ENABLING PRECISE INTERPRETATIONS OF SOFTWARE CHANGE DATA

by

David Kawrykow

School of Computer Science
McGill University, Montreal

August 2011

A THESIS SUBMITTED TO MCGILL UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS OF THE DEGREE OF
MASTER OF SCIENCE

Copyright © 2011 by David Kawrykow

Abstract

Numerous techniques mine change data captured in software archives to assist software engineering efforts. These change-based approaches typically analyze change sets – groups of co-committed changes – under the assumption that the development work represented by change sets is both meaningful and related to a single change task. However, we have found that change sets often violate this assumption by containing changes that we consider to be *non-essential*, or less likely to be representative of the kind of meaningful software development effort that is most interesting to typical change-based approaches. Furthermore, we have found many change sets addressing multiple *subtasks* – groups of isolated changes that are related to each other, but not to other changes within a change set. Information mined from such change sets has the potential for interfering with the analyses of various change-based approaches.

We propose a catalog of non-essential changes and describe an automated technique for detecting such changes within version histories. We used our technique to conduct an empirical investigation of over 30 000 change sets capturing over 25 years of cumulative development activity in ten open-source Java systems. Our investigation found that between 3% and 26% of all modified code lines and between 2% and 16% of all method updates consisted entirely of non-essential modifications. We further found that eliminating such modifications reduces the amount of false positive recommendations that would be made by an existing association rule miner. These findings are supported by a manual evaluation of our detection technique, in which we found that our technique falsely identifies non-essential method updates in only 0.2% of all cases. These observations should be kept in mind when interpreting insights derived from version repositories.

We also propose a formal definition of "subtasks" and present an automated technique

for detecting subtasks within change sets. We describe a new benchmark containing over 1 800 manually classified change sets drawn from seven open-source Java systems. We evaluated our technique on the benchmark and found that the technique classifies single- and multi-task change sets with a precision of 80% and a recall of 24%. In contrast, the current "default strategy" of assuming all change sets are single-task classifies single- and multi-task change sets with a precision of 95% and a recall of 0%. We further characterized the performance of our technique by manually assessing its false classifications. We found that in most cases (78%), false classifications made by our technique can be further refined to produce useful recommendations for change-based approaches. Our observations should aid future change-based seeking to derive more precise representations of the changes they analyze.

Résumé

De nombreuses techniques de génie logiciel exploitent l'information stockée dans des systèmes de gestion de versions. Ces techniques analysent généralement des groupes de changements (ou change sets) sous l'hypothèse que le travail de développement contenus dans ces change sets est à la fois pertinent et relié à une seule tâche. Nous avons constaté que les change sets violent souvent cette hypothèse lorsqu'ils contiennent des changements que nous considérons comme non-essentiels, c'est-à-dire, non-représentatif des changements normalement associés au développement de logiciel. Par ailleurs, nous avons trouvé de nombreux change sets qui contiennent plusieurs sous-tâches – des groupes de changements isolés qui sont reliés les uns aux autres, mais pas à d'autres changements du même change set. L'information extraite de change sets contenants des changements non-essentiels ou des changements reliés à plusieurs sous-tâches peut interférer avec les diverses techniques qui exploitent des systèmes de gestion de versions.

Nous proposons un catalogue de modifications non-essentielles et une technique automatisée pour détecter de tels changements dans les systèmes de gestion de versions. Nous avons utilisé notre technique pour mener une étude empirique de plus de 30 000 change sets dans dix logiciels libres en Java. Notre étude a révélé que entre 3% et 26% de toutes les lignes de code modifiés et entre 2% et 16% de toutes les méthodes modifiées sont modifiés seulement par des modifications non-essentielles. Nous avons également constaté que l'élimination de telles modifications réduit la quantité de fausses recommandations qui seraient faites par un analyse de type "associtation rule mining." Ces conclusions sont appuyées par une évaluation manuelle de notre technique de détection, par laquelle nous avons constaté que notre technique identifie faussement des méthodes non-essentielles

dans seulement 0,2% des cas. Ces observations devraient être tenues en compte dans l'interprétation des résultats d'analyse de données contenues das des systèmes de gestion de versions.

Nous proposons aussi une définition formelle de “sous-tâches” et présentons une technique automatisée pour détecter les sous-tâches dans les change sets. Nous décrivons un benchmark contenant plus de 1800 change sets tirées de sept systèmes Java. Nous avons évalué notre technique sur cette référence et avons trouvé que la technique classifie des change sets mono-tâche et multi-tâche avec une précision de 80% et un rappel de 24%. En revanche, la “stratégie par défaut” qui assume que tous les change sets sont mono-tâches classifie les change sets avec une précision de 95% et un rappel de 0%. Nous avons également caractérisé la performance de notre technique en évaluant manuellement ses classifications erronées. Nous avons constaté que dans la plupart des cas (78%), les classifications fausses faites par notre technique peuvent être améliorées pour produire des recommandations utiles.

Acknowledgments

The author thanks his advisor, Martin P. Robillard, for his many useful insights and detailed feedback about all ideas presented in this work. The author thanks Barthélémy Dagenais for his expertise in and continued technical support of SEMDIFF and PPA. The author also thanks Yijia Xu, Tristan Ratchford, Karl Kettenring, Sammy Scheibenhauer, and Martha Mansternacker for their personal support, as well as NSERC for its funding.

Contents

Abstract i

Résumé iii

Acknowledgments v

Contents vi

List of Figures ix

List of Tables x

1 Introduction 1

1.1 Background 3

1.1.1 Version Repositories 3

1.1.2 Change Sets 4

1.1.3 Edit Scripts 5

1.1.4 Current Data Refinement Model 6

1.2 Motivation and Problem Statement 7

1.3 Proposed Solution 9

2 Detecting Non-Essential Changes 10

2.1 Motivating Example 11

2.2 A Catalogue of Non-Essential Changes 12

2.3 Detecting Non-Essential Changes 17

2.4 Viewing Detected Changes 26
2.5 Empirical Study 28
2.5.1 Set up 28
2.5.2 Prevalence of Non-Essential Differences 31
2.5.3 Impact on Association Rules 36
2.5.4 Impact on Bug-Fixing Change Sets 38
2.5.5 Precision of the Detection Technique 40
2.5.6 Discussion 43

3 Detecting Subtasks **46**
3.1 Definitions and Problem Statement 48
3.2 Motivating Example 49
3.3 Approach 51
3.3.1 Keyword Connections 53
3.3.2 Dataflow Connections 54
3.3.3 Context Connections 58
3.3.4 Hierarchy Connections 58
3.3.5 Combining Connections 58
3.4 Evaluation 59
3.4.1 Creating the Benchmark 60
3.4.2 Quantitative Results 62
3.4.3 Qualitative Analysis 65
3.4.4 Discussion 67

4 Related Work **71**
4.1 Change Descriptions 71
4.1.1 Basic Differencing Tools 71
4.1.2 Tools Detecting Basic High-Level Changes 72
4.1.3 Tools Detecting Systematic Changes 73
4.1.4 Similarity Detection Tools 73
4.2 Change Interpretations 75

4.2.1 Significance of Low-Level Change Types 75
4.2.2 Classifications of Development Activity 76
4.2.3 Impact of Code Changes 76
4.2.4 The Quality of Mined Data Sets 77

5 Final Discussion **78**

List of Figures

1.1 Data Refinement Model for Change-Based Approaches 6
1.2 Non-Essential Changes in Ant 8
1.3 Proposed Data Refinement Model for Change-Based Approaches 9

2.1 Trivial Changes between two Java Files 11
2.2 Catalogue of Non-Essential Changes 13
2.3 DiffCat Output Format 21
2.4 Overview of the ChangeDistiller Wrapper 22
2.5 DiffCat Implementation 24
2.6 Viewing Detected Changes 27

3.1 Detecting Subtasks in Change Sets 52
3.2 Dataflow Connections between Variables 56

List of Tables

2.1 Characteristics of Target Systems 29
2.2 Code Churn in Target Systems (in kLOC) 32
2.3 Method Updates in Target Systems 33
2.4 Non-Essential Methods in Change Sets 34
2.5 Recommendation Quality 38
2.6 Bug-Fixing Change Sets 39
2.7 Characteristics of Selected Change Sets 41
2.8 Precision of the Technique (in %) 42

3.1 Characteristics of the Benchmark 60
3.2 Split Change Sets 63
3.3 Split Change Sets by Category 65

Chapter 1
Introduction

Software development teams typically use version repositories to manage the evolution of their software systems. Version repositories are useful because they allow developers to commit (persist) changes made to a set of software artifacts and to later rollback those changes to retrieve previous versions of those artifacts. Standard version repositories such as CVS[1] or SVN[2] typically enable this service by storing both the current snapshot of a given software artifact and a representation of all the changes that were applied to previous versions of that artifact.

Given the widespread use of version repositories in software development, numerous research techniques now propose mining the change data stored in version repositories to assist software engineering efforts. For example, the mining of change data has been used to measure code decay in aging systems [9], to predict defects in software modules [16, 34], and to detect non-obvious relationships between code elements [13, 44, 46]. We refer to these kinds of approaches as *change-based* approaches.

Although their techniques and end goals may vary, most change-based approaches operate on *change sets*, or groups of changes that were co-committed as part of a single commit operation to the repository [45]. This approach is motivated by the fact that changes within a given change set often correspond to the development work that was carried out as part of a single and coherent *change task*, such as a bug fix or feature enhancement. Change-based

[1] http://www.nongnu.org/cvs
[2] http://subversion.tigris.org

approaches exploit this natural correspondence between change sets and change tasks to infer useful properties about the underlying software system being changed. For example, Zimmermann et al.'s ROSE tool mines change sets to identify code elements that are frequently co-modified together, and then reminds developers of these associations whenever the developers are found to modify a given subset of those elements [46].

Unfortunately, change-based approaches cannot guarantee that data mined from a version repository actually conforms to the assumptions that are made about that data. Specifically, it is not always the case that all changes mined from a change set reflect interesting development work or effort that was necessarily part of a single and coherent change task. In fact, as we show in Chapter 2, developers for some software systems often commit *non-essential* modifications – changes that are so basic in nature that they do not actually reflect the kind of meaningful software development effort that is most interesting to change-based approaches. For example, whenever a developer renames a given code element using a modern Integrated Development Environment (IDE) such as Eclipse,[3] all references to that element will be *automatically* updated by the IDE without the developer doing any actual work. Such automated *rename-induced* modifications are then less likely to carry valuable information than other changes, such as modifications to the system's control flow, for example. Furthermore, as we show in Chapter 3, developers may also co-commit groups of changes that are clearly related to multiple change tasks. Such multi-task change sets may then contain numerous non-essential or accidental associations between elements that only happened to be committed together, but that did not actually need to be. Without further processing, this non-essential data contained within change sets can interfere with the analyses of typical change-based approaches. For example, in the case of the ROSE tool, non-essential or unrelated changes could potentially decrease the overall precision of its approach by introducing accidental associations between groups of unrelated code elements. Our goal is to help change-based approaches detect and eliminate such non-essential information from their input data.

[3]www.eclipse.org

1.1 Background

Software developers use version repositories to store versions of their source code files. Many software engineering techniques then seek to mine these repositories to infer useful properties about the underlying software. To better formulate our problem statement in Section 1.3, we first outline our assumptions about the kind of information that is available to typical repository-mining techniques and how these techniques process that information to infer useful properties about the system being analyzed.

1.1.1 Version Repositories

Typical version repositories such as CVS or SVN store source code and other software artifacts as plain text or binary files. They thus provide no support for encoding any possible syntactic or semantic structure contained within the files they store. Change-based approaches must reconstruct this structure after retrieving the files from the repository.

For a given file F, typical repositories maintain an explicit copy only of the latest version of F, say F_i, with each prior version $F_{j<i}$ committed to the repository represented only as a delta δ_j. Each δ_j describes the changes between F_j and F_{j+1} in terms of the *lines of text* in both file versions that do not have a matching line in the (respective) other file version. These deltas are typically computed using a line-based *diff* utility, e.g., the UNIX DIFF tool [21]. Whenever a developer commits a new version of a given file F_i, the repository stores the computed δ_{i-1} and replaces the previous version F_{i-1} with this latest F_i. An explicit copy of a given prior version F_j can then be reconstructed and retrieved by applying the computed deltas $\delta_{i-1}, ..., \delta_j$ to the current F_i in succession. Consequently, to facilitate our subsequent discussion of how this information is actually mined by change-based approaches, we will simply assume that change-based approaches always have direct access to the explicit representation of each committed file version, as opposed to some repository-specific format.

Version repositories do not monitor the specific development work made to the files stored within them. Similarly, diffing tools such as DIFF only express the *overall* differences between two files, not the specific sequence of change operations that were actually

performed by developers. Consequently, although there exist frameworks for monitoring developers as they work, e.g., SPYWARE [40], we assume that it is generally impossible to retrieve the actual development work separating the file versions within a version repository.

Typical version repositories allow users to commit multiple files at the same time. For a given n-file commit (n $\geq$ 1), a developer may associate a comment with the commit. These comments typically consist of a short, free-form text description summarizing the changes being committed, or a default tag in case no such description was entered. The repository then associates this comment with each file F_i that was co-committed along with that comment. However, although each repository associates the same comment with each file from the same n-file commit, only some repositories, e.g., SVN, explicitly encode which files were actually committed together [38]. Others, such as CVS, only associate with each file a *time stamp* that describes the time that the repository actually persisted that file [45]. However, given the availability of widely used heuristics for reconstructing what files were co-committed as part of the same commit operation, we assume that each n-file commit can be recovered from the repository [45].

1.1.2 Change Sets

Many change-based approaches, e.g., the ROSE tool, mine *change sets* retrieved from a version repository to infer higher level knowledge that might be useful to software developers. A change set consists of a group of file pairs (F_o^1, F_n^1),...,(F_o^k, F_n^k), where the files F_n^1,...,F_n^k represent the group of files that were co-committed by a developer to the repository (i.e., an n-file commit), and the files F_o^1,...,F_o^k represent their respective previous versions in the repository. Each change set also contains some metadata, which generally includes the comment associated with the commit, the author of the commit, and the time the files that were committed.

Change sets represent partial programs, i.e., a subset of all files in the system they come from. Consequently, given only a change set, it is generally impossible to obtain the same level of information about the code within its files as can be obtained from a complete and compileable program snapshot. For example, given an arbitrary element expression (e.g.,

a method invocation) within a Java-based change set, it is not guaranteed that this element expression can be resolved to its fully qualified element signature (e.g., its fully qualified method signature), because the file in which that element (e.g., the method) was declared might not be part of the change set being analyzed. In contrast, element resolution of this nature is always possible for complete and compileable program snapshots.

Change-based approaches can use one of three possible strategies to deal with this difficulty: They may *i)* avoid the more complex analyses like element resolution and simply limit themselves to deriving information on a textual or syntactic level, they may *ii)* maintain a snapshot of the system being analyzed, or they may *iii)* use heuristics to to infer as much of the missing information as possible. For example, change-based approaches can use *partial program analysis* (PPA) to resolve a high proportion of the element expressions within a change set to their fully qualified element signatures [7]. Although PPA typically enables less element resolution than a complete program snapshot, it is not always feasible to maintain such snapshots when mining version repositories, for example, because developers may commit changes that cause the system to no longer compile.

1.1.3 Edit Scripts

Change sets implicitly encode the differences between file pairs. To operate on these implicit differences, an analysis must therefore express them using some sort of explicit representation, also known as an *edit script*. Edit scripts describe the changes between two files as a sequence of fine-grain edit operations (o_i, e_i), where each (o_i, e_i) encodes some fine-grain element e_i and some operation o_i that describes what happened to e_i between the two file versions. Edit scripts allow change-based approaches to link code elements across program versions and to then assess how those elements were changed [26].

The most general edit scripts are computed by the UNIX DIFF tool, which describes the differences between two files in terms of line insertions and deletions [21]. In contrast, more advanced diffing tools compute edit scripts in terms of actual code blocks or program statements, and whether these were modified or moved, not just inserted or deleted [11]. The tradeoff in this case is that more expressive diffing tools allow change-based approaches to reason more precisely about the specific elements that were modified

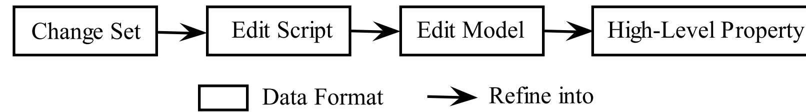

Figure 1.1: Data Refinement Model for Change-Based Approaches

between versions, but that they do so with higher computational overhead, or only for specific inputs, e.g., only for Java files.

1.1.4 Current Data Refinement Model

Typical change-based approaches, such as the ROSE tool, take as input a collection of change sets from a given software system's version repository and produce as output some kind of higher-level knowledge that is usable by a software developer. To help abstract the specific details of how each individual change-based approach achieves this, we can consider all change-based approaches to be roughly equivalent to a series of data refinements, which we depict in Figure 1.1. As we show in the figure, change-based approaches start by extracting from a given change set an edit script that describes the changes encoded by that change set. For example, ROSE infers the non-empty lines of code that were added or removed as part of the change set. This edit script is then converted into a high-level edit model that describes the low-level details of the edit script in terms more closely aligned with the goals of the approach. For example, ROSE maps the added and deleted code lines of the edit script to the functions that were modified by those operations. Finally, this edit model is used to infer high-level properties that might then be useful to a software developer. For example, ROSE uses groups of modified functions to boost support and confidence values for an association rule describing functions that are frequently co-modified together.

1.2 Motivation and Problem Statement

Typical change-based approaches often assume that a given change set captures software development effort that is both coherent and essential. For example, the ROSE tool implicitly assumes that methods updated as part of the same change set *i)* are all related to a single change task and *ii)* all capture what can be referred to as *essential* software development effort, such as changes related to bug fixes or feature enhancements, as opposed to those that were automatically performed by an IDE, for example. However, this assumption does not always hold in practice. Developers might commit changes related to multiple change tasks, or they might commit what we refer to as *non-essential* changes, which do not represent the kind of meaningful development effort that is most interesting to change-based approaches. To illustrate this, we can consider two change sets drawn from the version histories of two open-source Java projects, the first from ANT, a task management system,[4] the second from XERCES, an XML processing tool.[5]

The first change set includes updates to the syntactic content of six methods, among other changes.[6] The change set's commit comment indicates that it is related to an "update to use `getLogger().<method>()` rather than `log()`," which involves a "move [of the] `EchoLevel` inner class to [a] top level class." However, in this case, a manual inspection of the change set reveals that it would be inappropriate for a change-based approach to associate all of the changes in the change set with the effort related to the main or any other important development task. Specifically, apart of the work related to the main task, the developer also performs a number of less meaningful code clean-ups, among them the renaming of three fields (e.g., `file` to `m_file`) and a number of *non-essential* modifications to two of the six methods, some of which are shown in Figure 1.2. These non-essential modifications are so basic that they do not really capture the kind of meaningful software development effort that is most interesting to change-based approaches, e.g., updates to method invocations, control or data flow, etc. It would thus be inappropriate to treat the two method updates resulting from these very simple modifications in the same way as the

[4]http://ant.apache.org
[5]http://xerces.apache.org/#xerces2-j
[6]Committed by `donaldp` on 2001-12-29 at 07:16:00

```
/**
 * Sets the file attribute.
 *
 * @param file The new File value
 */
public void setFile( File file )
{
    this.file = file;
}
```

```
/**
 * Sets the file attribute.
 *
 * @param file The new File value
 */
public void setFile( final File file )
{
    m_file = file;
}
```

Figure 1.2: Non-Essential Changes in Ant

four resulting from the more meaningful work related to the main change task described in the commit comment. For example, the association between these two and the other four methods, as might be captured by the ROSE tool, would be less meaningful than the associations between the four methods associated with the main change task.

The second change set includes structural updates that affect four methods and one field definition.[7] Without further insight into the changes within this change set, a typical change-based approach might then assume that these five element-level updates are part of a single and coherent change task, and perhaps infer additional properties based on this information. However, a manual inspection of the change set reveals that this interpretation would be incorrect. In fact, as is explained in its commit comment, the change set performs changes related to *three* separate subtasks: *i)* “1. fixing bug [623]” in the `URI` class (one method), *ii)* “2. fixing bug [2451]” within the `DatatypeValidatorFactoryImpl` and `TraverseSchema` classes (two methods), and *iii)* “3. fixing an error message” within the `SchemaMessages` and `TraverseSchema` classes (one method, one field definition). In this case, it would be more appropriate to split each method or field update based on the coherent subtask it actually corresponds to. For example, the ROSE tool could avoid possible noise in its detected associations by splitting this change set into three separate change tasks, one per bug fix.

As these two examples show, it is not always appropriate to assume that all the work within a change set is both coherent and relevant. In the first example, the developer performs work that includes a number of changes that should not be treated in the same was as the work related to the change set’s main change task. In the second, they committed work that relates to several change tasks. Ideally, such changes should be appropriately

[7]Committed by `sandygao` on 2001-07-13 at 01:54:00

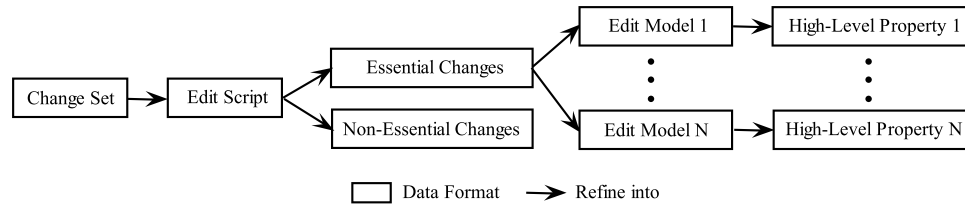

Figure 1.3: Proposed Data Refinement Model for Change-Based Approaches

catalogued for the change-based approaches that operate on them. However, we know of no automated technique that can help change-based approaches make these kind of distinctions between the changes within change sets.

1.3 Proposed Solution

We propose an automated technique that *i)* computes a fine-grained edit script describing changes within a change set, *ii)* identifies changes that we consider to be *non-essential*, and *iii)* identifies subtasks within change sets, or groups of changes that are related to each other, but unrelated to other groups of changes within the same change set.

Figure 1.3 illustrates how the proposed improvements of our technique relate back to the data refinement process used by change-based approaches shown in Figure 1.1. The first goal of our technique is to allow change-based approaches to eliminate changes that are less likely to reflect the kind of meaningful effort that one typically associates with coherent change tasks. The second goal is to further split this meaningful effort into separate and cohesive subtasks, thus allowing a more fine-grained high-level property inference than would otherwise be possible.

We further elaborate on each of these goals in the rest of this dissertation. Chapter 2 further motivates our detection of non-essential differences and outlines our efforts to detect them. Chapter 3 presents our technique for splitting change sets into subtasks, as well as an evaluation of its usefulness. Chapter 4 presents an overview of prior research related to our two goals. Finally, Chapter 5 presents a final discussion about our findings. We have also released all code, experimental designs, and generated data on a companion website: http://www.cs.mcgill.ca/~dkawry/thesis.

Chapter 2
Detecting Non-Essential Changes

Change-based approaches generally seek to model only those changes that are most likely to represent the kind of software development effort that is most relevant to the actual analyses of those approaches. For that reason, diffing tools such as Fluri et al.'s CHANGEDISTILLER automatically filter out what can be referred to as *trivial* changes, e.g., those arising from simple code rearrangement, the insertion or removal of unnecessary curly braces, or updates to the lincensing information at the top of a file [11]. The idea is that changes like these are less likely to be associated with the most meaningful kind of development effort, such as effort related to bug fixes or feature enhancements. For example, CHANGEDISTILLER deems the two Java files in Figure 2.1 to be identical because the only differences between them are trivial in nature: None of them affect the behavior or structure of the code they modify. All that changed between the two versions is *i)* the location of the `field` attribute w.r.t. `method`, *ii)* the location of the curly brace near `method`, and *iii)* the indentation of the two integer declarations within `method`.

As part of our manual investigations of change sets within software archives, we have come across an additional group of changes that, like trivial changes, are less likely to represent the kind of development work that is most likely to be indicative of meaningful software development activity. However, unlike trivial changes, no modern change-based approach actually detects or removes these kinds of *non-essential* changes when analyzing change sets. In this section we provide examples of what we consider to be non-essential changes and describe DIFFCAT, our novel diffing infrastructure for detecting these kinds

```
@SuppressWarnings("unused")
public class V1 {

    private Object field;

    public void method () {
        int x = 5;
        int y = 6;
    }
}
```

```
2
3 @SuppressWarnings("unused")
4 public class V1 {
5
6     public void method ()
7     {
8      int x = 5;
9      int y = 6;
10    }
11
12    private Object field;
13
```

Figure 2.1: Trivial Changes between two Java Files

of changes within Java change sets. We used DIFFCAT to study the prevalence of non-essential differences in change sets capturing over twenty-five years of development history in ten long-lived open-source software systems. In doing so, we found that approximately 10% of all modified lines of code and 8% of all updated methods are updated entirely by the kinds of non-essential differences we detect. We also found that non-essential method updates can have a non-negligible impact on the kinds of association rules that might be inferred by tools like Zimmermann et al.'s ROSE tool [46]. These and other observations are important to keep in mind when studying changes mined from version histories.

This chapter is an extension of our previous paper on non-essential changes in version histories [23]. Everything presented in that previous paper is our own work.

2.1 Motivating Example

We illustrate the concept of non-essential differences and their potential for interfering with higher-level information extracted by change analyses with an actual change set retrieved from the revision history of AZUREUS, a highly downloaded media sharing application.[1] The change set includes modifications to 77 methods, among other structural changes. The modified methods are spread out across 55 classes, which are themselves spread out across 24 packages. The method modifications all involve structurally meaningful updates to method invocations, `if`-statement conditions, or variable assignments. In all, over 700 lines of code are affected by the change, none of them whitespace or documentation-related. All of this information can be readily extracted using currently available automated differencing techniques.

[1] http://www.vuze.com

As the change appears to be quite large, to span a significant number of elements, and to feature non-trivial structural changes, analyses operating at any of these levels of abstraction might infer that the change is likely to introduce a bug [34] or be symptomatic of a decaying system [9]. Other analyses might mine the many pairwise associations between the modified methods and eventually detect non-obvious dependencies between them [46]. However, the developer who committed the changes characterizes the commit in another way. Their commit comment reads: "[Renamed] az3 constants class to ConstantsV3 to make it easier on my brain." Indeed, the developer renamed the `Constants` class to `ConstantsV3` and then committed all files that were trivially modified because of references to the `Constants` class.

Based on this manual assessment, automated interpretations of this change set based on lines, fine-grained structural differences, or the set of updated methods, are likely to yield an inaccurate interpretation of the software development activity or effort behind the change, and may thereby yield incorrect conclusions about potential bugs, system complexity, or non-obvious associations between methods. In this case, a detection of rename-induced and other non-essential differences would have supported a more meaningful abstraction of the change set.

2.2 A Catalogue of Non-Essential Changes

We informally define non-essential changes to be low-level code changes that are *i)* cosmetic in nature, *ii)* generally behavior-preserving, and *iii)* unlikely to yield further insights into the roles of or relationships between the program entities they modify. We keep our definition open-ended to emphasize that the true "essentiality" of code changes still depends on the individual contexts in which they are studied. We focus on those changes that are unlikely to capture meaningful information about the development effort behind a change in many contexts.

To help catalog the kinds of non-essential differences we studied, we refer to two versions of the same Java code, which we show in Figure 2.2. The two versions of the `NE1` class exhibit a number of differences, almost all of which are non-essential. On line 9, a

```
1 package sand;
2
3 import java.util.LinkedList;
4 import java.util.List;
5
6 @SuppressWarnings({"unused","rawtypes","unche
7 public class NE1 {
8
9     private Object val1 = null;
10    private Object val2;
11
12    public NE1 () {
13
14    }
15
16    void method1 () {
17        java.util.List l = new LinkedList();
18        l.add(this.val1);
19        this.val2 = 2;
20        return;
21    }
22
23    void method2 () {
24        boolean x = false;
25        if (x==false) {
26            int[] y = new int[5];
27        } else {
28            String y = "Hello World";
29        }
30        System.out.println("boo");
31    }
32 }
```

```
1 package sand;
2
3 import java.util.LinkedList;
4 import java.util.List;
5
6 @SuppressWarnings({"unused","rawtypes","unch
7 public class NE1 {
8
9     private Object val1;
10    private Object m_val2;
11
12    void method1 () {
13        List list = new LinkedList();
14        list.add(this.val1);
15        m_val2 = 2;
16    }
17
18    void method2 () {
19        boolean x = false;
20        if (! x) {
21            int y[] = new int[5];
22        } else {
23            String y = "Hello" + " World";
24        }
25        String s = "boo";
26        System.out.println(s);
27    }
28 }
29
```

Figure 2.2: Catalogue of Non-Essential Changes

redundant `null` assignment to the `val1` attribute is deleted. Lines 12 through 14 see the deletion of a redundant default constructor. On line 17, the variable `l` is renamed to `list`, and its declared type is trivially updated. The variable rename also induces a rename-induced difference on line 18. Lines 19 and 20, see the deletion of redundant `this` and `return` keywords, respectively. On line 25, an `if`-statement is trivially modified, line 26 sees a redundant array bracket rearrangement, the `String` on line 28 is trivially split into two substrings, and on line 30 the "`boo`" expression is placed into a single-use temporary variable. Finally, the `val2` attribute is renamed to `m_val2` (an essential change), which then induces a non-essential rename-induced difference on line 19.

Although all of these program differences may be of interest for certain change-based approaches, we believe that they are unlikely to contribute relevant information for many techniques seeking to measure meaningful software development effort. It would be unlikely, for example, for a developer to perform the kinds of modifications affecting the `NE1` class to advance the implementation of a cohesive change task, such as developing a new feature or fixing a complicated bug. We further justify this reasoning by considering each type of non-essential difference in isolation.

Trivial Assignment Updates

Java assigns default values to all declared variables whenever those variables are not specifically initialized at the time of their declaration. Specifically, it assigns a value of 0 to numeric variables, `false` to boolean variables, and `null` to all other variables. Consequently, adding or removing these default values to or from a variable declaration (e.g., line 9) has no effect on the runtime behavior of the program. These kind of trivial assignment updates are thus far less likely to be indicative of interesting development work than other kinds of assignment updates.

Redundant Default Constructor Updates

Java automatically creates a public default constructor for all classes that do not explicitly declare at least one other constructor. Consequently, when no other constructors are present, inserting or deleting these default constructors in a class (e.g., lines 12 through 14) does not change the semantic or structural properties of that class.

Trivial Type Updates

Textual updates to an entity's declared type are non-essential if the actual declared type is not affected by the update (e.g., lines 17 and 26). Specifically, a trivial replacement of a type's qualified name with its simple name does not affect how the declared entity is handled at runtime. Similarly, in Java, moving the angled brackets from after the type to after a variable name does not affect the actual type of the variable. In some situations, updates of this nature are made to distinguish the type from a second recently-imported type that shares the same simple name. In these situations, we still consider trivial type updates to be non-essential because they preserve existing program behavior given other essential changes, such as newly inserted references to the newly imported type. We note that such changes cannot be detected by considering only their textual properties, since with only textual information, it would not be obvious whether the `List` word refers to `java.util.List` or some other `List` type, e.g., `java.awt.List`.

Local Variable Renames

Developers may rename local variables only to increase the overall readability of the code. While a cosmetic change of this nature might be interesting for a study of code readability, in the general case it is typically unimportant to a change task or bug fix. In those cases where a variable name update truly does imply a change in the role of the variable, then this role change will be accompanied by other essential code changes, e.g., modifications to method invocations or control flow involving that variable.[2]

Rename-Induced Modifications

Whenever a developer renames a program entity (i.e., class, field, method, parameter, or local variable), any code statement referencing that entity will be textually modified as part of the rename (e.g., lines 18 and 19). These secondary textual changes are generally not relevant when studying program differences, as they are only a necessary by-product of existing program structure, which must be adapted to avoid compilation or runtime errors. In fact, many IDEs, e.g. Eclipse, even help developers perform rename refactorings by automatically updating all references to renamed entities. These kinds of automated reference updates are thus far less likely to contribute meaningful insight about the development effort behind a change than the actual renaming of the code element itself. Therefore, we consider the actual renaming of the code element to be an essential change, but argue that the textual reference updates induced by that renaming are non-essential. This argument echoes one made in previous work by Neamtiu et al., which presents a differencing technique that corresponding rename-induced updates as a single difference between program versions [35]. Similarly, in their work on change significance classification, Fluri et al. suggest that the *effects* of parameter renames should not be considered "significant" when analyzing code changes [10]. Like these approaches, we also consider rename refactorings to constitute a single (essential) developer modification that also induces a number of non-essential side effects as a by-product.

[2] In very rare cases, a variable renaming might cause the variable to shadow an existing attribute. We consider these renames to be essential because they can cause potentially noticeable changes in the behavior of the program.

Trivial Keyword Modifications

In Java, prepending the `this` keyword to a program entity only affects program behavior in a limited number of cases.[3] In Figure 2.2, the deletion of the `this` keyword (line 19) has no effect on the behavior of the `method1` method. While changes involving the `this` keyword might improve readability of the code, they can be considered non-essential in most contexts. Similar to type updates, in those cases where adding a `this` keyword is necessary to preserve an existing field access, we still consider such an insertion to be non-essential because it preserves existing behavior given other unrelated changes. In this situation, missed `this` keyword insertions actually have a greater impact on program behavior than redundant insertions. In this category we also currently include trivial insertions or deletions of `return` statements at the end of `void`-returning methods (e.g., line 20) and trivial insertions or deletions of default `super` invocations occurring at the top of default constructors (not shown).

Trivial If-Statement Updates

We have come across cases where developers trivially modify `if`-statements, for example, by replacing their simple boolean conditions with other equivalent expressions (e.g., line 25). Such rewrites might improve the readability of the code, but do not usually modify the way that code behaves at runtime. We do not include in this category `if`-statement updates that consist of reorderings of composite expressions involving field or array accesses, or method invocations, because such reorderings, although semantically equivalent during standard program execution, are not always identical during exceptional program execution, e.g., in the case of `null` pointers.

Cosmetic String Splits

In some cases, developers might split long `String` literals into a series of smaller concatenations involving the + operator. Although such concatenations can introduce slight

[3]Specifically, the `this` keyword is required to reference a field inside scopes found to declare a local variable sharing the same name as the field.

additional runtime overhead, it is unlikely that such cosmetic `String` splits are meant to modify anything except the readability of the code.

Local Variable Extractions

Developers may improve the readability of code by using temporary variables to store expressions and using those variables instead of the expressions in a *single* subsequent program statement. Such local variable extraction refactorings are cosmetic in nature, have no effect on a program's behavior, and do not need to be performed in the context of a related change task.

Whitespace and Documentation-Related Updates

Whitespace and documentation-based modifcations are already ignored by other change analysis tools, such as CHANGEDISTILLER [11]. investigation of non-essential differences, we took steps to eliminate these modifications from our input data. We thus only report on the prevalence and possible impact of the non-essential differences outlined above.

2.3 Detecting Non-Essential Changes

All of the non-essential changes outlined in Section 2.2 affect individual programming language statements or expressions. Consequently, to detect these changes in change history requires analyzing changes at a level of granularity finer than statement-level differences. Detecting the non-essential differences in our catalog also requires resolving arbitrary program expressions within statements to their fully qualified element signatures, a technically challenging task given that the files analyzed are not part of a complete and compilable system. For example, to detect trivial `this` keyword insertions or deletions requires an analysis that detects not only the additional or missing `this` keywords within statements, but also verifies that no element expressions were actually altered by their insertion or removal. Detecting non-essential changes in version histories thus requires a differencing technique that is both fine-grained (working at the level of expressions within statements)

and semantically-sensitive (to reason about the impact of changes on the program behavior). Furthermore, although existing change analysis tools already support fine-grained differencing of individual program statements (e.g., CHANGEDISTILLER [11]), we know of no change differencing tool that is both fine-grained and sensitive to the semantics at the expression (sub-statement) level.

To detect non-essential differences, we thus developed a novel differencing technique that is both fine-grained and able to reason about the element references within individual element expressions. We implemented our technique in a tool called DIFFCAT. Similar to existing change analysis tools, DIFFCAT takes as input a group of co-committed source files retrieved from a software repository (a change set) and returns as output a description of the various structural modifications characterizing changes to statements within the change set (a statement-level edit script). In addition to previous techniques, however, our technique is also able to reason precisely about the elements referenced within individual element expressions and to identify and label structural changes that we consider to be non-essential. For example, DIFFCAT detects and labels cases where a program statement was modified only by the trivial insertion of one or more `this` keywords. DIFFCAT is currently implemented to handle Java code stored in CVS and SVN. Readers can download a version of our tool from our companion website: http://www.cs.mcgill.ca/˜dkawry/thesis.

Reused Components

To resolve element expressions within the Java files it analyzes, DIFFCAT reuses an existing implementation of partial program analysis, called PPA [7]. PPA takes as input a collection of Java files (e.g., the old or new files within a change set) and produces as output a collection of resolved Abstract Syntax Trees (ASTs) representing those files. The nodes within each AST are also supplemented with *bindings*, which link elements referenced within an AST node (e.g., method invocations, variable names, etc.) to their fully qualified element signatures. We chose to detect non-essential differences using partial program analysis because we know of no existing infrastructure that enables efficient retrieval and compilation of a separate program snapshot for each individual change set.

To further facilitate our detection of fine-grained non-essential differences between the

ASTs produced by PPA, we use CHANGEDISTILLER [11], a state-of-the-art differencing tool that identifies various kinds of statement-level structural changes between Java AST pairs. These differences are outlined in a separate catalog [10]. DIFFCAT enhances the output computed by CHANGEDISTILLER with PPA-inferred bindings and then performs additional processing to detect non-essential differences. We chose to perform AST-differencing as opposed to, for example, token- or graph-based differencing, to avoid the conceptual challenges of working with two distinct program representations. We chose to specifically reuse CHANGEDISTILLER because, although other AST-differencing tools also exist (e.g., DIFFTS [17]), CHANGEDISTILLER is well-documented and also used by other software engineering researchers [25].

Finally, we have integrated DIFFCAT within SEMDIFF [8], a change analysis framework that runs within the Eclipse IDE. SEMDIFF retrieves change sets from CVS and SVN repositories, performs partial program analysis using PPA, and provides hooks that enable third-party diffing tools to identify and persist fine-grained differences between the PPA-enhanced files within each change set.

Two Challenges

To implement DIFFCAT, we were forced to address two important challenges. The first challenge arose due to a mismatch in the input/output representations of two of our reused components PPA and CHANGEDISTILLER. PPA produces element-resolved JDT-Core based ASTs and we need to run CHANGEDISTILLER on pairs of these ASTs to identify differences between them. However, we found that CHANGEDISTILLER *i)* only operates on Java files (read from an Eclipse project) and *ii)* expresses detected differences between those files using a custom output format, i.e., not in terms of AST nodes. Because we found it too difficult to modify CHANGEDISTILLER's code base, we were forced to write an adaptor to map between PPA's and CHANGEDISTILLER's input/output formats. The implementation of this adaptor was non-trivial.

The second challenge arose because CHANGEDISTILLER does not characterize program updates at granularities finer than individual program statements. Instead, it relies on measures of *textual similarity* between statement versions to detect cases where a statement

was modified, rather than inserted or deleted [11]. This means that, given a high enough textual disparity between statements, CHANGEDISTILLER flags unmatched statement pairs as deletions/insertions, rather than updates. For example, depending on its specific input similarity thresholds, CHANGEDISTILLER, might identify the following statement pair as arising due to a deletion and an insertion, rather than an update:

```
this.old = val; //v1        newValue = arg; //v2
```

Although high textual disparity between candidate statement pairs is generally a good indication that the pair corresponds to an insertion-deletion pair and not a modified statement, in some cases, high textual disparity between versions of modified statement can also arise because of *non-essential differences*, e.g., rename refactorings involving textually dissimilar names. For example, if a developer renames a field called `old` to `newValue` and a local variable called `val` to `arg`, then the statement pair above would actually correspond to three non-essential statement updates (a trivial `this` keyword deletion and two rename-induced updates), instead of a statement deletion/insertion pair. Given that all of the non-essential differences outlined in our catalog occur within *modified* statements, we were required to address this challenge to avoid mislabeling a potentially large amount of non-essential differences.

ChangeDistiller Wrapper

To address the input/output mismatch between PPA and CHANGEDISTILLER, we implemented a wrapper that takes as input any two resolved AST nodes (called the old and new ASTs) and returns as output an enhanced representation of all CHANGEDISTILLER-inferred changes (diffs) between them. Each diff contains several attributes:

1. The old AST node affected by the change, which we call the `leftNode`.
2. The new AST node affected by the change, which we call the `rightNode`.
3. A textual descriptor that summarizes the change, which we call the `changeType`. Our descriptors extend those presented in Fluri and Gall's previous catalog of fine-grain changes [10].

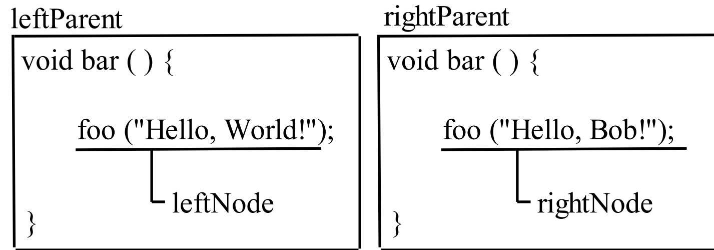

Figure 2.3: DiffCat Output Format

4. The old/new AST nodes that constitute the logical parents of the old/new AST nodes, which we call the `leftParent` and `rightParent`, respectively.

5. Additional intermediate information to facilitate later processing

We illustrate these attributes with the help of Figure 2.3. The figure depicts the old and new version of a hypothetical `bar` method, the body of which was modified by an update to an invocation of a hypothetical `foo` method. The `changeType` for this change is a "method invocation update," the `leftNode` and `rightNode` are the AST nodes representing the old and new invocations of `foo`, and the `leftParent` and `rightParent` are the AST nodes representing the old and new `bar` method declarations. In contrast, CHANGEDISTILLER would represent the change using only textual descriptors for the various AST nodes corresponding to the old and new `foo` invocations and the two `bar` parent methods.

Figure 2.4 presents a conceptual overview of our wrapper. The wrapper is coordinated by a `CDWrapper` component, which coordinates the analyses of several helper components. After receiving as input the old and new ASTs, `CDWrapper` uses its `ASTFilePrinter` to convert the ASTs into two Eclipse-based Java files. Those files are then fed to our reused `ChangeDistiller` component, which outputs the changes (diffs) between them. `CDWrapper` then uses its `DiffMapper` component to convert these diffs into those outlined above. Finally, it uses a `SupplDiffFinder` to find additional diffs not found by CHANGEDISTILLER. For example, the `SupplDiffFinder` detects updates to a method declaration's thrown exceptions.

The `DiffMapper` component takes as input the diffs δ_i detected by `ChangeDistiller`, as well as the old and new ASTs, and maps each δ_i to the actual pair of AST nodes

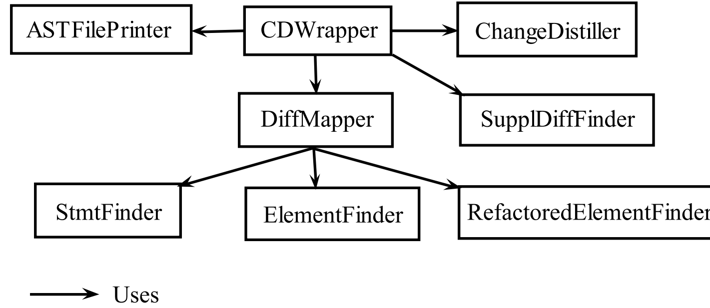

Figure 2.4: Overview of the ChangeDistiller Wrapper

it describes. This process consists of traversing the resolved ASTs and searching for the exact AST nodes that match the textual descriptors associated with each δ_i. Finding the exact AST nodes based on `ChangeDistiller`'s textual descriptors is not always straightforward because the δ_i sometimes contained *synthetic* method signature descriptors, for which no equivalent AST nodes can be found in the source code. Such discrepancies can arise whenever a method signature was affected by multiple signature refactorings. In these cases, `ChangeDistiller` uses synthetic signature descriptors to separately represent the old version of the method for each individual refactoring in isolation, rather than the actual method signature to which the refactoring was applied. For example, suppose an old method signature `method(Type1)` was refactored into a new method signature `method(Type3,Type2)` by replacing `Type1` with `Type3` and by inserting `Type2`. In this case, `ChangeDistiller` might then refer to a synthetic signature `method(Type1,Type2)` to express the starting signature on which the update from `Type1` to `Type3` was applied. However, given that the insertion of `Type2` was also applied, no such signature can actually be found in the old version of the code. Consequently, the use of by `ChangeDistiller` of such synthetic signatures made it difficult to map individual δ_i to the actual methods in which they were discovered.

To deal with this problem, our `DiffMapper` maps diffs in two steps. In the first, it collects all refactoring diffs δ_i that yield the same element signature as their final product. For example, we collect all method signature refactorings δ_i that `ChangeDistiller` associates with the final `method(Type3,Type2)` refactoring. For each such signature e_i

and its diffs δ_i, we then use a `RefactoredElementFinder` to visit all the elements e_j declared in the old AST and check whether applying *all* the δ_i to e_j yields e_i. We then use these discovered pairings (e_j, e_i) in a second phase to map the individual δ_i to their respective AST nodes. This is achieved using a number of AST visitors, e.g., a `StmtVisitor`, that look for specific nodes corresponding to δ_i when traversing an AST.

Detecting Statement Updates

We observed that the difficulties in detecting some statement updates sometimes arises because of rename refactorings. Rename refactorings can increase both the textual disparity between individual program statements and the general difficulty of operating on AST-based representations of code change. For example, discovering the non-essential statement update outlined in Section 2.3 is far more difficult than discovering the same update minus the effects of rename refactorings:

```
this.old = val; //v1        old = val; //v2
```

This latter update exhibits a higher degree of textual similarity, making it easier to identify it as a statement update in the first place. Furthermore, it only textually differs because of `this` keyword deletions, which, in our setting, makes it easier to detect and verify the non-essentiality of the statement update.

Our overall technique for detecting non-essential differences is based on the realization that the effects of rename refactorings should be eliminated when differencing source files. We thus use a *two-phase* tree-differencing technique to identify fine-grained modifications between source files and to label those that are non-essential. In the first phase, we use CHANGEDISTILLER and our own analyses to detect rename refactorings. We then *roll back* those renames in the files we analyze by resetting the textual descriptors of all renamed-affected program references to display their old names. We then re-run CHANGEDISTILLER on the modified files and further process the detected updates to identify those that were affected only by the non-essential differences outlined in our catalog.

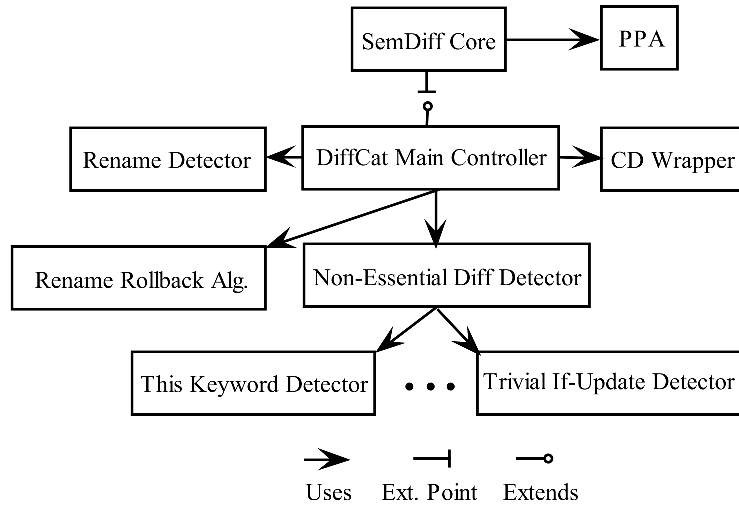

Figure 2.5: DiffCat Implementation

Implementation

Figure 2.5 presents a conceptual overview of our DIFFCAT implementation. For a given change set, DIFFCAT's `MainController` first receives the old and new resolved ASTs from SEMDIFF. It then then uses CHANGEDISTILLER (via a `CDWrapper`) to detect structural changes between the input ASTs; it stores these changes in a special set, called `firstRoundDiffs`. A `RenameDetector` then processes the `firstRoundDiffs` to collect those that represent rename refactorings and to detect additional rename refactorings not detected by CHANGEDISTILLER. The `MainController` then rolls back those rename refactorings by traversing the new ASTs and modifying the textual descriptors of all element expressions found to reference a renamed element. Next, the `MainController` again uses CHANGEDISTILLER, this time to detect a second round of refined structural changes between the old and new ASTs, which are stored in a second set, called `secondRoundDiffs`. The `firstRoundDiffs` and `secondRoundDiffs` are then *reconciled*, i.e., those rename-related differences present in `firstRoundDiffs` but not in `secondRoundDiffs` (because of the rename rollback) are merged into `secondRoundDiffs` and

tagged as either rename refactorings or rename-induced differences. Finally, the `secondRoundDiffs` are sent through a series of detectors, each of which flags diffs that correspond to the specific kinds of changes that we consider to be non-essential differences, e.g., trivial keyword modifications, trivial if-statement updates, etc. Whenever such a non-essential difference is identified by a detector, the detector modifies the diff's `changeType` property to reflect the specific kind of non-essential difference it embodies. For example, the trivial keyword detector might update the "method invocation update" `changeType` detected by CHANGEDISTILLER.

Detecting Class Renames

CHANGEDISTILLER does not identify class renames. DIFFCAT detects these by detecting class insert-delete pairs sharing a high proportion of identical field and method signatures (≥ 0.5). We chose this threshold because we found it to work well during prototyping.

Detecting Field Renames

CHANGEDISTILLER detects field renames by comparing their declaration statements using the Levenshtein similarity measure [31]. In certain cases, CHANGEDISTILLER is unable to recognize a renamed field because of a high textual disparity between its declaration pairs. We try to augment the number of detected field renames by iterating over all possible field insert-delete pairs within each class and checking whether references to the old field were always replaced by references to the new field. We check this condition in all statement updates stored in `firstRoundDiffs`. Our analysis rejects a field insert-delete candidate if even a single statement update does not satisfy our criterion.

Rename Reconciliation

The reconciliation of `firstRoundDiffs` and `secondRoundDiffs` is necessary to properly identify rename-induced non-essential differences that were eliminated by the rename rollback. For example, the rename-induced statement update

```
old = val; //v1        newValue = val; //v2
```

will be detected by our first CHANGEDISTILLER pass because of the textual disparity between the `old` and `newValue` entity. However, after rename rollback, the two statements will be textually equivalent and the update will no longer be detected by CHANGEDISTILLER in our second pass. To cope with this, we collect all statement-based structural differences from `firstRoundDiffs` and verify whether these are again present in `secondRoundDiffs`. If a change was no longer detected in the second phase, we conclude that the change was rename-induced and add it to our list of detected changes. Without this additional step, our procedure would miss these updates.

Detecting Trivial `if`-Statement Updates

We currently detect only a subset of all possible equivalent `if`-statements. Specifically, we only process `if`-statements that involve combinations of `+`, `*`, `=`, `!=`, `||`, and `&&` operators, and for these, we mainly look for updates involving pairwise expression re-orderings (e.g., `a==b` is equivalent to `b==a`) or certain patterns (e.g., `!x` is equivalent to `x==false`). This means that we cannot detect complex re-orderings such as, e.g., cases where `a==b==c` is equivalent to `c==a==b`, or cases involving numerical equivalence, e.g., cases where `1+1==2` is equivalent to `2==2`. Our analysis also conservatively skips expressions involving both object dereferences and either a `||` or `&&` operator, because we cannot safely determine how those dereferences behave at runtime. For example, although `x!=null && x.foo()` might always be equivalent to `x.foo() && x!=null` at runtime, our analysis cannot safely detect the equivalence using only a static analysis.[4] To detect more complicated `if`-statement updates, future implementations of DIFFCAT could use semantic clone detection tools such as H. Kim et al.'s MECC [24].

2.4 Viewing Detected Changes

As part of our overall implementation of DIFFCAT, we have also implemented an Eclipse-based viewing plugin that can be used to conveniently view changes detected by DIFFCAT,

[4] Due to the order in which the Java runtime evaluates `if`-statement conditions, if `x` is indeed `null`, the second `if`-statement will crash the program, but the first will not. We thus cannot assume that the second is equivalent to the first.

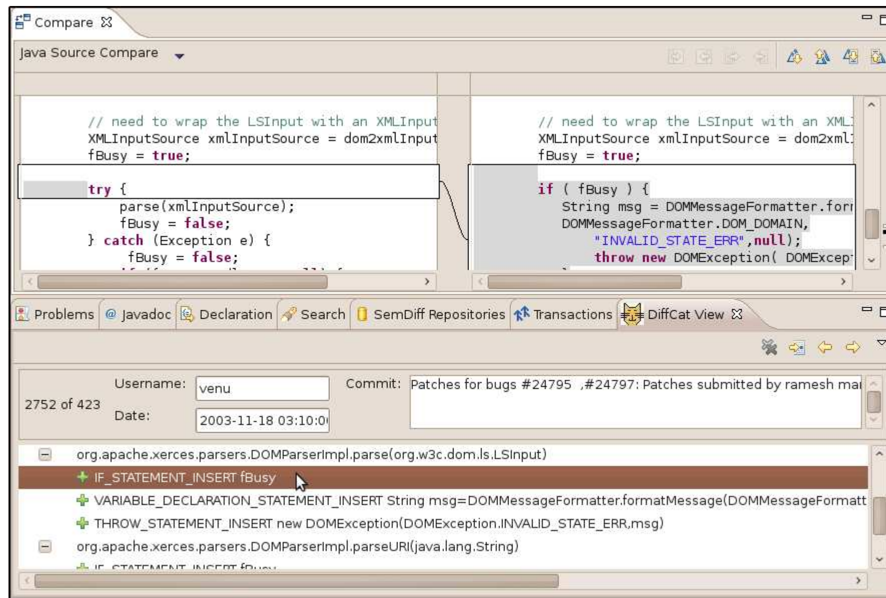

Figure 2.6: Viewing Detected Changes

to associate user-defined keywords with groups of changes, filter out changes based on their keywords, and remove various kinds of non-essential changes from the view using pre-defined filters. The viewer, which we show in Figure 2.6, was implemented to have the same look and feel as SEMDIFF's transaction viewer.

After using SEMDIFF to select a given repository and to run DIFFCAT on a change set from that repository, users can use our viewer to examine the changes that were detected by DIFFCAT within the change set. The viewer automatically groups changes based on the elements they modified. For example, in Figure 2.6, our viewer displays changes that modified the `parse` and `parseURI` methods, respectively. Each change is textually represented by *i)* a textual marker indicating the kind of change it embodies and *ii)* a textual summary of the code snippet(s) associated with that change. For example, in the Figure, the user has highlighted a change corresponding to the insertion of the `if`-statement involving an `fBusy` variable. Double-clicking on a given change automatically opens up an Eclipse comparison view that shows the old and new files affected by the change and the approximate location of the change itself. For example, in the figure, the user has just double-clicked on a change and that change is displayed in the comparison view above. To display changes DIFFCAT

reuses the Eclipse comparison view, which only displays the textual changes between two files; it then approximates the location of a given change by finding the diff-region within the comparison view that most closely correspond to the more fine-grained changes detected by DIFFCAT. For example, in the figure, the highlighted code region in the Eclipse comparison view captures the selected `if`-statement insert, as well as all the other changes within the `parse` method. In the future, we could improve the granularity of the Eclipse comparison view by directly incorporating changes detected by DIFFCAT.

Our viewer provides several other services. Besides from allowing users to view individual changes, it also allows them to assign text keywords to each change and to filter changes based on their keywords. It also allows users to automatically filter out all non-essential differences from the view, thus reducing the cognitive burden of abstracting over those changes when inspecting change sets. Although simple in nature, our viewer greatly facilitated the numerous amounts of manual inspections we performed as part of this thesis. For a download of our viewer, the reader is refered to DIFFCAT's main webpage.

2.5 Empirical Study

We sought to understand the potential impact of non-essential differences on higher-level abstractions of software development effort. To this end, we used DIFFCAT to analyze change sets from ten open-source Java systems and to collect essential and non-essential differences between committed file-pairs. We then determined *i)* the relative code churn associated with non-essential differences and *ii)* how often change sets include methods that were modified only via non-essential differences. We used our results to estimate how non-essential differences would interfere with the information measured by change-based approaches. We have released all our data and a full description of our setup on our website: http://www.cs.mcgill.ca/˜dkawry/thesis.

2.5.1 Set up

Table 2.1 describes the systems used for our evaluation. Columns in the table include the number of change sets studied for each system (Chg. Sets) and the number of days spanned

Table 2.1: Characteristics of Target Systems

System	**First**	**Last**	**Days**	**Chg. Sets**
Ant	6 Dec 2001	17 Jul 2007	2,048	3,853
Azureus	12 Nov 2003	14 Jul 2004	244	3,103
Hibernate	4 Dec 2003	19 Aug 2005	623	3 922
JDT-Core	17 Jan 2002	15 Jul 2003	544	4 192
JDT-UI	20 Aug 2001	15 May 2002	268	3 081
JEdit	11 Feb 2001	10 Jun 2011	3 792	3 143
Nutch	2 Jun 2005	23 May 2011	2 182	678
Spring	1 Feb 2004	6 Feb 2006	736	3 627
Struts	16 Jul 2000	16 Sep 2009	3 350	2 370
Xerces	17 May 2001	8 Nov 2007	2 366	2 681
Total			9 813	30 650

by those change sets (Days). Seven of the systems we studied were previously investigated by Robillard and Dagenais [41]; for these systems, we selected the same time frames as those used by Robillard and Dagenais. The other three systems (JEDIT, NUTCH, STRUTS) are all well-known and widely used open-source projects; for these, we considered all available change sets, starting with the 50th. We did not consider these systems' initial change sets to avoid processing the very large changes that may be committed as part of a system's initial introduction to its version control system; however, our specific selection of the 50th change set was otherwise completely arbitrary. For each system, we studied all change sets that occur within the ranges reported in Table 2.1. In very few cases ($74 < 0.0025\%$), we aborted processing change sets because we could not retrieve the files for those change sets from their repositories or because PPA did not appear to terminate on the retrieved files.

We used DIFFCAT to determine the differences within change sets. Like other differencing tools, DIFFCAT does not report any differences arising from white spaces. We also ignored all differences affecting comments and Javadocs, i.e., *we did not consider whitespace-, documentation-, or comment-based differences in any of our results*. We used

the remaining differences to compute each change set's total *code churn* (modified LOC) and to identify the methods that were modified by each change set. We then identified all non-essential differences to compute non-essential code churn and to identify which methods were modified only by non-essential modifications.

We computed code churn by considering the LOCs involved in each reported structural difference. Our code churn measure thus differs slightly from that which would be computed by purely line-based differencing techniques. For example, because of our rename rollback, DIFFCAT may identify that a LOC was updated, while other differencing techniques may report this difference as a LOC insertion-deletion pair. We chose to use DIFFCAT to compute churn to obtain the most precise estimate of the true churn arising from non-essential differences.

A change set was considered to modify a method if it updated the body of that method via one or more structural differences (i.e., we never considered documentation-related differences as updates to a method). For simplicity, we refer to the number of methods found to have been updated by a change set as the number of *method updates* for that change set. We labeled a method update as *non-essential* if the method update consisted only of non-essential differences. All other method updates were considered *essential*. The total number of method updates for a system corresponds to the total number of method updates found across all change sets.

We explicitly tracked method signature refactorings throughout our evaluation, i.e., we did not treat methods modified by such refactorings as method insertion-deletion pairs. If a method's signature and body were both updated by a change set, then we included the refactored method within the set of method updates for that change set. If only the method's signature was updated, then we did not include the method within the method update count for that change set. We did not include method deletions or insertions within the method update count because our investigation focused on the modified methods for each change set.

We ran CHANGEDISTILLER on its default settings. We set PPA to compile change sets using Java 1.5.

2.5.2 Prevalence of Non-Essential Differences

Table 2.2 records the overall code churn for each target system (in kLOC). The table shows the total number of code lines that were *modified* for each system (Modified). It shows how many of the modified lines were caused by four major classes of non-essential differences detected by our approach: differences induced by renames (R-Induced), trivial keyword updates (Key), local variable refactorings (Local), and other kinds of non-essential differences (Other). The "Local" column aggregates local variable extractions, local variable renames, and trivial updates to local variable declared types and initializers. The "Other" column aggregates trivial string splits, redundant class casts, trivial `if`-statement updates, and trivial updates to declared and return types of methods and fields. The combined non-essential line modifications are reported in the final column (Non Ess). The percentages displayed in this column correspond to the proportion of all modified code lines (Modified) that were found to be non-essential. We did not display the total deleted or inserted lines for each system because we found those measures remained virtually unaltered after we removed the kinds of non-essential differences that can affect them. In other words, insertions and deletions of trivial constructors, trivial `super()` invocations , and trivial `return` keywords had virtually no impact on the total deleted or inserted lines for each system.

Table 2.2 helped us derive the following observation:

Between 2.8% and 25.8% of modified code lines were updated only via non-essential differences.

This suggests that for some systems, non-essential differences can significantly increase line-modification-based abstractions of change.

From the table, we also see that across the target systems, a combined 266 kLOC were modified. We see that 26.3 (10%) of the total 266 modified kLOC were modified only by non-essential differences. However, based on a previous definition of *total code churn* [34], which combines modified and inserted code lines into a single measure, we find that the 26.3 non-essential kLOC constitute only a small fraction (<2%) of the overall *churned* code ($\approx$ 1.56 mLOC) across the ten systems. This observation suggests that the kinds of

Table 2.2: Code Churn in Target Systems (in kLOC)

System	Modified	R-Induced	Key	Local	Other	Non Ess
Ant	32.2	6.7	.8	.5	.3	8.3 (25.8%)
Azureus	95.1	2.5	.1	.1	0	2.7 (2.8%)
Hibernate	27.6	2.8	.1	.2	.1	3.2 (11.6%)
JDT-Core	15.3	1.8	.4	.3	.1	2.5 (16.3%)
JDT-UI	16.3	1.3	0	.1	0	1.5 (9.2%)
JEdit	21.4	.7	.2	.1	.1	1.1 (5.1%)
Nutch	10.0	.3	0	0	0	0.4 (4.0%)
Spring	23.2	3.8	.5	.2	.2	4.7 (20.3%)
Struts	10.0	.6	0	.2	0	0.8 (8.0%)
Xerces	14.9	1.0	0	.1	0	1.1 (7.3%)
Total	266.0	21.5	2.1	1.8	.8	26.3 (9.9%)

non-essential differences studied in our investigation do not affect measures of *total code churn* (that include added and modified lines).

Table 2.2 also enabled us to infer the following property:

> *Out of the non-essential differences currently detected by our approach most non-essential modifications were induced by rename refactorings or updates involving trivial* `this` *keywords.*

In particular, of the 26.3 non-essential kLOC reported in the table, approximately 82% consisted of rename-induced statement updates, 8% consisted of trivial updates involving `this` keywords, 5% consisted of local variable renames, and 5% involved the remaining kinds of non-essential differences.

Table 2.3: Method Updates in Target Systems

	Total	Non-Essential	R-Induced	Keyword	Local	Other
Ant	17 793	2 870 (16.1%)	2 196	542	141	131
Azureus	8 786	257 (2.9%)	223	32	6	2
Hibernate	15 975	1 189 (7.4%)	1 123	46	51	15
JDT-Core	8 867	622 (7.0%)	504	113	92	26
JDT-UI	9 690	443 (4.6%)	414	28	13	2
JEdit	13 803	238 (1.7%)	125	50	16	56
Nutch	3 570	100 (2.8%)	83	18	5	3
Spring	11 046	1 789 (16.2%)	1 497	257	73	60
Struts	5 515	228 (4.1%)	128	26	80	24
Xerces	8 409	247 (2.9%)	226	17	5	4
Total	103 454	7 983 (7.7%)	6 519	1 129	482	323

Non-Essential Method Updates

Table 2.3 records the total number of method updates that were detected for each target system. The table shows the total number of method updates (Total) and the number of those updates that were induced entirely by non-essential differences (Non-Essential). It also records how often different classes of non-essential differences contributed to a non-essential method update. We recorded this number for rename-induced updates (R-Induced), keyword updates (Keyword), updates to local variable (Local), and various other non-essential differences, such as redundant updates involving `if`-statements, string splits, and class casts (Other). We note that the sum across the individual columns is higher than the total number of non-essential updates because some non-essential method updates involved multiple classes of non-essential differences.

From the table, we see that out of a combined 103 454 method updates across the ten systems, 7 983 (7.7%) were non-essential. The table also enables us to make the following observation:

Table 2.4: Non-Essential Methods in Change Sets

System	Total	Non-Essential	R-Induced	Keyword	Local	Other
Ant	2 580	319 (12.4%)	251	89	47	63
Azureus	2 870	56 (2.0%)	56	24	5	2
Hibernate	3 059	325 (10.6%)	286	39	39	6
JDT-Core	2 017	158 (7.8%)	134	29	41	12
JDT-UI	2 155	157 (7.3%)	137	25	12	1
JEdit	2 619	136 (5.2%)	65	50	13	21
Nutch	506	28 (5.5%)	26	4	3	2
Spring	2 401	487 (20.3%)	391	117	61	47
Struts	1 509	82 (5.4%)	50	21	22	9
Xerces	2 038	75 (3.7%)	60	13	5	3
Total	21 754	1 823 (8.4%)	1 456	411	248	166

In the individual systems analyzed, between 1.7% and 16.2% of all method updates were non-essential.

This suggests that for some systems, non-essential differences can distort method-based abstractions of *change span*.

Distribution of Non-Essential Method Updates

Table 2.4 shows how many of the analyzed change sets included non-essential method updates. The table records the total number of change sets that included modifications to at least one method (Total). The remaining columns record the number of change sets found to include at least one non-essential method update (Non-Essential), one non-essential method update featuring a rename-induced difference (R-Induced), a keyword difference (Keyword), a local variable refactoring (Local), or an update involving a redundant class cast, trivial `if`-statement, or trivial string split (Other).

From the table, we see that out of 21 754 change sets found to modify at least one

method, 1 823 (8.4%) included at least one non-essential method update. The table also enabled us to make the following observation:

In some systems, non-essential differences distorted method-level change representations of over 10% of change sets.

This suggests that non-essential differences can impact method-level representations of a non-negligible number of change sets.

We next observed that method updates in smaller change sets were less likely to be non-essential than method updates in larger change sets. For example, we found that only (723/23302 ≈) 3.1% of method updates within "small" change sets (e.g., those 15 327 change sets modifying 1 to 3 methods) were found to be non-essential. This ratio increases to (3028/43102 ≈) 7.0% for "regular" change sets (e.g., those 5 605 change sets modifying 4 to 19 methods) and (4232/37125 ≈) 11.4% in "large" change sets (e.g., those 826 change sets modifying 20 or more methods). We observed similar proportions for other ranges. This data enabled us to draw the following conclusion:

Non-essential differences had the highest impact on method level representations of larger change sets.

This observation is important because it means that change-based approaches could both eliminate a majority of non-essential method updates and mitigate their most significant relative impact on method-level representations by using alternate differencing strategies for larger change sets. For example, we found that aside from featuring relatively high densities of non-essential method updates (11.4%), change sets modifying 20 or more methods also contained an overall majority (53%) of *all* detected non-essential method updates. Change-based approaches could exploit this general observation when scanning change sets by first using a lightweight differencing technique to compute a change set's method level *change span* and then switching to a more sophisticated differencing technique only in cases where the measured *change span* exceeds a certain threshold, e.g., 20. This kind of strategy is advantageous because larger change sets tend to appear relatively infrequently

in change history (e.g., in the data we analyzed, only 2.7% of all change sets modify 20 or more methods), which means change-based approaches could avoid the computational burden of partial program analysis in most cases, while still detecting a relevant proportion of non-essential method updates within change sets.

Finally, we observed that non-essential method updates were *interleaved* with other essential method updates in most (79%) change sets. This result corroborates findings of a previous investigation by Murphy-Hill et al., which showed that developers often interleaved refactorings with other modifications [33]. These observations suggest that in cases of interleaved changes, a fine-grained detection of non-essential differences can help change-based approaches obtain precise representations of the meaningful software development work behind a change (as opposed to capturing the effects of tool-assisted refactorings or trivial keyword updates).

2.5.3 Impact on Association Rules

To help us further asses the possible impact of non-essential differences on the results of existing change-based approaches, we implemented a simple method-pair association rule mining analysis similar to that of Zimmermann et al. [46] and evaluated how the *quality* of the recommendations produced by our analysis was affected by the kinds of method updates used to train the analysis. Specifically, we sought to compare the quality of the recommendations produced when all method updates were used to learn rules against their quality when only essential updates were used.

Our analysis takes as input a given sequence of change sets extracted from a system's change history, records the methods that were modified as part of each change set, and then uses this information to produce recommendations for a developer. Specifically, similar to Zimmermann et al.'s ROSE tool [46], our analysis supports developers who have modified some initial method m_i as part of some change set t_k and who would like to find additional methods m_j that also need to be changed along with method m_i. Our analysis helps developers by inferring association rules $(m_i \rightarrow m_j)$, from which we can return a ranked list of methods m_j that were found to have been frequently co-modified with m_i in prior change sets $H_k := t_0, \ldots, t_{k-1}$. We rank recommendations (m_j) for a change set t_k based on the

confidence of the inferred rule $(m_i \rightarrow m_j)$. We use their *support* values as tie breakers. Finally, like Zimmermann et al., we also filter out recommendations with confidence lower than 0.1 and cap the number of recommendations at ten [46].

To compare the quality of the recommendations produced by our analysis when trained using all methods (the regular setup) against their quality when we train it only on essential methods (our proposed setup), we compared several metrics used by Zimmermann et al. in their evaluation [46]. To compute these metrics, we replayed the change history intervals of the ten target systems (see Table 2.1) and determined which ranked recommendations m_j our analysis would have made for method updates m_i in t_k w.r.t. rules learned up until then from H_k.[5] We then recorded whether m_j was also updated as part of t_k and used this to tag each ranked seed-recommendation pair (m_i, m_j) in t_k as either "helpful" or not. Overall, this produced two sets of *non-empty* ranked recommendation lists for 39 246 different seed methods. We found that the recommendations were different in 12 208 (31.1%) cases. We then compared the quality of the recommendations for these 12 208 cases.

Our metrics allowed us to to make the following observation:

> *In those cases where a removal of non-essential method updates affected the quality of a seed method's recommendations, the overall precision of the recommendations increased by 10% and their recall decreased by 3%.*

Table 2.5 presents this observation in more detail. For the 12 208 cases considered for each setup, the table records the total number of recommendations made by our approach (Tot Rec), the number of method changes for which at least one recommendation was made (Feedback), and the proportion of recommendations that were found to have been helpful (Prec). It also records the proportion of changed methods for which at least one helpful recommendation was found in the top three recommendations (L3) and the proportion for which no helpful recommendations were made (Only Err).

From the table, we see that the precision of the approach improved by (.253/.230 ≈)

[5] We only considered *essential* method updates as candidate seeds to eliminate all spurious methods that were only indirectly modified via one or more rename refactorings, and hence not legitimate candidate seeds for our experiment.

Table 2.5: Recommendation Quality

Setup	Tot Rec	Feedback	Prec	L3	Only Err
Reg	111 712	12 200	0.230	0.453	0.215
Prop	97 950	11 109	0.253	0.483	0.178

10% and its total number of helpful recommendations decreased from (.230*111712 =) 25 661 to (.253*97950 =) 24 882, or by around 3%. We also see that the proportion of changed methods for which at least one helpful recommendation was found in the top three recommendations increased by (.483/.453 ≈) 6.6% and that the proportion for which only erroneous recommendations were made decreased by (.215/.178 ≈) 20.8%. Hence, given this general reduction in the number of false positives produced by our approach, and despite the slight loss in recall, we argue that the overall quality of the recommendations produced by our association analysis was improved after we removed non-essential method updates from consideration.

2.5.4 Impact on Bug-Fixing Change Sets

Some change-based approaches mine the individual changes within bug-fixing change sets to identify *fix-inducing* (i.e., bug-introducing) changes [42]. Fix-inducing changes can then be used to identify, for example, faulty components or future faulty changes. To further assess the possible impact of non-essential differences on change-based approaches, we measured how often bug-fixing change sets contain rename-induced and other non-essential changes. Such changes are less likely to embody the actual bug-fixing activity of bug-fixing change sets, and may thus result in inaccuracies when determining fix-inducing changes.

Change-based approaches can detect two kinds of bug-fixing change sets – those fixing a specific bug (referenced via bug id) and those that perform a general fix with or without specifying a bug. To detect the former, we use common heuristics similar to those used by previous approaches [42]. Our heuristics parse the lower case version of the commit message for each change set to identify those messages containing bug-related substrings (“bug,” “fix,” and “patch”) or some other system-specific bug marker (“nutch,” “pr,” “spr,”

Table 2.6: Bug-Fixing Change Sets

System	Bug ID	Bug ID +Nemu	Fix	Fix +Nemu	No Fix	No Fix +Nemu
Ant	780	21 (3%)	1 189	52 (4%)	2 664	267 (10%)
Azureus	27	0 (0%)	509	8 (2%)	2 594	48 (2%)
Hibernate	690	42 (6%)	1 137	82 (7%)	2 785	243 (9%)
JDT-Core	585	10 (2%)	665	13 (2%)	3 527	145 (4%)
JDT-UI	348	9 (3%)	597	21 (4%)	2 484	136 (5%)
JEdit	391	10 (3%)	1 303	41 (3%)	1 840	95 (5%)
Nutch	328	13 (4%)	420	20 (5%)	258	8 (3%)
Spring	47	2 (4%)	331	24 (7%)	3 296	463 (14%)
Struts	516	9 (2%)	798	22 (3%)	1 572	60 (4%)
Xerces	565	11 (2%)	1 136	31 (3%)	1 545	44 (3%)
Total	4 277	127 (3.0%)	8 085	314 (3.9%)	22 565	1 509 (6.7%)

"hb," and "hhh,"), followed by any amount of arbitrary text that includes at least one digit, which we then assume to be a bug id. To detect more general "fixes," we take the set of all change sets detected by the above heuristic and add to it any change sets with commit messages containing a fix-related substring ("fix", "bug," "defect," "repair," and "patch"). Finally, we record which of the matched change sets also contain at least one non-essential method update.

Table 2.6 records the results of this process. The table shows the number of detected change sets fixing a specific bug (Bug ID) and how many of those change sets also featured at least one non-essential method update (Bug ID +Nemu). It also displays the number of detected change sets featuring a general fix (Fix) and how many of those contained at least one non-essential method update (Fix +Nemu). Finally, the table displays those change sets that were not identified as containing a fix (No Fix) and how many of those contained at least one non-essential method update (No Fix +Nemu). We point out that, for each system, all change sets fixing a specific bug (Bug ID) are also counted as change sets featuring a

general fix (Fix). The number of change sets featuring no fix (No Fix) was derived by subtracting the number of change sets featuring a fix from the *total* number of change sets scanned for that system, i.e., the number reported in Table 2.1.

From the table, we see that, of those change sets referencing a specific bug id, 3.0% also contain at least one non-essential method update. This ratio increases slightly to 3.9% when we consider all bug-fixing change sets. In contrast, non-essential method updates appear within 6.7% of regular change sets, or approximately (6.7/3.9 $\approx$) 70% more frequently. This suggests that non-essential method updates are less likely to yield inaccurate method-level representations of bug-fixing change sets than they are for regular change sets. In fact, the data allows us to make the following observation:

Non-essential method updates are less likely to appear in bug-fixing change sets than in regular change sets.

A Chi-square test reveals the above relationship to be statistically significant with $p < 10^{-4}$.

To validate the precision of our detection of bug-fixing change sets, we selected 200 random change sets that our heuristics identified as containing a bug id and 200 random change sets that our heuristics labeled as containing a general fix. We then manually investigated the commit comments for these change sets and found that our heuristics correctly classified (181/200 $\approx$) 90% of all change sets in the former category and (190/200=) 95% of change sets in the latter category. We thus conclude that the above observation is grounded in a reasonably precise estimate of the overall bug-fixing change sets within the systems we analyzed.

2.5.5 Precision of the Detection Technique

We performed a manual inspection to verify the precision with which DIFFCAT identified rename refactorings and non-essential method updates. We focused on rename refactorings and non-essential method updates, rather than all reported non-essential differences, because erroneous classifications of rename refactorings and method updates are more likely to have a negative influence on the representations used by change-based approaches than

Table 2.7: Characteristics of Selected Change Sets

System	**CS**	**NEMUs**	**CR**	**MR**	**FR**	**PR**	**VR**
Ant	23	1 154	0	4	675	203	91
Azureus	4	115	2	28	3	2	31
Hibernate	38	609	30	169	62	106	34
JDT-Core	22	333	2	17	52	136	74
JDT-UI	20	201	27	48	15	40	29
JEdit	12	64	3	29	16	33	12
Nutch	4	42	1	21	9	17	9
Spring	45	773	43	240	123	452	38
Struts	9	77	1	8	19	4	9
Xerces	8	110	13	98	17	19	1
Total	185	3 478	122	662	991	807	328

erroneous classifications of isolated statement updates.

To select change sets for a given system, we selected a cutoff n and removed all change sets featuring fewer than n rename-induced method updates. We selected n so that the remaining change sets accounted for approximately *half* of all the rename-induced method updates for that system. In this way, we investigated approximately half of all rename-induced method updates reported in Table 2.3, while limiting our inspection to just 185 change sets across the ten systems.

Table 2.7 records, for each system, the number of change sets studied (CS) and the number of non-essential method updates that were detected by DIFFCAT. It also records the number of class, method, field, parameter, and variable rename refactorings (CR, MR, FR, PR, VR) that were detected by DIFFCAT. We investigated the correctness of these reported refactorings and non-essential method updates. We assessed reported rename refactorings by carefully inspecting all available code, the relative placement of inserted and deleted entities within code, documentation, and the commit comment of each change set. We used our rename classifications to judge the correctness of rename-induced statement updates

Table 2.8: Precision of the Technique (in %)

System	NEMUs	CR	MR	FR	PR	VR
Ant	100	n/a	50	100	100	98
Azureus	100	100	100	100	100	100
Hibernate	99.8	100	93	97	80	91
JDT-Core	100	100	76	98	100	96
JDT-UI	99.5	89	83	87	98	83
JEdit	98.4	100	79	88	85	42
Nutch	100	100	100	100	100	89
Spring	100	100	97	100	99	95
Struts	96.1	100	50	90	50	100
Xerces	98.2	100	96	94	95	100
Total	99.8	98	93	99	97	93

that were detected by DIFFCAT. We used the correctness of rename-induced statement updates and other non-essential differences to judge the correctness of each non-essential method update. Based on our manual investigation, we were able to assert that:

99.8% of all method updates that were classified as non-essential by DIFFCAT *were correctly classified.*

Table 2.8 presents the precision of our detected rename refactorings and non-essential method updates in more detail. The table displays the proportion of correct classifications for each of the results reported in Table 2.7. From the table, we see that the overall precision of our approach for rename detection ranges from 93% (method and variable renames) to 99% (field renames). The table also shows that our approach identified non-essential method updates within individual systems with a precision ranging from 96.1% to 100%.

The precision of non-essential method updates was higher than that of detected rename refactorings because only a small number of all erroneously classified insertion-deletion

pairs actually resulted in erroneously classified statement updates, and only a few of those statement updates were sufficiently isolated within methods to cause an entire method update to be erroneously classified.

2.5.6 Discussion

Non-Essential Differences

The true "essentiality" of modified code lines and updated methods is tied to the specific goals of individual change-based approaches. We believe that accounting for the kinds of non-essential changes detected by our approach will be most useful for change-based approaches that aim to analyze only specific classes of software development effort, such as effort related to feature implementations or bug fixes. The ultimate goal of our research is to enable change-based approaches to more precisely select the low-level modifications on which they base their higher-level change representations.

Our current catalog of non-essential differences did not include a number of additional fine-grained differences that may be considered non-essential in some contexts. For instance, change-based approaches might also be interested in ignoring updates involving trivial `final` keywords in local variable declarations or other updates to code that are less likely to provide meaningful insight into the kind of development work that is of interest to these approaches. Ideally, change-based approaches should be able to parameterize their change representations to include only those code changes that are most relevant for their analyses. Because the types of non-essential differences that can be detected is open, it should be noted that the numbers we report are an underestimate of all the possible non-essential changes that may exist in the histories of the software systems we studied. Moreover, we did not attempt to estimate the recall of our technique. In general, we designed the technique to be precise (i.e., to characterize differences as non-essential only in the presence of strong evidence). For this reason, hard-to-classify differences that may turn out to be non-essential in practice would not have been included in our results, further contributing to our numbers representing a lower-bound estimate of the prevalence of non-essential differences.

Our empirical investigation produced a number of observations about non-essential differences that we believe are relevant to a variety of change-based approaches. For example, we observed that between 1.7% and 16.2% of a system's method updates can be described exclusively in terms of non-essential differences, and that these kinds of method updates interfere with a non-negligible number of frequent pairwise method associations supported by change data. Eliminating non-essential method updates should thus have a positive impact on the results of change-based approaches seeking to detect meaningful associations between non-obvious method pairs. Based on other observations, we also expect non-essential method updates to be most relevant for change-based approaches that do not specifically pre-filter large or modification-intensive change sets from their analyses. We also expect non-essential differences to be most relevant to change-based approaches that do not specifically analyze bug-fixing change sets.

Generalizability of the Results

Our investigation focused on ten open-source Java systems. We expect our observations on non-essential differences to most readily generalize to other systems of similar size and developed using similar development practices as those used by the developers of our studied target systems. Except for AZUREUS and JEDIT, the systems we analyzed are all developed in association with major open-source software distributors (Spring, Apache, JBoss, and Eclipse). The development of AZUREUS is coordinated by a digital media technology company (Vuze) and that of JEDIT by an unaffiliated group of individuals. All analyzed systems included code commits from between 6 to 29 contributors, and 17 on average. The investigated projects all contain in the order of between 100 and 1000 kLOC. Our results may therefore not generalize to projects featuring significantly larger code bases or development teams, or those following more tightly regimented development practices. Systems developed in other programming language may not exhibit similar proportions of non-essential differences as those reported in our investigation.

Our investigation of non-essential differences makes no attempt to characterize the recall achieved by our differencing technique. We thus further qualify our individual observations by noting that our currently detected non-essential differences may represent distorted

representations of the actual proportions of these differences in change history. However, based on extensive manual assessments of the differences detected by our approach, we believe that our overall proportions of non-essential differences are not incorrect. We plan to further investigate the recall of our approach as part of future work.

Chapter 3
Detecting Subtasks

Change-based approaches often process change sets under the assumption that each change set constitutes a conceptually isolated and distinct *task*, such as a feature enhancement, code cleanup, or bug fix. This assumption allows these approaches to derive useful insights about the code elements that were affected by a given change. However, in our manual investigations of change history, we have come across many cases where change sets clearly contain changes related to multiple distinct *subtasks*. For example, as we show in our motivating example in Section 3.2, developers may simultaneously commit distinct patches for several independent bugs as part of the same change set. In other cases, developers may intersperse various code cleanups or other isolated minor modifications along with an unrelated bug fix or feature enhancement. In general, change-based approaches cannot guarantee that a given change set actually contains changes related to exactly one task.

We propose that change-based approaches automatically divide change sets containing changes related to multiple subtasks, so that change sets each contain updates related to precisely one task/subtask. We argue that splitting change sets in this fashion would allow change-based approaches to avoid harvesting accidental relations between co-committed code elements. We hypothesize that removing such accidental associations from inputs considered by various tools, such as, for example, the ROSE tool, could help those tools produce better results.

In the previous chapter, we proposed the concept of non-essential differences, or changes

that are unlikely to capture the kind of software development effort that is most meaningful to change-based approaches. In this chapter, we build on this work by presenting an automated approach that attempts to split change sets into *subtasks* – groups of smaller and conceptually related changes that are unrelated to other changes within a change set. Our approach takes as input the essential changes within a change set, as computed by DIFFCAT, and then automatically groups those changes into non-overlapping sets (subtasks) by identifying basic textual and static relationships between them. As far as we know, our approach is the first automated technique that splits change sets into subtasks.

To evaluate our technique, we constructed a new benchmark comprising over 1 800 change sets drawn from the revision histories of seven, long-lived open-source Java systems. We manually classified each change set as being either single-task or multi-task, and then manually split all multi-task change sets into their constituent subtasks. Running our approach on this benchmark showed that it correctly groups elements into subtasks for approximately 80% of all change sets. Specifically, the approach correctly identifies 84% of all single-task change sets and correctly splits 24% of all multi-task change sets. If we consider that the current "default strategy" simply groups all elements in multi-task change sets into single tasks, then we can also say that for over 90% of all multi-task change sets, the subtasks inferred by our approach are better, or at least no worse, than those produced by the current default stragey. Finally, we manually investigated the results of our approach for those single- and multi-task change sets for which the approach inferred erroneous subtasks. Our investigation revealed that, although our approach incorrectly splits 16% of all change sets, its erroneous results can still be considered reasonably useful for 78% of those change sets.

The contributions of this chapter include *i)* the precise formulation of a hitherto unstudied problem, *ii)* an automated technique that seeks to address that problem, *iii)* a reusable benchmark that can be used to evaluate and compare future techniques in this area, and *iv)* a detailed summary that describes the performance of our technique on the benchmark.

3.1 Definitions and Problem Statement

Given a change set C, we define an *updated element* of C to be any element (field, method, or class) that was either inserted or deleted as part of C, that had its declaration signature modified in any way as part of C, or (in the case of methods) that had its body modified as part of C.

Our goal is to separate the updated elements of C whenever those elements were modified as part of distinct and conceptually isolated subtasks. By default, we assume each change set addresses a single task. We say a change set C performs multiple distinct and conceptually isolated *subtasks* whenever the commit message of C explicitly refers to or names two or more changes that were performed as part of C, and the descriptions of those changes are textually isolated from another. We say that two subtask descriptions are textually isolated from another if they are separated using either a bullet-, comma-, or other token-centric list, line breaks, or other whitespace-related formatting, or if they appear within distinct phrases that are themselves separated by some form of punctuation (commas, full-stops, or semi-colons) and some concrete descriptive transition such as "also did," "also worked on," etc. The basic assumption behind this definition is that if a developer takes the time to split the description of their work into clearly separated sub-sections, then this is an indication that the developer considered the described changes to be conceptually separate.

To illustrate the reasoning behind our definition, we can consider a change set from ANT, in which the developer writes: "javadoc, some refactorings, attempt to delete VMS command file when process completes." In this case, because the developer's description explicitly isolates the refactorings from the VMS deletion activity, we assume that the changes associated with each activity are conceptually separate. In contrast, in another change set from ANT, the developer writes: "Add some preliminary test cases..." Here, because the added test cases are not further distinguished from one another through some kind of textual separation, we assume that their addition constitutes a single task, even if each test case could have been committed separately. Similarly, in a change set from AZUREUS, a developer writes: "fix signature calculation and add some debug." Again, because the two changes are not textually separated, we do not consider them to belong to

different subtasks. In this case, if the developer had written: "fix signature calculation. Also add some debug," we would assume that the textual separation between the two activities is indicative of a stronger conceptual split between them, and hence a corresponding division into separate subtasks.

Our goal is to help change-based approaches operate on each described subtask in isolation. Formally, we seek to develop an automated technique that, given a change set consisting of element updates e_i, associates each e_i with one subtask t_i, such that the description of each t_i is separate in the change set's commit message.

We note that our definition of a subtask is restrictive in several ways. First, aside from missing all subtasks that are not explicitly commented on by the developer, we also miss all cases where a developer uses a single phrase to describe multiple potentially isolated actions (e.g., "implemented/fixed X and Y," "fixed some bugs," "made minor tweaks," "cleaned up the code," "last commit of the day," etc.). Furthermore, we miss cases where a developer outlines only one main task within their change set while also performing several other more minor change tasks alongside the described main task. We prefer these restrictions to avoid possible bias or ambiguities in what we consider to be a "conceptually isolated subtask" and to facilitate the expression of our problem statement above. However, as our qualitative assessment in Section 3.4.3 shows, our proposed approach is able to detect many cases where changes can be considered to be isolated subtasks, even if they are not explicitly outlined in the change set's commit comment.

3.2 Motivating Example

To motivate the problem we seek to address, let us consider a change set[1] from the revision history of XERCES, a Java-based XML-processing toolkit.[2] The change set inserts one method and modifies three others. One of the modified methods is declared within `AbstractDOMParser`; the other three methods are declared within `DOMParserImpl`. The change set's commit comment reveals that the change set commits two separate patches

[1]Committed by author `venu` on 2003-11-18 03:10:00.
[2]http://xerces.apache.org/xerces2-j/

for two unrelated bugs: bug #24795[3] and bug #24797.[4] A manual inspection of the bug reports and the two submitted patches for the two bugs reveals that the two issues are conceptually distinct and their changes not linked in any way. Specifically, the changes for bug #24795 update a single if-statement within the `AbstractDOMParser` method from

```
if (fDOMFilter.getWhatToShow() & NodeFilter.SHOW_TEXT)!=0)
```

to

```
if (child.getNodeType() == Node.TEXT_NODE &&
    (fDOMFilter.getWhatToShow() & NodeFilter.SHOW_TEXT)!=0)
```

to avoid the duplicate traversal of certain child nodes. In contrast, the changes for bug #24797 insert a new method `abort` with body

```
if ( fBusy ) { reset(); fBusy = false;}
```

and two identical code blocks, each of the form

```
if ( fBusy ) {
  String msg =
    DOMMessageFormatter.formatMessage(
      DOMMessageFormatter.DOM_DOMAIN,
      "INVALID_STATE_ERR", null
    );
  throw new DOMException(
    DOMException.INVALID_STATE_ERR, msg);
}
```

Hence, in this case, given that the change set commits changes related to two isolated and conceptually distinct subtasks, we want to separate the two groups of changes to be able to process each in isolation.

[3]https://issues.apache.org/jira/browse/XERCESJ-838
[4]https://issues.apache.org/jira/browse/XERCESJ-839

3.3 Approach

There exist numerous possible avenues for identifying multi-task change sets and for mapping the changes within these change sets to their respective subtasks. For example, an approach might parse the commit message of the change set, identify textual sub-regions within that message, and then map textual keywords associated with the changes of the change to keywords in each sub-region. In other cases, e.g., in the case of our motivating example, a strategy could be to search for the specific patches associated with a commit and disambiguate the commit's changes by mapping them back to a specific patch. However, we decided to investigate the applicability of a third strategy. Specifically, we sought to assess how well the raw structural changes themselves could be used to identify and map subtasks within change sets. We did so for several reasons. First, the two strategies outlined above depend on input data that might be incomplete, misleading, or even completely unavailable. For example, in typical software systems, detailed commit comments or separate patches describing subtasks within a change set simply do not exist for some commits. Furthermore, commit comments themselves can be misleading in cases where textual artefacts like bullet lists are used for reasons other than listing subtasks. For example, we have seen many commit comments that use bullet lists to present justification for a single change, rather than outline multiple changes, e.g., "This [change] has many benefits: 1. [...] 2. [...] 3. [...]."[5] In contrast, the raw changes themselves are always present and are less easily misinterpreted than natural language. Finally, working with changes allowed us to build on our existing DIFFCAT diffing infrastructure.

During the development of our approach, we made two important observations: *i)* most commits contain changes related to a single task and *ii)* it is always safer to miss splitting a multi-task change set than it is to erroneously split a single-task change set, since in the former case we do no worse than the current "default strategy," whereas in the latter case we might cause a change-based approach to miss potentially useful associations between co-committed elements. Our underlying strategy for splitting change sets thus assumes that all updated elements are generally related, unless we find very strong evidence to the contrary. In this way, we sought to do no worse than the current "default strategy" in most

[5] Committed by `sandygao` to XERCES on 2002-09-16 01:07.

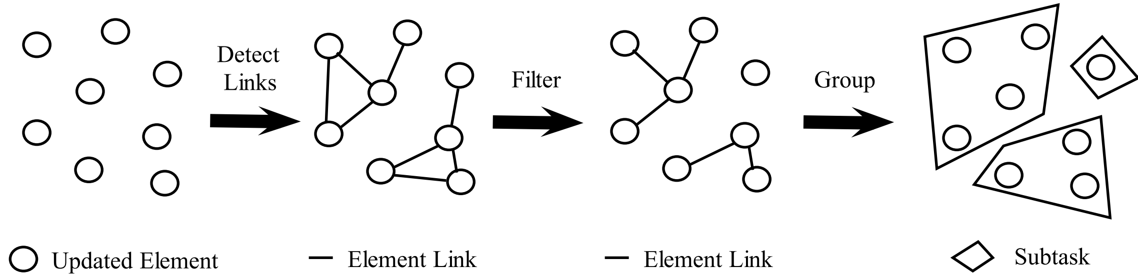

Figure 3.1: Detecting Subtasks in Change Sets

cases, and possibly do better than this strategy in the remaining cases.

Our overall approach consists of three steps, which we outline in Figure 3.1. In the first step, we use four basic heuristics to detect weighted *connections* between pairs of updated elements. Each of our heuristics parses the low-level changes affecting each updated element and connects pairs of elements if their low-level changes meet specific similarity criteria. We then combine the basic element-level connections from each heuristic into combined weighted connections between elements and use these to build a weighted element graph, where nodes in the graph represent updated elements, and weighted edges denote combined weighted connections between updated elements. In the second step, we filter out those connections that we deem to be too weak. Finally, in the third step, we detect all connected sub-graphs within our element graph and label each connected sub-graph as an isolated subtask. The idea behind this approach is that we use lenient linking criteria to eagerly connect as many elements as possible into common subtasks, so that two updated elements e_i, e_j are very likely to appear within the same subtask unless we find very strong evidence to the contrary, i.e., not a single path from e_i to e_j anywhere in our graph. The eagerness of our approach is motivated by the two observations outlined in the previous paragraph.

We next outline the details of each of our four heuristics and general procedure. We have released a prototype implementation of our technique on our website: http://www.cs.mcgill.ca/~d-kawry/thesis.

3.3.1 Keyword Connections

During our general manual investigations of change sets, we noticed that developers often reuse names and keywords when working on a specific subtask. For example, in our motivating example in Section 3.2, two of the three changes related to the second subtask are textually identical, and all three changes feature a reference to the `fBusy` field. In contrast, the change related to the first subtask has very little textual overlap with any of the three changes of the second subtask. Specifically, the change of the first subtask involves keywords such as `child`, `node`, `type`, `text`, etc., whereas the combined changes of the second subtask involve keywords such as `busy`, `reset`, `msg`, `message`, `format`, etc. In all, the textual cohesion between the change related to the first subtask and any of the changes related to the second subtask is much lower than the textual cohesion between any pair of changes related to the second subtask.

Given this insight, our first heuristic is based on the hypothesis that changes (diffs) featuring similar textual keywords are more likely to be related to the same task/subtask than diffs featuring dissimilar keywords. In other words, we hypothesize that if the diffs associated with one updated element share similar textual keywords as diffs associated with another element update, then those two element updates are likely to be part of the same task/subtask. Conversely, if the keywords associated with two element updates are dissimilar, then it is less likely that the updates are related.

To associate keywords with an updated element e, we first collect all diffs δ_i associated with e. We say a δ_i is associated with e if δ_i inserts or deletes e, if it modifies the declaration signature of e or, in the case of methods, if it modifies the body of e. We then traverse the `leftNode` and `rightNode` AST nodes of each δ_i and collect all text-based elements (simple and qualified names, and words within string literals) that appear within these nodes. We then tokenize each text-based element by splitting it based on common token separators (periods and underscores), camel case word separations, and groups of capitalized letters appearing together (e.g., "DOM"). We eliminate non-essential or low-value tokens by filtering out one-letter tokens and common stop words,[6] and by ignoring words picked up from primitive and basic Java types (e.g., `int`, `Object`, `String`, etc.). Finally, we convert

[6] http://www.textfixer.com/resources/common-english-words.txt

all tokens to lower case and filter out duplicates, so that a given token is associated at most once with a given element e.

Given two elements e_1, e_2 and their token sets S_1, S_2, we calculate the weighted connection between e_1 and e_2 based on the size of the normalized intersection of S_1 and S_2. Specifically, we compute

$$W(e_1, e_2)_{\text{key}} = \frac{|S_1 \cap S_2|}{\min(|S_1|, |S_2|)} \tag{3.1}$$

We take a normalized size so that smaller changes involving few keywords can still be connected to larger changes involving many keywords. We also note that if S_1 or S_2 is empty, then $W(e_1, e_2)_{\text{key}} := 0$.

In our motivating example, our heuristic associates the keywords `child`, `node`, `type`, `text`, `dom`, `filter`, and `show` with the method updated as part of the `AbstractDOM-Parser` subtask. None of these words appear in the the change inserting the `abort` method, which yields keywords `busy`, `reset`, and `abort`, so that $W_{\text{key}} = 0$. In contrast, the insertion of the `abort` method has a relative overlap of $W_{\text{key}} = \frac{1}{3}$ with the two other changes in the change set, because those also feature the `fBusy` field, and hence the `busy` keyword.

3.3.2 Dataflow Connections

Our keyword heuristic only connects updates whenever a developer explicitly works with similar textual content. When this is not the case, we can still detect connections between diffs if we consider basic dataflow dependencies between them. For example, we can consider the case where a developer updates the conditions of a hypothetical `if`-condition within a method `m`:

```
if (!  foo) { // deletion of the ``!''  character
```

and then adapts a related boolean assignment to preserve the program's behavior at a call site of `m`:

```
bar = true; // becomes bar = false
```

```
m (bar);
```

In this case, using a keyword heuristic like the one outlined above, the first change might only generate the keyword `foo`, whereas the second would only generate the keyword `bar`, which do not textually overlap. However, the results of a basic dataflow analysis would detect that the assignment to `bar` in the second change is related to the condition on `foo` in the first change, because the value of `bar` might determine the value of `foo` at runtime. In other words, the overall *dataflow* of the second diff is generally related to the dataflow of the first diff. Given this insight, we hypothesize that updated elements involved in similar dataflow have a higher likelihood of being related than elements that are not involved in similar dataflow.

We next outline our detection such dataflow connections between elements.

Detecting Basic Dataflow Connections

We first perform a light-weight inter-procedural dataflow analysis to build a dataflow dependency graph between all variables that are referenced within the old or new files of the change set. Each node in the graph represents a referenced variable as part of the change set and edges between variables v_i, v_j indicate that the value of either v_i or v_j influences the value of the other. Specifically, we build an edge between two variables v_i and v_j whenever v_i and v_j appear together in one of four basic relations. We further illustrate these relations in Figure 3.2. The figure displays a representative code snippet on the left and the various basic variable connections on the right. Given this, we say that v_i and v_j are connected if

1. v_i is directly assigned to v_j in an assignment statement in an old or new file. For example, in Figure 3.2, we connect `y` and `x` because `y` is assigned to `x` in the code.

2. v_j is assigned a value inside the scope of a conditional block and the conditional statement of that block references v_i. For example, in Figure 3.2, we connect `s` with both `r` and `t` because `s` appears within the condition of an `if`-block assigning values to `r` and `t`.

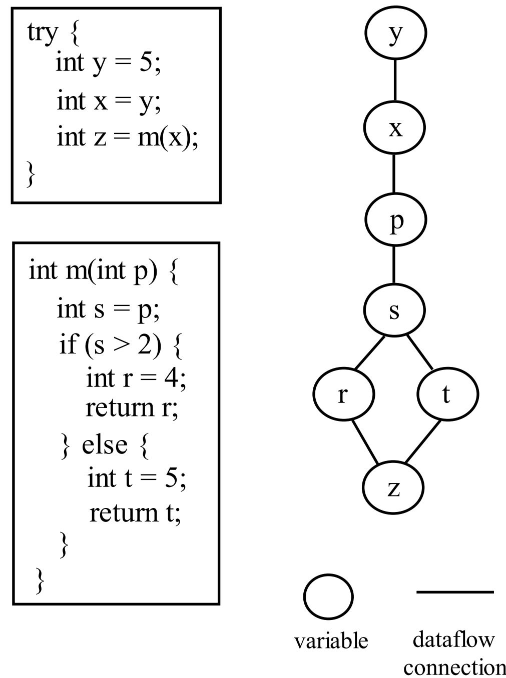

Figure 3.2: Dataflow Connections between Variables

3. v_j corresponds to the kth declared parameter within a method body m and v_i is the kth input argument to an invocation of m in another part of the source code. For example, in Figure 3.2 we connect `x` and `p` because `x` is passed as the first input argument to an invocation of `m` and `m`'s first formal parameter is `p`.

4. v_j is returned from within a method body m (i.e., via a `return` statement) and v_i is assigned the returned value of an invocation of m (e.g., `vi = m()`). For example, in Figure 3.2, we connect `z` with both `r` and `t` because `r` and `t` are both returned by `m` and `z` is assigned the returned value of an invocation to `m`.

Several things should be noted about the construction of our dataflow dependency graph. First, we only pick up inter-procedural connections for all method bodies that were committed as part of the change set. Consequently, we do not pick up any connections from method invocations referring to unknown method bodies. Second, to deal with Java's dynamic dispatch mechanism, we associate a given method invocation m with the bodies of *all* declared and available methods sharing the same name and simple parameter types

as m. We do not take into account additional control-flow or information related to the program's type hierarchy because this information is too often incomplete when analyzing change sets. Similarly, we do not link a given thrown exception e with any caught exceptions in the code because we do not have enough structural or type information to refine our associations in a consistent fashion; while prototyping our approach, we tried linking a given exception with all other caught exceptions, but found that this strategy caused too many unrelated changes to be linked by the resulting, often erroneous dataflow connections. Finally, we also note that our analysis deals with nested method invocations, e.g., `m(f())`, by unrolling these through imaginary temporary variables to store intermediate expressions, e.g., by rewriting nested expressions as `t=f(); m(t)`.

Connecting Updated Elements

Having once established connections between individual pairs of variables, we next use these to identify weighted element-level connections between updated elements. To do this, we first collect all variables referenced as part of the diffs for each updated element e_i to construct a set of initial seed variables S_i. For example, in the case of our motivating example, we would associate the set { `child`, `TEXT_NODE`, `fDomFilter`, `SHOW_TEXT` } with the updated `AbstractDOMParser` method. We then use the basic dataflow dependencies between all variables to compute the transitive closure $T(S_i)$ for each S_i. Conceptually, $T(S_i)$ represents all variables that influence or are influenced by the variables in S_i. We then compute a weighted connection between e_i and e_j using the normalized size of the intersection between $T(S_i)$ and $T(S_j)$:

$$W(e_i, e_j)_{\text{df}} = \frac{|T(S_i) \cap T(S_j)|}{\min(|T(S_i)|, |T(S_j)|)} \tag{3.2}$$

The idea is that if the variables of one updated method reach those of another updated method via our basic dataflow connections, then the transitive closures of their variable sets will overlap and we will associate them with the same task/subtask. In the case of our motivating example, we find that the transitive closure of the `AbstractDOMParser` method has zero overlap with those declared in `DOMParserImpl`, whereas the methods in

the latter group all share an overlap of 1.0.

3.3.3 Context Connections

Not all related elements can be connected based on keywords or dataflow alone. For example, in object oriented programs, elements are often permanently connected via inheritance relations, so that pairs of elements are often intrinsically related because they, for example, override the same method, implement the same interface, or share the same name. We thus assign $W(e_i, e_j)_{\text{con}} := 1$ to two updated elements e_i, e_j whenever they share similar declaration signatures, i.e., have the same name, override the same method, or extend or implement the same type. Otherwise we say that $W(e_i, e_j)_{\text{con}} := 0$. This heuristic echoes previous approaches linking code elements or changes based on their sharing similar structural context [19, 27].

3.3.4 Hierarchy Connections

Given two elements e_i and e_j, we automatically assign $W(e_i, e_j)_{\text{hi}} := 1$ if e_i is either a newly inserted or deleted element or e_i's declaration signature was modified in some way, and if e_j is modified to contain a newly inserted or deleted reference to e_i. In these cases, we assume that changes to e_j cannot be independent from e_i because of these direct adaptations involving both elements.

3.3.5 Combining Connections

We have implemented our connection heuristics by extending the SEMDIFF framework. Our prototype implementation takes as input the old and new files of a change set (represented as resolved ASTs) and all the essential differences (diffs) between those files, as detected by DIFFCAT. We then compute four weighted connections between each pair of elements e_i, e_j using our heuristics and combine these weighted connections into a single weighted connection using the following linear combination:

$$W(e_i, e_j) = \frac{1}{4}W(e_i, e_j)_{\text{hi}} + \frac{1}{4}W(e_i, e_j)_{\text{con}} + \frac{1}{4}W(e_i, e_j)_{\text{key}} + \frac{1}{4}W(e_i, e_j)_{\text{df}} \tag{3.3}$$

Next, we remove all connections that do not satisfy our selected threshold of $W(e_i, e_j) \geq \frac{1}{4}$. The combined effect of our linear combination and selected threshold is that we eliminate all connections between pairs of updated elements that *i)* do not share any contextual or hierarchical connections and *ii)* have at most very low combinations of keyword and dataflow similarity. We selected the specific threshold of $W(e_i, e_j) \geq \frac{1}{4}$ based on experience gained while prototyping the approach. Finally, we use the remaining connections to construct a graph G, where each vertex represents an updated element and edges denote connections between elements. We then say that each connected component within G represents an isolated subtask. For example, if G consists of two connected components S_1 and S_2, then we say that all the elements within S_1 constitute one subtask and all those within S_2 constitute another subtask. Conceptually, this means that every updated element within S_1 does not share a connection with *any* element within S_2, and vice versa. We interpret this as constituting strong evidence that changes within S_1 are conceptually separate from those in S_2, and likely to embody the kind of subtasks that might be mentioned separately within the change set's commit comment.

Finally, we note that we always remove non-essential differences when processing the diffs associated with a given updated element. We do this to avoid processing non-essential method updates and isolated changes that are less likely to be associated with documented tasks/subtasks. For example, a trivial `this` keyword insertion is less likely to be part of a documented task/subtask than, say, an update to an `if`-statement. We assume that processing statements that were updated by such a trivial insertion would only pollute our results in most cases.

3.4 Evaluation

We sought to evaluate whether our approach can accurately split change sets containing changes related to multiple subtasks. To do so, we retrieved 1 805 change sets from the change histories of seven of the ten target systems introduced in Section 2.5.1 and used the commit comments of the selected change sets to manually split them into subtasks.[7]

[7] We did not analyze HIBERNATE and JDT-UI because we were unable to download some of the requisite files from their repositories. We did not analyze STRUTS because we had not yet selected it for analysis in

Table 3.1: Characteristics of the Benchmark

System	Total Ch Sets	N-Task Ch Sets	N-Task Authors	1-Task Avg Size	N-Task Avg Size	N-Task Avg #Sub	N-Task Ambig
Ant	274	13	8	11	17	5	17
Azureus	264	10	1	8	4	2	10
JDT-Core	92	0	- -	20	- -	- -	0
JEdit	309	19	7	13	19	2	20
Nutch	308	18	5	21	18	2	9
Spring	284	10	3	13	9	2	11
Xerces	274	25	15	11	8	2	4
Total	1 805	95	39	13	13	2.5	71

We then applied our approach to each change set and compared the results of its analysis against the manually-inferred subtasks in our benchmark. We have released our benchmark, details of our experimental setup, and all data on our website: http://www.cs.mcgill.ca/~d-kawry/thesis.

3.4.1 Creating the Benchmark

We randomly selected 500 change sets from each of the transaction histories of the seven selected benchmark systems.[8] We then used DIFFCAT to identify the essential element updates within each change set and removed all change sets modifying less than two elements. To avoid ambiguities, we also removed all those change sets for which there was no commit message because we could not easily estimate the number of subtasks within such change sets. This left us with 1 805 candidate change sets, all of which had a non-empty commit comment and two or more updated elements.

We next read the commit message for each selected change set to determine which of

Chapter 2.

[8]Given the relatively short history of NUTCH, we selected 500 consecutive change sets from its change history, starting with the 100th.

these included updates related to two or more subtasks. Whenever a commit message did not clearly indicate the presence of multiple subtasks, we assumed that all updated elements for that change set belonged to a single task. Whenver a change set's commit message did clearly outline at least two subtasks, we used the DIFFCAT viewer to investigate the elements that were updated as part of the change set, as well as the actual fine-grained structural changes that modified each element. During this step, we labelled each updated element based on what concrete subtask(s) that element was modified as part of. To avoid bias, and in keeping with our definition of subtasks, we always referred specifically to the subtasks outlined in the change set's commit comment when labelling updated elements. We did not attempt to label element updates in change sets that had vague commit comments or changes that were too complex; in these cases, we flagged the change sets as being "ambiguous, " and made no further attempt to classify updated elements therein. Furthermore, whenever we found that an updated element was assigned to more than one subtask, we fused those subtasks and all their updated elements into a single task. In total, this process yielded 95 multi-task change sets for which we could collect clear evidence that they represented more than one subtask. Furthermore, we identified 71 change sets as having commit messages that suggested the presence of multiple subtasks, but for which we were unable to clearly relate changes to those subtasks.

Table 3.1 summarizes properties of the benchmark. The table shows the total number of change sets included for each system (Total Ch Sets) and how many of these change sets described clearly separable subtasks within their commit comments (N-Task Ch Sets). The table also shows the total number of contributors (authors and any contributors of patches) for the change sets featuring two or more subtasks (N-Task Authors), as well as the average number of updated elements for both single- and multi-task change sets (1-Task and N-Task Avg Size, respectively). The benchmark includes changes by 39 contributors, thus limiting any bias that may arise from studying changes made by only a small number of authors. The average number of elements updated as part of the multi-task change sets in the benchmark (N-Task Average Size) is roughly equivalent to the average number of elements updated as part of the single-task change sets (1-Task Average Size). The multi-task change sets also contained about 2.5 subtasks on average, with most change sets featuring two subtasks (N-Task Avg #Sub). Finally, the table displays the number of change

sets that feature commit messages implying the presence of multiple subtasks, but for which we were unable to produce unambiguous element classifications (N-Task Ambig). We also note that the benchmark features numerous combinations of subtasks, including change sets featuring multiple isolated bug fixes or enhancements, as well as change sets featuring both a fix and an enhancement, a fix and a code cleanup, an enhancement and a cleanup, or a combination of all three.

We note that the average multi-task change set size recorded in the table is not likely to be an estimate of the true size of general multi-task change sets because we found at least 71 change sets with commit comments that suggested the presence of multiple subtasks, but which we were unable to disentangle. Furthermore, given the restrictions of our definition of a subtask, the benchmark does not include multi-task change sets containing conceptually isolated, albeit undocumented changes. Consequently, the benchmark does not allow us to measure the overall recall of our approach, instead giving us insight into how well the approach can identify the kind of smaller, more concise multi-task change sets for which we were able to manually split their updated elements. We also note that since we did not attempt to process change sets for which there was no commit message, our evaluation cannot reveal how well our approach performs on these kinds of change sets. However, we prefer these limitations to avoid ambiguities in the element assignments within our benchmark.

3.4.2 Quantitative Results

We applied our approach to the change sets within the benchmark. Whenever the approach split a change set into subtasks, we compared the inferred subtasks against those in the benchmark to categorize the result into one of five categories, which are shown in Table 3.2. For each system, the table records how often our approach *i)* incorrectly split a single-task change set into several subtasks (1-Task Incorr), *ii)* correctly split a multi-task change set into its subtasks (n-Task Corr), *iii)* identified at least one correct subtask within a multi-task change set, but fused others into a single subtask (n-Task Part), *iv)* incorrectly separated the updated elements of a multi-task change set into subtasks different than those defined in our benchmark (n-Task Err), or *v)* split elements into more than one subtask for one of the

Table 3.2: Split Change Sets

System	1-Task Incorr	n-Task Corr	n-Task Part	n-Task Err	Ambig
Ant	35/244 (14%)	2/13 (15%)	1/13 (8%)	3/13 (23%)	1/17 (6%)
Azureus	35/244 (14%)	1/10 (10%)	0/10 (0%)	1/10 (10%)	1/10 (10%)
JDT-Core	15/92 (16%)	- -	- -	- -	- -
JEdit	69/270 (26%)	5/19 (26%)	1/19 (5%)	1/19 (5%)	7/20 (35%)
Nutch	42/281 (15%)	4/18 (22%)	0/18 (0%)	1/18 (6%)	4/9 (44%)
Spring	44/263 (17%)	1/10 (10%)	1/10 (10%)	1/10 (10%)	2/11 (18%)
Xerces	30/245 (12%)	6/25 (24%)	1/25 (4%)	1/25 (4%)	2/4 (50%)
Total	270/1639 (16.5%)	19/95 (20%)	4/95 (4%)	8/95 (8%)	17/71 (24%)

71 "ambiguous" change sets (Ambig).

As shown in Table 3.2, our approach split (270+19+4+8+17=) 318 of the 1805 change sets in the benchmark. Of those, 270 were found to be single-task change sets, so that overall, the aproach correctly classified (1639-270) = 1369 of the 1639 single-task change sets in the benchmark (83.5%). The approach produced fully or partially correct splits for (19+4=) 23 of the 95 multi-task change sets (24%), and incorrectly split eight of the 95 multi-task change sets (8%); the remaining (95-23-8=) 64 multi-task change sets were not split and thus incorrectly classified as single-task change sets (67%). Finally, we found that the approach splits 17 of the 71 "ambiguous" change sets (24%).

Precision and Recall

We define the *precision* of our approach as the overall proportion of unambiguous change sets for which the approach produces correct or partially correct subtasks. We define its *recall* to be the proportion of correct or partially split multi-task change sets. Based on these definitions and the values in Table 3.2, we see that our approach has an overall precision of (1369+19+4=1392/1734 $\approx$) 80% and a recall of 24%. In contrast, the current "default strategy" of splitting no change sets would have a precision of (1639/1734 $\approx$) 95% and a

recall of 0%.

Properties of Inferred Subtasks

Whenever our approach split change sets, it typically split updated elements into exactly two subtasks (75% of the 318 split change sets). In almost all cases (96%), the approach split change sets into no more than five subtasks. The approach also typically grouped most elements into one large subtask, with the remaining elements being divided into addtional and typically very small subtasks. Specifically, 59% of all inferred subtasks consisted of a single "breakaway" element and 83% of all inferred subtasks consisted of no more than five elements.

Correctly Split Multi-Task Change Sets

We found that the 23 multi-task change sets that were correctly or at least partially split by our approach tended to contain fewer subtasks and elements per subtask, on average, than the 72 multi-task change sets that were not split or only incorrectly split by our approach. Specifically, when we combined these two measures to define the "size" of a change set as

$$\text{size} = \#\text{Subtasks} + \frac{\#\text{Elements}}{\#\text{Subtasks}} \tag{3.4}$$

we found that the 23 change sets in the former group had an average size of approximately 5, whereas the average size of the 72 change sets in the latter group was around 8.5. If we then say that a "small" multi-task change set is one whose size was less than the average size of all multi-task change sets ($\approx$ 7.7), then our approach correctly classified 20 of 43 small change sets (47%), versus just 3 of 52 large change sets (6%). A Fischer's Exact Test showed this relationship to be statistically significant, with $p < 0.0001$. From this result, we infer that our approach is more likely to correctly split a multi-task change set if it features both few subtasks and few elements per subtask. We add that we also noted a similar relationship when we defined "size" to be only the average number of elements per subtask ($p = 0.0057$). However, we noted no similar statistically significant relationship when considering only the number of subtasks for each change set.

Table 3.3: Split Change Sets by Category

Change Group	Undoc Atomic	Unused Elements	Limit Approach	Unusable Changes	Other Indirect
Single-Task	116 (43%)	17 (6%)	13 (5%)	64 (24%)	60 (22%)
Multi-Task	3 (38%)	0 (0%)	1 (13%)	2 (25%)	2 (25%)
Ambig	7 (41%)	0 (0%)	1 (6%)	6 (35%)	3 (18%)
Total	126 (43%)	17 (6%)	15 (5%)	72 (24%)	65 (22%)

3.4.3 Qualitative Analysis

We manually investigated the results of our approach for those (270+8=) 278 cases where the approach incorrectly split the updated elements of a single- or multi-task change set, and for those 17 cases where the approach split an ambiguous change set. To do this, we used the DIFFCAT viewer to study the updated elements for the change set and to identify reasons why some of those updated elements were not linked to the appropriate subtask. The following is a discussion of the different causes we identified, which are also summarized in Table 3.3. The table displays, for each of the three groups of split change sets (rows), the number and proportion of change sets that were split for a given reason (columns).

Undocumented Atomic Changes

In (126/295=) 43% of the change sets, our approach identified what we call "undocumented atomic changes," or changes that, although not explicitly described in the change set's commit comment as separate subtasks, are nevertheless completely isolated from the change set's other changes. Changes in this category include isolated updates to logging- or debug-related outputs, updates to methods describing the current build version, isolated insertions or deletions of `final` keywords, reductions in an element's accessibility, code-cleanups involving generics or other minor rewrites, checkstyle concessions, isolated micro-bug fixes, insertions of separate utility methods, and other clearly isolated changes. This category

included 116 single-task change sets, seven ambiguous change sets, and three incorrectly classified multi-task change sets.

Deletions of Unused or Unusable Elements

In (17/295=) 6% of the change sets, our approach isolated deletions of unused elements or null-returning methods. In these cases, change sets deleted isolated code elements but were not adapted in any other way to deal with the deletion. In some cases, the deleted elements were either private, deprecated, stub implementations, or simple delegates to methods declared in a parent class. In other cases, the elements were visible outside of their class scope, but no other code changes were committed as part of their deletion, making it difficult to determine whether these elements were truly unused or whether the change ended up breaking the build. This category consisted entirely of single-task change sets.

Limitations in our Approach

In (15/295=) 5% of the change sets, element updates were split due to limitations in our approach. In most of these cases, our approach failed to link updated elements that were updated via similar changes. For example, in several change sets, we could have linked an updated field to a related updated method because both the declared type of the field and the formal return type of the method were replaced by the same new type. In other cases, incomplete bindings returned by PPA or bugs in CHANGEDISTILLER caused us to miss information that would have helped us link udated elements. In this category, 13 change sets were single-task, one was ambiguous, and one was an incorrectly classified multi-task change set.

Unusable Changes

In (72/295=) 24% of the change sets, our approach could not link elements that were modified only via changes involving either not enough or too much text. For example, in many cases, elements were updated only via an increase in their accessibility, e.g., by changing a `private` keyword to `public`. In other cases, elements were updated only via insertions or deletions of `return` or `break` statements, additional exceptions in their signature, updates

to `catch` clauses, insertions of `this()` constructors, updates to large natural-language string messages featuring too many keywords, updates involving `ints` or `booleans`, updates to fields with very small names, and other changes featuring no or only very little reusable textual keywords or variables. In these cases, the precision of our approach could be improved by automatically linking elements updated only via very small changes to all other elements. This category consisted of 64 single-task change sets, six ambiguous change sets, and two incorrectly classified multi-task change sets.

Other Indirect Changes

In the remaining (65/295=) 22% of the change sets, our approach failed to link updated elements because the keywords and variables associated with some updated elements simply did not overlap with those of other updated elements. Such changes can arise because of indirect dependencies between updated elements, for example, when they are are controlled by third-party GUI frameworks. This category included 60 single-task change sets, three ambiguous change sets, and two multi-task change sets.

3.4.4 Discussion

Precision of the Approach

Our assessments show that our approach incorrectly splits 16.5% of all single-task change sets, 8% of all multi-task change sets, and 24% of all ambiguous change sets (295/1805 $\approx$ 16% overall). In the worst case, erroneously splitting change sets can induce clients of our approach to miss potentially valuable associations between co-committed elements. However, we found that in many cases (43% of 295), our approach identified *atomic changes*, or changes which did not appear to be related to the other changes in the change set. In these cases, we hypothesize that any lost associations between co-updated elements are less likely to have a negative impact on client results and might in fact improve those results. In other cases, our approach split change sets because some updated elements featured too few or too many keywords (24%) or because the deletions of those elements were not directly linked to other changes (6%). To avoid lost element associations, our approach can

be modified to automatically categorize change sets featuring such changes as single-task. Similarly, in a few cases (5%), future improvements to our heuristics or reused components could avoid any lost associations altogether. Hence, given the qualitative breakdown of our results, we found that our general linking strategy was completely inappropriate for only 16% of the 295 change sets we considered, or (65/1805=) 4% of all the change sets in the benchmark. Consequently, although the overall precision of our approach was only 80%, we find that our general heuristics are not unreasonably imprecise in about 96% of all cases.

Comparison Against the Default Strategy

Our approach incorrectly classified about two thirds of all unambiguous multi-task change sets as single-task. In these cases, the outcome of our approach is identical to the current "default strategy," which assumes all change sets are single-task. Consequently, given that our approach also correctly splits 24% of all unambiguous multi-task change sets, we say that our approach performs better or no worse than the "default strategy" on 92% of all unambiguous multi-task change sets. However, it performs worse than the "default strategy" on the remaining multi-task change sets (8%).

Identifying Small Subtasks

Our assessments suggest that our heuristics are more likely to correctly (or at least partially) split multi-task change sets if those change sets contain both a small number of subtasks and few elements per subtask. Whether an identification of such small subtasks in change sets can actually help client analyses will depend partly on the relative frequency of such change sets in change history. Our benchmark contained only very few small and unambiguous multi-task change sets (46 out of 1805, based on our proposed definition of "smallness"), which suggests our approach is unlikely to have a strong impact on the results of change-based approaches in the general case. However, our approach also managed to identify many small and undocumented atomic changes in an additional (126/1805 $\approx$) 7% of all change sets. Hence, if we expand our definition of "subtask" to include such small multi-task change sets, our identification of *small subtasks* could be relevant for approximately (126+46=172/1805 $\approx$) 10% of all change sets in the benchmark.

Further Improvements to the Approach

Our current linking strategy does not use artefacts in the change set's commit comment, possibly linked bug database entries, or any other data except for the raw structural changes themselves. This allows us to perform equally well given potentially confusing commit comments or missing bug entries. However, our approach could also be improved by incorporating such additional knowledge when available and reliable. For example, in the case of our motivating example in Section 3.2, our approach could have found the appropriate (and separate) patches for the two bug fixes on the XERCES bug repository website. Or, in the case of lengthy commit messages, text-based analyses could be used to identify those messages featuring no textually separated sub-regions of any kind.

One additional limitation of our use of structural changes is that our approach does not *name* the subtasks it infers, i.e., we provide no links between our inferred subtasks and identifiable entities such as bug ids, named features, or concrete development activity. Consequently, approaches that wish to operate on the subtasks we infer would not know the relevant portions of the commit comment describing those subtasks. Consequently, naming our inferred subtasks remains as interesting future work.

Threats to Validity

Possible misinterpretations of commit comments or individual changes during the construction of our benchmark could have influenced our quantitative results. We tried to minimize possible misinterpretations by using very strict criteria for what constitutes a separate subtask and by separately labelling those classifications that we aborted due to ambiguities about a change set's changes. We thus emphasize that the 95 multi-task change sets within our benchmark are likely to be less complex than general multi-task change sets.

An additional threat to validity is a possible selection bias arising from our choice of target systems. We tried to minimize this bias by selecting long-lived projects that have been previously studied and that are distributed by different vendors. However, as all of our projects are medium-sized, open-source, and developed by small- to medium-sized core development teams, we cannot generalize our results to larger projects or those following propietary development models. For example, several of our studied projects regularly

include patches by other contributors, which, when reviewed and committed together by a core developer, might be more easily disentangled than changes that are unrelated but committed as part of a regular commit. The overall proportion and characterizations of multi-task change sets might thus not hold for systems that are not updated through patches from external contributors.

Chapter 4
Related Work

Our investigation of non-essential differences and multi-task change sets in change history complements existing research that seeks to increase the precision with which software changes can be abstracted and incorporated into software engineering tools.

4.1 Change Descriptions

There exist numerous techniques for summarizing the differences (or changes) between two software elements. Our work is similar to these prior techniques in that we also seek to summarize changes in terms of their essentiality to change-based approaches, as well as their relatedness to each other.

4.1.1 Basic Differencing Tools

There exist many general-purpose differencing tools that operate on various program representations (e.g., text or Abstract Syntax Trees) and at different levels of granularity (e.g., lines or element references) to compute edit scripts summarizing the changes between program versions. Many of these techniques have been summarized in an existing survey by M. Kim et al. [26]. Similar to these differencing tools, our goal is to identify and describe changes that occur within a change set. However, our overall goal is to also classify these

changes in terms of their relevance to higher-level representations of development effort and to discover what changes are related to each other.

Most existing differencing tools partially focus on eliminating the effects of spurious textual differences from their computed edit scripts. For example, similar to our work, Neamtiu et al. developed an AST-based differencing technique that compares program snapshots and detects rename refactorings and rename-induced statement updates [35]. In contrast, we describe a more general category of change that includes rename-induced updates and other non-essential differences.

Our own differencing tool builds specifically on CHANGEDISTILLER, Fluri et al.'s tool-supported differencing technique that identifies statement-level differences between Abstract Syntax Trees [11, 12]. Like other tools, CHANGEDISTILLER ignores whitespace-related differences and identifies documentation-related updates. Our approach extends its technique by using PPA-inferred bindings to further categorize these changes.

Our use of Dagenais and Hendren's partial program analysis [7] to infer type bindings and support program differencing at the granularity of referenced program elements echoes previous work by Dagenais and Robillard on framework evolution [8], in which the authors identify differences between the call graphs of the old and new files within a change set to identify call-change relations. In contrast, we work with fine-grained modifications to detect non-essential differences occurring at the sub-statement level and to group these differences into subtasks.

4.1.2 Tools Detecting Basic High-Level Changes

Our research complements existing approaches that aim to summarize groups of low-level changes mined from version history in terms of higher-level change patterns. Such approaches include tools detecting refactorings [43], ad-hoc method splits and merges [15, 28], or moves of arbitrary lines of code between methods [5, 6]. Our work could benefit from such approaches when detecting rename-refactorings and other kinds of changes at or above the element level. For example, although we currently detect rename refactorings using CHANGEDISTILLER and our own custom analyses, we could also validate these detected refactorings using any of the techniques outlined above.

4.1.3 Tools Detecting Systematic Changes

Several techniques detect what can be referred to as systematic changes between program versions. Work in this area include techniques that mine change history to find aspects or cross-cutting concerns in source code [1, 4, 36]. Results inferred by these aspect-mining approaches could help our technique link changes whenever they modify the same aspect. However, in keeping with their goal to identify aspects, the techniques used by these aspect-mining approaches usually require changes to exhibit certain pre-defined patterns, so that they naturally fail to link arbitrary groups of changes. For example, Breu and Zimmermann's technique links changes to a common aspect if those changes delete or insert similar method invocations [4]. Similarly, Nguyen et al.'s aspect-mining technique links changes if they modify similar code snippets [36]. In keeping with our goal of finding subtasks within a change set, our change-linking heuristics are more lenient in that they cluster all changes as long as there exist certain loose and transitive relations between them.

One additional approach in this area is M. Kim and Notkin's LSDIFF, which compares program snapshot pairs to detect groups of coarse-grained low-level changes exhibiting logical high-level structural patterns [27]. For example, LSDIFF detects cases where all sub-types of a given type added calls to a new method `foo`, or where all methods of a certain name no longer call `bar`. In our work, we use similar ideas to link changes. For example, in the two cases outlined above, our keyword heuristic would link the changes based on their references to a common `foo` or `bar` keyword. However, as our goal is to discover more basic links between all kinds of changes, we process more kinds of fine-grained structural changes than LSDIFF and also use more lenient linking criteria to cluster these changes together. Furthermore, we also apply our analysis on change sets, not program snapshots.

4.1.4 Similarity Detection Tools

Our work complements existing tools identifying similar or nearly-identical code elements. Tools in this area include dozens of code clone detection tools, many of which have been outlined in an existing survey by Bellon et al. [2]. Similar to clone detection, our aim is to detect pairs of code fragments (e.g., methods) that are identical except for non-essential differences. In particular, all of the non-essential differences currently detected by our

approach are or could be used by existing code clone detectors to detect similar program fragments [2]. However, the converse is not true, i.e., clone detectors generally ignore additional kinds of differences that we consider to be essential. For example, a clone detector might detect code fragments that differ only by the insertion of an additional method invocation or updated variable assignment, whereas we consider such updates to be essential. Furthermore, unlike our approach, clone detectors do not typically label the differences between fragments, preventing potential clients of these approaches from operating on the differences themselves.

H. Kim et al.'s MECC detects what they refer to as *semantic clones*, or small code snippets that may not be syntactically similar, but which implement the same or similar functionality [24]. Their approach compares pairs of code snippets by modelling how those snippets access memory; snippets with similar memory accesses are then deemed to be semantic clones. H. Kim et al. show that this approach can detect numerous kinds of non-essential differences, such as general `if`-statement reorderings and various kinds of redundant statement reorganizations. However, like other clone detection tools, MECC only requires *similar* memory usage in its detection of clones, whereas we search specific for a restricted number of particular fine-grained changes. Similarly, unlike our approach, MECC does not describe the exact differences between clones.

Other approaches in this area include our previous work on detecting API method imitations [22], in which we compare element references (field access and method invocations) to detect similar code snippets, and work by Long et al. [32], which attempts to cluster related API functions into modules based on their use of shared program state and common private functions. Our keyword and dataflow heuristics are similar to these similarity heuristics in that they link updated elements sharing similar dataflow and using similar keywords (e.g., calling similarly named methods, referencing similarly named fields, etc.). However, in keeping with our desire to link changes to change tasks, our heuristics operate primarily on the changes affecting updated elements, and not their entire content.

4.2 Change Interpretations

Previous techniques have sought to assess the general meaning of a change, either in terms of its impact on the underlying system, or the kind of development activity that it embodies. Our work complements these techniques in that we seek to characterize the overall essentiality of fine-grained changes and to identify change sets that are likely to feature development effort related to multiple subtasks.

4.2.1 Significance of Low-Level Change Types

Our investigation of non-essential differences is related to a previous case study by Fluri and Gall, which showed that an interpretation of a change set's "significance" is tied to the particular representation with which its low-level deltas are represented [10]. Similarly, we also measured how higher-level change representations can be impacted by different low-level change characterizations. In their work, Fluri and Gall specifically contrasted a purely line-based representation of change significance against one based on their taxonomy of fine-grained structural differences. In contrast, we compared *non-essential* modifications against fine-grained structural differences and by evaluating impact in the more concrete terms of a method level representation, as opposed to a general notion of significance. Furthermore, Fluri and Gall's proposed significance measure of individual change types is partly based on their likelihood of inducing changes in other entities. In contrast, our notion of non-essentiality imposes stricter conditions on individual changes that is based partly on resolved element references and on the likelihood that the change is relevant to higher-level representations of software development effort. We also implemented a novel differencing technique to detect non-essential differences, which we used to further characterize their impact in an empirical investigation of a large number of change sets retrieved from multiple open-source systems.

Our work also complements a more recent investigation by Giger et al. [14], which showed that bug prediction techniques based on fine-grained structural changes can be more precise than those based only on line-based changes. Our work is similar in that we showed that eliminating non-essential differences from the input data used by a simple

association rule miner improved the precision of that miner.

4.2.2 Classifications of Development Activity

Our investigation of non-essential differences is related to existing approaches characterizing the development activity behind changes. These include an approach by Robbes and Lanza for eliciting higher-level properties of changes made during development sessions [39]. As part of this approach, all development activity is directly monitored, a strategy that could also be adapted to identify non-essential differences as they happen. Other approaches include the use of machine learning on commit metadata (e.g., commit comments) to classify large commits into different maintenance categories, such as code cleanups [18], or the use of pattern matching on commit comments to identify changes introducing or fixing a bug [42]. A detection of non-essential differences and multi-task change sets could help these approaches produce more precise labels of the specific development activity behind each change.

4.2.3 Impact of Code Changes

Our detection of non-essential differences is related to approaches that measure the possible impact of changes on the underlying system. These approaches can warn developers about changes that are likely to introduce bugs [29], affect program behavior [37], or introduce unexpected dynamic behavior at runtime [20]. Our work complements these approaches by identifying non-essential changes that are extremely unlikely to introduce bugs, require re-testing, or introduce unexpected dynamic behavior. In addition, unlike these approaches, our discovery of non-essential differences requires only the files within a change set, whereas most impact-assessment tools require additional input data. For example, S. Kim et al.'s detection of buggy changes uses machine learning to discover patterns from previous buggy changes; the approach is less effective in cases where such data is not yet available [29]. Similarly, although differential symbolic execution by Person et al. discovers non-essential method updates by identifying behaviour-preserving changes,

their approach requires a complete and compileable snapshot of each version being analyzed [37].

4.2.4 The Quality of Mined Data Sets

Our attempts to improve the quality of data mined from version repositories is motivated, in part, by previous work highlighting impurities that might exist in various archives and datasets that are frequently mined by software engineering techniques. For example, in 2009, Bird et al. showed that bug-fixing change sets with explicit links to the bugs they fix contain disproportionate amounts of simple fixes when compared to all the bugs that are found in a system's bug database [3]; the bias in these labeled bug-fixing changes may then affect default-prediction techniques that learn features from such changes, such as, for example, S. Kim et al.'s BUGCACHE approach [30]. Our work is similar in that we seek to detect non-essential differences and multi-task change sets and thereby improve the quality of data that can be mined from version histories.

Chapter 5
Final Discussion

Numerous techniques involve mining change data stored in software archives. Many of these techniques work with change sets under the assumption that the changes within change sets are all equally meaningful and all related to a single, well-defined development task. In this dissertation we provide empirical evidence suggesting that this may not always be true. For example, in Chapter 2, we found that between 3% to 26% of all modified lines of code and 2% to 16% of all method updates are due entirely to non-essential changes – minor cosmetic changes that are less likely to represent the kind of meaningful software development effort that is most interesting to change-based approaches. Furthermore, we found that over 80% of all non-essential differences were actually induced entirely by rename refactorings, which are typically entirely automated by modern IDEs such as Eclipse. These kinds of automated changes are less likely to provide meaningful or non-obvious information than other kinds of changes, such as those modifying the system's structure or control flow. Based on the qualitative assessments in Chapter 3, we also found that, among those commented change sets modifying two or more program elements, approximately 10% contain work related to either multiple documented subtasks or one or more undocumented *atomic changes* that are unrelated to the main task outlined in the change set's commit comment. These subtasks may then induce false associations between co-committed code elements. We believe these ratios should motivate change-based approaches to more precisely categorize the kind of data that they mine from version histories.

We have developed a general framework for detecting non-essential differences and

subtasks within change sets. Our framework makes use of element resolution provided by PPA [7] to precisely label individual changes and detect non-essential differences, and to link changes based on shared structural properties and thereby detect subtasks. Although we do not yet identify all possible kinds of non-essential differences and although we were not perfectly successful in identifying all the subtasks within our benchmark, we believe that future approaches can incorporate the ideas we present in this dissertation to reason more precisely about the changes they process in their analyses. For example, in Chapter 2, we showed that an elimination of non-essential differences can lead to a general improvement in the precision of a simple association rule miner.

We emphasize that the use of PPA limits the usefulness of our general strategy in several ways. First, our current techniques are limited specifically to the Java language. Extending our work to cover other popular programming languages will also require new implementations of PPA for those languages. Second, by being tied to specific programming languages, rather than more general text- or tree-based program representations, techniques using PPA must also adapt to the specific properties of the languages they process. Specifically, as programming languages evolve, steps must be taken to ensure that different regions of a system's change history are appropriately processed using the proper version of that language. In the case of PPA, clients must select under which version of Java they would like to compile the code they are analyzing, because different versions of Java support non-overlapping Java constructs. Failure to determine the appropriate version of Java for a specific system snapshot can lead to imprecisions in the ASTs inferred by PPA. Finally, although PPA allows very precise analysis of a change set's element references, the use of PPA significantly slows down the rate at which change sets can be processed. Informally, we noted that, for some systems, DIFFCAT processed less than 100 change sets per hour, with most of DIFFCAT's processing time spent using PPA. Furthermore, in some cases, PPA required many minutes to process a single change set. Although specific processing speeds are less relevant when conducting moderate empirical evaluations such as ours (30 000 change sets), they do matter in other contexts, for example, when processing millions of change sets. However, as we showed in Chapter 2, change-based approaches can also reduce the amount of PPA required for their analyses by considering *i)* that the

effects of non-essential differences are most noticeable in large change sets (those modifying 20 or more methods), which we estimated to constitute less than 3% of all change sets, and *ii)* that bug-fixing change sets are less likely to contain non-essential differences than other kinds of change sets. Consequently, change-based approaches that specifically process only very small or non bug-fixing change sets might should consider the tradeoff between PPA's (and hence DIFFCAT's) limitations and the potential gains from detecting non-essential differences and subtasks.

So far, we have loosely defined non-essential differences in terms of their possible relevance to certain forms of development activity, e.g., bug-fixing or work related to feature enhancements. However, informally, we have also noted that non-essential differences are also conceptually similar to the kind of undocumented atomic changes we describe in Section 3.4.3. Specifically, like undocumented atomic changes, we found that non-essential differences are often not documented as part of a change set's commit comment. It might thus be reasonable to redefine a "non-essential" difference as being a minor or automated modification that is both independent from other changes and also unlikely to be explicitly documented. Given such a modified definition, it would then be possible to reformulate the fairly separate problem statements of Chapters 2 and 3 as a single, more concise problem statement: Namely, that, rather than identify non-essential differences and split multi-task change sets, an approach could simply seek to identify *atomic changes* within change sets, or changes that are not in any way mentioned by a change set's commit message. Eliminating such undocumented atomic changes from a change set could then provide change-based approaches with a more concise mapping between a change set's described task(s) and the actual changes related to those tasks.

Bibliography

[1] B. Adams, Z. M. Jiang, and A. E. Hassan. Identifying crosscutting concerns using historical code changes. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 305–314, 2010.

[2] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, 33(9):577–591, 2007.

[3] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and balanced?: bias in bug-fix datasets. In *Proceedings of the the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 121–130. ACM, 2009.

[4] S. Breu and T. Zimmermann. Mining aspects from version history. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 221–230, 2006.

[5] G. Canfora, L. Cerulo, and M. Di Penta. Identifying changed source code lines from version repositories. In *Proceedings of the 4th International Workshop on Mining Software Repositories*, page 14, 2007.

[6] G. Canfora, L. Cerulo, and M. Di Penta. Ldiff: An enhanced line differencing tool. In *Proceedings of the 31st IEEE International Conference on Software Engineering*, pages 595–598, 2009.

[7] B. Dagenais and L. Hendren. Enabling static analysis for partial Java programs. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*, pages 313–328, 2008.

[8] B. Dagenais and M. P. Robillard. Recommending adaptive changes for framework evolution. In *Proceedings of the 30th ACM International Conference on Software Engineering*, pages 481–490, 2008.

[9] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus. Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1–12, 2001.

[10] B. Fluri and H. C. Gall. Classifying change types for qualifying change couplings. In *Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 35–45, 2006.

[11] B. Fluri, M. Wursch, M. Pinzger, and H. C. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, 2007.

[12] H. C. Gall, B. Fluri, and M. Pinzger. Change analysis with Evolizer and ChangeDistiller. *IEEE Software*, 26(1):26–33, 2009.

[13] H. C. Gall, M. Jazayeri, and J. Krajewski. CVS release history data for detecting logical couplings. In *Proceedings of the 6th International Workshop on Principles of Software Evolution*, pages 13–23, 2003.

[14] E. Giger, M. Pinzger, and H. Gall. Comparing fine-grained source code changes and code churn for bug prediction. In *Proceeding of the 8th Working Conference on Mining Software Repositories*, pages 83–92, 2011.

[15] M. W. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, 31(2):166–181, 2005.

[16] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(7):653–661, 2000.

[17] M. Hashimoto and A. Mori. Diff/TS: A tool for fine-grained structural change analysis. In *Proceedings of the 15th Working Conference on Reverse Engineering*, pages 279–288, 2008.

[18] A. Hindle, D. M. German, M. W. Godfrey, and R. C. Holt. Automatic classification of large changes into maintenance categories. In *Proceedings of the 17th IEEE International Conference on Program Comprehension*, pages 30–39, 2009.

[19] R. Holmes, R.J. Walker, and G.C. Murphy. Approximate structural context matching: An approach to recommend relevant examples. *IEEE Transactions on Software Engineering*, pages 952–970, 2006.

[20] Reid Holmes and David Notkin. Identifying program, test, and environmental changes that affect behaviour. In *Proceedings of the 33rd ACM/IEEE International Conference on Software Engineering*, pages 371–380, 2011.

[21] J. W. Hunt and T. G. Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20(5):350–353, 1977.

[22] D. Kawrykow and M. P. Robillard. Improving API usage through automatic detection of redundant code. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, page 111–122, 2009.

[23] D. Kawrykow and M. P. Robillard. Non-essential changes in version histories. In *Proceedings of 33rd ACM/IEEE International Conference on Software Engineering*, pages 351–360, 2011.

[24] H. Kim, Y. Jung, S. Kim, and K. Yi. MeCC: Memory comparison-based clone detector. In *Proceeding of the 33rd ACM/IEEE International Conference on Software Engineering*, pages 301–310, 2011.

[25] M. Kim, D. Cai, and S. Kim. An empirical investigation into the role of API-level refactorings during software evolution. In *Proceedings of the 33rd ACM/IEEE International Conference on Software Engineering*, pages 151–160, 2011.

[26] M. Kim and D. Notkin. Program element matching for multi-version program analyses. In *Proceedings of the 3rd International Workshop on Mining Software Repositories*, pages 64–71, 2006.

[27] M. Kim and D. Notkin. Discovering and representing systematic code changes. In *Proceedings of the 31st IEEE International Conference on Software Engineering*, pages 309–319, 2009.

[28] S. Kim, K. Pan, and E. J. Whitehead Jr. When functions change their names: Automatic detection of origin relationships. In *Proceedings of the 12th Working Conference on Reverse Engineering*, pages 143–152, 2005.

[29] S. Kim, E.J. Whitehead Jr, and Y. Zhang. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering*, 34(2):181–196, 2008.

[30] S. Kim, T. Zimmermann, E.J. Whitehead Jr, and A. Zeller. Predicting faults from cached history. In *Proceedings of the 29th International Conference on Software Engineering*, pages 489–498, 2007.

[31] V.I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet Physics Doklady*, volume 10, pages 707–710, 1966.

[32] F. Long, X. Wang, and Y. Cai. Api hyperlinking via structural overlap. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, page 203–212, 2009.

[33] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. In *Proceedings of the 31st IEEE International Conference on Software Engineering*, pages 287–297, 2009.

[34] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th ACM International Conference on Software Engineering*, pages 292–301, 2005.

[35] I. Neamtiu, J.S. Foster, and M. Hicks. Understanding source code evolution using abstract syntax tree matching. In *Proceedings of the 1st International Workshop on Mining Software Repositories*, pages 1–5, 2005.

[36] T. T. Nguyen, H. V. Nguyen, H. A. Nguyen, and T. N. Nguyen. Aspect recommendation for evolving software. In *Proceeding of the 33rd ACM/IEEE International Conference on Software Engineering*, pages 361–370, 2011.

[37] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu. Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 226–237, 2008.

[38] C. M. Pilato, B. Collins-Sussman, and B. W. Fitzpatrick. *Version Control with Subversion*. O'Reilly Media, 2008.

[39] R. Robbes and M. Lanza. Characterizing and understanding development sessions. In *Proceedings of the 15th IEEE International Conference on Program Comprehension*, pages 155–166, 2007.

[40] R. Robbes and M. Lanza. Spyware: A change-aware development toolset. In *Proceedings of the 30th ACM International Conference on Software Engineering*, pages 847–850, 2008.

[41] M. P. Robillard and B. Dagenais. Recommending change clusters to support software investigation: an empirical study. *Journal of Software Maintenance and Evolution: Research and Practice*, 22(3):143–164, 2010.

[42] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *Proceedings of the 1st International Workshop on Mining Software Repositories*, pages 1–5, 2005.

[43] P. Weissgerber and S. Diehl. Identifying refactorings from source-code changes. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 231–240. IEEE, 2006.

[44] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, 30:574–586, 2004.

[45] T. Zimmermann and P. Weißgerber. Preprocessing CVS data for fine-grained analysis. In *Proceedings of the 1st International Workshop on Mining Software Repositories*, pages 2–6, 2005.

[46] T. Zimmermann, P. Weissgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005.