inserted files would be completely wrapped with \textit{diff:delete}’s and \textit{diff:insert}’s, respectively.
A single diffPath expression represents one difference between two versions of a source code document/system. Each diffPath is an XPath expression that uniquely defines the location/address of a difference in a srcDiff document. Since diffPath utilizes srcDiff, which is an extension of srcML, each step in the XPath expression represents a syntactic construct. In other words, diffPath completely describes the syntactical elements affected by a source-code deletion, insertion or modification.

Each diffPath expressions is also the complete expression path, includes all the steps, to a single entire change. That is, when an if statement is added, there will be one diffPath expression for that if statement. Nothing within the if is a diffPath expressions, because they are included with the added if tag. At its simplest, to be unique and completely contain a difference operation, a diffPath expression needs to only specify the location of the operation using a tag name and a position for each step. Figure 7 shows two examples of a simple diffPath expression.

```
/srC:unit[1]/srC:function[1]/srC:type[1]/srC:name[1]/diff:delete[1]/text()[1]
/srC:unit[1]/srC:function[1]/srC:type[1]/srC:name[1]/diff:insert[1]/text()[1]
```

Figure 7. Two simple diffPath expressions of the first two changes in Figure 5. The steps/elements are bold for clarity. Together they represent the change of a return type.
As shown in Figure 7, each step, separated by a ‘/’, in the XPath is a level of syntax, or equivalently represents a node in the parse tree. Each subsequent step, is a child node in the parse tree. Since it is possible to have multiples of the same syntactic type at the same level in the parse tree, the position or numbering, indicated between square brackets (index starts at one), selects which child node of that type at that level.

The first diffPath in Figure 7 means that the first text node (often all the text) in the first delete, in the name of the first type, which is within the first function, that is within the first unit (file), was deleted. In other words, the text of the first function’s return type was deleted. The next diffPath means that the text of the function’s return type was inserted at the same point the first was deleted. In combination, they mean that the function’s return type was changed.

In a srcDiff document, multiple elements can be contained within the same diff tag. Each of these elements is a diffPath expression. Only elements directly contained within a diff tag will become a diffPath expression. A possible exception to this is when there is a difference operation contained within another difference. For instance, when an if statement is added around a pre-existing statement and a variable within the statement is changed, then the if statement is one change operation (diffPath) and the deleted and inserted variable (changed variable) within the if statement are each individual diffPath expressions. Figure 8 presents the complete srcDiff of an if statement where a variable name has been changed. Figure 9 reveals the resulting diffPath expressions that are created.
In Figure 8, an expression statement ‘a = b + c’ was wrapped with an if statement. However, the internal statement has been changed to ‘a = b + d’, i.e., the ‘c’ was changed to a ‘d’.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
    <diff:insert>
        <if>
            if <condition>
                (expr)
            </condition>
            then
            <diff:common>
                <expr_stmt>
                    <expr>
                        <name>A</name> = <name>b</name> + <name>c</name> <diff:delete type="change">d</diff:delete> <diff:insert type="change">e</diff:insert>;
                    </expr>
                </expr_stmt>
            </diff:common>
        </then>
    </if>
</diff:insert>
```

Figure 8. An added if statement with the variable c in the then changed to d. The original text is bold and the srcDiff is in italics. srcDiff common is in non italics.

In Figure 9, there are now three diffPath expressions and not one. Although the if above contains all the changes, it does not represent each individual change. There is still a deleted and inserted variable. Each one of these is an individual change in the srcDiff document. All three expressions are required to fully describe this change.

```
<src:unit[1]/diff:insert[1]/src:if[1]
/src:unit[1]/diff:insert[1]/src:if[1]/src:then[1]/diff:common[1]/src:expr_stmt[1]/
/src:expr[1]/src:name[3]/diff:delete[1]/text()[1]
/src:unit[1]/diff:insert[1]/src:if[1]/src:then[1]/diff:common[1]/src:expr_stmt[1]/
/src:expr[1]/src:name[3]/diff:insert[1]/text()[1]
```

Figure 9. The three diffPath expressions formed from Figure 8. The steps are highlighted in bold and the rest is left non bold.

Simple diffPath expressions such as those in Figure 7 and Figure 9, represent all the affected nodes in the parse tree including the node that was immediately inserted or deleted. Other than the syntactic qualities of the change, the simple diffPath does not
necessarily reveal much information about the change. It is possible to decorate the
diffPath with more information, creating a more complex XPath expression, a complex
diffPath. Figure 10 shows the same change as the first diffPath in Figure 7 as a more complex diffPath.

Figure 10. A more complex diffPath of the first diffPath in Figure 7. The steps are bold for clarity, and the decorated information, predicates are left non bold.

The diffPath has been annotated with information about the different elements. Annotating diffPath expressions reveals more information that can be used for such things as finding patterns. From Figure 10 the filenames of the two versions are now clearly distinguished. The function type and name have also been added as additional information to the function. Looking at the type of the function, it is apparent that the type has changed. The text that was deleted in the type is ‘int’, and it has been replaced with the type ‘double’. However, although the complete description of the function contains both the new and old type, the diffPath still only points to the deleted return type, which is now recognized as ‘int’. Not every element is decorated with additional information, as can be seen in Figure 10. For instance, type, diff:delete, and text() remain unchanged from Figure 7.

Currently, the elements that do have additional information defined in a complex diffPath are summarized in Table 4. Mostly names of specific constructs are added. However, some such as functions and declaration statements add a type as well. Table 4
does not represent a definitive list. diffPath can be decorated with any amount of information, as long as it conforms to XPath and will produce a unique result. For instance, the srcML/srcDiff format allows for the additional markup of literals, operators, and modifiers. These could be added to a complex diffPath expressions, if they are present in the srcDiff document.

Table 4. Summary of srcML/srcDiff elements with additional information in complex diffPath expressions.

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Description</th>
<th>Additional Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Generic name can have another name inside</td>
<td>Text within leaf names is added</td>
</tr>
<tr>
<td>function</td>
<td>Function definition</td>
<td>Function name and return type</td>
</tr>
<tr>
<td>function_decl</td>
<td>Function declaration</td>
<td>Function name and return type</td>
</tr>
<tr>
<td>constructor</td>
<td>Constructor definition</td>
<td>Constructor name</td>
</tr>
<tr>
<td>constructor_decl</td>
<td>Constructor declaration</td>
<td>Constructor name</td>
</tr>
<tr>
<td>destructor</td>
<td>Destructor definition</td>
<td>Destructor name</td>
</tr>
<tr>
<td>destructor_decl</td>
<td>Destructor declaration</td>
<td>Destructor name</td>
</tr>
<tr>
<td>struct</td>
<td>Struct definition</td>
<td>Struct name</td>
</tr>
<tr>
<td>struct_decl</td>
<td>Struct declaration</td>
<td>Struct name</td>
</tr>
<tr>
<td>class</td>
<td>Class definition</td>
<td>Class name</td>
</tr>
<tr>
<td>class_decl</td>
<td>Class declaration</td>
<td>Class name</td>
</tr>
<tr>
<td>union</td>
<td>Union definition</td>
<td>Union name</td>
</tr>
<tr>
<td>union_decl</td>
<td>Union declaration</td>
<td>Union name</td>
</tr>
<tr>
<td>decl_stmt</td>
<td>Declaration statement</td>
<td>Declaration variable name and type</td>
</tr>
<tr>
<td>call</td>
<td>Function or macro call</td>
<td>Function or macro call name</td>
</tr>
</tbody>
</table>

XPath expressions, as relevant to diffPath, allow for the addressing of children in two different ways. The children can be seen as an incremental list starting from one and indexing forwards (1, 2, 3, ...), or they can be referenced from the end of the list starting at the last element and indexing backwards (last(), last() - 1, last() -2, ..). Both can be used for simple and complex diffPath expressions, however, with the predicates in a complex diffPath expression, there is a small consequence to using forward indexing. If a name
changed from ‘b’ to ‘a::b’, the predicate with the path to the name ‘b’ in ‘b’ and ‘a::b’
would be different from each other. If sequence mining or another method of analysis
was to be used on the expressions, it is likely that these would not be recognized as the
same ‘b’. Figure 11 provides an example of the same function with and without a
namespace using forward and backward indexing. As you can see with forward indexing,
the ‘b’ predicate has a different index, but in backward indexing, it has the same
predicate.

```plaintext
// Forward indexing
   src:name[1]/src:name[2]='b']

// Backward indexing
src:function[1][src:type[last()]/src:name[last()]='void'][src:name[last()]=
   'b']
src:function[1][src:type[last()]/src:name[last()]=
   'void'][src:name[last()]/src:name[last() -
   1]='a'][src:name[last()]/src:name[last()]=
   'b']
```

Figure 11. Two ways of marking up the predicate paths. Shows what a function would
look like with and without a complex name for both ways. The single step is bold, and
the remaining is left normal.
CHAPTER V
SRCDIFF IMPLEMENTATION

Implementing a tool to automatically create a srcDiff document would be more straightforward if you use a tree differencing approach, since srcML like all XML documents forms a tree structure. However, the scaleability of performing tree differences can have high complexity, and for large projects, it could take an unacceptable amount of time. For instance, under certain assumptions, a tree difference algorithm has a time complexity of $O(NE + E^2)$, where $N$ is the number of leaf nodes in the tree, and $E$ is the weighted edit difference [13, 14]. In contrast, a sequence-based difference algorithm can run with time complexity of $O(ND)$, where $N$ is the sum of the length of both sequences and $D$ is the number of differences [15, 16]. We choose to follow a similar design philosophy to that of the current srcML engineers, i.e., to quickly, efficiently, and adequately provide a solution to the domain problem, enabling the timely use on real and large projects.

The researchers at SDML and ourselves have attempted several tool implementations to automatically and efficiently create srcDiff documents from two versions of source-code files/systems using line-based differencing information such as those provided by the `diff` utility [12]. This approach was decided to be too difficult, mostly in the maintaining a valid resulting XML document. In turn, we have completely
re-engineered the approach to creating srcDiff and have developed an associated tool, srcdiff, that can efficiently produce a srcDiff document from either pairs of files or pairs of directories that individually represent two versions of a source-code file/system. The new approach uses a similar algorithm to diff, however, it applies it recursively to the child nodes in the XML document as needed.

The development of srcdiff into a multi-platform deliverable as well as some of the underlying output of XML have been ported from old versions or related tools and/or have been produced with help outside of those of the author. We will focus on what is uniquely ours for this thesis.

First, we will look at the top level general algorithm used to compute the srcDiff from two individual versions of the same file. Next, each step will be expanded in detail. Afterwards, it will be described how from working on single files, srcdiff is extended to work on two versions of a software-system, i.e., two directories each with a different version of the same system.

General Algorithm

Figure 12 lays out the top-level general algorithm used to produce a srcDiff document. The first step, lines 2 and 3, is for each of the files to be converted into srcML. The library libsrcml is used to convert both documents. The library libsrcml provides the exact same srcML translation that is available in the srcML Toolkit tool src2srcml. In other words, the library components were extracted from the source-code used directly in the tool src2srcml, and then in turn, the library is now used in the compilation of src2srcml and srcdiff.
1. def srcdiff_translate(old_file, new_file) :
2.   srcml_old = translate2srcML(old_file)
3.   srcml_new = translate2srcML(new_file)
4.   nodes_old = collect_xml_nodes(srcml_old)
5.   nodes_new = collect_xml_nodes(srcml_new)
6.   node_sets_old = collect_node_sets(nodes_old[1 : len(nodes_old) - 2])
7.   node_sets_new = collect_node_sets(nodes_new[1 : len(nodes_new) - 2])
8.   nodes = [nodes_old, nodes_new]
9.   output_differences(nodes, node_sets_old, node_sets_new)

Figure 12. General algorithm to compute the srcDiff of two files. Handles the conversion to srcML, converting into an internal format, and the start of the differencing process.

After translation, in lines 4 and 5, the XML nodes are collected using libxml2’s xmlTextReader interface [17], and converted into an internal format. A collection of nodes is formed for each file. Each collection of nodes contains the complete set of start and end tag nodes as well as all text nodes, except for the root unit tag. In our internal representation we separate white-space text from non-white space text forming multiple nodes when necessary. Each white-space character forms its own node. This is used to produce fine-grained white-space differences both by themselves and as parts of larger changes that may involve other nodes. Parenthesis, brackets, and the comma are also separated into their own nodes. This is to avoid possible problems that may occur if we have ‘(‘ in one version and ‘(’ white space ‘)’ in another. Next, in line 6 and 7, sets of nodes are created from each collection of nodes. All the node sets from a single collection are formed from the children nodes of the root unit element. Each child node set contains all the nodes from the start of the child node (opening tag) to the end of the
child node (closing tag). All white space nodes that are children of the root element and within the node sets are ignored. All white space is handled separately and will be discussed in a later subsection.

In Figure 2, the root node unit would have approximately four children, depending on the breakup of the white space nodes: cpp:include, a white-space node, function, and another white space node. In the example, the only node sets that would be collected would the cpp:include node set, which consists of the cpp:include beginning and ending tags and its entire subtree, and the function node set, which consists of the function beginning and ending tags as well as its entire subtree. None of the internal white space in the subtrees is gathered.

Furthermore, each node in a node set is not a copy of its associated node, but is the index of the node into the base node collection. When the nodes are output, any non-consecutive numbers indexed into the base array indicates that white space occurred, and it can be handled appropriately. This will be discussed in detail, when the output of the specific differences is presented. Once the top level containers are gathered, then the actual difference computing algorithm begins at line 9.

Main Differencing Algorithm

The differencing algorithm is broken down into finer detail in Figure 13. At this level of detail, the main differencing algorithm can be thought of as an application of the Shortest Edit Script (SES) algorithm on the children of a root node. This function is recursive in nature. Lines 10, 12, or 16 can result in a subsequent call to output_differences, where one of the roots children becomes a new root node, and the
entire process is applied again on the subtree. Beginning with the SES algorithm on line 2, the parts of Figure 13 will be presented in detail.

**Shortest Edit Script**

Line 2 of Figure 13 computes the Shortest Edit Script (SES) of both node sets. The SES between two sequences is the minimal number of deletion and insertion operations, edits, required to transform the first sequence into the second sequence [15, 16]. This problem is equivalent to the Longest Common Subsequence (LCS) problem, and has been reduced to a time and space complexity of $O(ND)$, where $N$ is the sum of the length of both sequences and $D$ is the minimum number of differences or edits [15, 16]. The $O(ND)$ SES algorithm created by Miller and Myers [15, 16] has been implemented for *srcdiff*. Although a linear space refinement can be applied to Miller and Myers’ SES algorithm [16], this was not used and the $O(ND)$ space algorithm was selected. Through experimentation it was conducted that only excessively large values of $D$ exhausted enough memory to stall execution, and it does not seem to be a problem for the sequences of node sets used in *srcdiff*. However, in the future, the linear space refinement version may be implemented. For more information on the basic algorithm, please see [15, 16].

**Shortest Edit Script Structure**

The edit script obtained through our implementation of Miller and Myers SES algorithm is highly based on the approach presented in [15]. It contains the sequence of edit operation necessary to convert the first node set into the second. In other words, it stores the complete set of deletions and insertions.
def output_differences(nodes, node_sets_old, node_sets_new):
    edits = shortest_edit_script(node_sets_old, node_sets_new, nodes)
    mark_moves(edits, nodes, node_sets_old, node_sets_new)
    for edit in edits:
        if is_common(edit):
            output_common(nodes, node_sets_old, node_sets_new)
        elif is_change(edit):
            if is_one2one(edit):
                if is_syntax_match(edit, nodes, node_sets_old, node_sets_new):
                    output_single(edit, node_sets_old, node_sets_new)
                elif is_nested(edit, nodes, node_sets_old, node_sets_new):
                    output_nested(edit, nodes, node_sets_old, node_sets_new)
                else:
                    output_change(edit, node_sets_old, node_sets_new)
            else:
                output_many2many(edit, node_sets_old, node_sets_new)
        else:
            output_pure_operation(edit, nodes[0][0start:oend], nodes[1][nstart:nend])
    if edits.last.end < node_sets_old.last:
        output_remaining_common(edits.last, nodes, node_sets_old, node_sets_new)

The edit operation is limited to either a delete or insert. An individual edit also provides offsets to the start and end of a difference, and it also stores enough information...
to locate where common sets of node sets occur between edit operations, as well as, where a change, a delete followed directly by an insert, occurs.

Move Detection

In line 3, the differences are iterated over to find if any moves have occurred. Here we describe a move as a section of code, i.e., a node set, that has been deleted in one place, then in another place the exact same section has been inserted. Normally, the SES algorithm would match them, however, due to a large spatial distance or some other more optimal matching, it was marked deleted and inserted. In line 3, we use the edit script and the nodes sets to look for such an instance. If one is found in the code, we mark the beginning and the ending nodes of both with a unique move identifier. This is used later as part of a the move attribute on the \textit{diff} tags. A moved operation is treated as a pure operation, discussed later, and no other processing needs to be used. This means that once we detect a move and it is marked with a move id, the move id indicates to the rest of the algorithm that it can be ignored safely. To simplify later discussion, we will disregards moves unless otherwise stated. Figure 14 shows two sections of code where a move would occur. In Figure 15, the resulting srcDiff of the sections is shown.

<table>
<thead>
<tr>
<th>// old version</th>
<th>// new version</th>
</tr>
</thead>
<tbody>
<tr>
<td>a;</td>
<td>b;</td>
</tr>
<tr>
<td>b;</td>
<td>c;</td>
</tr>
<tr>
<td>c;</td>
<td>a;</td>
</tr>
</tbody>
</table>

Figure 14. Two versions of a piece of source code where a move occurred. The statement ‘a;’ has been moved from the beginning to the end.

As can be seen in Figure 15, the code is still repeated. This is necessary for the easy extraction of a single version of source code. Also note that that the white space is
not considered part of the move. The new line is both inserted and deleted and not marked as having been moved. This is because the white space is not contained within the syntactic construct being moved. This is a slight deficiency with the current approach, and it may be corrected in the future to include white space that also moved. However, because it is not included the white space can differ before and after both the delete and insert.

Another current limitation is that our move detection is limited to an exact move. Any change to any of the text and/or srcML markup however small will cause the match to fail. In the future we may extend it to accept a move that has a tolerable amount of change.

```diff
<diff:delete move="1"><expr_stmt><expr><name>a</name></expr>;</expr_stmt></diff:delete><diff:delete>
</diff:delete><expr_stmt><expr><name>b</name></expr>;</expr_stmt><expr_stmt><expr><name>c</name></expr>;
<diff:insert move="1"><expr_stmt><expr><name>a</name></expr>;</expr_stmt></diff:insert><diff:insert>
</diff:insert>
```

Figure 15. srcDiff of the move from Figure 14.

Handling Differences

Line 4 of Figure 13 loops through every edit operation obtained from running SES. Common node sets between both `node_sets_old` and `node_sets_new` are output before each edit operation in lines 5 and 6, and in lines 19 and 20, all the remaining common nodes sets are output. In the case when there are no edits, i.e., the node sets are the same, lines 19 and 20 will output all the node sets as common.
The remaining lines in the main differencing algorithm presented in Figure 13 handles the two types of edit operations that can occur. Either there can be a change, handled in lines 7-16 or the edit operation can be standalone, i.e., surrounded by common node sets and consisting of that single edit type. This second type of operation, we call a pure operation, is handled in line 18 following the else clause in line 17.

Output of Common Node Sets

The output of common node sets function output_common runs through all the nodes from both versions and outputs the nodes in common. It is guaranteed by the SES that all the XML nodes and non-whitespace text nodes are the same, so the only possible difference is white space. A simple narrowing of the amount of change for the white space is used. That is, whenever we reach two nodes that are different, one or both of the nodes must be white space. All white space that matched before the difference have already been output. So, we collect all the remaining white space up until a non white space node, compute any portion of the suffix that is in common, if both were white space nodes, and output the middle portion as different. As a precaution, if we exhaust the nodes from one source, the remaining is output as different.

Output of Pure Operations

In the case of a pure operation, with little exception, all the node sets just need to be output surrounded by the appropriate diff tag. This output also includes all internal white space. The only exception to the output of just the node sets is the white space that is a prefix to the node set and the white space that is a suffix to the node sets.
Figure 16 shows a simple diagram of an output of a pure operation. As shown in the diagram the prefix and the suffix of the node sets that are purely deleted or inserted is matched with the beginning of the other node set in the opposite operation in a greedy fashion. However, we purposefully do not match white space that exists on the same line before the node sets start. This is to try and maintain a programmers view of the deletion or insertion. That is, all the lines were deleted or inserted including the indentation. The only exception to this is if there is no previous new line, in which case we use all the white space that matches. As for the other node set, the middle portion that is different will be output after the pure operation. The remaining white space is in common and at least one following node set is also in common to both operations.

As an example off a pure operation, Figure 17 shows the before and after of a piece of code with a single statement inserted. In Figure 18, we show the resulting section of code in the srcDiff format. The statement ‘c;’ has been completely added on a line by itself. The entire addition including the new line has been wrapped with a single \texttt{diff:insert}. Although the example had no white space before the purely added ‘c;’, it also would have been included within the tag.

Output of Change Operations

A change is distinguished from a pure operation in that at the same relative location in the documents, two different types of operations have occurred. We identify two different types of changes in \texttt{srcdiff}: one-to-one, handled in lines 8-14 of Figure 13 and a many-to-many, encapsulated in a function in line 16. A one-to-one change is a change where there is exactly one deleted node set and one inserted node set.
Since, there is only one of each we can treat this as a one-to-one mapping. A many-to-many change is a change where both a delete and an insert occur with one of the
operations having one or more node sets and the other having more than one node set. First, we will look how srcdiff handles one-to-one changes and then extend it to work on many-to-many changes.

One-to-One Change

The pseudocode for a one-one change occurs in lines 9-14 of Figure 13. There are two different things that can happen in a one-to-one change. We can have a syntax match, i.e., the root srcML nodes of node sets are the same, which is handled in lines 9 and 10 of Figure 13. The other is a syntax difference, i.e., the root srcML nodes of the node sets are different, handled in lines 11-14. In the case of a syntax match this typically means we can tighten the difference. Unless it is decided that one is a subtree of the other. Discussion of the subtree case will be left until later.

So on a syntax match, recursion into the node sets is possible. First, the necessary white space is output in a prefix and suffix based manner similar to pure operations. The only differences are that all the white space (including indentation) is output and all the white space is before the start of the node set. Next, the common root element of the node sets is output. Then, all sub-node sets of the root nodes are collected, and the main differencing algorithm is called again.

This step is equivalent to repeating lines 6-9 of Figure 12. Finally, the remaining white space not issued in the recursion is output followed by the common ending tag for the root node sets. After the output of the common ending tag, any ending white space up to and including a single new line is output (if present).
White space in the end is also handled by computing the prefix and the suffix. To better illustrate this concept, Figure 19 provides a diagram of the process. As described previously, both white-space differences, those before and after the node sets, are computed by narrowing the white space from both ends using first the prefix in common and then the suffix in common. The recursive call in the middle outputs all the differences and commons internal to the deleted and inserted node set, while that output is then wrapped with the root tag that is in common to both.

Figure 19. One-to-one change handling for a syntax match. Shows handling of white space, output of common sequences, and the recursive call.

Once again, we will always have a change, because of the SES algorithm. However, it may be the case that the smallest change is just text or an empty XML node.
(XML node of the form <node/>). In these cases we have reached a base case, the smallest individual elements for which we allow a difference to occur.

Also, when two comments are encountered in a one-to-one change, then we switch to a slightly different method than the main differencing algorithm. The only difference is a possible simplification of the handling of areas in common and in changes, and it does not contain any special handling for a one-to-one or many to many. It simply outputs all differences as a change, first deleted then inserted. Pure operations are also handled in a similar manner as in the main differencing algorithm. The only reason comments are split off from the main algorithm is to allow for the contents of comments to be handled specially. For instance, by grouping and performing the output as paragraphs, lines, words, and the like. This is not currently used to much extent, however, the ability to extend does exist. In the future, this may be implemented.

In srcdiff, although it is not shown in either Figure 13 or Figure 19, there are two other circumstances than can occur before we recurse on a syntax match. srcdiff attempts to follow a programers view point of the differences as much as possible. For instance, if an expression is changed, at a certain point (to a programmer) the expression statement is not simply an edit of a few operations, but that the entire expression statement has changed. To reflect this we added a pre-check to heuristically check if we should descend down the tree. The heuristic currently used checks and sees if the amount of similarity of the node sets is at least 75 percent. The formula for the heuristic is:

\[
similarity(set\_one, set\_two) \geq 75\% \min \_length (set\_one, set\_two)
\]
Here the similarity is running the SES algorithm on the non-white space text nodes, finding the amount of deletions and subtracting it from the length of the number of text nodes in the deleted node set. Then, if the similarity is greater than or equal to 75 percent of the smallest text length, the syntax match described previously will be performed. Currently, we limit the blocking of recursion only for expression statements, declaration statements, and expressions. As mentioned before, the second case is that one of the node sets can be a subset of another, i.e., it can be nested within the other. This is handled in srcdiff by checking before we descend into the node sets. Nesting will be discussed in detail next when we talk about lines 11 and 12 of Figure 13.

When the root node of the node sets are different some extra processing needs to occur to check to see if one of the node sets is a subset of other, i.e., it can be nested within the other node set. This condition can occur for a number of reasons. For instance, a statement/set of statements can be wrapped with an if statement, or the if statement can be removed. Figure 20 shows an example of a piece of source-code that would be nested. In Figure 21 the resulting srcDiff is shown.

In Figure 20, the simple statement ‘a;’ is wrapped with ‘if(1)’. In srcML the first statement would be marked with an expr_stmt element, as can be seen in the combined srcDiff in Figure 21, and the new version with the ‘if(1)’ would be an if element.

The problem we have is that the same piece of code exists at different levels of the XML tree, and in this case, it is a complete subtree. If we were to just treat these as two disjoint sets we would unnecessarily repeat the expr_stmt, and the resulting srcDiff would be insufficient. However, as shown in Figure 21, if we can detect this case, then
we can use the *diff:common* to mark the common *expr_stmt* correctly within the subtree, while still expressing that the if condition was inserted.

```plaintext
// old source-code       // new source-code
a;       if(1)
  a;
```

Figure 20. Source-code where the statement ‘a;’ is wrapped with an if.

```xml
<diff:insert><if><condition>(<expr>1</expr>)</condition><then>
  <diff:common><expr_stmt><expr><name>a</name></expr>;</expr_stmt></
  diff:common></then></if></diff:insert>
```

Figure 21. *srcDiff* of the source-code that has been nested in Figure 20.

Simply speaking, we could just check if one of the node sets is completely contained within the other. However, this means that if there is even the slightest change in the nested element, the nesting would fail, and we would simple output them as completely deleted and inserted node sets. In *srcdiff*, we use a somewhat greedy heuristic to determine if a statement can be nested within another. That is, when we can, we try to nest, and if we fail we default to the output of separate un-nested operations.

There are a few important things to take into consideration before an attempt to nest is tried: can a node set always be syntactically nested within the other, is it possible for a node set of one type to be placed into the node set of another, and are they a syntax match. Currently, attempting to nest something in *srcdiff* is limited to a select group of structures. For those structures we define the structures that can always be syntactically nested, and those that can possibly be nested, this includes syntax matches. Table 5 summarizes the structures that we will attempt to nest and what structures we will try to nest in them.
Table 5. List of nest elements and what is possibly to nest.

<table>
<thead>
<tr>
<th>Element</th>
<th>Nest</th>
<th>Possible Nest</th>
</tr>
</thead>
<tbody>
<tr>
<td>block</td>
<td>expr_stmt, decl_stmt, return</td>
<td>block</td>
</tr>
<tr>
<td>if</td>
<td>expr_stmt, decl_stmt, else, return</td>
<td>block, if white, for</td>
</tr>
<tr>
<td>else</td>
<td>expr_stmt, decl_stmt, return</td>
<td>block, if white, for</td>
</tr>
<tr>
<td>while</td>
<td>expr_stmt, decl_stmt, return</td>
<td>block, if white, for</td>
</tr>
<tr>
<td>for</td>
<td>expr_stmt, decl_stmt, return</td>
<td>block, if white, for</td>
</tr>
<tr>
<td>function</td>
<td>expr_stmt, decl_stmt, if, while, for</td>
<td>-</td>
</tr>
<tr>
<td>class</td>
<td>decl_stmt, function_stmt</td>
<td>-</td>
</tr>
<tr>
<td>struct</td>
<td>decl_stmt, function_stmt</td>
<td>-</td>
</tr>
<tr>
<td>union</td>
<td>decl_stmt, function_stmt</td>
<td>-</td>
</tr>
</tbody>
</table>

As can be seen from Table 5, only a few constructs are handled, and not every possible case for each. Some cases may not be found typically in source-code, such as nesting and if, while, or for in a function. These must be present outside the function and then placed inside the function. However, since srcML and srcDiff do not require source-code that can compile, this was added, because of the ease in which it could be handled.

Moving from the simplest to the complex, if the node set that is being tested to be nested is one that is syntactically safe to nest i.e., the ‘Nest’ column, then an attempt will always be made to try to nest them. If it is a syntax mismatch and falls into the ‘Possible Nest’ column, then there will be a check to see if there is an element of that type within the other node set. If there is, then an attempt to nest the node set will be tried. Finally, if they is a syntax match a heuristic based on a best match is tried. For a node set that is being tested to nest, the heuristic finds all of the syntax matches within the node set being
tested to nest into, takes the one that matches best, and if it is close enough, then an attempt will be made to try to nest the node set.

We define \textit{best match} as the internal node set that has the highest degree of similarity, where similarity is calculated the same as for testing to recurse down the tree. Our heuristic, is if we have high similarity, low difference, and the amount of differences is smaller then if we did not nest it, we will try to nest it. Once again similarity is calculated as before, and difference is calculated as just the total amount of deletions and insertions. Once we have obtained the \textit{best match} node set, the similarity of the \textit{best match} and the item to nest, the difference of the \textit{best match} and the item to nest, and the difference of both complete sets, we apply the following heuristic:

\begin{equation}
\text{match\_similarity} > 90\% \text{ max\_size and match\_difference} \leq 10\% \text{ min\_size and match\_difference} < \text{ difference}
\end{equation}

In our formula, \textit{match\_similarity} and \textit{match\_difference} are the similarity and difference between the nest node set and \textit{best match} node set. \textit{max\_size} and \textit{min\_size} are the maximum and minimum size between the nest node set and the \textit{best match} node set, and \textit{difference} is the difference between the nest node set and the complete other node set that contains the \textit{best match} node set.

Once we have decided that we will try to nest something, we can move to the output step where we will either nest the item or fail and fall back on the output of the operations individually. First, we output all the prefix white space that match, as in a pure operation. Next, computation of the \textit{best match} is performed as was did with a syntax match. If a \textit{best match} is found, then the node set will be nested within the other
node set, otherwise the differences are output separately. Figure 22. shows how nesting functions.

**Superset Node Set**

**Nested Node Set**

Figure 22. Output of a nested node sets, including the handling for white space, the superset and the recursive call.

As illustrated in Figure 22, we output nodes in the node set that is the superset up until the *best match* and the white space before. Then, we compute the suffix on the white space. The main differencing algorithm is then run on the nest node set and the *best match* node set. After the main differencing algorithm finishes, the white space prefix and suffix that occurs after the internal node sets is output. The white space of the superset is completely written to output, but on the other nested node set the smallest length of available white space between the two is used. That is, if one node set has three
white space nodes and the other has two white space nodes, then only two for the nested node set and all of the superset node set is used. Next, we output the remainder of the superset node set. On either success or fail, we end with the output of the prefix/suffix of all the white space that remains. This is done in the same manner as in the beginning and the end of a recursive syntax match.

There are a few problems with our current implementation of nesting. As mentioned before, we are limited to a small subset of elements. Secondly, when we check for the possible elements to nest, it could be the case that either could be nested into the other. We do not check for the best possible nest between the two, or have some sort of threshold of acceptance, but try to nest if it is possible. Finally, our nesting is currently limited to only a one-to-one nesting. For example, we can not nest a group of statements into an if statement. Some experiments have been tried to get the multiple node sets into a single node set to some success, however, it is not complete at this point. So, it has been left out of the main srcdiff tool for now. We plan to add it in when it becomes more stable.

Finally, if we do not have a syntax match and we do not have a nest, we output the node sets as a change. The main difference between the output of a change and the output of a pure operation is when we have both deleted and inserted nodes to output, the diff tags have the attribute type set with the value change and the operations directly follow each other as a diff:delete and then a diff:insert. This was also done when nesting fails. However, when node sets are output directly as a change, instead of when we failed
while trying to nest, the white space still needs to be handled properly. Figure 23 illustrates the output of a change.

![Diagram](image)

**Figure 23.** Output of a change, including the handling of white space.

As shown in Figure 23, the prefix white space is output first (as was described previously). Then, the suffix is precomputed to see how much of the ending output is part of each individual operation, separating the different white space from what is in common. The precomputing suffix function sets the end of the change to the point in white space where the change ends for each set of node sets, instead of the ending tag of one of the root nodes. This is done because the changes are output as one long sequence each under a single *diff* tag. Once the output of the changes is finished, then the output of
all the white space that is available after the node sets is performed. This is done by computing the prefix and suffix of all the available white space.

**Many-to-Many Change**

A many-to-many change occurs when there are both delete operations and insert operations, and one of the operations has at least one node set, while the other operation has more than one. When a many-to-many change occurs, the only information that is known is that at this point none of the node sets in the one operation is in common with any of the node sets in the other operation, otherwise SES would have matched. Figure 24 provides an example pieces of source code that would create a many-to-many change.

<table>
<thead>
<tr>
<th>// old version</th>
<th>// new version</th>
</tr>
</thead>
<tbody>
<tr>
<td>a - 1;</td>
<td>a + 1;</td>
</tr>
<tr>
<td>b + 1;</td>
<td>b - 1;</td>
</tr>
<tr>
<td>c - 2;</td>
<td>c + 2;</td>
</tr>
</tbody>
</table>

Figure 24. Example source code where a many to many occurs.

With this trivial example the only differences are that all the ‘-’ and ‘+’ operators have been switched. A node set, as would be created in Figure 24, in one operation can be almost an exact copy of a node set in one of the other operation node sets. However, since they are not an exact match they are treated as different. The ideal solution is to force these different node sets into a one-to-one change by matching node sets between the different operations. This way everything that applies to a one-to-one change can be extended to each pair in the many-to-many.

When forcing the multiple one-to-one changes from a many-to-many, the naive case would be to match each node set to the node set in the other operation that it most
closely matches. This can unfortunately creates circumstances where the correct srcML versions can not be reproduced from the srcDiff. This is because order may not be preserved. Figure 25 illustrates this problem. Line 1 of the old version matches line 3 of the new version most closely and line 3 of the old file matches line 1 of the new version.

Figure 25. An incorrect mapping of statements where order is not preserved.

These statements can not both be matched, or they will be duplicated in both places when a version is extracted. Nor can the middle match occur with another match crossing the middle line. This may be correct when one version is extracted, however, the other version would be extracted incorrectly. A match can not occur in any way that does not preserve the order of both. The next approach that could be tried is to pick the best matching for a single node set, and then output the rest as best as can be done, in a similar method, with what remains. Unfortunately, although preserving order, this naive approach creates more code duplication than necessary, and it could be considered suboptimal.

Another difficulty with many-to-many which is not shown in Figure 24 or Figure 25, is that there does not have to have the same number of deletes and inserts. This
means when node sets are paired to each other, there can be node sets that are left not paired. Another thing to consider, is it also does not make much sense to try and pair node sets that are syntactically different.

Our solution is a dynamic programming algorithm that attempts to minimize the number of unmatched node sets, were an unmatched node set is a node set that is either syntactically different or can not be paired because of the unavailability of a node set to pair. Then, the algorithm maximizes similarity to find the optimal parings between those with the same number of unmatched node sets. Figure 26 presents the outline of our dynamic programming algorithm.

From looking at the outline in Figure 26, the solution has a \( O(N^2) \) worst space and time complexity algorithm, where \( N \) is the maximum size between both node sets. The two nested loops in lines 3 and 4 compose the \( N^2 \) portion of the algorithm. The remainder of the algorithm except for line 8 runs in constant time. The similarity is the same similarity measure discussed earlier and uses SES that has a space and time complexity of \( O(MD) \), where \( M \) is sum of the length of the two node sets and \( D \) is the number of differences. So, the worst time and space complexity of our algorithm is \( O(N^2MD) \). However, it is worth mentioning that the similarity only run on syntax matches, and that substitution of another algorithm would change the overall complexity.

In line 2, the initialization of the data structure that contains the information that will be computed during the algorithm is performed. Lines 21-26 show what information is stored in the data structure as it is being updated each iteration. Line 5 checks if there is a syntax mismatch and lines 6-8 calculates the amount of similarity between the two
node sets. Since, the number of unmatched node sets and similarity are tracked separately, similarity is zero when the node sets are unmatched.

```python
def match_node_sets(nodes, node_sets_old, node_sets_new):
    differences = []
    for i in range(len(node_sets_new)):
        for j in range(len(node_sets_old)):
            unmatched = not is_syntax_match(nodes, node_sets_old[j], node_sets_new[i])
            similarity = 0
            if not unmatched:
                similarity = similarity(nodes, node_sets_old[j], node_sets_new[i])
            max_similarity = -1
            num_unmatched = MAX_INT
            matched = False
            direction = NULL
            if i == 0 and j == 0:
                max_similarity = similarity
                num_unmatched = 0
                if unmatched:
                    num_unmatched = 2
                matched = not unmatched
            if j > 0: ...
            # move on x(j) axis
            if i > 0: ...
            # move on y(i) axis
            if i > 0 and j > 0:
                ... # move on diagonal
            differences[i][j].marked = marked
            differences[i][j].num_unmatched = num_unmatched
            differences[i][j].similarity = max_similarity
            differences[i][j].direction = direction
            differences[i][j].opos = j
            differences[i][j].npos = i
    return differences
```

Figure 26. Main dynamic programming algorithm to pair node sets.
Lines 9-12 store the information and optimal values that will be computed for the
dynamic programming table. *max_similarity* stores the total similarity,*num_unmatched*
stores the total number of unmatched node sets, and *matched* is a flag that indicates those
node sets matched. *matched* is primarily used for constructing the solution from the
dynamic programming table. *direction* stores what way was moved from in the dynamic
programming table to reach this position.

There are four possible values for *direction* used in the dynamic programming
used in the dynamic programming
table: the starting direction/position, moving from left to right along the x-axis, moving
from down to up along the y-axis, or moving along the diagonal. These are handled in
lines 13-17 and lines 18-20. The first is the starting point of the dynamic programming
table, handled in lines 13-17. This *direction* is only an indicator for the start of the
dynamic programming table. The starting point initializes the beginning of the dynamic
programming table. The starting direction differs from the other directions in that there is
no previous information to compare, it is optimal. If there is an unmatched set at (0,0),
then there are two node sets that are unmatched: one for both the old node sets and the
new node sets. Otherwise, the starting amount of similarity with no nodes unmatched is
set. While the starting direction is not really a true direction, the remaining three
correspond to directions taken through the dynamic programming table in order to reach
the end and our optimal solution.

In the dynamic programming solution, the table is growing in each direction by
adding node sets along those paths. The first direction moving from left to right,
increasing j, along the x-axis, corresponds to the situation of adding one more node set to
the old node sets. The second direction moves from down to up, increasing i, along the
y-axis, and corresponds to the situation of adding one more node set to the new node sets.

Finally, the diagonal is a movement that increases i and j, and corresponds to adding both
a new node set to each the old node sets and new node sets. Figure 27, Figure 28 and
Figure 29. shows the expanded pseudo-code for all three directions.

```text
# move on x(j) axis
18-1.  if j > 0 :
    # set x axis move
18-2.   direction = “x-axis”
    # assume do not match this j
18-3.   max_similarity = differences[i][j - 1].similarity
18-4.   num_unmatched = differences[i][j - 1].num_unmatched + 1
    # check if better to match j
18-5.   int match_j_unmatched = i + j + (unmatched ? 2 : 0)
18-6   if match_j_unmatched < num_unmatched
          or (match_j_unmatched == num_unmatched
                  and similarity > max_similarity) :
18-7.       max_similarity = similarity
18-8.       num_unmatched = match_j_unmatched
18-9.       matched = !unmatched
```

Figure 27. Pseudo-code for checking in the x-axis direction.

Since the x-axis is usually the first case to test every iteration, this is assumed to
be the correct direction (default direction) in line 18-2. The next step is to calculate the
new similarity and number of unmatched node sets as if the appended node set was not
included, that is it is treated it as unmatched. Since in the beginning the node set is
treated as being unmatched, line 18-3 and 18-4, the similarity is set to the old similarity
and the number of unmatched is incremented by one from the previous value. Next, a
check to see if it is better to completely disregard all previous matches, un-matching any previous matches (line 18-5) and matching the current node set i and j is more optimal is needed. If this new configuration is found more optimal, line 18-6, the optimal information is updated in lines 18-7 through 18-9. In line 18-9 a flat is set to indicate if a match occurred so that the path can be rebuilt latter.

After checking a move along the x-axis next is a check along the y-axis. In Figure 28, it can be seen that much of the code remains very similar with lines 18-3 to 18-8 corresponding roughly to lines 19-2 to 19-7. What is new is lines, 19-8 through 19-12. The previous lines calculated what was the optimal option for the y-axis, and lines 19-8 determine which path is more optimal among the x-axis and y-axis. It is possible at this point in the code that the y-axis test has been entered, yet the default x-axis was not entered, when \( j = 0 \) and \( i > 0 \). However, since the \( \text{num}_\text{unmatched} \) is initially set to \( \text{MAX}_\text{INT} \), maximum value an integer can hold, the condition on line 19-8 should always pass. If \( \text{num}_\text{unmatched} \) ever exceeds \( \text{MAX}_\text{INT} \), then signed overflow would have occurred, this would require an exceedingly large amount of unmatched node sets on the y axis.

Finally, the diagonal path is checked. As can be seen from Figure 29, the diagonal path is a bit different. The diagonal increments both sides, so no breaking up of any already formed pairs is done, as was done in the x-axis and in the y-axis. The diagonal in affect increases similarity, and possibly increases the number of unmatched pairs.

In the case of unmatched pairs, there are two node sets that are unmatched so the value is increased by two in line 20-3. After determining the affect moving on the
diagonal would have, in lines 20-2 through 20-3, a test to see if this is more optimal than
the other paths is done in line 20-4. If it is found that it is more optimal, the optimal
information and other information is updated in lines 20-5 through 20-8.

```plaintext
# move on y(i) axis
19-1. if i > 0 :
    # assume do not match this i
19-2.   y_max_similarity = differences[i][j - 1].similarity
19-3.   y_num_unmatched = differences[i][j - 1].num_unmatched + 1

    # check if better to match i
19-4.   int match_i_unmatched = i + j + (unmatched ? 2 : 0)
           if match_i_unmatched < y_num_unmatched
               or (match_i_unmatched == y_num_unmatched
                   and similarity > y_max_similarity) :
19-5.       y_max_similarity = similarity
19-6.       y_num_unmatched = match_i_unmatched
19-7.       y_matched = !unmatched

    # check if better to use y-axis
19-8.   if y_num_unmatched < num_unmatched
               or (y_num_unmatched == num_unmatched
                   and y_max_similarity > max_similarity) :
19-9.       max_similarity = y_max_similarity
19-10.      num_unmatched = y_num_unmatched
19-11.     matched = y_matched
19-12.    direction = 'y-axis'
```

Figure 28. Pseudo code for checking in the y axis direction.

Once the dynamic table is complete the result are returned in lines 27 in Figure
26. Then, the information can be read back to build a list of node-set pairs. Reading
back the structures is simple as tracing backward the direction information, and when wa
node that is marked is reached, then it is a pair. Since, it is possible in an earlier step that
one of the node sets was paired differently before, what has already been paired needs to

48
be tracked. This can be done with just a single list of positions, and marking what has been paired. The only place where pairs are broken are on x-axis and y-axis junctions, but when this is done, the new position is marked as matching (if it is a matching set). Since, the information being built up is being read backward, the newest optimal information is encountered first. That is, new matchings will be encountered before any old matching that was broke is encountered.

```plaintext
# move on diagonal
20-1. if i > 0 and j > 0 :
    # add similarity and if unmatched
    diagonal_similarity = differences[i - 1][j - 1].similarity + similarity
    diagonal_num_unmatched = differences[i - 1][j - 1].num_unmached
    + (unmatched ? 2 : 0)

    # check if better to use diagonal
20-4   if diagonal_num_unmatched < num_unmatched
        or (diagonal_num_unmatched == num_unmatched
            and diagonal_similarity > max_similarity)
20-5   max_similarity = diagonal_similarity
20-6   num_unmatched - diagonal_num_unmatched
20-7   matched = !unmatched
20-8   direction = “diagonal”
```

Figure 29. Pseudo code for checking on the diagonal.

Now, that the pairing of the correct nodes has been obtained, the pairs can each be treated as a one-to-one change. A little attention needs to be focused on the node sets that were not paired. These are output as a normal change operation in a one-to-one when necessary, with the exception that more than one node set may be output unmatched and as a change. This is done so that as few diff tags as needed are used.
To avoid code duplication between the one-to-one code and the many-to-many, all changes are handled as a many-to-many, since it requires the same operations. The only thing that needs to performed before treating the operation as a many-to-many is the detection of nesting. This is done, and the nodes are marked. These are then filtered out with any moves before the dynamic programming algorithm runs and they are output as necessary.

One limitation of the dynamic programming solution, is that it does not currently consider nesting, although this might be easy to add if the similarity measure computed a proper measurement. However, as of now this is not the case. Nesting has mostly been left out on purpose, because of the possibility of nesting multiple node sets within a single node set. The dynamic programming solution aims at creating one-to-one relationships. A little progress has been made at adding this as a post process that runs on unmatched node sets, however, it is not yet stable, and it has been left out for now.

Handling Software Systems

All previous descriptions have focused on working on two versions of a single file. To extend this to multiple files or systems, all that needs to be done is to run the project on matching files. For this, it is assumed that file names have not changed, and a sorted list of the files from the different directories is used to form the appropriate srcDiff archive. Since, renaming files is not necessarily uncommon in source-code systems, it is proposed that in the future the new filename mappings will be allowed as separate input.

It is important to note that the turning of srcdiff from working on single files to working on full systems is not a contribution to this thesis. It was developed with help
from others besides the author. However, it is used in the evaluation, and it is also a point of future work as discussed previously.
CHAPTER VI

DIFFPATH IMPLEMENTATION

The tool \textit{diffpath} has been implemented to create \textit{diffPath} expressions from srcDiff documents. Currently, only the correct creation of simple \textit{diffPath} expressions is supported. The implementation also ignores pure white space changes. Furthermore, although, a srcDiff document is required for \textit{diffpath}, it would be possible to use source code by using the components in \textit{srcdiff}. Since, translation is contained within a single object in \textit{srcdiff}, this would only require the exposing of the translation in the form of a library similar to libsrcml [18].

SAX2

The implementation for computing the \textit{diffPath} is based off of SAX2 (Simple API for XML version 2) [19]. SAX2 is an event driven interface to handle the parsing of XML documents. The C language implementation of the SAX2 interface provided by libxml2 is used [17]. As SAX2 is an event driven process to parse an XML document as a stream, a number of hooks for callback functions are provided whenever an event is encountered. These events provide the flexibility to handle application needs. The event handlers of interest for \textit{diffPath} are \textit{startDocument}, \textit{endDocument}, \textit{startElementNs}, \textit{endElementNs}, and \textit{characters}. The implementation of these are provided as separate figures in the APPENDIX.
Simple diffPath Creation

The creation of simple diffPath expressions is now presented. *startDocument* and *endDocument* are called at the beginning and end of the XML parsing. They are quite simple. The main purpose of *startDocument* is to initialize information before parsing the XML. The information includes a list of flags and two stacks: one for the current difference operation and another for all open elements. The difference operation stack tracks what difference operation is currently open and begins in a state with all elements in common. The difference stack holds other information besides what current difference is, such as how deep in the XML tree the current open element is relative to the current difference. *endDocument* only outputs an ending newline.

*startElementNs* updates the stacks in response to the opening of an element in the XML. It updates the number of children of each type for the parent of the opened element and puts the newly opened element on the open element stack. If a *diff* element is encountered, it also updates the difference stack. Otherwise, it increments the current depth of elements relative to the current difference operation.

A difference can occur in one of two places, after a new element is opened, the depth is equal to one, or at a depth of zero, when there is a difference in text. When one of these conditions is true, then a diffPath expression is build using the current information and output. Since it is output as soon as we reach a difference, the information, such as the offset into the children, will be correct at this point. Although a simple diffPath does not normally contain any information, since a *unit* tag’s filename is easily accessed through an attribute, it is collected as part of a simple diffPath.
The \textit{endElementNs} function performs much of the same operations as \textit{startElementNs}, only reversing some of the properties. Where \textit{startElementNs} is called each time an element opens, \textit{endElementNs} is called each time an element ends. The \textit{endElementNs} remove an element from the open element stack. In response to the closing of a \textit{diff} element, a difference element is also removed from the difference stack. Otherwise, the depth is decremented.

The last remaining function is \textit{characters}, which handles text nodes. \textit{characters} can be called multiple times for the same text node, so, some special handling is needed to avoid issuing multiple \texttt{diffPaths} for the same change. This requires storing a flag indicating if the last called event handler was \textit{characters}, setting it in \textit{characters} and clearing the flag in \textit{startElementNs} and \textit{endElementNs}.

Text is considered an element, so, \textit{characters} updates the number of text element children for the currently opened element. However, special care is taken so that the same node is not added twice (checking the text flag). A difference can occur when the current depth is equal to zero. At this point the element needs to be push on the open element stack, and the difference is output using the open element stack.

Since, we do not output white space differences, the first time \textit{characters} is called it may not generate a \texttt{diffPath}, however, in a subsequent call in the same text node there may be a non-white space character, so a \texttt{diffPath} expression will need to be output. However, the text flag has already been set, indicating a previous call. To overcome this problem, the first time \textit{characters} is called a test is made to see if a \texttt{diffPath} would need to be output if it was non-white space. This sets a flag where afterwards we check for
non-white space. The complexity is necessary so that more than one diffPath is not generated and so that we do not miss the output of a diffPath when the first call contains all white space. After a successful output or a call to startElementNs or endElementNs the flag is cleared.

Next, we will briefly describe the output off a diffPath expression. Printing a diffPath simply requires iterating over the stack of open elements. Since, we stored the child information in the parent, we can retrieve this information and add it as part of the predicate. The root tag is defaulted to be the first child. For a unit we create the appropriate name tag from what was collected when it was opened. The appropriate ‘/’ is appended before each node is formed so that the last node does not have a ‘/’. After completing the output of a diffPath, a new line is output to separate the diffPath from future diffPath expressions.

Although it is not expressed directly, nor reflected in the code in the APPENDIX, the complete path is not the default output of diffpath when a srcDiff archive is supplied. In this instance, the first unit tag is dropped. This is to match what is required for srcml2src. An option does exist to output the root unit, which is similar to the srcml2src option.. Both options are called --apply-root [18].
CHAPTER VII

CASE STUDY

A case study will be performed on both srcdiff and diffpath. Both tools will be evaluated using the extensive history of the GNU GCC Compiler starting from version 2.95 up until version 4.6.3. Only the source code for gcc-core will be used for the study. GCC was chosen because it has well documented releases of both bug-fix and feature releases. The information is summarized in Table 6. Three different types of releases were identified using the change logs [9]: major feature releases, which may also contained bug fixes, releases that consisted exclusively of bug fixes and/or improvements, and releases whose classification could not be found or was unclear.

Table 6. List of versions of GCC used for the srcDiff and diffPath case study.

<table>
<thead>
<tr>
<th>Type</th>
<th>GCC Versions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feature</td>
<td>3.0, 3.1, 3.3, 3.4.0, 4.0.0, 4.1.0, 4.2.0, 4.3.0, 4.4.0, 4.5.0, 4.6.0</td>
</tr>
<tr>
<td>Bug Fix</td>
<td>2.95.1, 2.95.3, 3.0.3, 3.2.3, 3.3.1, 3.3.2, 3.4.1, 3.4.2, 3.4.3, 4.0.1, 4.3.2, 4.3.3, 4.4.1, 4.4.2, 4.4.3, 4.5.2, 4.6.1, 4.6.2, 4.6.3</td>
</tr>
<tr>
<td>Other</td>
<td>2.95, 2.95.2, 3.0.1, 3.0.2, 3.0.4, 3.1.1, 3.2, 3.2.1, 3.2.2, 3.3.3, 4.0.2, 4.1.1, 4.1.2, 4.2.1, 4.2.2, 4.2.3, 4.3.1, 4.5.1</td>
</tr>
</tbody>
</table>

For each major release, the tar.bz2 file was downloaded, and for each subsequent minor release, the patch was downloaded. The main branches of development were followed. That is, if their was a patch from the same version to a major release and one for a minor release, the major release was selected. After extracting the directories from
the major release’s tar.bz2, it was copied and the first patch was applied. *srcdiff* was run on the first directory pair and then the directories were updated with the correct patch to produce the next set of input directories. This was accomplished by discarding the old version, copying the patched version, and applying the next patch. *srcdiff* was run using a Macbook Pro 15” with a 2 GHz quad-core i7 and 16GB of RAM. The 16GB of RAM is not necessary, but was what was available on the machine. The data was stored on an external 6 TB My Book Thunderbolt Duo. For srcDiff, the following research questions will be addressed:

- RQ1: Does srcDiff correctly capture both versions?
- RQ2: Does srcDiff produce an acceptable optimality?
- RQ3: Was srcDiff able to produce the document in a reasonable amount of time?

In order to answer our first research questions. The output of srcDiff was compared to the output of *src2srcml*. Since, the srcML Toolkit is only able to extract single srcML files from srcDiff and not yet on archives, both *src2srcml* and *srcdiff* were run on the files individually for each version. Both tools were applied to files whose language extensions are by default recognized by *src2srcml* and *srcdiff*. Table 7 provides the default extension to language extensions used by *src2srcml* and *srcdiff* [10].

<table>
<thead>
<tr>
<th>Language</th>
<th>Extensions</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>.h,.c</td>
</tr>
<tr>
<td>C++</td>
<td>.H,.C,.hpp,.cpp,.hxx,.cxx,.h++,.c++, .hh,.cc,.tcc</td>
</tr>
<tr>
<td>Java</td>
<td>.java</td>
</tr>
<tr>
<td>Aspect J</td>
<td>.aj</td>
</tr>
</tbody>
</table>
Files with no change were disregarded, as was checked using the utility `cmp`. They were not computed or compared. Completely added and deleted files were also ignored, since they did not have both srcML documents to compare, and the srcDiff computation is simple. The tool `srcml2src` was used to extract the individual srcML files from the srcDiff files, and the two results were compared using `cmp`. No difference was seen between any of the srcML documents and the equivalent extracted srcDiff documents. That is, `srcdiff` perfectly preserved both srcML versions within a single srcDiff document, satisfying RQ1. Since GCC, is a large C project, more validation needs to be run on other languages to further validate srcDiff. GCC also does not provide a complete cross-section of types of software systems. So, more validation is needed to completely confirm RQ1.

In order to validate RQ2, `src2srcml` was used to create srcML archives of every version of GCC in a manner similar to how the srcDiff archives were created. The lines of code in srcDiff were compared with the lines of code in srcML for both versions. Table 8. summarizes the results. The first column is the previous version or base version, and column three is the subsequent or next version. Columns two and four contain the lines of code for both srcML versions, and column five contains the lines of code for the srcDiff of both versions. Column six takes the largest srcML version and finds how much greater the srcDiff version is in terms of lines.

Column seven is the number of lines found different by the `diff` utility (diffutils 2.8.1) [12]. Only the files with extensions that `srcdiff` would convert to srcDiff were used. That is, every file that had the extensions in Table 7 were located.
Table 8. Results of applying srcML and srcDiff to GCC.

<table>
<thead>
<tr>
<th>Previous LOC</th>
<th>Next LOC</th>
<th>srcML LOC</th>
<th>srcDiff LOC</th>
<th>Lines Added</th>
<th>Diff Lines</th>
<th>Lines Saved</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.95</td>
<td>637623</td>
<td>637789</td>
<td>637989</td>
<td>200</td>
<td>784</td>
<td>584</td>
</tr>
<tr>
<td>2.95.1</td>
<td>637789</td>
<td>638022</td>
<td>638171</td>
<td>149</td>
<td>659</td>
<td>510</td>
</tr>
<tr>
<td>2.95.2</td>
<td>638022</td>
<td>638856</td>
<td>639139</td>
<td>283</td>
<td>1810</td>
<td>1527</td>
</tr>
<tr>
<td>2.95.3</td>
<td>638856</td>
<td>3.0</td>
<td>709581</td>
<td>199694</td>
<td>485557</td>
<td>285863</td>
</tr>
<tr>
<td>3.0</td>
<td>709583</td>
<td>3.0.1</td>
<td>716480</td>
<td>348</td>
<td>7905</td>
<td>7557</td>
</tr>
<tr>
<td>3.0.1</td>
<td>716480</td>
<td>3.0.2</td>
<td>716908</td>
<td>289</td>
<td>1176</td>
<td>887</td>
</tr>
<tr>
<td>3.0.2</td>
<td>716908</td>
<td>3.0.3</td>
<td>717517</td>
<td>866</td>
<td>1933</td>
<td>1067</td>
</tr>
<tr>
<td>3.0.3</td>
<td>717517</td>
<td>3.0.4</td>
<td>723311</td>
<td>1294</td>
<td>9112</td>
<td>7818</td>
</tr>
<tr>
<td>3.1</td>
<td>795748</td>
<td>3.1.1</td>
<td>797582</td>
<td>1820</td>
<td>4862</td>
<td>3042</td>
</tr>
<tr>
<td>3.1.1</td>
<td>797582</td>
<td>3.1.2</td>
<td>797617</td>
<td>9</td>
<td>81</td>
<td>72</td>
</tr>
<tr>
<td>3.2</td>
<td>797617</td>
<td>3.2.1</td>
<td>798462</td>
<td>2438</td>
<td>7225</td>
<td>4787</td>
</tr>
<tr>
<td>3.2.1</td>
<td>798462</td>
<td>3.2.2</td>
<td>799473</td>
<td>2170</td>
<td>6797</td>
<td>4627</td>
</tr>
<tr>
<td>3.2.2</td>
<td>799473</td>
<td>3.2.3</td>
<td>800473</td>
<td>2976</td>
<td>9772</td>
<td>6796</td>
</tr>
<tr>
<td>3.2.3</td>
<td>800749</td>
<td>3.3</td>
<td>840816</td>
<td>111806</td>
<td>257163</td>
<td>145357</td>
</tr>
<tr>
<td>3.3</td>
<td>840816</td>
<td>3.3.1</td>
<td>841797</td>
<td>575</td>
<td>2401</td>
<td>1826</td>
</tr>
<tr>
<td>3.3.1</td>
<td>841797</td>
<td>3.3.2</td>
<td>842195</td>
<td>781</td>
<td>2148</td>
<td>1367</td>
</tr>
<tr>
<td>3.3.2</td>
<td>842195</td>
<td>3.3.3</td>
<td>843983</td>
<td>2246</td>
<td>6836</td>
<td>4590</td>
</tr>
<tr>
<td>3.3.3</td>
<td>843983</td>
<td>3.4</td>
<td>847334</td>
<td>218544</td>
<td>367460</td>
<td>148916</td>
</tr>
<tr>
<td>3.4</td>
<td>847334</td>
<td>3.4.1</td>
<td>848278</td>
<td>2950</td>
<td>2131</td>
<td>367460</td>
</tr>
<tr>
<td>3.4.1</td>
<td>848278</td>
<td>3.4.2</td>
<td>852031</td>
<td>14471</td>
<td>13793</td>
<td>52131</td>
</tr>
<tr>
<td>3.4.2</td>
<td>852031</td>
<td>3.4.3</td>
<td>852595</td>
<td>242</td>
<td>1460</td>
<td>1218</td>
</tr>
<tr>
<td>3.4.3</td>
<td>852595</td>
<td>4.0</td>
<td>944387</td>
<td>164427</td>
<td>450820</td>
<td>286393</td>
</tr>
<tr>
<td>4.0</td>
<td>944387</td>
<td>4.0.1</td>
<td>951032</td>
<td>1152</td>
<td>3563</td>
<td>2411</td>
</tr>
<tr>
<td>4.0.1</td>
<td>951032</td>
<td>4.0.2</td>
<td>951939</td>
<td>261200</td>
<td>147452</td>
<td>147452</td>
</tr>
<tr>
<td>4.0.2</td>
<td>951939</td>
<td>4.1</td>
<td>988204</td>
<td>2742</td>
<td>1582</td>
<td>2742</td>
</tr>
<tr>
<td>4.1</td>
<td>988204</td>
<td>4.1.1</td>
<td>998712</td>
<td>2090</td>
<td>4351</td>
<td>2261</td>
</tr>
<tr>
<td>4.1.1</td>
<td>998712</td>
<td>4.1.2</td>
<td>990021</td>
<td>47909</td>
<td>182896</td>
<td>134987</td>
</tr>
<tr>
<td>4.1.2</td>
<td>990021</td>
<td>4.2</td>
<td>1065121</td>
<td>4327</td>
<td>2681</td>
<td>134987</td>
</tr>
<tr>
<td>4.2</td>
<td>1065121</td>
<td>4.2.1</td>
<td>1065304</td>
<td>1646</td>
<td>4327</td>
<td>2681</td>
</tr>
<tr>
<td>4.2.1</td>
<td>1065304</td>
<td>4.2.2</td>
<td>1064936</td>
<td>8656</td>
<td>7287</td>
<td>134987</td>
</tr>
<tr>
<td>4.2.2</td>
<td>1064936</td>
<td>4.2.3</td>
<td>1068274</td>
<td>8970</td>
<td>5584</td>
<td>7287</td>
</tr>
<tr>
<td>4.2.3</td>
<td>1068274</td>
<td>4.3</td>
<td>1450832</td>
<td>603988</td>
<td>513533</td>
<td>513533</td>
</tr>
<tr>
<td>4.3</td>
<td>1450832</td>
<td>4.3.1</td>
<td>1452116</td>
<td>3751</td>
<td>2427</td>
<td>2427</td>
</tr>
<tr>
<td>4.3.1</td>
<td>1452116</td>
<td>4.3.2</td>
<td>1453175</td>
<td>4396</td>
<td>3994</td>
<td>2427</td>
</tr>
<tr>
<td>4.3.2</td>
<td>1453175</td>
<td>4.3.3</td>
<td>1454739</td>
<td>696</td>
<td>3534</td>
<td>2838</td>
</tr>
<tr>
<td>4.3.3</td>
<td>1454739</td>
<td>4.4</td>
<td>1550990</td>
<td>71550</td>
<td>206750</td>
<td>206750</td>
</tr>
<tr>
<td>4.4</td>
<td>1550990</td>
<td>4.4.1</td>
<td>1551584</td>
<td>432</td>
<td>2002</td>
<td>1570</td>
</tr>
</tbody>
</table>
When the file existed in both versions, `diff` was run on them and only the number of lines beginning with a “<” or “>” were counted. If the files did not exist in one of the versions those lines were added to the lines of `diff`. Column eight shows the number of lines saved using `srcdiff`, that is, `diff` lines minus `srcdiff` lines added. As seen in the table, `srcdiff` produced a significantly fewer number of additional lines that `diff` said changed. Running `diff` with out white space as significant may significantly reduce the lines it produces. However, information would be lost, while `srcdiff` would still produces the white space differences. `srcdiff` respects syntax boundaries, not line boundaries, so multiple changes appear on the same line. If a change or multiple changes appeared on a line, then `diff` would report those two lines, however, `srcdiff` should mark up all changes and should not reproduce the common line. We conclude that `srcdiff` produces an acceptable level of optimality. Of note, both tools seem to produce a much larger delta in feature releases than bug-fix releases. This most likely indicates are larger maintenance effort.

For RQ3, we use the bug and feature releases of GCC. We report the time (using the real time reported by the `time` utility) srcDiff takes to convert two versions source code in terms of the number of changes (non-white space), as indicated by the number of
diffPaths. The number of non-white space changes is used, because the underlying Shortest Edit Script algorithm complexity is dependent on the number of changes and does not use white space [15, 16]. Figure 30 shows the scatter plot of number of changes and time. As can be seen in Figure 30, the time increases with some degree of dependence with the number of changes. The highest number of changes occurs in version 3.3.3 to 3.4.0, which shows a significantly smaller amount of time, and appears to be an outlier. Although this may be explained by the smaller number of lines output by srcdiff, this does not seem to be the case, since the second highest number of changes occurs between 2.95.3-3.0 which has fewer lines of srcDiff.

![Figure 30. Scatter plot of the time to srcDiff plotted against the number of non white space changes.](image)

Ignoring the outlier, the time it takes to execute seems to be approximately linear. From this we are able to validate that RQ3, that srcdiff does appear to produce srcDiff documents within an acceptable amount of time.

For diffpath we want to address the following research questions:
• RQ1: Does diffpath correctly address changes in the srcDiff document?

• RQ2: Is diffpath scaleable?

• RQ3: Does the format provide an useful/efficient way of gathering information?

For RQ1, we wrote a validation tool that takes a srcDiff document as input. From this input, it runs diffpath, and then executes each diffPath expression on the srcDiff document. Only the GCC releases that correspond to bug fixes and feature releases were used. We also executed the following XPath expression:

```
//diff:*[not(self::diff:common)]/node()
```

Which collects all the changes in the srcDiff document. Discarding those that are only white space, we then compare the results using a direct string comparison. Due to efficiency reasons and the time it takes to run individual XPath expressions on a large XML file, we had to extract the files that had changes to form separate srcDiff documents. diffpath and the XPath expression were run on each changed file individually. The extraction of the individual files and the execution of diffPath and XPath expressions was done using the tool srcml2src. We were able to capture the exact information. That is every diffPath corresponded to exactly a single change, and contained all of that change.

To answer RQ2, we use the size of the input srcDiff document as the independent variable, and plot the time (using the real time reported by the time utility) as a scatter plot. Figure 31 shows the scatter plot of the time it takes to create diffPath expressions from srcDiff documents. As can be seen, the plot forms a fairly regular line, meaning we appear to have a linear growth rate. Since, the range of srcDiff lines of code is from 62
around 600,000 to around 1.7 million, we conclude that simple diffPath expressions are highly scaleable.

![Scatter plot of time to create diffPaths from srcDiff using srcDiff lines of code as the independent variable.](image)

Figure 31. Scatter plot of time to create diffPaths from srcDiff using srcDiff lines of code as the independent variable.

So far we have shown that srcdiff can create an accurate delta that completely preserves both srcML documents. We have also shown that srcdiff can do this efficiently, and with an acceptable level of optimality. Furthermore, we have shown that diffpath can quickly and accurately capture the location of changes in a srcDiff document. What is left to be shown is RQ3, that is, is the diffPath format useful. Completely validating a case study on the analysis of the structures in the diffPath expressions requires also validating that the original srcML is correctly marking up the source code. Such validation is beyond the scope of this thesis. Note: that the correct markup of srcML was not required for either the time to produce a correct delta or the time to produce correct
diffPath expressions. Instead we will validate RQ3 by describing the information that can be easily extracted from the diffPath expressions.

The number of diffPaths provides a direct relation to the number of changes performed. We are also able to parse the diffPaths, and obtain the statistics on the overall length of the diffPath expressions such as max and average length. The length of the diffPath expression may correlate to the granularity of the changes, and it could also indicate the overall complexity of the changes. The longer the diffPath the more syntactic elements affected and the more complicated the change. Also, the longer the diffPath the finer grained the difference, or more specific the address. A shorter diffPath means the changes affect less structures and that it is higher up the Abstract Syntax Tree (AST). Higher up changes in the AST may encompass a larger change.

Information from the diffPaths can be used to identify the number of all the structures affected with deletes and inserts, as well as, the number of each structure deleted and inserted. Using this information, a distribution of the percentage of each element type inserted and deleted, as well as, an average of the number of times that element appears in a given change can be ascertained. This information can possibly be used to ascertain what the common change activities were relative to a certain maintenance task (such as bug-fix releases and feature releases) or simply can be used to summarize the overall change to a system. The percentage of each element added and deleted can also be separated into specific structures of interest. It can also be used to indicate the impact of a change. That is, the addition/deletion of many functions should have a greater impact that changes localized to within expressions.

64
Finally, the overall percentage of deletions and insertions can be obtained from the diffPath expressions. This can indicate growth of a system. The insertions can also be broken down between the different elements to show what elements were most commonly inserted or deleted.

A description of the information that can be obtained from a diffPath has been described. We believe that information provided by diffPath expressions are useful in describing changes to large systems. This shows that diffPath provides a useful function, answering RQ3.

Threats to Validity

Our approach is based entirely on the accuracy of srcML in marking up source-code. Although, the srcML toolkit has a long history and is in constant improvement, markup errors occur. For instance, in some versions of GCC, K and R C was used. srcML does not handle K and R C function parameter definitions correctly. srcML attempt at producing the programmers view of source code, also means it generally runs on code without running the preprocessor. This means languages such as C and C++ are not context-free. However, since we did not perform analysis on the structures, but rather described them, this threat has been minimized.

We only validated our tools on a single language, C. In order to properly assess the usefulness and accuracy of both tools, they should be run on additional systems with different languages. GCC also represents only one type of software system. A larger set of systems from varying application is needed to properly access and validate our claims.
However, in both circumstances, srcdiff’s knowledge is related to srcML constructs and not directly related to a particular language or system.

Our work is also limited to complete releases. These releases are formed over a longer period of time, and are only snapshots of the systems before and after a group of changes occurs. It is possible that multiple changes occurred in the same location, however, we missed them because the release period was too large. Along these lines, it has been observed that a large percentage of changes to correct bugs, will themselves create bugs that need to be corrected latter [20]. Investigating versions more frequently will allow us to more accurately describe the changes to a system.

A case study of the structural information of diffPath expressions has not been accomplished. To completely indicate if diffPath expressions are useful aids to understanding changes to a system. A complete case study is needed. However, this is left as future work.

Classification of the gcc-core releases was accomplished and rechecked by hand. It is entirely possible that something was overlooked while manually classifying the releases. However, the only conclusion we drew is that feature releases produced a larger delta.
CHAPTER VIII

RELATED WORK

The srcML format and toolkit was selected for the author’s familiarity as well as it being the foundation for srcDiff. srcML also completely preserves the original text [7, 8, 10], which is a necessity for proper difference information. However, other XML based approached to the mark up of source-code exist such as JavaML [21, 22], which is an XML format for Java source code.

Alternative approaches to using srcML/srcDiff or XML as a basis for analysis includes Baxter’s Design Maintenance System (DMS) [23], and more specifically their syntax aware Smart Differencer [24]. Both provide capabilities comparable or sometimes exceeding that of srcML/srcDiff and diffPath and are able to handle a larger variety of languages. However, DMS and the Smart Differencer take a heavy-weight parser based approach. They require well-structured code (for at least C and C++) for their approach to work. They also ignore white space and comments. srcML/srcDiff and diffPath does not require either complete programs or compilable code. Even in differencing the srcDiff format seeks to preserve the programmers view of the source code and preserve white space and comments and the corresponding changes. However, white space, as was in diffPath, can be ignored as can comments. In spite of these distinctions, we do not see these as alternative approaches, but more as complements of each other. Were the DMS
approach might be more equipped to handle more advanced analysis, such as pointer aliasing, and our approach is capable of handling poorly structured code.

The following application to source-code differencing produce a delta, however, may not be useful for change analysis. The GNU *diff* tool [12], which performs line based differencing based off of Miller and Myers Shortest Edit Script algorithm with a linear space refinement [15, 16] is perhaps the most frequently used tool for differencing of source code. The tool supports some features that enable change conceptualization, such as side-by-side comparisons, surrounding lines for context, and what function the change occurs (C functions). It also has an option to ignore white space changes. However, the line difference tool does not have true support for the syntax information provided by srcML/srcDiff. However, based only on text line differencing, *diff* is language neutral and extremely fast. Ldiff is an extension of the *diff* tool that that enhances *diff* to identify fragment changes and moves [25].

JDiff is a differencing tool for Java that understands object oriented features [26]. The outputs are pairs of nodes sets and the difference operation. JDiff also outputs the matching classes, interfaces and methods. Semantic Diff tries to perform a meaningful difference operation using the changes of inputs and outputs (a semantic difference) that is unrelated to the underlying syntax [27]. UMLDiff performs differencing at the design level, taking class models as input and outputs a change tree [28].

Related to differencing on XML documents, a calculus for XML edit operations is presented in [29]. X-Diff performs tree differencing on XML documents treating them as unordered models [30]. The treatment as unordered only respecting hierarchy is invalid
for source code and srcML. In X-Diff, they contrast this with XML TreeDiff and XyDiff which they describe as ordered model tree differencing [30]. These approaches view the problem as trees, and the XML approaches have no specific knowledge of what is marked up. srcDiff on the other hand is not based on tree differencing, and instead applies the sequence-based Shortest Edit Script algorithm of Miller and Myers recursively. srcDiff is also developed specifically to handle source-code with knowledge build in about source-code obtained from the underlying srcML representation. Since, we have information from the Abstract Syntax Tree (AST), we are able to make general conclusions about what types of elements can be placed where, without validating to a DTD or Schema.

In [20], Purushothaman and Perry correlated the size of a change with the likelihood of it introducing an error. They provide evidence that the greater the size of a change, the more likely it was to have a defect.

Our work is also related to the work presented in [31]. Alali et al. characterize a typical commit in terms of number of files, number of lines, and number of consecutive lines (hunks). They also combine the size information with the most frequent vocabulary used in the commit messages of each type of commit. Our work could be applied as an extension to this, adding structural information to the typical types of commits.

Dex is a syntax and semantic source code difference tool that uses graph differencing on Abstract Semantic Graphs (ASG) created from patches (difference between two systems) to systems [32]. A tree differencing approach termed change distilling is presented by Fluri et al. that does fine-grained change extraction [33]. Their
approach uses a combination of node matching and a minimum edit script, and it enables them to identify the changes according to their taxonomy.

The srcML format and srcDiff format and its proposed use for source-code difference analysis was presented in [5, 6, 34]. Previous implementation of srcDiff have been used to support automatic/semiautomatic co-evolution of traceability links in [35], as well as, the automatic detection of changes that have an affect on code-to-design traceability in [36, 37]. The previous tools have also been used to break down large changes or commits into smaller changes in [38]. This work also incorporated XPath expression to address into the srcDiff document. Lastly, it has been used as part of a change impact analysis that used Information Retrieval (IR) techniques and software repository mining techniques [39].

To the best of our knowledge no work on the addressing of differences of source-code have been done. This makes diffPath a very unique and potentially new area of study. We believe that similar addresses for XML documents like diffPath could possibly be applied to other types of XML documents.
CHAPTER IX
CONCLUSION AND FUTURE WORK

This thesis contributes a new implementation of a tool to produce srcDiff documents. We discuss in detail the implementation of \texttt{srcdiff}, and we prove that \texttt{srcdiff} can produce srcDiff documents from two versions of source code and that we are able to reproduce both versions from the single srcDiff document. We also showed that \texttt{srcdiff} can produce the srcDiff document from source code in an acceptable amount of time. \texttt{srcdiff} also produces an acceptable optimality, however, there is still room for improvement.

Leveraging the srcDiff documents, we present a new representation for changes in source code. The format \texttt{diffPath} is explained, and a implementation of a tool to create \texttt{diffPath} expressions, \texttt{diffpath}, is presented. We show that the tool accurately produces expressions that uniquely address a change. We also show that the tool is scaleable. Finally, we also describe how the format may be useful in the analysis of change to support the software evolution process.

\texttt{srcdiff} is a tool that can and should undergo continual improvement building in more information about the underlying srcML. The ability to nest constructs within another, especially multiple constructs within another, is a large area of expansion. This would largely help the optimality of \texttt{srcdiff}. 71
Although we implemented and validated *diffpath* for simple diffPath expressions, we did not do so for complex diffPath expressions. There is more work to be done on extending diffPath for use with complex expressions and more work on validating its output.

Our work focused on the differences between releases of source code. Applying the same procedure to individual commits, could produce finer grained changes and more accurate diffPath expressions while possibly improving results. Since, *srcdiff* is largely based off of the Shortest Edit Script problem, which itself is related to the Longest Common Subsequence problem, the smaller the changes, the more efficient *srcdiff* should be at the output of the srcDiff.

Our research was applied to an open-source system. Another study applied to a closed system in industry should also be the focus of future work, to further validate *srcdiff* and *diffpath* and to show the applicability on such systems.

diffPath expressions require a complete case study to validate their usefulness. This should be the immediate focus of future work.
BIBLIOGRAPHY


APPENDIX

DIFFPATH EVENT HANDLER IMPLEMENTATIONS

1. def startDocument(ctx) :
2.     path_generator = ctx._private
3.     path_generator.diff_stack.push(diff("common"))
4.     path_generator.collect_text = false

5. def endDocument(ctx) :
6.     print_newline()

Figure 32. startDocument and endDocument handlers used in diffpath.
1. def startElement(ctx, localname, prefix, URI, attributes) :
2.     path_generator = ctx._private
3.     path_generator.collect_text = false
4.     path_generator.diff_stack.back().output_text = false
5.     if URI == “http://www.sdml.info/srcDiff” :
6.         path_generator.diff_stack.push(diff(localname))
7.     element = element()
8.     element.set_info(prefix, localname, URI)
9.     if localname == “unit” : element.signature = form_filename(attributes)
10.    if path_generator.elements.size() > 0 :
11.        add_child(path_generator.elements.size().back(), element);
12.    path_generator.elements.push(element)
13.    if URI != “http://www.sdml.info/srcDiff” :
14.        path_generator.diff_stack.back().level += 1
15.    if path_generator.diff_stack.back().operation != “common”
16.        and path_generator.diff_back().level == 1 :
17.        output_path(path_generator)

Figure 33. startElementNs event handler used in diffpath.
```python
1. def endElement(ctx, localname, prefix, URI):
2.     path_generator = ctx._private
3.     path_generator.collect_text = false
4.     path_generator.diff_stack.back().output_text = false
5.     if URI == “http://www.sdml.info/srcDiff”:
6.         path_generator.diff_stack.pop()
7.     path_generator.elements.pop()
8.     if URI != “http://www.sdml.info/srcDiff”:
9.         path_generator.diff_stack.back().level -= 1
```

Figure 34. endElementNs event handler used in diffpath.
def characters(ctx, localname, text):
    path_generator = ctx._private

    element = element()
    element.name = "text()"

    if not path_generator.collect_text:
        add_child(path_generator, element)

    if path_generator.diff_stack.back().operation != "common"
        and text != "" and not path_generator.collect_text
        and path_generator.diff_stack.back().level == 0:
        path_generator.diff_stack.back().text_num
            = num_child(path_generator, element)
        path_generator.diff_stack.back().output_text = true

    if path_generator.diff_stack.back().operation != "common"
        and text != ""
        and path_generator.diff_stack.back().output_text
        and (path_generator.diff_stack.back().text_num
            == num_children(path_generator, element)
            and has_non_whitespace(text)
            and path_generator.diff_stack.back().level == 0):
        path_generator.diff_stack.back().output_text = false
    path_generator.elements.push(element)
    output_path(path_generator)
    path_generator.elements.pop()

    path_generator.collect_text = true

Figure 35. characters event handler used in diffpath.
1. def output_path(path_generator) :
2.     previous_element
3.     for element in path_generator.elements :
4.         print_character("/")
5.         child = 0
6.         if !is_root(element)
7.             child = previous_element.get_child_count(element)
8.         print_element(element, child, path_generator.diff_stack.back().operation)
9.     previous_element = element
10.    print_newline()

Figure 36. Pseudo code of how to output a diffPath expression in diffpath.

1. def print_element(element, child, operation) :
2.     name = element.prefix
3.     if name == "" :
4.         name = "src:"
5.     else :
6.         name += ":"
7.     name += element.name
8.     print name
9.     if child <= 0 :
10.        child = 1
11.        print "[" + str(child) + "]"
12.     if element.name == "unit" :
13.         print "[@filename=" + combine_filename(element) + "]"

Figure 37. Pseudo code of how to print an element in diffpath.