Factoring Differences for Iterative Change Management

Michael L. Collard, Huzefa Kagdi, and Jonathan I. Maletic
Department of Computer Science
Kent State University
Kent Ohio 44242
{collard, hkagdi, jmaletic}@cs.kent.edu

Abstract

An approach for factoring source-code differences is presented. A single large difference between two versions of a program is decomposed into factors (i.e., smaller changes). The application of all the factors is equivalent to the application of the single large difference. The factors are obtained by user-defined criteria. They include changes that are limited to a specific syntactic construct or ones that occur throughout a file. The factors can be applied individually or in combination to generate intermediate forms of the large change. This directly supports iterative software change management by decomposing large changes into smaller factors.

The approach uses srcDiff, an XML representation of multiple versions of a source-code file and their differences. XML transformations are used to factor a change according to an XPath expression. The approach is applied to changes made to a large open-source system. The results indicate the approach is flexible and efficient, and can be integrated with common differencing tools.

1. Introduction

Many large-scale open source development projects require that changes to the system be committed in small increments [17]. This limits the impact of the change and simplifies testing and integration. Additionally, small changes can be well understood by the many developers on the project, thus giving the entire team more confidence that any adverse side effects will be identified early on.

However, in practice, developers find committing these small incremental changes problematic. That is, development of a new feature typically will require many changes to a system, some of which will be revoked or done differently as development progresses. As implementation moves forward, flaws in the initial design are uncovered and previously implemented code is reworked or refactored. For example, it is not uncommon to add a new attribute to a class to support a new feature, but then later change your mind and rename or move this attribute to another class.

In practice, developers will implement a large part (or all) of a new feature and then go back and try (manually) to break apart the additions so they can be committed in a more incremental fashion. This avoids the "commit, change your mind, and recommit" problem. Development is rarely a nice clean path to the final solution but rather a search for the solution. However, we do want our commits to represent as clear a path to the solution as possible. Due to the nature of open source development this clear-path becomes even more important. Without ways of logically breaking up changes, the change moderator must either accept or reject the entire change. There is no easy way to iteratively provide feedback on a change, i.e., part of the change is accepted and part is given back. In the worst case, the change is too pervasive and too difficult to break apart and may never be integrated into the system [17].

Current differencing approaches (*diff* and *patch*) are not sufficient and only easily allow a large change to be broken apart at the file level. While a change may also be broken apart at a line level, line-level changes do not match with the semantics or syntax of the changes.

Ideally we would have a means to take a large change and factor it into a set of prime changes. These prime changes are syntactically meaningful and as small as possible. They can later be composed to form larger factors that represent related (e.g., logically or semantically) incremental changes. The developer would then select the appropriate sets of factors to define an understandable and clear path of commits to implement the new feature.

The approach taken here extends our previous work on srcDiff (multi-version difference formats) and meta-differencing (querying source-code differences) to extract factors of a change. XPath expressions, based on syntactical and documentary structure of the source, are applied to the srcDiff format and used to extract prime factors. These factors can be used to form a path for the iterative application of changes that more closely matches the requirements of change integration, but may differ from the path of changes used to create a large change.

The paper is organized as follows. The next section formalizes the problem with the help of an example. In section 3 srcDiff and meta-differencing approach, and the toolset used are outlined. Evaluation of the approach is presented in section 4 using the example from section 2. Discussion on open issues is presented in section 5. This is followed by related work in section 6 and finally conclusions and future work in section 7.

```
     API changes:
[A1] * Attribute doesn't need to track which ranges are using it... that's
       just overkill
[A2] * Start working on exposing the dynamic highlighting effects... which
       aren't written yet (sorry for getting your hopes up)
[A3] * Mouse and cursor enter/exit notification for ranges.  Needs polishing
       on how to actually request that they be delivered (SmartInterface
       needs a bit of refactoring)
[A4] * made SmartRange::deepestRangeContaining() provide a method for
       returning which SmartRanges were iterated to get to the answer
[A5] * changed attachAction to associateAction for consistency (and there's
       no ownership, it's just a relationship)
[A6] * intersect and encompass functions for Range
[A7] * a few extra handy parent-related functions for Range (it's amazing how
       much you find out what is missing from your interface when you start
       using it yourself)

     Kate part changes:
[K1] * moved the RenderRanges stuff out into its own file
[K2] * hook up the mouse movement logic again
[K3] * clean up KateSmartRange a bit
[K4] * track deleted ranges better - less dangling pointers (probably some
     * still remaining)
[K5] * started porting the word wrap indicators - seems to not be working yet
[K6] * attempt to fix mouse from changing positions

     Needed to commit this as it was getting... a bit big :)
```

Figure 1. A commit message from KDE revision 473657, performed on 2005-10-24, and impacting text editor interfaces and Kate in kdelibs. In addition to the original text, labels appear in bold (e.g., [A3]) corresponding to individual factors.

2. Defining the Problem

Let us now look at a specific, nontrivial, example that is used as a running example in the paper. Figure 1 contains an actual commit message for a large change from *KDE* (K Desktop Environment). *KDE* is a successful open-source system with more than 4000 KLOC and 800 contributors. A commit message is the text that a developer enters when a changeset is committed to the version control system, in this case Subversion. From the text we can derive that this change contains an API change along with changes to a client application (i.e., the editor *Kate*). This commit message is a list of issues corresponding to specific functionality in the API and how this functionality was incorporated into Kate.

We manually annotated (prefixed with bold labels) the commit message so that each of the individual changes can be referenced in the paper. Each of the individual changes could have been committed separately. As the author of the commit message clearly indicates, there are potentially more individual changes (see [A2] in Figure 1 and the "closing remark" of the commit message). These individual changes appear to be separate atomic changes irrespective of whether a single large commit or multiple small commits were made.

We cannot precisely infer the original order in which changes were performed. However, the order in which these changes are accepted (i.e., updating a working copy) depends on the task and the size of each change. One may want to apply all of the API changes first (i.e., labeled A1 – A7), then the client changes (i.e., labeled K1 – K6), or partition the changes based on which parts of the source code are involved. Therefore the commit order (and granularity) may not be the same as the acceptance order.

Using this example we now state more formal definitions. A system S consists of a set of files, $\{f1,f2,\ldots,fn\}$. The *difference* Δ_{i+1} between two versions of a system in a repository, S_i and S_{i+1}, is defined as the changes to the version S_i needed to produce the version S_{i+1}, i.e., $\Delta_{i+1}(S_i)=S_{i+1}$. In our running example, S_i corresponds to the system before the application of the change, and S_{i+1} corresponds to the system after the application of the change. The application of the difference between the two versions of the system (the change described in the commit message of Figure 1) corresponds to Δ_{i+1}. The difference Δ_{i+1} produces another set of files that contains modified files from, added files, to, and/or removed files from the set $\{f_i1,f_i2,\ldots,f_in\}$.

The commits for a system form a finite *commit sequence* $\Delta_n(\Delta_{n-1\ldots}(\Delta_{i+1}(S_i)))$. The system S_{i+j} corresponds to the version produced after the composition of the first j differences, i.e., $\Delta_j(\Delta_{j-1\ldots}(\Delta_{i+1}(S_i)))$. A commit sequence of length n corresponds to $n+1$ different versions of the system.

Two commit sequences, $\Delta_j(\Delta_{j-1\ldots}(\Delta_{i+1}(S_i)))$ and $\Delta_k(\Delta_{k-1\ldots}(\Delta_{i+1}(S_i)))$ are considered equivalent if they produce the same system, i.e., the same set of files. The two sequences may contain different elements, or the same elements in a different order. The existence of equivalent reorderings depend on the location of changes (e.g., disjoint sets of files for each commit) and the mechanism of the patch mechanism (e.g., line-based path mechanisms require proper ordering due to line number changes).

The notation so far applies to large changes to a system, i.e., the changes that map to a commit. However, each large change may also consist of individual parts, (i.e., factors). In our example, these parts could be the individual parts of the commit, e.g., A1. A *factor,* δ_{i+1} between two versions of a system, S_i and S_{i+1}, is defined as the changes to the version S_i needed to produce the version S_{i+1}, i.e., $\delta_{i+1}(S_i) = S_{i+1}$. A *factor sequence* can be defined similarly to a commit sequence, $\delta_n(\delta_{n-1}\ldots(\delta_{i+1}(S_i))\ldots)$. The system S_{i+j} corresponds to the version produced after the composition of the first *j* factors, i.e., $\delta_j(\delta_{j-1}\ldots(\delta_{i+1}(S_i))\ldots)$. Each commit difference corresponds to a factor sequence, i.e., $\Delta_{i+1}(S_i)=\delta_n(\delta_{n-1}\ldots(\delta_{i+1}(S_i))\ldots)$

A *prime factor* is a factor whose further division is not necessary for a considered task and/or view of a change. The issue of whether a factor is sufficiently prime depends on the granularity and type of change. At the textual level, a single character factor is the most prime. However, for purposes of producing prime factors, we feel the indivisibility should depend on syntactic features of the source code being modified. Prime factors may also depend on the task of application of the differences.

The next section presents our approach to extracting and manipulating the factors of a change, i.e., extracting a factor sequence $\delta_n(\delta_{n-1}\ldots(\delta_{i+1}(S_i))$ from a commit change $\Delta_{i+1}(S_i)$.

3. The Approach

Our approach is to transform the textual differences into syntactic differences. The factors are then formed by querying and manipulation of the syntactic differences. To solve this problem we use our srcML and srcDiff representations. Figure 2 depicts the overall process. The process starts with the initial text difference. This difference is lifted to the srcDiff representation (i.e., syntactic differences). The factoring of a difference is then reduced to an XML transformation on the srcDiff format. In the following subsections, we expand on the process including the srcML and srcDiff formats.

3.1. srcML Source-Code Representation

The srcML format [3, 4] is used for the representation of source code in srcDiff. srcML is an XML representation of source code where the source code text is marked with elements indicating the location of syntactic elements. The format supports the representation of all parts of a source code file, including preprocessor directives, white space, and comments. The srcML format has a 1-1 mapping with the text in the original source code file, i.e., a source code file can be put in the srcML representation and later extracted without any loss of text.

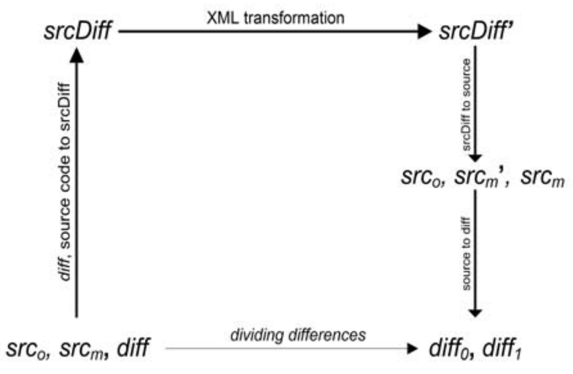

Figure 2. Using srcDiff XML transformations can be used to factor a difference. First, the difference is converted to the srcDiff format. Second, the srcDiff undergoes an XML transformation. This modified srcDiff can be used to extract a new version of the source code. This new version has only the changes that remain in the new srcDiff. In addition, the intermediate version of the source code can be used to generate factored textual differences.

The srcML toolkit includes translators to and from the srcML format. The srcML file is typically 3.5 times the text size and the translation speed to srcML is over 10 KLOC/sec. The srcML toolkit is available both under a GPL and a commercial license (www.sdml.info). For more information about the srcML format, we refer the readers to [4].

The srcML representation and translator are extended to support compound srcML documents. The compound srcML format facilitates the representation of a set of files, e.g., all the source-code files of a KDE commit, in a single srcML document. Each file is represented in an individual *unit* element with directory and filename stored in attributes. The individual unit elements are nested in a single root unit element. Also, the capability of the srcML toolkit to operate on compressed files allows an entire project to be stored in a single compressed file that is typically 1.5 times the size of the equivalent compressed text files.

3.2. srcDiff Difference Format

srcDiff [20] is an intensional format for representing differences in XML, i.e., it contains both versions of the source code and their differences. The srcDiff format is a direct extension of srcML. An example of srcDiff representation is given in Figure 3. The srcML of two versions of a file (i.e., old and new) are stored. The difference elements *diff:common*, *diff:old*, and *diff:new* represent sections that are common to both versions, deleted from the old version, and added to the new version respectively. The sections are well-formed with the srcML elements by the addition of nested *diff:common* elements. Although, the example demonstrates srcDiff for only two versions of a file, it allows representation of any number of versions. Similar to srcML, srcDiff also supports a compound format to represent differences between sets of files.

3.3. Generation of Difference Format

The approach to srcDiff generation uses the *diff* utility to find the textual differences of two versions of a file. The combined output of *diff* that marks differences using the preprocessor directives *#ifdef*, *#else*, and *#endif* is used. This output is translated into srcML, and then a series of textual substitutions replaces the preprocessor directives with empty difference elements. Further analysis is performed to obtain a finer granularity of differences than is available from the diff utility. This processing is linear with respect to the number of srcML tokens. The empty-difference elements are finally converted to srcDiff elements that are well-formed with respect to the srcML elements.

The generation of srcDiff is based on line-based differences because of their robustness and speed. In order to be practical the speed of generating srcDiff is very important. In our previous work [20] the srcDiff format was generated at a speed in the range of 100 LOC/second. In this work the approach takes advantage of new features in the srcML translator. The speed of the generation of the srcDiff format improved to 6 KLOC/second, an over 60 times speedup. In addition to the speedup, the improved srcDiff generation provides a finer granularity of changes.

3.4. Meta-Differencing

Once source code is in the srcDiff representation, changes can be analyzed and manipulated using common XML tools. We term this *meta-differencing* as it allows the extraction of information from differences, including their context. Meta-differencing is an extension of queries on the srcML representation. Elements in the source code (syntactic and documentary) can be located by an XPath expression to the srcML element, e.g., to locate a function with the name of sort, the XPath expression is */unit//function[name='sort']*. Determining the syntactic context of a change (or the change context of a syntactic element) is performed by using the difference elements, e.g., to locate all the added code in the function with the name of sort the XPath expression is */unit//function[name='sort']//diff:new*. The result is all the code that is contained in these additions, including text and other elements.

Note that because of need to keep the srcML elements nested in a difference well-formed, nested *diff:common* elements are used. Thus, a srcML element in the difference element *diff:new* may also be contained in a difference element *diff:common*. In order to determine whether an element is added, deleted, or in common, the parent of the element along the difference axis is used.

3.5. Transformation of Differences

In our approach the differences are represented in the srcDiff format, and transformation of differences is a transformation on the srcDiff format. In this section we will begin by explaining how we can make these transformations.

First we note that from the srcDiff format two different versions of a file may be extracted. For the difference δ_m, where f_o is the original file and f_m the modified version, version f_o is formed from the content of the difference elements *diff:common* and *diff:old*, and version f_m is formed from the content of the difference elements *diff:common* and *diff:new*. In order to understand how nested difference elements are handled, consider a stack that stores the open difference elements. Whenever the start of a difference element is reached it is pushed on the stack, and whenever the end of a difference element is reached the difference element is popped from the stack. The decision as to when a srcML or text node is placed on the output is determined by the top element of the stack, i.e., *diff:common* and *diff:old* for f_o and *diff:common* and *diff:new* for f_m.

Transformation of a difference δ_m to δ_m' changes the version f_m that is produced to a version f_m'. The transformation changes the difference elements, and may change the srcML elements nested inside. There are three different operations that that can be applied during the transformation. We can reject a change that deletes code or reject a change that adds code. We can also transform a replacement change (old replaced by new), as a combination of two.

Eliminating a code deletion is based on changing *diff:old* elements. A *diff:old* element occurs in only the original version, while a *diff:common* appears in both versions. We must change it so that it appears in both versions and can do so by changing the *diff:old* to a

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

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

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

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

Figure 3: A portion of a srcDiff document showing the old, new, and common difference sections. The old text is strikethrough and the new text is bold.

diff:common. All the text and nested srcML elements must also be preserved. Any nested *diff:common* or *diff:new* elements remain intact.

Elimination of a code addition is based on changing *diff:new* elements. A *diff:new* element occurs in only the new version. In this case the *diff:new* element and its content must be removed. Any nested *diff:common* or *diff:old* however must be preserved.

3.6. Factoring Differences

There are multiple ways of factoring a change. As a result criteria specific to the task and/or desired view of a change are needed for factoring. For example, the acceptance criteria may require two factors based on changes to a particular entity. We may want to factor out changes to a particular function from changes to the rest of the system. Recursive application allows a change to a system to be factored into individual changes corresponding to changes to individual functions. Another possible criteria is factoring based on the type of the language element. We may want to allow comments changes to existing comments/elements to occur first.

Criteria are expressed using the XML addressing language XPath applied to a srcDiff document. The criteria XPath indicate which parts of the XML document is to be part of the change. The criteria could specify which differences are to be included, or excluded. Specifying inclusion is probably more natural since that is used in an XPath expression evaluation tools, and template matching in XSLT. Note that the expressiveness of the XPath addressing on the srcDiff format allows for any combination of the two.

3.7. Factoring Tool

The previous discussion showed the wide variety of XPath expressions that can be formed and used on the srcDiff representation. In this section we describe the tool that we developed and used in the evaluation.

The tool, *difffact* factors differences. The input to the tool is a compound srcDiff document that is a collection of all the files involved in the change, and the output is the transformed srcDiff that can produce factors of a change.

The tool takes two parameters. The first parameter specifies the type of change that is to be filtered out, either addition, deletion, or both. The second parameter is the location in the srcDiff document where the transformation is to occur. The location is given as an XPath expression. For example, to filter out all added statements in a function definition the command is *difffact add "/src:unit/src:unit//src:function/src:block"*.

The tool is written using TextReader, a streaming XML API that is part of the libxml2 (xmlsoft.org). As each element *unit* in the compound srcDiff document is reached that part of the XML tree is expanded and the XPath parameter is evaluated. At the end of the element *unit* this part of the expanded XML tree is removed. Our experience with this approach in a query tool on compound srcML documents has shown it to be quite efficient, e.g., minutes to query on a srcML representation of the Linux kernel. However, this is not the only way to perform difference factoring on srcDiff, i.e., XSLT, and other XML API's can be used.

After generating the modified srcDiff document the files can be extracted using another tool, *diffver.* The tool *diffver* takes a srcDiff file and extracts either the original or modified form as a compound srcML document. The srcML toolset can be used to extract the text files from the srcML document. If needed, textual differences can be generated using the utility *diff* on the extracted text files.

4. Evaluation

The primary goal of the evaluation is to demonstrate our approach in factoring a large change performed in a large-scale software system developed in a highly collaborative environment. *KDE* is used as a subject system. A prerequisite is to acquire an instance of a large change that serves in evaluating the expressiveness (i.e., represent factors at the various syntactic levels) and effectiveness (i.e., realistic performance) of the approach. In the rest of this section, we discuss how one such large change was obtained and factored.

File	Change (# of lines)
arbitraryhighlighttest.cpp	A2(43)
arbitraryhighlighttest.h	A2(2)
attribute.cpp	A1(4), A2(13), A5(14)
attribute.h	A1(13), A2(13), A5(16)
attribute_p.h	A1(4)
katedocument.cpp	A2(6), K4(18)
katedocument.h	A2(8), A5(13)
katerenderer.cpp	K1(223)
katesmartcursor.cpp	A3(9)
katesmartcursor.h	K2(8)
katesmartmanager.cpp	K1(6), K4(2)
katesmartmanager.h	K1(4), K4(3)
katesmartrange.cpp	K2(13), K3(12), K6(19)
katesmartrange.h	K2(37)
kateview.cpp	A2(15), K4(4)
kateview.h	A2(19)
kateviewhelpers.cpp	ND(42)
kateviewinternal.h	A2(19)
kateviewinternal.cpp	A2(51), A4(2), K2(17), K4(15), ND(1)
range.cpp	A6(13)
range.h	A6(40)
rangefeedback.cpp	A3(16)
rangefeedback.h	A3(95)
smartinterface.h	A2(6)
smartrange.cpp	A1(8), A4(35), A5(12), A7(11), K3(8), K4(36)
smartrange.h	A3(10), A4(13), A5(23), A7(17)

Table 1. Files involved in the example commit change. For each file the individual change and the number of lines in that change are given. The lines that are not clearly associated with any of the individual changes are labeled ND.

4.1. Changeset Acquisition

The source-code repositories of *KDE* are managed by *Subversion*. *Subversion* preserves all the changed files submitted in a single commit operation as an atomic *changeset*. Additionally, each changeset is annotated with metadata such as the committer's identify, date, a text message provided by a committer, and names of the changed files. *Subversion* stores the metadata of changesets as log records. These log records (specifically the text messages) are utilized in acquiring changes that are potential candidates in evaluating our approach.

One approach to extract the log records from a *Subversion* repository is using the client command *svn log*. However, this approach requires a working copy of the repository. Clearly, this approach is not feasible for use-cases in which a desired subset is obtained from a search space of a large system with thousands of changesets. Therefore, we developed the tool *changeextractor* that uses *pysvn* (i.e., *Subversion* API for *Python*) to extract changesets (without a working copy) from the repository. The tool *changeextractor* takes a set of search terms (e.g., *refactoring*) and the repository URL and outputs a set of changesets containing any of the search terms in their text messages. Thus, the extraction of changeset of interest basically reduces to the specification of search terms.

We restricted our interest to the refactoring and API changes. These types of changes are typically composed of small incremental steps. In such cases, the changeset may correspond either to a small step (desired good practice) or the entire end result (potentially problematic). Therefore, the changesets corresponding to the end results are of primary interest. The application of *changeextractor* to the *KDE* repository with search terms refactoring and API matched 32 changesets. The text messages of matched changesets were manually examined and a changeset was selected. The selected changeset is a large complex change consisting of 28 files involving API (text highlighting features), refactoring (rename methods), and other minor changes. This changeset mainly involves changes to the text editor *Kate*. Table 1 shows the files and the number of lines changed in them on account of individual changes. Changes to (two) files related to build configuration (*Makefile.am*) and a *TODO* list are ignored.

The text message of this changeset contains a very short (one-two line) description of each individual change. In order to further verify that this changeset is truly complex, one of the authors manually examined the changed files and associated each changed line with the corresponding individual change. Some of the individual changes crosscut multiple files and syntactic structures. Associating individual changes to the line-change factors took a considerable amount of manual effort (approximately three hours). A large number of the API and refactoring changes in this changeset substantially consisted of comment changes more than the other types of syntactic structures. If the syntactic composition and context of a change is available, the reviewing and acceptance process could be better organized. In the case of a large API change, the changes in methods can be examined first and the comment changes last.

4.2. Conversion to the srcDiff format

Once the example changeset was available it was converted into the srcDiff format using the toolset described in Sections 3.2 and 3.7. All 26 files were individually converted to the srcDiff format then merged into a compound srcDiff document. The file sizes were reasonable with the resulting srcDiff document 2.1 MB, while the equivalent text size is approximately one fourth the size at 567 KB. The size of XML for representing AST or parse-tree information is a concern [1]. As is recommended for dealing with large XML files [10], the entire toolchain was enabled to process gzipped files. The size of the gzipped srcDiff document was 153 KB, and the size of the equivalent gzipped text was 102 KB for a ratio close to 1.5. The speed of the conversion was under 3 seconds.

4.3. Individual Factors

We focus on the changes in the file *attribute.cpp* to demonstrate factoring into individual changes. Table 1 shows that this file is involved in three individual changes: A1, A2, and A5. For each of these changes we individually factor out a separate version that only contains these changes so that they can be applied individually, i.e., create factors δ_{A1}, δ_{A2}, and δ_{A5}.

Each change occurs in the file *attribute.cpp*, so the XPath expression begins with a selection of this particular file (which we leave out of the following examples for clarity):

/unit/unit[@filename='attribute.cpp']

Factor δ_{A2} is the addition of functions *effects* and *seteffects*. They are the only added functions to this file. So the new sections that have an added function are located and marked. The XPath expression for extracting this factor is:

//diff:new[function]

Factors δ_{A5} are changes in *if*-statements. The if-statements that have new additions are located and marked. We want to remove both types of modifications (i.e., additions and deletions). The XPath expression to select this factor is:

//if[.//diff:old or .//diff:new]

The last factor, δ_{A1}, is the commented-out functions *addRange* and *removeRange*. Since this is the only remaining change, it can be found by extracting (and applying) the other factors, δ_{A2} and δ_{A5} first.

All of these factors were generated using the *difffact* tools on the compound srcDiff document of the entire example (all 26 files). The generation of a modified srcDiff took less than a second.

4.4. Standard Factors

The factors previously presented are very task specific. However, there are some factors that are common to many tasks and which could be applied without knowledge of a specific task at hand. They are based on the types of syntactic entities in the context of differences.

The first distinction is between documentary and non-documentary changes, specifically comment changes. The ability to factor out comment changes (which are probably accepted without much examination) from non-comment changes (which may require a detailed review) can simplify the change acceptance process. To demonstrate an entire commit change was divided into two factors, δ_c and $\delta_{\sim c}$ which are for comment changes and non-comment changes respectively.

Updated comments, i.e., existing comments that have modification to their text, can often be accepted without much review. So as a first example we form a factor δ_c consisting of these comments. More specifically, comments that contain *diff:old* or *diff:new* elements. The XPath expression to select the comments in this factor is:

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

22 comments were found by applying the above XPath expression.

Another variation of comment extraction is comments that were deleted. In general, we are interested in comments directly inside of the difference elements *diff:old* and *diff:new*, and not comments inside of other entities that are deleted or added. The XPath expression to select factors of comments that were directly added or deleted is:

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

17 comments that were directly added and 15 comments that were directly deleted were found by applying the above XPath expression. Note that the XPath path operator or "|" can only be used at the top level in XPath 1.0. Each part of the XPath can be applied separately.

And as a final example, it may be interesting to factor modifications to existing return-statements using an approach similar to modifications of existing comments. There were only three of these return-statements in the entire system. They happen to be in a single file, *attribute.cpp*, which is the example given in the previous section.

5. Discussion

The undertaken approach and its evaluation bring forward various open issues. This section discusses these issues and outline possible directions in addressing them.

5.1. Manual Effort and Dependency

The undertaken approach requires the manual generation of the XPath criteria for factoring an individual change. Also, the classification of the files and lines involved in a change into types of individual changes is performed manually.

The XPath to extract the factors based on the list in the commit message was performed manually. As can be seen, the generation of these XPath expressions was not simple. A natural question is whether these criteria could be extracted (semi) automatically. One problem with automatic extraction is the task dependent view of the prime factor. The specification of a factor would have to be made manually. This could be used to automatically generate the factoring XPath. For example, if the user is interested in a specific set of source code entities, e.g., a set of files, classes, and /or functions involved in a change, then the XPath criteria could form two complementary factors: one pertaining to the entities of interest, and the other addressing the remainder of the change. It is possible to use the information from the software repositories to automatically determine the potential set of entities that may affect a developer's contribution.

The changed lines between two versions of a file were manually classified by examining only the textual differences of the two versions of a file and the commit message. Neither of the two complete versions of a file was used to determine the context of the change. This exposes a threat to the misclassification of a change into inappropriate individual change. Moreover, the classification is also highly dependent on the "quality" of the commit message. Document similarity comparison methods could be employed for classification. The commit messages, and identifiers and comments in source code can be utilized in such an approach. In the absence of a "quality" commit message, the locality of changes to a syntactic structure may prove to be a viable option in clustering a change into a hierarchy of individual changes. For example, the changes in a particular method, that in turn is (possibly with changes in other methods) in a particular class.

5.2. Scalability

The scalability of the approach directly corresponds to the srcDiff representation (size and time of generation), the XPath criteria (size and complexity), and the generation of the factors (time). The size of the srcDiff format is typically less than 4 times that of a text representation as applied to large systems such as *Linux* and *KDE*. Moreover, the speed of srcDff generation is of the order of seconds. This evaluation endorses the scalability of the srcDiff representation and its generation. The srcML format and its translation from source code shows a similar degree of scalability (~3.5 times the space of the text, and ~12 minutes to translate the Linux kernel). As for the size and complexity of the XPath criteria, as the number of files increases, there is a high likelihood that the desired factors may also generalize. Therefore, such factors lead to drastic reduction in the complexity of the XPath expressions. As for the generation of the factors, since this is an application of an XPath expression on a srcDiff file our experience with querying srcML using XPath expressions (minutes for the Linux kernel) shows that this will also scale well.

In summary, although our approach is evaluated here on a medium-sized example, our previous experiences with srcML and srcDiff on a number of large systems for a variety of querying and transformation tasks support the scalability of the approach.

5.3. Granularity

One of the advantages of the srcDiff generation is the use of line differences as a first stage. As we noted in this work additional, low-cost, stages were used to provide finer granularity. These stages used linear comparisons of the line differences avoiding the potential cost of applying LCS (Longest Common Subsequence) to a group of lines involved in a change. The addition of these stages had very little effect on the overall time of processing.

The largest effect on the efficiency of differencing is the granularity of the difference. Line-based differencing is fast because it only considers differences of lines. Complete syntactic differencing, such as AST-based approaches, can take time that is of the quadratic order of the number of nodes. Partial syntactic differencing takes less time, but takes as a single, large change, whenever an element is inserted that wraps around existing elements, e.g., inserting an *if*-statement around previously existing statements.

Additional stages could be added to further improve the granularity. These stages could be based on an as-needed basis and depend on individual statements and syntax. The availability of the srcML markup in the context of the differences allows for a rich infrastructure to create entity-specific granularity improvements.

5.4. Factor Operators

XPath is an extremely location expressive language. Not only can it refer to specific locations via a path, but it can also express complex relationships between elements. The use of these complex relationships may lead to the automatic merging of separate factors.

One example is a change to portions of source code related to each other which do not have an explicit way of stating (in the programming language) their relationship. For example, it is common to put a comment before a function describing its purpose. If we have two separate factors, one a change to the function and one a change to a comment, it would be useful to combine these factors. This has to be done using external knowledge of these relationships.

A factor operator would take a set of factors and try to combine as many as it can. The output is a new set of factors. These factor operators look for specific relationships between the factors. For example, a *comment-function* merge factor would look to see if any of the comment factors could be merged with any of the function factors.

5.5. Integration with Version-Control Systems

The two versions of a file and their differences are the only external inputs to the overall toolset for factoring changes. As shown in 4.1, the tool *changeextractor* obtains these inputs from the software repository by using the API of a version control system (i.e., Subversion). The tool *changeextractor* produces the difference in the unified-difference format. However, the generation of srcDiff requires the compound format with preprocessor directives (see section 3.2). The tool *changextractor* can be easily extended to produce the difference format needed by srcDiff. With this extension, the tool *changeextractor* and the rest of the toolset can be seamlessly integrated with the version-control systems such as Subversion.

6. Related Work

Differencing is performed for many tasks including comprehension, patching, merging, and automatic change detection. The particular task affects what kind of change is considered a difference and the representation of the difference.

The main categories of differencing include textual, syntactic, and semantic [12]. Textual differencing detects changes in textual lines as in the utility *diff* [13] which uses the LCS (Longest Common Subsequence) algorithm [14]. Syntactic differencing is concerned with changes in an AST, as in LTDIFF [12] and Dex [26], or changes in a parse tree as in our previous work on meta-differencing [20] and is typically applied to merging [21]. Semantic differencing is concerned with changes in behavior. Since detecting changes in behavior is undecidable, heuristics are used as in [11] [16] and [2].

While textual differencing may be applied to XML documents, there is a great deal of interest in differencing which takes advantage of the format. These differencing tools include [5, 15, 19, 22, 25, 28, 29] with formats including [6, 18, 23, 24, 30]. These tools either have quadratic or higher complexity, or when applied to the srcML representation ignore ordering and white space. One exception is [28] which applies the LCS algorithm to a flattened XML tree.

The formats produce edit scripts that do not reflect the context of a change, or only allow the marking of changes to elements, not to text (which is a problem for changes in the text of comments). Recent work has been performed on providing a better context for changes [27]

There has been recent interest in detection of certain types of differences especially refactorings. Gall et al. [7, 8] use a lightweight AST differencing approach. Görg et. al. [9] use a lightweight parsing approach to determine syntactic elements which are then compared from version to version for changes.

There is a tool that provides minimal support for separating differences. WinMerge (winmerge.sourceforge.net) is an open source tool for differencing and merging text files. It provides a line-based view that can be filtered using regular expressions based on the content of an individual line. In merge mode it can be used to selectively pick which change to apply. However, the merging is completely line-based and manual with no ability to use the syntactic context of a change or to match large numbers of changes in a single expression.

7. Conclusions

We presented an approach that factors single commit-level changes into a series of smaller changes by a syntactic criteria. The approach provides iterative acceptance and feedback mechanisms to directly support management of large changes. .

The syntactic criteria are shown to be both expressive and effective. Differences at a syntactic level were found efficiently. Both the generation of the srcDiff format and the factoring is very efficient and therefore scalable to large systems.

Future work is to create standard factoring criteria so that large changes can be automatically factored without manual writing of the criteria. This also involves the issue of concept location for individual parts of a commit change. In addition, we are working on the packaging of the factoring tools and making it available under the GPL license.

8. References

[1] Anderson, P., "The Performance Penalty of XML for Program Intermediate Representations", in Proceedings of Fifth IEEE Internation Workshop on Source Code Analysis and Manipulation, Budapest, Hungary, Sep 30 - Oct 1, 2005 2005, pp. 193-202.

[2] Apiwattanapong, T., Orso, A., and Harold, M. J., "A Differencing Algorithm for Object-Oriented Programs", in Proceedings of 19th IEEE International Conference on Automated Software Engineering (ASE '04), Linz, Austria, September 20-25 2004.

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

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

[5] DeltaXML, "DeltaXML.com Change Control for XML in XML", Date Accessed: May 16, http://www.deltaxml.com, 2004.

[6] DeltaXML, "How DeltaXML Markup Represents Changes to XML Files", Date Accessed: May 16, www.deltaxml.com/core/deltaxml-changes-markup.html, 2004.

[7] Fluri, B. and Gall, H., "Classifying Change Types for Qualifying Change Couplings", in Proceedings of 14th IEEE International Conference on Program Comprehension, Athens, Greece, Jun 14-16, 2006 2006, pp. 10 pages.

[8] Fluri, B., Gall, H., and Pinzger, M., "Fine-Grained Analysis of Change Couplings", in Proceedings of Fifth IEEE Internation Workshop on Source Code Analysis and Manipulation, Budapest, Hungary, Sep 30 - Oct 1, 2005 2005, pp. 66 - 74.

[9] Görg, C. and Weißgerber, P., "Detecting and Visualizing Refactorings from Version Archives", in Proceedings of 13th IEEE International Workshop on Program Comprehension, St. Louis, Missouri, USA, 2005, pp. 205-214.

[10] Harold, E. R., Effective XML 50 Specific Ways to Improve Your XML, Addison-Wesley, 2004.

[11] Horwitz, S. and Reps, T. W., "The Use of Program Dependence Graphs in Software Engineering", in Proceedings of International Conference on Software Engineering (ICSE), Melbourne, Australia, May 11 - 15 1992, pp. 392 - 411.

[12] Hunt, J. J. and Tichy, W. F., "Extensible Language-Aware Merging", in Proceedings of IEEE International Conference on Software Maintenance (ICSM'02), Montreal, Canada, October 3-6 2002, pp. 511-520.

[13] Hunt, J. W. and McIllroy, M. D., "An Algorithm for Differential File Comparision", AT&T Bell Lab. Inc. 1976.

[14] Hunt, J. W. and Szymanski, T. G., "A Fast Algorithm for Computing Longest Common Subsequences", CACM, vol. 20, no. 5, May 1977, pp. 350 - 353.

[15] IBM, "XML TreeDiff", Date Accessed: May 16, http://alphaworks.ibm.com/tech/xmltreediff, 1998.

[16] Jackson, D. and Ladd, D. A., "Semantic Diff: A Tool for Summarizing the Effects of Modifications", in Proceedings of IEEE International Conference on Software Maintenance (ICSM'94), Victoria, British Columbia, Canada, September 19-23 1994, pp. 243-252.

[17] Kraoh-Hartman, G., "HOWTO do Linux kernel development", Date Accessed: Apr 5, http://www.kroah.com/log/linux/howto.html, 2005.

[18] Laux, A. and Martin, L., "XUpdate Working Draft", http://exist-db.org/xmldb/xupdate/xupdate-wd.html, 2000.

[19] Logilab, "xmldiff", Date Accessed: May 16, http://www.logilab.org/projects/xmldiff, 2003.

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

[21] Mens, T., "A State-of-the-Art Survey on Software Merging", IEEE Transactions on Software Engineering, vol. 28, no. 5, May 2002, pp. 449 - 462.

[22] Microsoft, "Microsoft XML Diff and Patch 1.0", Date Accessed: January 19, http://apps.gotdotnet.com/xmltools/xmldiff, 2002.

[23] Microsoft, "Microsoft XML Diff Language v.1.0 Beta", Date Accessed: January 19, http://apps.gotdotnet.com/xmltools/xmldiff, 2002.

[24] Mouat, A., XML Diff and Patch Utilities, Heriot-Watt University, Edinburgh, Scotland, Senior Project, 2002.

[25] Mouat, A., "diffxml", Date Accessed: May 16, http://diffxml.sourceforge.net, 2004.

[26] Raghavan, S., Rohana, R., Podgurski, A., and Augustine, V., "Dex: A Semantic-Graph Differencing Tool for Studying Changes in Large Code Bases", in Proceedings of 20th IEEE International Conference on Software Maintenance (ICSM'04), Chicago, Illinois, September 11 - 14 2004, pp. 188-197.

[27] Schubert, E., Schaffert, S., and Bry, F., "Structure-Preserving Difference Search for XML Documents", in Proceedings of Extreme Markup Languages, Montreal, Quebec, Aug 7-11, 2006 2005, pp. 10.

[28] Walsh, N., "diffmk", Date Accessed: May 11, http://sourceforge.net/projects/diffmk/,

[29] Wang, Y., DeWitt, D. J., and Cai, J.-Y., "X-Diff: An Effective Change Detection Algorithm for XML Documents", in Proceedings of 19th International Conference on Data Engineering (ICDE'03), Bangalore, India, 2003, pp. 519-530.

[30] XML:DB, "XML:DB Initiative for XML Databases", Date Accessed: April 11, http://www.xmldb.org, 2002.