Non-Essential Changes in Version Histories

David Kawrykow and Martin P. Robillard
McGill University
Montréal, Canada
{dkawry,martin}@cs.mcgill.ca

ABSTRACT

Numerous techniques involve mining change data captured in software archives to assist engineering efforts, for example to identify components that tend to evolve together. We observed that important changes to software artifacts are sometimes accompanied by numerous *non-essential* modifications, such as local variable refactorings, or textual differences induced as part of a rename refactoring. We developed a tool-supported technique for detecting non-essential code differences in the revision histories of software systems. We used our technique to investigate code changes in over 24 000 change sets gathered from the change histories of seven long-lived open-source systems. We found that up to 15.5% of a system's method updates were due solely to non-essential differences. We also report on numerous observations on the distribution of non-essential differences in change history and their potential impact on change-based analyses.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms

Measurement, Experimentation, Algorithms

Keywords

Mining software repositories, software change analysis, differencing algorithms

1. INTRODUCTION

Source code repository systems have been in use since the 1970s to keep track of the different versions of a system's artifacts and, by extension, of the changes made between versions [22]. Numerous techniques now involve mining change data captured in software archives to assist software engineering efforts. For example, mining change data has been used to measure code decay in aging systems [5], to predict defects in modules [10, 16], and to detect non-obvious relationships between code elements [8, 23, 25]. We refer to approaches operating on change data as *change-based approaches*.

Typical version control systems store changes as line-based textual deltas between committed code files. In contrast, change-based approaches generally aim to operate on more meaningful representations of change, such as, for example, the individual methods that were updated as part of a developer commit to the repository. More meaningful representations of software changes support more accurate reasoning about software development activity and effort.

A critical problem for change-based approaches is thus to bridge this conceptual gap between the low-level deltas stored in version control systems and the abstractions used to represent software development activity. A first step, implemented by most modern change-based approaches, is to ignore trivial low-level changes, like those induced by white spaces or other formatting-related modifications. The general assumption behind this strategy is that these groups of low-level differences are less likely to yield meaningful abstractions of the actual development effort behind a code change. For example, many change-based approaches ignore trivial updates when determining the set of methods that were modified as part of a code commit.

As part of our ongoing investigation of software archives, we observed that many additional kinds of minor (or *non-essential*) code changes can also cause change-based approaches to infer inaccurate high-level representations of software development effort. For example, every time a developer performs a rename refactoring, all methods that include references to the renamed element will also be textually modified; a naive abstraction of these non-essential rename-induced statement updates can then result in a bloated high-level representation of the change that appears to span many lines of code, methods, and files, despite corresponding to a single developer modification (that is generally a very simple tool-assisted operation). Given the growing importance of change analysis in software engineering, our long-term goal is to enable change-based approaches to incorporate information about the *essentiality* of code changes into their analyses. With this information, change-based approaches will be able to more precisely select the individual low-level modifications they use to derive their high-level representations of development activity or effort.

We investigated the potential impact of non-essential differences on the abstractions that are typically analyzed by many change-based approaches. In particular, we sought *i)* to characterize the prevalence of non-essential differences in change history, and, *ii)* to measure their impact on the *code churn* and *method updates* associated with code commits, two facets of code change that are considered in existing empirical studies involving change data [5, 16] and change task oriented analyses [25].

ICSE'11, May 21–28, 2011, Waikiki, Honolulu, HI, USA

Analyzing change history to detect the kinds of non-essential differences mentioned above is far from trivial. An automated detection of non-essential differences requires both a characterization of structural changes occurring within statements and an analysis of their impact on the underlying system. In addition, to avoid reconstructing an entire program snapshot for every committed change, the impact of changes must be determined given only a change set, or group of files that were co-committed by a developer [24]. We know of no existing tool that supports this type of precise sub-statement-level change analysis on partial programs (change sets). For example, although existing standalone syntactic- or token-based differencing techniques can be used to process change sets and detect cases of reference replacements within statements, none use type resolution to precisely infer reference replacements that were induced specifically by rename refactorings.

We developed tool support for detecting non-essential differences in change sets mined from change history. Our technique identifies cases where a modified program statement was updated by one or more predefined types of fine-grained differences. The non-essentiality of candidate statement updates is then verified by resolving and operating on the types of relevant sub-expressions within a statement update. For example, our technique detects cases where a statement was modified only because it contained references to one or more renamed program entities. We investigated the precision of our technique and found that 98.8% of the method updates it identified as non-essential were accurately classified.

We used our technique to analyze over $24\,000$ change sets from the revision histories of seven long-lived open-source Java systems. Based on the systems studied, our investigation found that between 2.6% and 15.5% of all method updates in a system's change history consisted entirely of non-essential differences. We also observed that these non-essential method updates can have a significant impact on the kind of higher-level recommendations produced by certain well-known change-based approaches. Specifically, we observed that non-essential method updates can interfere with the recommendations that might be produced by a standard method-pair association-rule mining analysis in 26% of the cases where the analysis makes at least one recommendation. Furthermore, we found that the overall *quality* of the recommendations produced in these cases was significantly improved after non-essential method updates were no longer taken into consideration. These and additional insights regarding the prevalence of non-essential difference in change histories are important to keep in mind when making decisions based on change data.

This paper contributes a tool-supported differencing technique for identifying non-essential code modifications in change histories, and a number of high-level observations characterizing the prevalence of non-essential differences and their possible impact on change-based approaches. Our observations are supported by a detailed empirical investigation of over $24\,000$ change sets from the revision histories of seven, long-lived, open-source Java systems. Readers can download a version of our tool and the experimental data behind our observations from our website.[1]

2. MOTIVATING EXAMPLE

Our motivation for studying non-essential differences stems from our previous work on change task clustering, where we noticed that non-essential method updates often generated meaningless associations between change sets [21].

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.[2] 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.

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 [16] or be symptomatic of a decaying system [5]. Other analyses might mine the many pairwise associations between the modified methods and eventually detect non-obvious dependencies between them [25]. 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 and change sets. In this case, a detection of rename-induced and other non-essential differences would have supported a more meaningful abstraction of the change set.

3. NON-ESSENTIAL DIFFERENCES

Existing tool-supported techniques can analyze change sets and detect and eliminate *trivial* differences such as whitespace- and documentation-related modifications [7]. Our goal is to measure the impact of *non-essential* differences on change-based approaches. We define non-essential differences 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 ones that are unlikely to capture meaningful information about the development effort behind a change.

To help catalog the kinds of non-essential differences we studied, we refer to two versions of the same code, shown below.

```
// Version N                    // Version N+1
Object field = ...              Object m_field = ...

  void sample() {               void sample() {
1.  List l = ...                  java.util.List list = ...
2.  l.add(this.field);            list.add(m_field);
3.  m(l.size());                  int size = list.size();
4.  return;                       m(size);
  }                             }
```

The two versions of the `sample` method exhibit a number of differences. In the first line of version N, the local variable `l` is renamed to `list` and its declared simple type modified to its equivalent fully-qualified type. In the second line, a trivial instance of

[1] www.cs.mcgill.ca/~swevo/diffcat

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

the `this` keyword is removed. The second line is also textually altered by the effects of a rename refactoring of the `field` attribute. The third line is modified by a local variable extraction refactoring that stores the input expression to `m` in a temporary variable `size`. Finally, in the fourth line, a redundant `return` keyword is deleted.

Although all of these program differences may be of interest in certain situations, we believe that they are unlikely to contribute relevant information for many change-based approaches seeking to measure meaningful software development effort. It would be unlikely, for example, for a developer to perform the kinds of modifications affecting the `sample` method 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 Type Updates

Textual updates to an entity's declared type are non-essential if the actual declared type is not affected by the update. Specifically, a trivial replacement of a type's simple name with its fully-qualified name does not affect how the declared entity is handled at runtime. We note that such changes cannot be detected without type resolution, since without type bindings, it would not be obvious whether `List` refers to `java.util.List` or some other `List` type, e.g., `java.awt.List`.

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.

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. 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 reports rename refactorings and all corresponding rename-induced updates as a single difference between program versions [17].

Trivial Keyword Modifications

In our investigations of Java code history, we encountered cases where developers redundantly insert or delete instances of the `this` keyword. In Java, prepending the `this` keyword to a program entity only affects program behavior in a limited number of cases.[3] In the `sample` method, the deletion of the `this` keyword has no effect on the behavior of that method. While changes involving the `this` keyword might improve readability, they can be considered non-essential in most contexts. In this category we also currently also include trivial insertions or deletions of `return` statements at the end of `void`-returning methods and trivial insertions or deletions of default `super` invocations occurring at the top of default constructors. In Java, such modifications have no effect on a program's behavior. We do not currently include other keyword updates, such as those involving affecting the visibility, modifiability, or overridability of a code entity.

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.[4]

Whitespace and Documentation-Related Updates

Whitespace and documentation-based modifcations are already ignored by other change analysis tools, such as CHANGEDISTILLER [7]. In our 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.

4. DETECTION TECHNIQUE

All of the non-essential differences outlined in Section 3 occur within programming language statements or expressions. Consequently, to detect them in change history requires analyzing changes at a level of granularity finer than statement differences. Detecting the non-essential differences in our catalog also requires resolving the type bindings of expressions within statements, a technically challenging task given that the files analyzed are not part of a complete and compilable system.[5] For example, to detect trivial `this` keyword insertions requires an analysis that detects not only the additional `this` keywords, but also verifies that no type bindings were altered by their insertion. 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 type-sensitive (to reason about the impact of changes on the program behavior). Although existing change analysis tools already support fine-grained differencing of individual program statements, we know of no change differencing tool that is both fine-grained and type-sensitive at the expression (sub-statement) level.

To detect non-essential differences, we therefore developed a mining technique that is both fine-grained and type-sensitive; we implemented our technique in a tool called DIFFCAT. Similar to existing change analysis tools, DIFFCAT takes 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 that change set. In addition to previous techniques, however, our technique also uses type resolution to further identify and label structural changes that are also non-essential. For example, DIFFCAT detects and labels cases where

[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.

[4] In very rare cases, a variable renaming might cause the variable to shadow an existing variable. In these cases, our differencing technique classifies the rename as non-essential and the resulting method update as essential.

[5] Software archives do not store enough meta-data to systematically regenerate a fully built system to match each individual change in isolation.

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 repositories.

4.1 Reused Components

Working with program references requires type bindings, or ways of resolving possibly ambiguous references to a program element to the corresponding fully-qualified element name. Type bindings are generally unavailable when working with partial programs, or collections of code files with missing dependencies, like those retrieved as part of a change set from typical version repositories such as CVS or SVN. Missing dependencies make it difficult to unambiguously identify the individual program entities being referenced in various statements, thus restricting the amount of information that can be inferred about those statements.

To help recover missing bindings, we use Partial Program Analysis (PPA). PPA is a general technique for resolving bindings in partial programs. In the case of Java, a recent implementation of PPA has been shown to recover over 90% of missing bindings in partial programs [2]. For example, using PPA allows change analyses to confidently determine whether a newly inserted `this` keyword affects the element reference to which it was prepended.

Our reused PPA implementation works with Abstract Syntax Tree (AST) representations of Java code. AST representations of Java files can easily be constructed using existing tool-support, such as the Eclipse-based JDT-CORE plug-in. Consequently, given PPA's AST-based output, we decided to detect non-essential differences using AST-differencing to more easily incorporate PPA-inferred type bindings into our differencing technique. We decided against using a other kinds of differencing techniques, e.g., token-based ones, to avoid the conceptual challenge of working with multiple program representations.

To further facilitate our detection of fine-grained non-essential differences between ASTs, we use CHANGEDISTILLER, a state-of-the-art tool that identifies statement-level structural changes between Java AST pairs [6, 7]. We enhanced the output computed by CHANGEDISTILLER with PPA-inferred bindings and performed additional processing to detect non-essential differences.

4.2 Identifying Modified Statements

CHANGEDISTILLER expresses differences between two source code files as edit scripts, or sequences of edit operations (e.g., insertions, deletions, or updates) involving structural entities at varying levels of granularity. In particular, CHANGEDISTILLER does not express program updates at granularities finer than individual program statements. Instead, it relies on a measure of textual similarity between statement versions to detect cases where a statement was modified, rather than inserted or deleted. This means that, given a high enough textual disparity between statements, CHANGEDISTILLER flags unmatched statements as deletions or insertions, rather than updates. 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 following assignment statement pair

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

exhibits a high degree of textual disparity, despite being functionally identical. 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.

4.3 Approach

We observed that the challenge described in Section 4.2 typically 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 above is more difficult than discovering the same update minus the effects of rename refactorings:

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

because the latter update exhibits a higher degree of textual similarity, making it easier to identify it as a statement update in the first place. Furthermore, the latter update 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 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.

4.4 Implementation

We reuse infrastructure provided by SEMDIFF [3], a change analysis tool for studying framework evolution. SEMDIFF retrieves and represents change sets as collections of file pairs. Each file pair corresponds to the two versions of a file found to have been modified as part of the change set. Each file-pair is provided to DIFFCAT as a pair of ASTs. With DIFFCAT, we then identify fine-grained structural differences (including non-essential ones) through a sequence of nine high-level operations. In the description of this sequence, we assume that identified differences are stored in a set, which we refer to as `changes`. We proceed as follows:

1. We run PPA on all input ASTs to resolve type bindings in statements and expressions.
2. We use CHANGEDISTILLER to identify structural differences between AST pairs. We collect all method and field rename refactorings from these differences.
3. We add to the list of rename refactorings by detecting renames of classes and local variables, as well as additional cases of field renames.
4. We process AST pairs representing renamed classes to detect rename refactorings within the renamed classes themselves.
5. All detected rename refactorings are stored in `changes`.
6. Using the renames in `changes`, we traverse the ASTs of each modified file pair to roll back the textual identifiers of all element references affected by a rename.
7. We re-run CHANGEDISTILLER on the modified AST pairs and collect all reported structural differences.
8. We process the structural differences and identify those that are non-essential. We store all structural differences in `changes`.

9. We reconcile changes inferred in Step 2 that were no longer reported in Step 7 because of our rename rollback. We label these changes as non-essential and add them to `changes`.

We provide additional details about our procedure below.

Detecting Class Renames

CHANGEDISTILLER does not identify class renames. We detect these by detecting class insert-delete pairs sharing a high proportion of identical field and method signatures (≥ 0.5).

Detecting Field Renames

CHANGEDISTILLER detects field renames by comparing their declaration statements using the Levenshtein similarity measure. 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 found in Step 2. We note that our analysis rejects a field insert-delete candidate if even a single statement update does not satisfy our criterion.

Rename Reconciliation

Step 9 is necessary to properly identify rename-induced non-essential differences that were eliminated by the rename rollback in Step 6. For example, the rename-induced statement update

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

will be detected in the 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 the second pass. To cope with this, we collect all statement-based structural differences from Step 2 and verify whether these were again detected in Step 7. 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.

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 seven 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 by non-essential differences. We used our results to estimate how non-essential differences would interfere with the information measured by change-based approaches.

5.1 Set up

Table 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 by those change sets (Days). We studied the same systems as those analyzed in a prior study on change clusters [21] to help us assess the effects of non-essential differences on the results of client analyses. We studied all change sets that occur within the ranges reported in Table 1.

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* (LOC added, deleted, or modified) 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.

Table 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
Spring	1 Feb 2004	6 Feb 2006	736	3 627
Xerces	17 May 2001	8 Nov 2007	2 366	2 681
Total			6 463	24 459

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 PPA and CHANGEDISTILLER on their default settings.

5.2 Prevalence of Non-Essential Differences

Table 2 records the overall code churn for each target system (in kLOC). The table shows the total number of code lines that were deleted (-), inserted (+), or modified (~) for each system. It shows how many of the modified lines were caused by three major classes of non-essential differences detected by our approach: differences induced by renames (R), trivial keyword updates (K), and local variable refactorings (L). The "L" column aggregates local variable extractions, local variable renames, and trivial updates to local variable declared types. The combined non-essential line modifications are reported in the final column (Non Ess.). The per-

Table 2: Code Churn in Target Systems (in kLOC)

System	-	~	+	R	K	L	Non Ess.
Ant	113	35	301	6.8	.8	.4	8.0 (22.9%)
Azureus	49	95	108	2.6	.0	.1	2.7 (2.8%)
Hibernate	63	35	196	2.9	.0	.2	3.1 (8.9%)
JDT-Core	47	16	73	1.8	.6	.2	2.6 (16.3%)
JDT-UI	72	23	100	1.3	.0	.1	1.4 (6.1%)
Spring	43	27	126	3.8	.6	.2	4.6 (17.0%)
Xerces	62	15	196	1.0	.0	.1	1.1 (7.3%)
Total	449	246	1 100	20.2	2.0	1.3	23.5 (9.6%)

centages displayed in this column correspond to the proportion of all modified code lines (~) that were found to be non-essential.

Table 2 helped us derive the following observation:

Between 2.8% *and* 22.9% *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 246 kLOC were modified. We see that 23.5 (9.6%) of the total 246 modified kLOC were modified only by non-essential differences. Based on previous definitions of *total code churn* [16], we also see that only 23.5 (1.7%) of the overall (1 100 + 246 =) 1 346 *churned* kLOC were modified only by non-essential differences, because the large number of added code lines dwarfs the impact on existing, changed lines. This suggests that the kinds of non-essential differences studied in our investigation do not affect measures of *total code churn* (that include added and modified lines).

Table 2 also enabled us to infer the following property:

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

In particular, of the 23.5 non-essential kLOC reported in the table, 86% consisted of rename-induced statement updates, 9% of trivial keyword updates, and the remaining 5% of local variable refactorings. A further breakdown of the individual non-essential difference classes revealed that more than 99% of non-essential keyword updates consisted of trivial `this` keyword insertions and deletions, over 90% of all detected local variable refactorings consisted of local variable renames, and that almost no variable refactorings ($< 1\%$) involved trivial local variable type updates.

Non-Essential Method Updates

Table 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), and local variable updates (Local). 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 80 378 method updates across the target systems, 7 211 (9.0%) were non-essential.

Table 3: Method Updates in Target Systems

	Total	Non-Essential	R-Induced	Keyword	Local
Ant	17 792	2 759 (15.5%)	2 227	531	110
Azureus	8 731	229 (2.6%)	227	1	5
Hibernate	15 881	1 153 (7.3%)	1 136	6	52
JDT-Core	8 837	673 (7.6%)	542	133	98
JDT-UI	9 681	426 (4.4%)	424	0	13
Spring	11 047	1 715 (15.5%)	1 508	216	74
Xerces	8 409	256 (3.0%)	250	6	5
Total	80 378	7 211 (9.0%)	6 314	893	357

Table 4: Non-Essential Methods in Change Sets

System	Total	Non-Essential	R-Induced	Keyword	Local
Ant	2 579	283 (11.0%)	263	86	39
Azureus	2 866	65 (2.3%)	63	1	4
Hibernate	3 024	303 (10.0%)	295	6	35
JDT-Core	2 015	175 (8.7%)	153	33	45
JDT-UI	2 152	145 (6.7%)	143	0	14
Spring	2 400	454 (18.9%)	396	100	62
Xerces	2 037	77 (3.8%)	73	5	5
Total	17 073	1 502 (8.8%)	1 386	231	204

The table also enabled us to make the following observation:

In the individual systems analyzed, between 2.6% *and* 15.5% *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 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 non-essential difference (R-Induced), a keyword difference (Keyword), or a local variable refactoring (Local).

From the table, we see that out of 17 073 change sets found to modify at least one method, 1 502 (8.8%) 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 10.0% *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 2.6% of method updates within "small" change sets (e.g., those modifying 1 to 3 methods) were found to be non-essential. This ratio increases to 7.8% for "regular" change sets (e.g., those modifying 4 to 19 methods) and 14.2% in "large" change sets (e.g., those 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 high densities of non-essential method updates (14.2%), change sets modifying 20 or more methods also contained an overall majority (55.9%) 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 2.4% 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 a majority (79%) of cases. This result corroborates findings of a previous investigation by Murphy-Hill et al., which showed that developers often interleaved refactorings with other modifications [15]. 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 insertions).

5.3 Impact of Non-Essential Differences

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. [25] 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, 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 [25], 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 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, we also filter out recommendations with confidence lower than 0.1 and cap the number of recommendations at ten [25].

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 [25]. To compute these metrics, we replayed the change history intervals of our seven target systems (see Table 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.[6] 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. This produced two sets of *non-empty* ranked recommendation lists for 38 047 different seed methods. We found that the recommendations were different in 10 218 cases, or for 26.9% of those 38 047 seed methods where there was at least one recommendation. We then compared the quality of the recommendations for these 10 218 cases.

Table 5: Recommendation Quality

Setup	Tot Rec	Feedback	Prec	L3	Only Err
Reg	93 576	10 214	0.219	0.442	0.220
Prop	81 162	9 242	0.242	0.475	0.183

Our metrics allowed us to to make the following observation:

For those changed methods for which at least one recommendation was made by our analysis, removing non-essential method updates improved the overall precision of the recommendations by 10.5% *and decreased their recall by* 4.2%

Table 5 presents this observation in more detail. 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 3 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 $(.242/.219 \approx)$ 10.5% and its total number of helpful recommendations decreased from $(.219 \times 93576 =)$ 20 501 to $(.242 \times 81162 =)$19 631, or by around 4.2%. We also see that the proportion of changed method for which at least one helpful recommendation was found in the top three recommendations increased by $(.475/.442 \approx)$ 7.5% and that the proportion for which only erroneous recommendations were made decreased by $(.22/.183 \approx)$ 20.2%. 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.

5.4 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 non-essential method updates, rather than all reported non-essential differences, because erroneous classifications of method updates are more likely to have a negative influence on the representations used by change-based approaches than erroneous classifications of isolated statement updates.

To select change sets for a given system, we first sorted the system's change sets according to the number of non-essential method updates they contained. We then went down this list in descending

[6] 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 6: Characteristics of Selected Change Sets

System	CS	NEMs	CR	MR	FR	PR	VR
Ant	29	1 386	0	46	813	208	103
Azureus	4	113	2	28	3	2	31
Hibernate	33	580	30	134	52	88	24
JDT-Core	16	336	1	10	47	135	74
JDT-UI	23	213	29	57	14	42	28
Spring	51	857	44	240	134	469	43
Xerces	11	132	13	103	26	18	1
Total	167	3 617	119	618	1 089	962	304

Table 7: Precision of the Technique (in %)

System	NEMs	CR	MR	FR	PR	VR
Ant	99.9	n/a	95.6	99.9	100	90.3
Azureus	100	100	100	100	100	83.9
Hibernate	99.7	96.7	88.8	94.0	90.9	70.1
JDT-Core	99.1	100	60.0	95.7	100	89.1
JDT-UI	93.3	72.4	73.7	50.0	47.6	67.9
Spring	97.7	93.1	86.7	97.8	91.9	88.4
Xerces	97.7	100	98.1	96.2	94.4	100
Total	98.8	89.9	88.5	98.7	92.8	85.5

order and selected all change sets until we had accounted for *half* of all the non-essential method updates within the system. In this way, we limited our manual inspection to 167 change sets across the seven systems.

Table 6 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 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:

> DIFFCAT *correctly classified non-essential method updates* 98.8% *of the time.*

Table 7 presents the precision of our detected entity renames and non-essential method updates in more detail. The table displays the proportion of correct classifications for each of the results reported in Table 6. From the table, we see that the overall precision of our approach for rename detection ranges from 85.5% (variable renames) to 98.7% (field renames). The table also shows that our approach identified non-essential method updates within individual systems with a precision ranging from 93.3% 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.

5.5 Discussion

Non-Essential Differences

The true "essentiality" of code differences, method updates, and pairwise method associations is tied to the specific goals of individual change-based approaches. We believe that accounting for the kinds of non-essential method updates 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, redundant class casts, 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 2.6% and 15.5% 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 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.

Generalizability of the Results

Our investigation focused on seven 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. The systems we analyzed are all developed in association with major open-source software distributors (Spring, Apache, JBoss, and Eclipse), except for AZUREUS, the development of which is coordinated by a digital media technology company (Vuze). All analyzed systems included code commits from between 10 to 23 contributors, except for JDT CORE, which included commits from just six. The sizes of the investigated projects are in the order of between 100 and 500 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.

6. RELATED WORK

Our investigation of non-essential differences 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.

Differencing Tools

Our research is related to existing 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 align code elements between program versions [12]. In contrast, our goal is to identify changes between code elements and to also classify them in terms of their relevance to higher-level representations of development effort.

Our differencing tool builds specifically on CHANGEDISTILLER, a tool-supported differencing technique that identifies statement-level differences between ASTs [7]. Like other tools, CHANGEDISTILLER ignores whitespace-related differences and identifies documentation-related updates. Our approach extends its technique by using type resolution to detect non-essential differences.

Other tools have also focused on eliminating spurious textual differences between programs. 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 [17]. In contrast, we describe a more general category of change that includes rename-induced updates and other non-essential differences. We also detect such changes in individual change sets, not between snapshots.

Our use of partial program analysis [2] 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 [3]. However, instead of operating on differences between call graphs to detect call-change relations, we work with fine-grained modifications to detect non-essential differences occurring at the sub-statement level.

Code Clone Detection

Our work on detecting non-essential differences complements research on code clone detection. Similar to clone detection, our aim is to detect pairs of code fragments (e.g., methods) that are identical modulo 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 [1]. 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.

Higher-Level Structural Patterns

Our detection of non-essential differences complements existing approaches that summarize groups of low-level changes in terms of higher-level patterns. Such approaches include performing origin-analysis to detect method splits [9], refactoring detection tools [4], or research on identifying recurring bug fixes in change data [18]. Our work complements these approaches by classifying specific groups of low-level changes as non-essential differences. Furthermore, although we currently partly rely on CHANGEDISTILLER to detect rename refactorings, we could also extend our current detection of non-essential differences to incorporate renames detected by other tools. Other work with goals similar to our research includes Kim and Notkin's approach for discovering groups of low-level changes exhibiting logical high-level structural patterns [13]. The abstraction of changes used by their tool (LSDIFF) could also be used to help change-based approaches ignore groups of systematic updates involving only non-essential differences. However, LSDIFF currently processes only coarse-grained differences between programs and is implemented to handle program snapshots, not change sets. A detection of fine-grained type-resolved structural differences within change sets could enable LSDIFF to extend its rule generation to cover broader classes of changes and enable it to operate on individual developer commits.

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 [14] or affect program behavior [19]. Our differencing technique complements these approaches by identifying low-level changes that are unlikely to introduce bugs or require extensive re-testing.

Differential symbolic execution (DSE) also identifies a potentially unlimited number of classes of non-essential method updates by identifying updates that have no impact on a method's symbolic execution summary, e.g., trivial keyword insertions [19]. Alternatively, a detection of non-essential changes could help DSE ignore certain groups of modified methods.

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 [6]. 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, their 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-essential imposes stricter conditions on individual changes that are partly based on type bindings 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.

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 [20]. 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, including code cleanups [11]. Detecting non-essential changes within change sets could result in a more precise classification of code cleanups than one based on commit metadata alone.

7. SUMMARY

Numerous techniques involve mining change data captured in software archives to assist software engineering efforts. We described non-essential differences, or minor modifications stored in software archives that can cause inaccuracies in high-level interpretations of software development effort. We developed a differencing technique that detects non-essential differences in change histories and used our technique to conduct an empirical investigation of change data retrieved from seven open-source Java systems. Our evaluation found that between 2.6% and 15.5% of all method updates consisted entirely of non-essential modifications, and that these affected the association rules that can be mined from change data. These observations should be kept in mind when interpreting insights derived from change histories.

Acknowledgments

The authors thank Barthélémy Dagenais, Tristan Ratchford, and the anonymous reviewers for their helpful comments on earlier drafts of this paper. This work is funded by NSERC.

8. REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automated detection of refactorings in evolving components. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, pages 404–428, 2006.

[5] 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.

[6] 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.

[7] 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.

[8] H. 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.

[9] 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.

[10] 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.

[11] A. Hindle, D. M. German, M. W. G, 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

[17] I. Neamtiu, J.S. Foster, and M. Hicks. Understanding source code evolution using abstract syntax tree matching. *ACM SIGSOFT Software Engineering Notes*, 30(4):5, 2005.

[18] T.T. Nguyen, H.A. Nguyen, N.H. Pham, J. Al-Kofahi, and T.N. Nguyen. Recurring bug fixes in object-oriented programs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 315–324, 2010.

[19] 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.

[20] 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.

[21] 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.

[22] Marc J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, SE-1(4):364–370, 1975.

[23] 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.

[24] 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.

[25] 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.