Identifying Behavioural Implications of Source Code Changes

by

Abdullah El-Sayed

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2013

Abstract

The dynamic behaviour of a software system changes as a consequence of developer's static source code modifications. In this thesis, we improve upon a previous approach that combines static and dynamic analyses to categorize behavioural changes by greatly improving its accuracy through polymorphic mapping. We further refine the previous model by introducing a change-centric state transition model that captures the flow of call pairs among different partitions based on static and dynamic call graphs. We also extend the approach by incorporating complete dynamic call stacks into the analysis. Finally, we perform a longitudinal analysis of three software systems to categorize how they have dynamically evolved across 100 program versions.

In our evaluation, the polymorphic mapping algorithm decreased mismatches between the static and dynamic analyses by 53%. In particular, we decreased the mismatch by 71% in the most important category of changes from the developer's point of view. We found that developers introduce new behaviour more often than eliminating old behaviour. Our results show that developers are more likely to remove unexecuted/dead code than code that is executed dynamically. In terms of change types, we found that changes made to fix defects encountered the least inconsistent and unexpected behaviour, while changes made to add new functionality experienced the highest unexecuted behaviour. Finally, we argue that augmenting the dynamic analyses with call stacks provides useful information that helps developers analyze the implications of the call pairs highlighted by our analyses.

Acknowledgements

First, I would like to thank my supervisor Prof. Reid Holmes for his continuous and invaluable support throughout my research. His knowledge and guidance allowed me to successfully complete this thesis. I would also like to express my acknowledgments to the members of the SWAG group for their great feedback.

I am also grateful to my coworkers and friends at Waterloo, especially Khaled Ammar, Mohammed Sabri, and Mostafa Gaber for their insightful discussions and feedback. Last but not least, I would like to express my sincere appreciation to my family for their continuous motivation, support, and guidance.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Developers continually evolve software systems throughout their lifetimes to fix defects, add new features, and improve performance and design [31]. While making a source code modification, developers aim to implement the intended change correctly, and to make sure their change does not induce any unintended regressions. Therefore, after implementing a source code change, developers often try to ensure their change did not break the system in unintended ways (e.g., by executing a regression test suite). While failing a regression test suite can indicate that a change has caused unintended side effects, a test suite cannot ensure that no regression has taken place; in some cases, the behavioural impact of a change may only appear over time through manual testing or after a test suite is further altered to consider new test cases.

To tackle these challenges, Holmes and Notkin proposed the Inconsistency Inspector, an approach that partitions program call dependencies that influence software behaviour [16]. Given two versions of a program, the static and dynamic call graphs are extracted from each version, then call dependencies are partitioned based on their presence in each of the four graphs. Certain partitions contain calls that deserve investigation; for instance, a call that appears dynamically in the second version without any corresponding static change may occur because of a change to a configuration file. Another partition contains calls that are reported statically in both versions but executed dynamically in the second version, which may occur if a call is located in a condition block where the condition is satisfied in the second version. Finally, other partitions contain calls that represent normal behaviour and are unlikely to concern the developer; for example, if a developer adds a new method call and it gets captured statically and dynamically in the second version but not in the first version.

One of the major shortcomings of the Inconsistency Inspector was that it failed to consider type hierarchies and polymorphism when matching static and dynamic analysis; this prevented a large number of static changes to be properly mapped with their dynamic counterparts. While partitioning changes, the Inconsistency Inspector matches call dependencies between static and dynamic call graphs by comparing the exact signature of the caller and callee for each call pair. Since the tool does not consider type hierarchies during the mapping process, this leads to mismatched elements in the partitions. For instance, the static call graph may contain `Collection.add(int)`, but it would appear dynamically as `ArrayList.add(int)`; the tool would consider these as two different calls. This is problematic on a practical level: since the Inconsistency Inspector failed to combine even these simple calls, it made it difficult for developers to trust the correctness of the approach for more complicated scenarios. We address this major issue by implementing polymorphic mapping and utilizing type hierarchies when mapping static and dynamic call dependencies. Our approach reduced mismatch between static and dynamic analyses by 53% compared to the Inconsistency Inspector.

In addition to greatly improving the precision of this approach, this thesis further refines the model by introducing a state-transition model that captures the flow of call dependencies among different partitions based on static and dynamic call graphs. Compared to the previous model which is region-centric, our proposed model is change-centric which makes it easier for developers to understand the flow and impact of their changes. We further extend the Inconsistency Inspector, which considers call pairs only, by incorporating complete dynamic call stacks into the analysis. Furthermore, in order to produce more manageable and meaningful results for developers, we divide some partitions into subpartitions that represent calls of frequently occurring types.

While a variety of papers have described how software evolves statically over time (e.g., [13], [32], [23]), very little has been reported that describes the dynamic evolution of systems. We have investigated the dynamic impact of developer changes for 100 versions of three different systems. The primary intent of our evaluation was to address the following research questions: How do different source code changes affect the behaviour of a system? How are change characteristics related to software behaviour? How useful are call stacks, compared to call pairs, for behaviour analysis? to answer these questions, we investigated the impact of changes on software behaviour, quantitatively and qualitatively, over the three systems.

The primary contributions of this thesis are:

- A change-centric state-transition model that describes the flow of call dependencies among different partitions captured by static and dynamic call graphs.
- Improving the practical effectiveness of the Inconsistency Inspector by incorporating complete dynamic call stacks into the analysis and reducing mismatched elements in the results by implementing polymorphic mapping and utilizing type hierarchies.
- An investigation of the behavioural evolution of three software systems over 300 changes from two open source systems and an industrial system in order to identify the behavioural implications of source code changes.

The remainder of this thesis proceeds as follows. A concrete motivating scenario is provided in Chapter 2. Related work is covered in Chapter 3. Our proposed state-transition model and implementation details appear in Chapter 4. The evaluation and results are covered in Chapter 5. Chapter 6 provides discussion, threats to validity, and future work. Finally, our conclusion is provided in Chapter 7.

Chapter 2

Motivating Scenario

To motivate our work, we present a concrete scenario of the Inconsistency Inspector with our enhancements. As mentioned earlier, the approach requires two versions of a system: one before and one after a change. In this scenario, we choose two consecutive versions from JodaTime[1], an open source date and time java library; the two versions are `6b1b99` and `53eadf`.

The developer making this change intends to "support parsing of date-time zone IDs like Europe/London" as mentioned in the second commit status. The developer changes four classes listed in Figure 2.1a. After making the modifications, the developer runs our tool to compare JodaTime before and after the change. The tool analyzes the static and dynamic call graphs from both versions and forms partitions of call dependencies based on their presence in each of the four call graphs. In this scenario, we will focus on changes that represent divergences between the static and dynamic call pairs. We focus on these calls because they are likely to capture unforeseen behavioural changes.

Running the tool without our enhancements results in 46 call pairs with these properties, while running the tool with our enhancements results in 25 call pairs. This indicates that our polymorphic mapping algorithm reduced the mismatched elements by 45%, leading to a smaller and a more manageable set of call pairs for the developer to investigate.

[1] http://joda-time.sourceforge.net/

```
DateTimeFormatter
DateTimeFormatterBuilder
DateTimeParserBucket
TestDateTimeFormatterBuilder
```

(a) Modified classes in version two.

```
LenientChronology.withZone(DateTimeZone) → LenientChronology.withUTC()
```

(b) Dynamically detected, but statically non-obvious call pair.

```
public Chronology withZone(DateTimeZone zone) {
   ...
   if (zone == DateTimeZone.UTC) {
        return withUTC();
   }
   ...
}
```

(c) Source code of LenientChronology.withZone().

```
LenientChronology::withUTC()
LenientChronology::withZone(DateTimeZone)
DateTimeFormatter::parseInto(ReadWritableInstant, String, int)
ZoneInfoCompiler::parseTime(String)
...
```

(d) Identical part of stacks including withZone() where calls are ordered from bottom to top.

Figure 2.1: JodaTime example details.

Looking at the inconsistent partitions, the developer notices that three call pairs appear in classes that were unchanged in his commit. For illustrative purposes, we focus on one of these calls, shown in Figure 2.1b. This call appears in both static call graphs, indicating no change of source code in the `LenientChronology` class, but only executes dynamically in version two. This requires further investigation because when a developer makes a change he usually expects new call edges to appear in his changed classes; in this example, a call pair is executed dynamically in a class that was not modified.

To investigate this, the developer inspects the source code of the method `Lenient-Chronology.withZone()`, shown in Figure 2.1c, and notices that `withUTC()` is called when a condition is satisfied based on the parameter `zone`. The developer starts inspecting the dynamic call stack partitions; more specifically, the partition that includes new dynamic stacks that appear in version two. He finds that `LenientChronology.withZone()` is called in seven stacks. Among these stacks, the last part of the stack is identical, shown in Figure 2.1d, where `DateTimeFormatter.parseInto()` calls `withZone()`. Since the developer modified `DateTimeFormatter` in version two, he can analyze the changes to decide if this call chain is normal or problematic. Applying this approach to other call pairs allows the developer to detect and reason about unexpected behavioural changes.

Chapter 3

Related Work

In this chapter, we provide an overview of the Inconsistency Inspector. Then we discuss previous work in two related areas: software evolution and change impact analysis.

3.1 Inconsistency Inspector

As mentioned earlier, the Inconsistency Inspector was proposed to identify specific program call dependencies that influence software behaviour [16]; especially in situations where behavioural changes are not easily identified by testing techniques or manual inspection. The tool requires two versions of a system: one before and one after a change. The two versions are preferably, but not necessarily, consecutive. The first step in the approach is to generate four call graphs: a static call graph for each version (denoted as V1S and V2S), and a dynamic call graph for each version (denoted as V1D and V2D). Each call graph consists of call pairs between methods, and class and method declarations, and class type hierarchies. In our work, we extend the existing model with complete call stacks for the dynamic call graphs.

The static call graphs are generated using the Dependency Finder Java framework where external library code is not considered, which results in a graph close to what a developer may encounter in a manual code inspection. It is worth mentioning that, in a

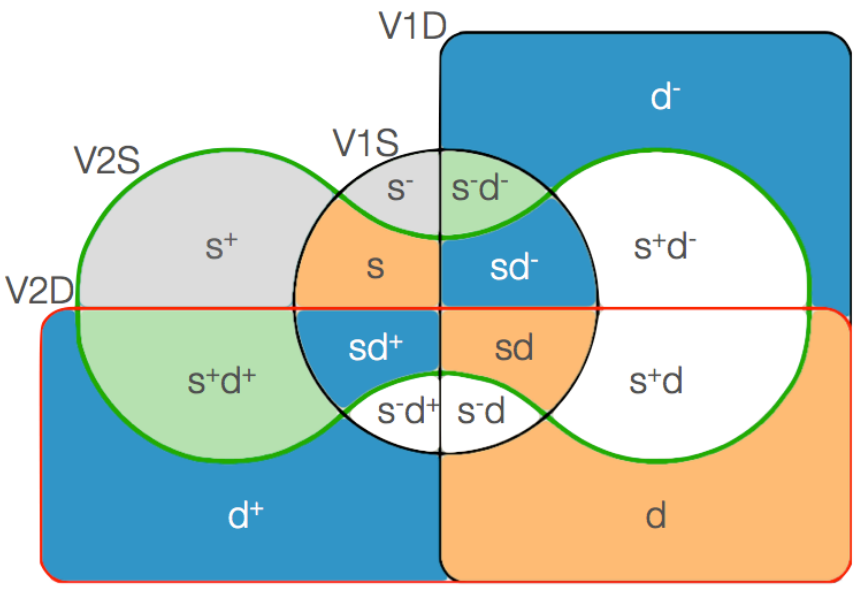

Figure 3.1: Inconsistency Inspector conceptual model.

static call graph, not all reported calls can be executed at run-time and not all calls that can arise at run-time are reported. For example, any code that depends on a condition, that is never satisfied, will be reported statically but not executed dynamically. Oppositely, a static call such as `Collections.add()` could invoke an `equals()` method during run-time for comparison with an existing object, but the call would not be reported statically.

The dynamic call graphs are generated using a custom program tracer. The dynamic graph is collected by running test suites or any arbitrary execution of a system. During system execution, the tracer maintains a call stack and creates a method call relation whenever a method is invoked.

From the four call graphs, the previous model considers all set intersections using a four-set Venn diagram, as shown in Figure 3.1. The circle represents the call pairs observed statically in the first version (V1S), and the barbell-shape represents the pairs observed statically in the second version (V2S). The vertical rectangle on the right represents the pairs observed dynamically in the first version (V1D), and the horizontal rectangle denotes the call pairs observed dynamically in the second version (V2D).

After generating the graphs, the Inconsistency Inspector forms partitions that contain call pairs based on their presence in the four call graphs. The partitions are described in the following notion:

- An `s` if the call pairs are statically observed in both versions.
- An `s+` if the call pairs are statically observed in the second version but not in the first version.
- An `s-` if the call pairs are statically observed in the first version but not in the second version.

Similarly, partitions containing at least one dynamically observed pair are marked with `d`, `d+`, or `d-`. The static property of a partition is described first, if any exists, followed by the dynamic property of the partition, if any exists. For instance, `s+` includes the pairs that are statically observed in the second version (V2S) but not in the first (V1S), and that were not dynamically observed in either version (V1D or V2D). The partition `s-d-` contains only pairs that are statically and dynamically observed in the first version (V1S and V1D) but not in the second version.

The conceptual model in Figure 3.1 is challenging for developers to interpret because the partitions are distributed arbitrarily in regions without illustrating the relations between these partitions. We propose a change-centric state-transition model that captures the flow of call dependencies among different partitions. Our model improves on this model in that it makes it easier for the developer to understand the flow of call pairs among different partitions, and therefore comprehend the behavioural categories of changes more easily, as will be shown later in the thesis.

3.2 Software Evolution

Evolution is an essential and critical trait of software systems [6]. Throughout its life cycle, a system will be continuously modified. As widely mentioned in the literature, software changes may lead to unintended, expensive, or even disastrous effects (e.g., [22, 8]). Thousands of computer-related risks have been documented by the Risks Digest[1] since 1985; where a large portion of these risks are traced to unintended consequences of changes. Based on an observational study of developers in a large software company, Ko et. al [17] found that feedback about the fidelity of changes is among the most-sought piece of information. In a similar study, aiming to identify questions programmers ask during software evolution tasks, Sillito et. al [30] found that developers are keenly interested in knowing the direct and total impact of their changes. Our approach aims to alleviate unintended consequences by helping developers identify the static and dynamic impact of their changes, and thus, build confidence in their changes.

According to a common classification of software changes by Swanson [31], software changes are one of four types: corrective, adaptive, perfective, or preventive. Corrective maintenance is applied to fix defects that appear after the software is released and used. Adaptive maintenance is performed in response to changes in the external environment and usually translates into new features. Perfective changes are changes that improve non-functional properties of the system such as increasing performance, eliminating inefficiency, and improving maintainability. Finally, preventive changes are changes that correct latent faults in software before they become effective faults. In our work, we aim to understand the behavioural implications of changes in light of this classification and how do different types of changes influence the system.

In terms of software changes factors, Purushothaman & Perry [24] performed an extensive study on a commercial software system to investigate the nature of small source code changes. They found that nearly 50% of changes are small changes. They also found that only 10% of changes altered a single line of code, and out of these less than 4% lead to faults. However, nearly 40% of changes intended to fix a fault lead to further faults. In terms of types of changes, they found that the majority of changes were adaptive and that nearly 10% of the changes were for perfective purposes. We aim to extend this study by analyzing the dynamic implications of source code changes.

A variety of papers described how systems evolve statically over time. Godfrey and Tu

[1]http://catless.ncl.ac.uk/Risks/

[13] investigated the evolution of the Linux operating system kernel based on several metrics such as number of lines of code, number of source files, and number of global functions and variables. They found that the Linux code base experienced linear growth as it became bigger in its latest stages. In a study of software evolution, Rysselberghe and Demeyer provided an approach that visualizes the change history in terms of changed files and the dates of changes [32]. The aim of their approach was to identify unstable components, consistent entities, and changes in team productivity. Mockus et. al [23] performed a study comparing open source development to commercial development. In their study, they investigated the development process of the Apache web server by quantifying code properties, developer participation, and problem resolution interval. We augment these studies by examining the dynamic evolution of software and investigating the behavioural implications of source code changes over the lifetime of a system.

3.3 Software Change Impact Analysis

Impact and change propagation are identified as influencing factors of software evolution [7]. The aim of change impact analysis, simply known as impact analysis, is to identify possible consequences and effects of program changes [2]. Impact analysis techniques are mainly divided into two classes: techniques that predict potential effects of changes before they are applied and techniques that measure and evaluate consequences of changes after they are made; our work falls into the latter category. Numerous algorithms have been proposed and utilized to perform impact analysis including program slicing (e.g., [36] [12]), dependence call graphs (e.g., [25] [34]) , execution traces (e.g., [18] [14]), and history mining (e.g., [37] [35]). We discuss relevant approaches to our work below.

3.3.1 Static Impact Analysis

Comparing call dependence graphs is a common approach to impact analysis. The dependence graphs used for such analysis can be static, dynamic, or a combination of both. A collection of graph comparison techniques relies on static dependence graphs to perform safe regression test algorithms: "most techniques select tests based on information about the code of the program and the modified version" [28]. These algorithms work on eliminating all tests from an original program test suite that cannot expose a fault in the modified software. Many variants of these algorithms have been analyzed by Rothermel & Harrold [28], and a meta-analysis of empirical results is available as well [11]. Badri et. al proposed a model based on control call graphs of static analysis taking into consideration the decision and conditional points of a program, their approach can be used for safe regression testing as well [4]. Our approach augments existing regression testing approaches by integrating dynamic analysis and capturing more behavioural data which provides further analysis despite of an assertion failure in a regression test.

Approaches using static analysis only are considered to be safe as they consider all possible impact sets of the system; however, regardless of the algorithm used, these techniques can only distinguish partitions including static call pairs. Pure dynamic partitions (such as `d`, `d+`, and `d-`) cannot be inferred using these techniques. For instance, a dynamic call that appears in the new version of a program cannot be identified. Also, a developer will not be able to detect behaviour similar to the one captured in our motivating scenario using static analysis only.

3.3.2 Dynamic Impact Analysis

Several approaches rely on dynamic executions traces to form their analysis. PathImpact, a path-based technique that relies on instrumentation to collect dynamic data from a running system [19], records multiple execution traces to calculate change impact sets. Given a set of changes, PathImpact finds all methods that execute after a change and consider it to be affected by the change. Apiwattanapong et. al extended this approach by analyzing partial traces that are executed after a change instead of complete traces which reduces the cost of the algorithm [1]. Another technique extends the path-based algorithm by performing its analysis completely during program execution to alleviate the need of producing a whole program path trace [5]. Rohatgi et al. presented an approach that extracts features by comparing dynamic traces then ranking the returned components based on their static relationships to each other [27]. Also, Eisenbarth et al. proposed a feature location approach that uses static and dynamic analysis where dynamic traces are generated based on a set of scenarios, then formal concept analysis are applied to the traces to determine the relation between features [9].

3.3.3 Hybrid Impact Analysis

Another class of techniques utilizes static and dynamic graphs in their analysis. Lhotak presented a call graph difference tool that compares the static graph of a program to a dynamic graph of the same version and provides the calls ranked by likelihood of causing a difference [21]. Other approaches execute two distinct test suites across a single program. The Tripoli system [29] compares two random executions of a system and determines their coverage differences assuming that the source code remains unchanged. Wilde and Scully proposed an approach to identify parts of a program that implement a particular feature by exercising it in the first test suite but not the second [33]. Eisenberg and de Volder extended this approach to relax the explicit requirement of exhibiting and non-exhibiting test suites [10]. A similar approach was used to identify programs that might be susceptible to problems such as Y2k [26]. This type of approaches that uses two dynamic graphs and one static graph can infer up to eight partitions. However, purely static partitions: `s`, `s-`, or `s+`, cannot be distinguished. In some cases, these partitions may prove useful for the developer to investigate. For instance, if an edge is expected to appear in `s+d+` but appears in `s+` only then the developer needs to investigate the change.

3.3.4 Root Cause Analysis

Chianti, a change impact analysis tool for Java [25], reports potentially affected regression tests by analyzing two versions of a program in terms of atomic changes. The aim of the tool is to identify changes that causes a particular test case to fail. Another approach that aims to identify root cause analysis was proposed with semantic-aware trace differencing to identify precise and useful details about the underlying cause for a regression [15]. Babenko et. al proposed the AVA approach [3], which extends the concept of root cause analysis to include the capability of reasoning to differentiate between passing and failing tests. In contrast, our approach does not try to determine the root cause behind any change; instead, we provide the developer with a manageable set of interesting behavioural changes located in meaningful partitions.

Chapter 4

Implementation

As mentioned earlier in Section 3.1, we improve the Inconsistency Inspector by proposing a change-centric state-transition model that makes it easier for developers to understand behavioural changes. We also greatly improve matching static and dynamic calls by implementing polymorphic mapping and utilizing type hierarchies. Finally, we incorporate complete dynamic call stacks into the analysis. In the following sections, we provide an overview of our proposed state-transition model, our polymorphic mapping algorithm, and dynamic call stack analysis.

4.1 State-transition Model

Our change-centric state-transition model, shown in Figure 4.1, captures the flow of call pairs among different partitions based on static and dynamic call graphs analysis. As shown in the model, any call pair added to the system will move from the empty state to one of the transitional partitions, represented as arrows, first. Then, if the call pair persists in the next version, it will appear in one of the unchanged partitions represented as states: `s`, `d`, or `sd`. For instance, in order for a call pair to appear in `d`, it has to appear first in `d+`, then if it persists in the next version of the program, it will appear in `d`. Similarly, there are two ways to reach the partition `s`: either a new call pair appears in `s+` then persists in

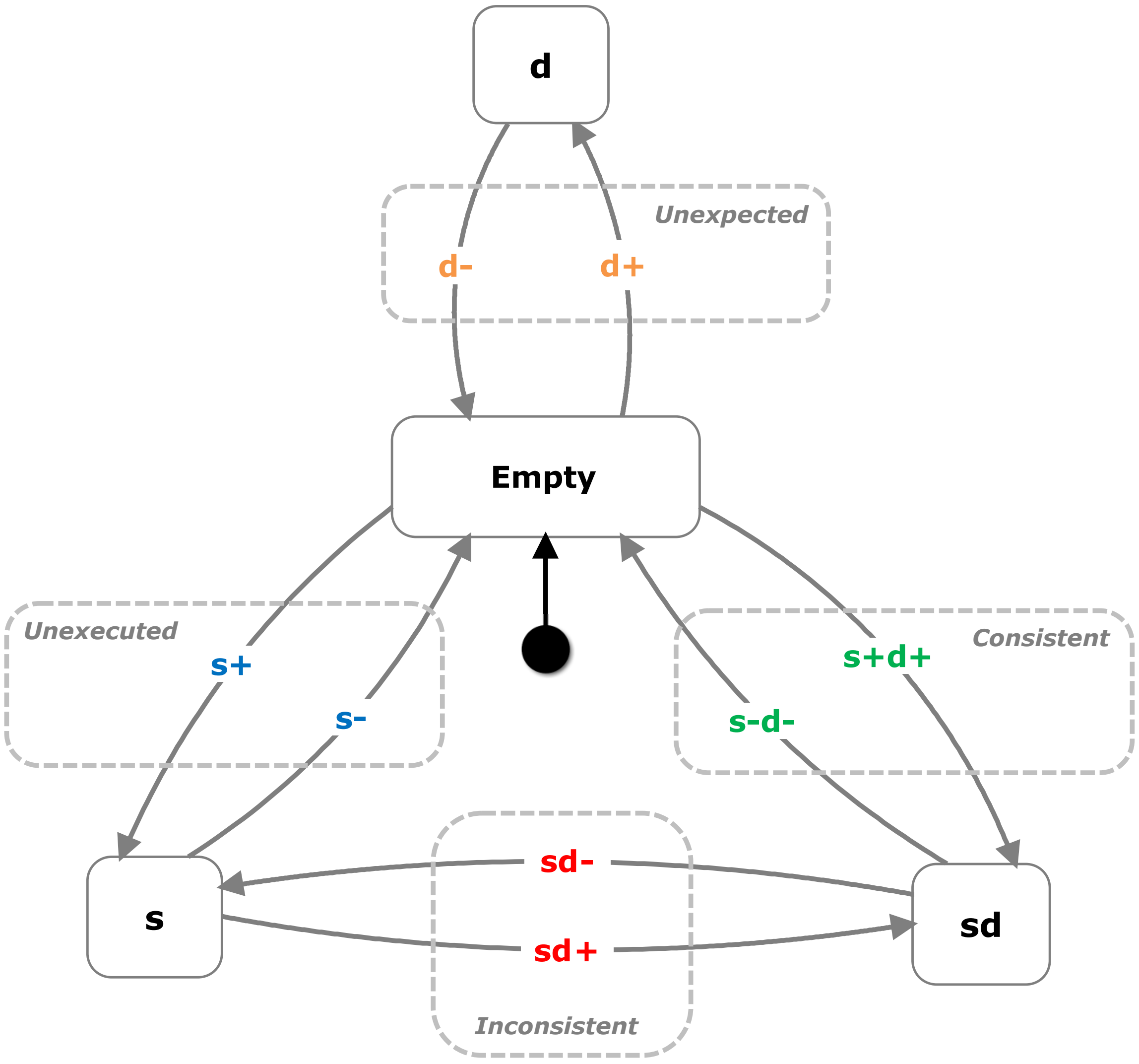

Figure 4.1: State-transition model capturing the flow of call pairs among partitions.

the next version so it appears in `s`, or a pair that used to be in `sd` moves through `sd-` then falls into `s`.

The state `sd` denotes the partition of calls that are observed statically and dynamically in both versions. There are two ways to reach this state: adding a new call pair that gets observed statically and dynamically, appearing in `s+d+`, then persisting in the next version and moving into `sd`, or dynamically exercising a pair that used to be in `s`, moving it through `sd+`, then into `sd` in the following version.

In the previous work [16], the partitions were grouped into five categories arbitrarily, meaning that there was no direct relation between the categories and the conceptual model. Furthermore, some of the categories were impossible to occur. We introduce a meaningful categorization of the partitions based on their existence in our model. We group every two adjacent transitional partitions together, and the partitions represented as states (`s`, `d`, and `sd`) together. Grouping partitions in this way makes it easier for the developer to remember the implications of each category, as they are easily recognized from the state-transition model. We describe our categories in the following.

4.1.1 Unchanged Partitions

The unchanged partitions, represented as states in our model, are `s`, `d`, and `sd`. These partitions contain unchanged call pairs among the two versions. Call pairs that are persistent statically, dynamically, or both before and after a change are highly unlikely to be surprising for the developer. Based on our results, the majority of call dependencies are located in these partitions.

It is worth mentioning that once a call pair reaches one of the unchanged partitions, it persists there through the next versions of the system unless it gets transferred to another partition through one of the transitional partitions shown in our model. For instance, a call pair that appears in `s+` in version four will appear in `s` in version five, and if not removed through `s-` in a future version it will persist in `s`.

4.1.2 Consistent Partitions

The consistent partitions, labeled in green, are the transitional partitions `s+d+` and `s-d-`. These partitions are coherent and consistent, statically and dynamically, among the two versions. A developer making a modification will examine the consistent partitions to ensure that added calls or removed calls are consistent and are performing as expected. A call pair appearing in `s+d+` implies that it was added statically and executed dynamically as well. Oppositely, `s-d-` contains calls that are removed statically and are no longer executed. As shown in our model, these partitions represent the transition of call pairs between the empty state and the unchanged `sd` partition. Call pairs appearing in `s+d+` will be located in `sd` in the next version and call pairs in `s-d-` will be unavailable in the following version.

4.1.3 Unexecuted Partitions

The unexecuted partitions, labeled in blue, are the transitional partitions `s+` and `s-`. These partitions represent transitions between the empty state and the unchanged partition `s`. A call pair appearing in `s+` means that it was added statically but is not observed dynamically in either version. This could arise if, for instance, a call is added inside a method that is never called. As a result, the call will not execute dynamically and will appear in `s+` and not `s+d+`. A call pair in `s-` means that it was removed statically in the new version and it is not exercised dynamically in either version. An example from JodaTime is shown in Figure 4.2, where the red lines indicate removed code after the change and the call pairs that appear in `s-` are shown; `<init>` indicates a call to a constructor. The calls appear in `s-` as they were not executed before the change, due to `iChronology` not being null, and are removed after the change. This category, along with the consistent category, provides useful indications for the developer to know if the system is running as intended. After making a modification, a developer would check the unexecuted partitions to decide whether the unexecuted calls are normal or they should appear in the consistent partitions.

In code:

```
Object readResolve() {
- if (iChronology == null) {
-    return new LocalDateTime(iLocalMillis, ISOChronology.getInstanceUTC());
- }
...
}
```

In s-:

```
LocalDateTime.readResolve() → ISOChronology.getInstanceUTC()
LocalDateTime.readResolve() → LocalDateTime<init>(long, Chronology)
```

Figure 4.2: Example of unexecuted call pairs in s-.

4.1.4 Unexpected Partitions

The unexpected partitions, labeled in orange in our model, are the transitional partitions `d+` and `d-`. This category contains call pairs that are not reported statically but are observed during run-time, indicating an unexpected dynamic behaviour. These partitions are worth investigating by the developer as they are the most likely to capture unforeseen behavioural changes. The partition `d+` contains call pairs that are observed dynamically after the change and not observed statically before or after the change. In contrast, `d-` contains call pairs that are no longer observed dynamically although there were no corresponding static modifications. Call pairs may appear in `d+` or `d-` because of environmental changes — changing a configuration file, or a non-source file in general. Another collection of calls that appear in unexpected partitions are call backs from external libraries; these calls are not reported statically because they originate from external libraries. For instance, as shown in Figure 4.3, this is a call from the `java.util.Map` class to an internal overridden `hashCode()` method in the internal class `CachedDateTimeZone`.

```
java.util.Map.get(java.lang.Object) → CachedDateTimeZone.hashCode()
```

Figure 4.3: Sample of a call back in d+.

4.1.5 Inconsistent Partitions

This category contains partitions that represent divergences between statically and dynamically observed pairs. The inconsistent partitions, labeled in red, are the transitional partitions: `sd+` and `sd-`. The calls appearing in these partitions are worth investigation as they are very difficult to detect by manual code inspection. More importantly, from a developer's perspective, the inconsistent partitions often capture behavioural changes occurring in classes that were not modified in a change. The partition `sd+` represents calls that are present statically in both versions but were exercised dynamically after the change only. In contrast, a call pair appearing in `sd-` indicates that the pair is no longer exercised dynamically but is still persistent statically. An example of an inconsistent call pair is shown in our motivating scenario in Chapter 2 where `withUTC()`, a method call inside a condition block, is executed because the condition is satisfied after the change. As a result, a call pair appears in `sd+` as shown Figure 4.4.

```
LenientChronology.withZone(DateTimeZone) → LenientChronology.withUTC()
```

Figure 4.4: Sample of a call pair in sd+.

4.2 Polymorphic Mapping

The Inconsistency Inspector failed to account for polymorphism, this meant that large numbers of dynamic calls were not matched with their static counterparts. This is especially significant for the unexpected partitions as it meant that, for example, many expected `s+d+` calls were being identified as two separate calls (unexecuted `s+` and an unexpected `d+`). For instance, the static call graph may contain `Collection.add(Object)`, but it would appear dynamically as `ArrayList.add(Object)`; the tool would consider these as two different calls. Assuming these two calls appear after a change, they will appear separately in `s+` and `d+` even though they should be matched to one call in `s+d+`.

Our key enhancement to the Inconsistency Inspector was to implement polymorphic mapping to reduce mismatch between static and dynamic analysis. After the partitioning phase, we map mismatched call pairs between partitions correspondingly: comparing `s+` with `d+` and mapping mismatched elements to `s+d+`, mapping mismatches from `s` and `d` to `sd`, .. etc. Our algorithm works as follow: first, we iterate through the static call pairs and compare each pair to dynamic call pairs with identical method names. If the classes of the caller and callee are identical or related by inheritance, we compare the number and types of parameters. The types of parameters are compared because it is possible to encounter polymorphism in parameters as well i. e., `new Vector(ArrayList)` and `new Vector(Collection)`. If the parameter types are matched, we map the call pair to the correct corresponding partition e.g., from `s+` and `d+` to `s+d+`. To clarify this, we provide an example from JodaTime in Figure 4.5 for two call pairs that appear in `s+` and `d+`. Given that the caller method is identical in both calls, and `FixedDateTimeZone` extends `DateTimeZone`, the call pairs are matched and moved to `s+d+` as a single call pair.

In d+:

```
TestDateTimeZone.testTimeZoneConversion() → FixedDateTimeZone.getOffset(long)
```

In s+:

```
TestDateTimeZone.testTimeZoneConversion() → DateTimeZone.getOffset(long)
```

Mapped to s+d+ as:

```
TestDateTimeZone.testTimeZoneConversion() → FixedDateTimeZone.getOffset(long)
```

Figure 4.5: Polymorphic mapping example.

To compare types, we use the type hierarchies data extracted along with the static call graph. Since it is possible for a class to be mapped to a super parent, in some cases, we need to iterate recursively through parents to find a match. Furthermore, we check if there is a mutual parent between classes when comparing static and dynamic pairs. If the static and dynamic method names are identical and the classes share a mutual parent or extend a common interface, the calls are considered to be similar. For instance, if class X is a child of Y and class Z is a child of Y as well, and Y has a method foo(), a static call such as objectX.foo() will be mapped to a dynamic call such as objectZ.foo() as they share a mutual parent. A concrete example is provided in Figure 4.6, where `AbstractConverter` and `PartialConvertor` both extend `Convertor`. In this case, these two calls are considered to be similar.

Dynamically:

`AbstractConverter.getChronology(Object, Chronology)`

Statically:

`PartialConverter.getChronology(Object, Chronology)`

Figure 4.6: Mutual parent example.

While examining the dynamic call graphs, we found direct calls from constructors to super constructors of classes that are two levels higher or more in the hierarchy. However, looking into the source code, we found that these constructors are not connected directly. For instance, if class A inherits from B which inherits from C, in the dynamic call graph, we would find a direct call from the constructor of A to the constructor of C; while statically the constructor of A calls constructor of B which calls constructor of C. We added some heuristics to detect and handle this case by matching the static and dynamic constructor calls together. A concrete example is shown in Figure 4.7 where the multiple static call pairs and the single dynamic call pair are shown; `<init>` indicates a constructor call.

Multiple static calls:

```
TimeOfDay.<init>(int, int) → TimeOfDay.<init>(...)
TimeOfDay.<init>(...)  → BasePartial.<init>(...)
org.joda.time.base.BasePartial.<init>(...)  → AbstractPartial.<init>()
```

Single dynamic call:

```
TimeOfDay.<init>(int, int) → AbstractPartial<init>()
```

Figure 4.7: Constructor calls example.

4.3 Forming Subpartitions

While testing the tool and examining the results, we noticed that there are frequently occurring types of calls that appear in the three exclusively dynamic partitions: `d`, `d+`, and `d-`. In order to make these partitions easier to understand, we divided each of the three partitions further into subpartitions that capture common types of calls. The subpartitions are as follow:

Call backs from JDK methods (cbjdk): This subpartition contains calls where the caller is a method from an external class from the `java.*` package, and the calle is an overriden JDK method in an internal class including methods such as `toString(), hashCode(), equals(), clone()`, and `finalize()`. For instance, a call that originates from `java.lang.Object.toString()` to `org.joda.time.LocalDate.toString()`.

Call backs (cb): This subpartition contains calls from external classes to internal methods other than the JDK methods; these calls originate from external libraries. For instance, a call that originates from `org.apache.cxf.jaxws.JaxWsProxyFactoryBean.create()` to the internal method `JuliToLog4jHandler.publish(LogRecord)`.

JUnit calls (junit): This subpartition contains calls originating from junit classes to methods from internal classes. For instance, a call from `junit.framework.TestSuite.-<init>(java.lang.Class)` to `org.joda.time.TestLocalDate.<int>()`.

4.4 Call Stack Analysis

Another main contribution in our work is extending the Inconsistency Inspector by incorporating dynamic call stacks into the analysis. We believe that call stacks provide additional information that can help the developer understand the behavioural changes in a program by analyzing the program paths that appear or disappear after a modification. In addition to the partitions mentioned earlier, we introduce three new call stacks partitions: `stack:d+`, `stack:d-`, and `stack:d`. The `stack:d+` partition contains call stacks that appear in version two only, `stack:d-` contains call stacks that appear in version one only, and finally, `stack:d` contains call stacks that appear in both versions.

```
1:  void methodA () {
2:    int x;
3:    x = methodB(true);
4:    x = methodB(false);
5:    x = methodB(true);
6:  }
7:
8:  int methodB (boolean flag) {
9:     if (flag) return methodC();
10:    return 5;
11:  }
12:
13:  int methodC () {
14:    return 4;
15:  }
```

Figure 4.8: Call stacks code example.

To implement this feature, we modified the dynamic tracer to store unique call stacks. We consider a call stack unique – and add it to our set of call stacks – once there is a pop operation detected after a push. To illustrate this, we provide a code snippet in Figure 4.8, where the execution starts from `methodA`. In line 3, `methodB` will be invoked and pushed to the stack; since the flag in `methodB` is true, the condition is satisfied and `methodC` will be pushed to the current stack. When `methodC` is executed, and before it is popped out of the stack, we check the previous operation; since the previous operation is a push, we add the current stack to our set. Then `methodB` will be popped and the stack is ignored as it is a pop after a pop. Next, `methodB` will be invoked and pushed again to the stack from line 4. Since the condition is not satisfied this time, `methodB` will be popped, but before it is popped, we add the current call stack to our set of stacks. For line 5, the same stack of line 3 will be generated but it will not be added to the set because we ignore duplicates. The final set of calls stacks is shown in Figure 4.9, where the class name is `Example`.

We store call stacks in this way to capture the unique program paths taken at runtime

Stack1:

```
Example::methodC()
Example::methodB(boolean)
Example::methodA()
```

Stack2:

```
Example::methodB(boolean)
Example::methodA()
```

Figure 4.9: Set of unique call stacks.

instead of considering the most dominant or largest stacks only.

To clarify the difference between call pairs and call stack analysis, we assume that the developer changes the code by removing line 4. This will result in removing Stack2 from the set, and therefore it will appear in the `stack:d-` partition when running our tool. On the contrary, the call pairs partitions will not change since the pair from `methodA()` to `methodB()` is still observed in line 3 and line 5. This is an example where call stacks may expose behavioural changes that cannot be inferred by call pairs analysis.

Chapter 5

Evaluation

Our evaluation sought to gain insight into the following three research questions:

- **RQ-1:** How do static source code changes impact the dynamic behaviour of a system?
- **RQ-2:** How are change characteristics and code attributes related to behaviour?
- **RQ-3:** How useful are call stacks, compared to call pairs, for behaviour analysis?

We evaluate our approach by applying it to two existing open source systems and an industrial system. The primary intent of our evaluation is to identify the behavioural implications of software changes by integrating static and dynamic analysis. We aim to understand the flow of call pairs and stacks among different partitions with respect to our model, and the reasons that cause certain call pairs to appear in certain partitions. In terms of the tool, we intend to evaluate our mismatch reduction algorithm by comparing results before and after the polymorphic mapping phase.

The two open source systems we evaluated are JodaTime[1] and Apache POI[2]. JodaTime is a date and time library for Java that improves upon the JDK date library. Apache POI is a Java library that provides APIs for manipulating files in Microsoft Office formats, such

[1] http://joda-time.sourceforge.net/
[2] http://poi.apache.org/

as Word, PowerPoint and Excel. Both systems are actively maintained and contain a set of test suites; we use the test suites to observe the dynamic call graphs in our analysis. In addition, both systems are available on open repositories which allows us to access past versions at per-commit granularity.

The industrial system we evaluated is a widely-used online marketing platform. The system undergoes active development using an agile methodology with delivered milestones after every three-week development sprints. The system heavily relies on external libraries and utilizes various development techniques such as mock objects, aspects, and parallelization. By considering an industrial system in our evaluation, we aim to understand the behavioural attributes of changes in industrial applications in contrast to open source libraries. Basic information about the evaluated systems is provided in Table 5.1.

System	**KLOC**	**# Commits**	**# Tests**
JodaTime	160	1575	3908
Apache POI	185	5143	4280
Industrial	184	31838	2570

Table 5.1: Evaluated systems details, indicating size, # of commits, and # of tests.

5.1 Methodology

We applied our approach to the 101 most recent versions with source code changes for each open source system. We only consider versions with source code changes because documentation changes have no impact on behaviour, and therefore result in empty partitions. Since the tool analyzes pairs of versions, considering 101 versions results in 100 entries for each system. For the industrial system, we had access to past versions in the form of nightly builds rather than commits, where each build includes a single or multiple number of commits. In order to perform a fair evaluation, we considered the latest builds that cover 101 commits with source code changes.

For each version in each system, we extract the static and dynamic call graphs using the techniques mentioned earlier; the dynamic call graph is collected by running the entire test suite of a system. Next, we run the tool to compare the versions and form partitions of call pairs and stacks. After running the tool between each two consecutive versions, we run our polymorphic mapping algorithm to reduce mismatched elements. As mentioned earlier, we detect mismatches between purely static partitions (`s`, `s+`, `s-`) and purely dynamic partitions (`d`, `d+`, `d-`), then we move the mismatched call pair to the proper corresponding partition. For instance, a mismatched call pair detected in `s` and `d+` will be moved to `sd+`.

Additionally, for each version in the open source systems, we extract the commit description, number of files changed, number of line insertions, and number of line deletions. Then we classify each change type according to a common classification of software changes by Swanson [31], where a change is one of four types: corrective, adaptive, perfective, or preventive. Corrective maintenance is applied to fix defects that appear after the software is released and used. Adaptive maintenance is performed in response to changes in the external environment and usually translates into new features. Perfective changes are changes that improve nonfunctional properties of the system such as increasing performance, eliminating inefficiency, and improving maintainability. Finally, preventive changes are changes that correct latent faults in software before they become effective faults.

5.2 Quantitative Results

Out of the 320 most recent commits of JodaTime, we found 101 commits with source code changes. As for Apache POI, we found 101 source code commits out of the latest 354 commits. For the industrial system, to cover 101 commits, we considered 20 nightly builds containing source code changes. In the following, we provide the quantitative results of our evaluation: first, we evaluate our polymorphic mapping algorithm, then we discuss the distribution of partitions and change charecteristics.

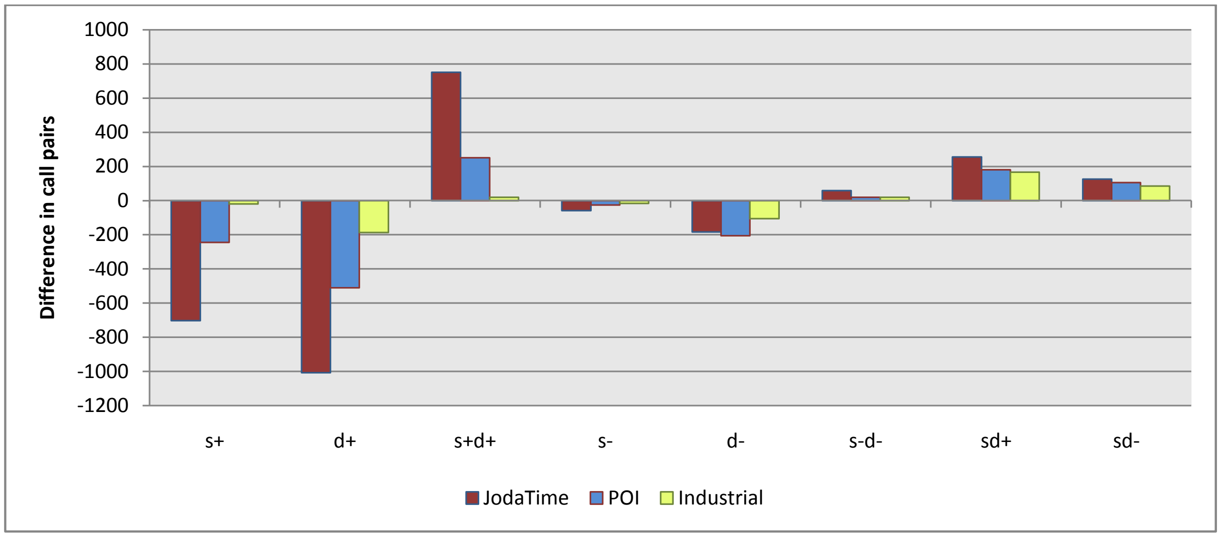

Figure 5.1: Mismatch reduction in partitions.

5.2.1 Impact of Polymorphic Mapping

We compared the partitions before and after the polymorphic mapping phase. Over the 100 version pairs of JodaTime, our polymorphic mapping algorithm decreased mismatches by 80% in purely static and purely dynamic partitions (from a total of 3186k call pairs to 636k). In POI, mismatches were reduced by 50% in purely static and dynamic partitions (3194k call pairs to 1579k). As for the industrial system, we reduced mismatch by 29% in the same partitions, which possibly indicates less usage of type hierarchies. Overall, the system size (total number of call pairs) of JodaTime decreased by 19%, POI decreased by 12%, and the industrial system decreased by 3%.

The changes in size of key partitions after polymorphic mapping is shown in Figure 5.1. The unchanged partitions (`s`, `d`, and `sd`) are not shown as their sizes are very large compared to other partitions, and they are not of developers interest. As shown in the figure, there is an evident reduction in the size of the purely static and dynamic partitions, especially `s+` and `d+`. Since the tool moves mismatched elements to the correct partitions, this leads to an increase in the size of the partitions: `sd`, `sd+`, `sd-`, `s+d+`, and `s-d-`. For instance, given that the class `ISOChronology` is a child of `Chronology` in JodaTime, if

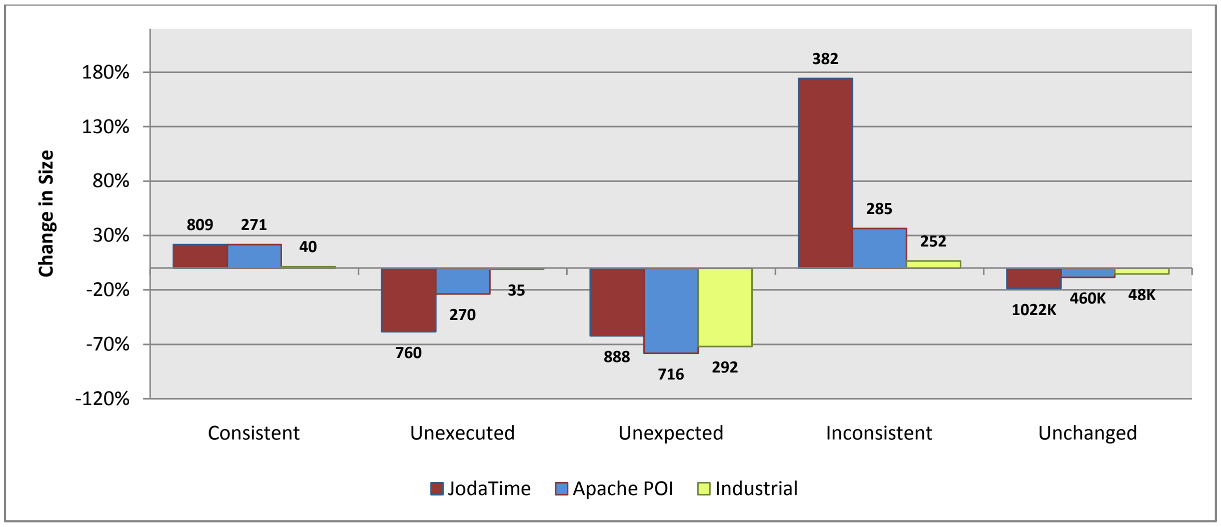

Figure 5.2: Mismatch reduction in categories.

`ISOChronology.withZone()` is reported in `d+` and `Chronology.withZone()` is reported in `s`, they will be matched and moved to `sd+` as the single call `ISOChronology.withZone()`. The reason why the number of removed elements is higher than added elements is because a mismatch results in two separate call pairs that, when matched, become a single pair. More importantly, it is possible that a single static call pair gets mapped to multiple dynamic call pairs.

In terms of partition categories, the percentages of size changes of categories due to mismatch reduction are shown in Figure 5.2 for the three systems. The number of added or removed elements are shown for each bar. Notably, there is an evident reduction in the unexpected partitions, which contains calls that are worth investigating by the developer. Even though the increase percentage in the inconsistent category in JodaTime is high (173%), the number of added elements (382) are significantly less than the eliminated elements, for instance, in the unexpected category (888). And even though the reduction percentage in the unchanged partitions may seem low, the number of removed elements are the highest among the partitions.

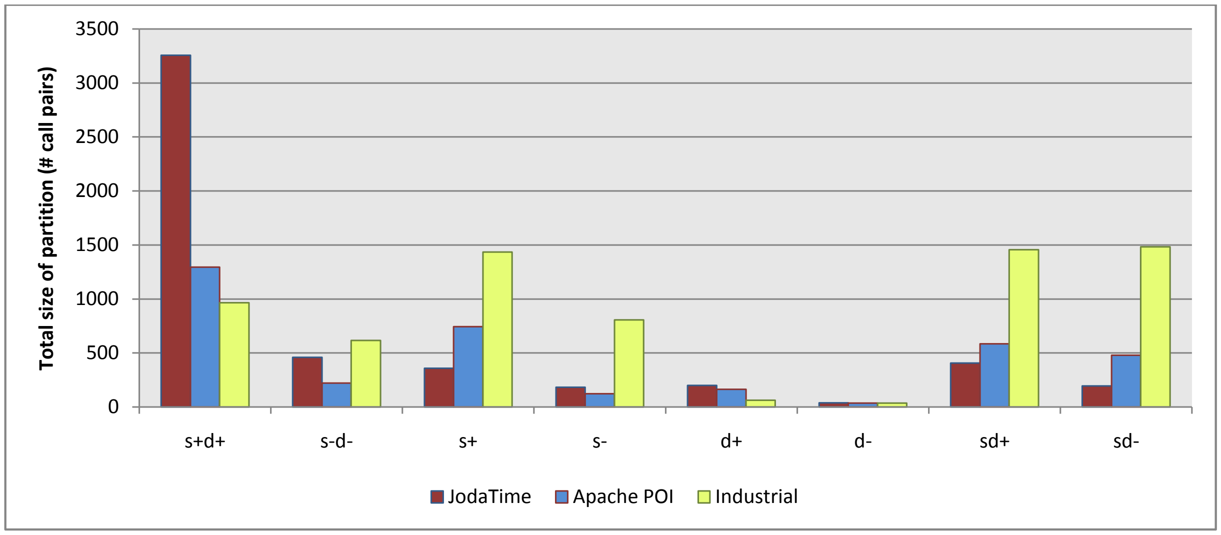

Figure 5.3: Distribution of partitions.

5.2.2 Distribution of Partitions

The total number of call pairs in the key partitions, after the polymorphic mapping phase, is shown in Figure 5.3. Since the unchanged partitions (`s`, `d`, `sd`) were nonempty in all versions and contained the largest amount of call pairs as expected, they are not included in the the figure. As shown, the unexpected partitions (`d+` and `d-`) are the lowest in the three systems. The consistent partitions (`s+d+` and `s-d-`) are relatively high in the three systems, indicating that the majority of developers changes are statically and dynamically coherent. Noticeably, for the industrial system, the unexecuted partition `s+` is higher than the consistent partitions, possibly because developers are adding code without testing it. Also, it is evident that the inconsistent partitions are almost as high as the consistent partitions in the industrial system.

Furthermore, more information about the key partitions is provided in Table 5.2, indicating the percent of versions where a partition is nonempty and the average size of each partition among nonempty versions. It is apparent from the table that partitions adding relationships (e.g., with a `s+` or `d+`) appear in more versions than those that remove relationships. The average size and number of nonempty versions of `s+d+` is higher than

	JodaTime		**Apache POI**		**Industrial**	
Partition	**Nonemp.**	**Avg.**	**Nonemp.**	**Avg.**	**Nonemp.**	**Avg.**
Consistent						
s+d+	87%	54	30%	7	68%	88
s-d-	44%	10	24%	7	56%	63
Unexecuted						
s+	53%	7	59%	6	70%	115
s-	19%	10	22%	13	62%	66
Unexpected						
d+	26%	8	30%	5	41%	6
d-	10%	4	14%	2	31%	4
Inconsistent						
sd+	40%	10	44%	13	80%	78
sd-	17%	11	26%	18	72%	81
Call stacks						
stack:d+	91%	1040	72%	1160	-	-
stack:d-	48%	469	54%	710	-	-

Table 5.2: Key partitions information.

`s-d-` among the three systems, indicating that new consistent behaviour arises more often than the removal of consistent old behaviour. Notably in all systems, `s+` appears in more versions than `s-`. However, in the open source systems, the average size of `s-` is greater than s+, whereas in the industrial system the average size of `s+` is greater.

In terms of call stacks, the `stack:d+` partition was nonempty in more versions than `stack:d-` for both open source systems; call stacks were not available for the industrial system as a previous version of the dynamic tracer was used to extract the dynamic graphs. Noticeably, the average number of stacks in `stack:d+` partition is 1040 stacks for JodaTime and 1160 for POI, which indicates a large set for the developer to investigate. Similarly, the average number of stacks in `stack:d-` is high as well.

Distribution of Sub-Partitions

As mentioned earlier, we divide the purely dynamic partitions (`d`, `d+`, and `d-`) into subpartitions that contain frequently occurring types of calls. The distribution of the `d+` subpartitions in the three systems is shown in Figure 5.4, and the distribution of `d-` subpartitions is shown in Figure 5.5. As shown in the figures, the call_backs subpartition is notably larger in the industrial system than the other systems; this is due to the heavy usage of external libraries in the industrial code base as opposed to libraries such as JodaTime and POI where external libraries are not used that often. It is also apparent from the figures that the `cbjdk` partitions are greater in the libraries than the industrial system, since libraries introduce new types that require custom overridden JDK methods such as `equals()` and `hashCode()`.

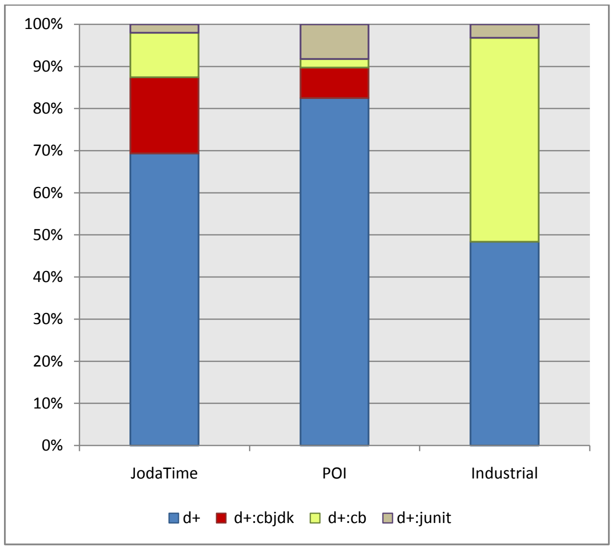

Figure 5.4: Subpartitions of d+.

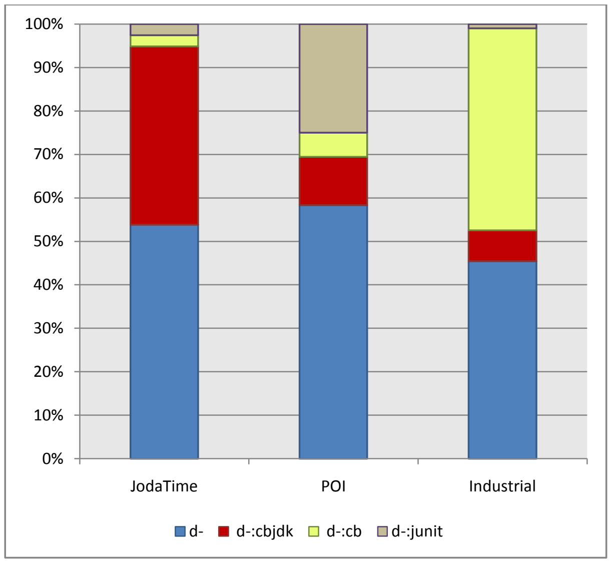

Figure 5.5: Subpartitions of d-.

5.2.3 Change Characteristics

In terms of change types, we found that 41% of the changes in JodaTime were corrective, 37% were adaptive, 19% were perfective, and 3% were preventive. As for Apache POI, 46% of the changes were adaptive, 29% were corrective, 22% were perfective, and 3% were preventive. The average number of call pairs in categories are shown for each type in Figure 5.6 and Figure 5.7. We found that the average size of unexecuted partitions is the highest in adaptive changes in both systems (5 call pairs in JodaTime and 11 call pairs in POI). This shows that when developers are adding new features or adapting the system to a new environment, they are less inclined to add tests that execute new edges as opposed to corrective or perfective changes. As shown in the figure, corrective changes experienced the least inconsistent and unexpected behaviour among both systems.

In terms of code changes in JodaTime, on average, 5 files were changed in each version, 106 lines of code are added, and 22 lines of code are removed. As for POI, 4 files are changed on average, 146 lines of code are added, and 24 lines of code are removed in each version. Looking into the relation between these attributes and our partitions, we found several interesting correlations; the correlations were measured using Pearson coefficient with a p-

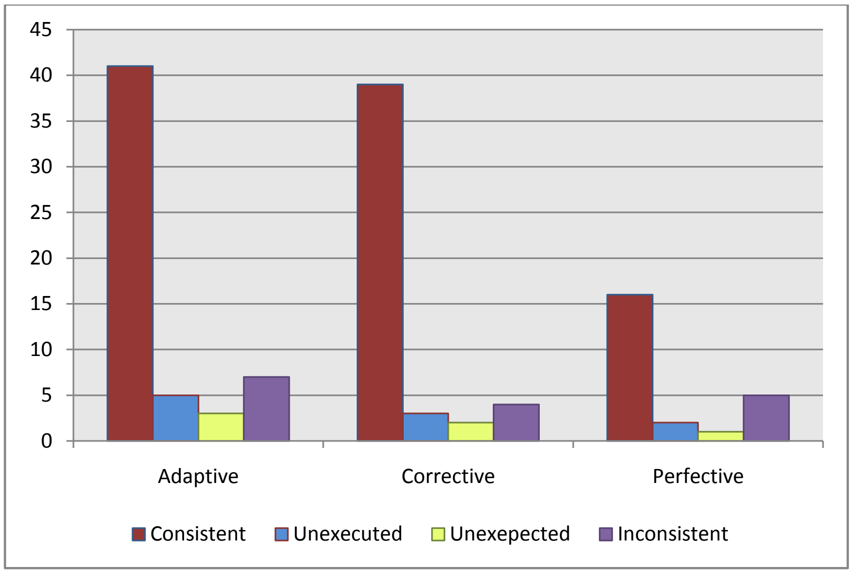

Figure 5.6: Average # call pairs in JodaTime for categories in different types.

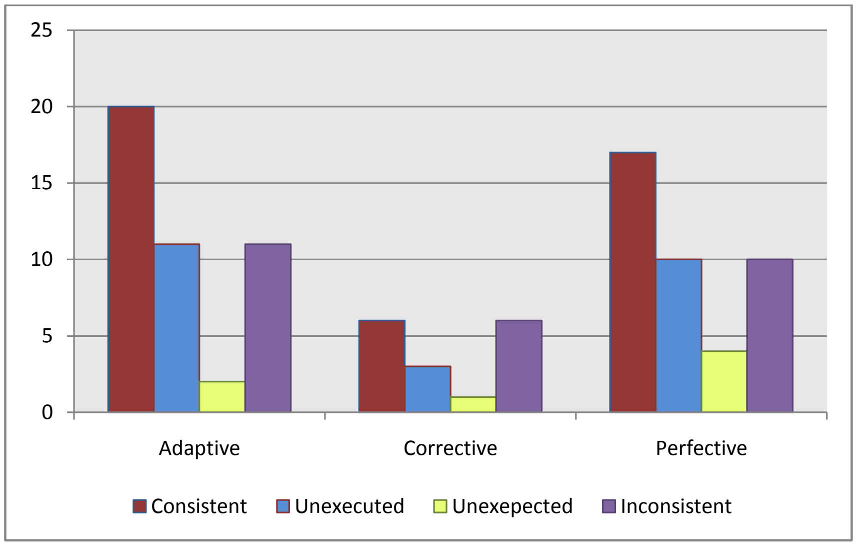

Figure 5.7: Average # call pairs in POI for categories in different types.

value smaller than 0.05 for all correlations. The correlations between code properties and partitions are shown in Table 5.3. As shown in the table, we found a strong correlation between line insertions and partitions with a `d+` label, indicating new behaviour: `d+`, `sd+`, and `s+d+`. Surprisingly, there was none or weak correlation between line deletions and partitions with a `d-` label (indicating old behaviour). Finally, we found a positive correlation between line deletions and `s-`, and similarly, between line insertions and `s+`. The fact that there is a correlation between line deletions and `s-`, but there is no correlation between deletions and partitions indicating old behaviour, shows that developers are more likely to remove unexecuted code more than executed code.

Correlation	**JodaTime**	**Apache POI**
Insertions/(d+, sd+, s+d+)	0.9	0.5
Deletions/(d-, sd-, s-d-)	0.1	0.0
Insertions/s+	0.8	0.5
Deletions/s-	0.8	0.4

Table 5.3: Correlations between code attributes and partitions.

5.3 Qualitative Results

We examined the partitions qualitatively to understand the types of call pairs that appear in different categories. We discuss our key results below.

5.3.1 Unexecuted Partitions

By inspecting the call pairs appearing in the unexecuted partitions, we found that the majority of calls originate from new methods that are added but never called from any test in the test suite. For instance, in JodaTime v1472, a call was added from `DateTimeZone.-convertLocalToUTC()` to `getOffset()` as shown in Figure 5.8. Since `convertLocalToUTC()` was never called from any test, `getOffset()` is not called and the call pair in Figure 5.8 appears in `s+`.

In code:

```
convertLocalToUTC(long instantLocal, boolean strict, long originalInstantUTC)
{

    int offsetOriginal = getOffset(originalInstantUTC);
    long instantUTC = instantLocal - offsetOriginal;
    int offsetLocalFromOriginal = getOffset(instantUTC);
    ...
}
```

In s+:

```
DateTimeZone.convertLocalToUTC(long, boolean, long) →
DateTimeZone.getOffset(long)
```

Figure 5.8: Unexecuted code example from JodaTime v1472.

Another collection of calls represent calls added inside a condition block where the condition is not satisfied. For example, in JodaTime v1545, the developer adds a return call inside a condition block where the condition checks if the object `iChronology` is equal to `null` as shown in Figure 5.9. Since the variable does not equal to `null` during execution, we find three call pairs appearing in `s+`; two sample call pairs are shown in Figure 5.9. Similarly, in Apache POI v4980, the developer adds a return call to `offset()` inside a condition block, as shown in Figure 5.10, but the condition is not satisfied and a call pair appears in `s+` accordingly.

Another portion of calls originate from tests to the jUnit `fail()` method. This is a technique used by developers when a tested code should raise an exception; they add a call to `fail()` after the test code. If `fail()` is executed this means that no exception was thrown. Therefore, if the call to `fail()` appears in the unexecuted partition, this indicates that the test code worked as intended. An example of this from JodaTime is shown in Figure 5.11.

Similarly, some of the call pairs appearing in unexecuted partitions are methods calling

In code:

```
Object readResolve() {
   if (iChronology == null) {
       return new LocalDateTime(iLocalMillis, ISOChronology.getInstanceUTC());
   }
...
}
```

In s+:

```
LocalTime.readResolve() → ISOChronology.getInstanceUTC()
LocalDateTime.readResolve() → LocalDateTime<init>(long, Chronology)
```

Figure 5.9: Unexecuted code example from JodaTime v1545.

In code:

```
AreaEval convertRangeArg(ValueEval eval) throws EvaluationException {
   ...
   if (eval instanceof RefEval) {
      return ((RefEval)eval).offset(0, 0, 0, 0);
   }

   throw new EvaluationException(ErrorEval.VALUE\_INVALID);
}
```

In s+:

```
Rank.convertRangeArg(ValueEval) → RefEval.offset(int, int, int, int)
```

Figure 5.10: Unexecuted code example from POI v4980.

an exception such as in Figure 5.12, or a `Null Pointer Exception`, as shown in Figure 5.13.

Another case where call pairs appear in unexecuted partitions is when developers remove dead code such as broken test cases, as in v1530 in JodaTime. In this case, after

In code:

```
void TestFieldUtilstestSafeMultiplyLongInt() {
...
try {
  FieldUtils.safeMultiply(Long.MIN\_VALUE, -1);
  fail();
} catch (ArithmeticException e) { .. }
...
}
```

In s+:

```
TestFieldUtils.testSafeMultiplyLongInt() → TestFieldUtils.fail()
```

Figure 5.11: Unexecuted code example from JodaTime v1555.

```
Rank.convertRangeArg(ValueEval) → EvaluationException.<init>(ErrorEval)
```

Figure 5.12: Exception call in s+ in POI v4980.

In code:

```
LocalDate now(Chronology chronology) {
...
if (zone == null) throw new NullPointerException("Zone must not be null");}
..
}
```

In s+:

```
LocalDate.now(Chronology) → NullPointerException.<init>(String)
```

Figure 5.13: Unexecuted code example from JodaTime v1521.

removing the broken tests, the developer could check the dynamic partitions with a `d-` label to verify that no behaviour was changed and that all deletions were unexecuted as intended.

5.3.2 Unexpected Partitions

Examining the unexpected partitions (`d+` and `d-`), we found that call backs to overridden JDK methods are often related to changes made to achieve stability or determinism. For instance, to "make hash code deterministic" as stated in the commit description of JodaTime v1541. Also, in JodaTime v1480, where the commit description is "Standard hashCode and equals for stability across serialization". A sample of these call pairs is shown in Figure 5.14. We also found that when few call pairs appear in the cb:jdk subpartition, there is a significant change in the call stack partitions. For instance, in JodaTime v1542, 5 call pairs appeared in `d+:cbjdk`, and correspondingly, 1208 new stacks appeared in `stack:d+`. The reason behind this is that JDK methods such as `equals()` and `hashCode()` are invoked very often during execution. It is also worth mentioning that these overridden methods usually appear at the edge of stacks in most changes.

```
java.util.Map.get(Object) → MockZone.equals(Object)
java.util.Arrays.equals(Object[], Object[]) → DateTimeFieldType.equals(Object)
java.util.Set.add(Object) → DurationFieldType.hashCode()
java.util.Map.get(Object) → CachedDateTimeZone.hashCode()
```

Figure 5.14: Sample of call backs to equals() and hashCode().

Another example for JDK call backs are calls to `toString()` as in POI v4783 where two call pairs, shown in Figure 5.16, appear in the d+:cbjdk partition due to the added code shown in Figure 5.15. The reason why `append()` calls `toString()` is because it uses a String representation of the object and appends to it.

Another example is in Jodatime v1472 where the line "`System.out.println(dt)`" that has an `AbstractInstant` object as a parameter, was removed. As a result, a call pair appears in `d-` from `java.io.PrintStream.println(java.lang.Object)` to `org.joda.time.-base.AbstractInstant.toString()`. Looking into the stacks, surprisingly, we find that the one line deletion resulted in 48 call stacks to disappear. This indicates that stacks are useful in knowing the actual impact of changes on taken program paths.

```
if ( ((Boolean) value).booleanValue() )
    {
         out.write(0xff);
         out.write(0xff);
    }
```

Figure 5.15: Code added in POI v4783.

```
java.lang.StringBuffer.append(Object) → ClassID.toString()
java.lang.StringBuffer.append(Object) → Section.toString()
```

Figure 5.16: Call pairs in d+:cbjdk in POI v4783.

Another portion of the call backs are calls to custom `readObject()` in serializable objects, as it is allowed for each subclass of a serializable object to define its own readObject method. This appears in JodaTime v1478 where an edge appears in `d+` from `java.io.ObjectInputStream.readObject()` to `org.joda.time.YearMonth.readResolve()`; the `readResolve()` implementation is shown in Figure 5.17.

```
Object readResolve() {
      if (DateTimeZone.UTC.equals(getChronology().getZone()) == false) {
          return new YearMonth(this, getChronology().withUTC());
      }
      return this;
}
```

Figure 5.17: readResolve() method in JodaTime v1478.

As for non-callbacks appearing in the remaining `d+` and `d-` partitions, we found a portion of calls to annotated methods. For instance, in JodaTime v1471, a call was added to `convertToString()` as shown in Figure 5.18, which calls the annotated `@ToString` method `getID()` shown in Figure 5.19, during run-time. Examining the new stacks partition, we only find one stack with two calls identical to the call pair shown in Figure 5.20. It is worth noting that annotations can be considered reflective as they are embedded in class files generated by the compiler then retained by the Java VM during run-time.

```
void testTimeZone() {
     DateTimeZone test = DateTimeZone.forID("Europe/Paris");
     String str = StringConvert.INSTANCE.convertToString(test);
     ...
}
```

Figure 5.18: ConvertToString() call in JodaTime v1471.

```
@ToString
String getID() {
    return iID;
}
```

Figure 5.19: DateTimeZone.getID() implementation in JodaTime v1471.

`TestStringConvert.testTimeZone()` $\rightarrow$ `DateTimeZone.getID()`

Figure 5.20: Call pair in d+ in JodaTime v1471

Another collection of calls appeared due to changes in configuration files. For instance, in POI v4980, the call pair in Figure 5.21 appears in `d+` even though the `WorkbookEvaluator` class was not modified. When investigating the change, we found that a change in a testing data file (`FormulaEvalTestData.xls`) resulted in this edge to be executed dynamically.

Finally, we found a portion of calls originating from inner classes to methods in other inner classes. For example, in JodaTime v1518, a call pair appears in `d+` from an inner static class `TimeZoneName` to another inner class `Composite` as shown in Figure 5.22. This call is executed dynamically but not reported statically; this seems to be a shortcoming of our static call graphs generator, as calls from inner classes should be reported statically as well.

```
WorkbookEvaluator.countTokensToBeSkipped(Ptg[], int, int) →
Area2DPtgBase.getSize()
```

Figure 5.21: Call pair in d+ in POI v4980.

```
DateTimeFormatterBuilder$Composite<init>(List) →
DateTimeFormatterBuilder$TimeZoneName.estimatedParsedLength()
```

Figure 5.22: Call pair in d+ in JodaTime v1518.

5.3.3 Inconsistent Partitions

In the inconsistent partitions, we found that a portion of calls appear due to new tests being added, where the new tests set parameters that causes previously unsatisfied conditions to be true, similar to the behaviour shown in the motivating scenario. The calls under these conditions appear in the inconsistent partitions, as they existed statically but were not exercised dynamically in old versions. For example, in JodaTime v1571, a new test was added that caused the condition in `BasicMonthOfYearDateTimeField.add()`, shown in Figure 5.23, to be true. As a result, the two calls: `partial.getValue()` and `return set()` appear in the `sd+` partition.

In other cases, some edges are no longer observed dynamically because certain calls are removed. For instance, in v1745 in JodaTime, the developer removes a call to `TimeZone.getDisplayName()`. As a result, any calls originating from `getDisplayName()` will be no longer executed and therefore will appear in `sd-`. This is worth investigating because the developer did not modify the `TimeZone` class directly. Looking into the `sd-` partition, we found one edge from `getDisplayName()` to another instance of `getDisplayName()` with parameters as shown in Figure 5.24. To further investigate this, we looked into the `stack:d-` partition, and we found that three program paths, containing `getDisplayName()`, have disappeared after the change. This shows that call pair analysis occasionally oversimplifies the impact of a change.

Another example for inconsistent call pairs is in POI v4783, where the added code is shown earlier in Figure 5.15. The few lines added resulted in 107 call pairs appearing in `sd+`; the change caused many edges originating from `toString()` methods to be invoked,

```
int[] add(ReadablePartial partial, int fieldIndex, int[] values,
int valueToAdd) {
   ....
      if (partial.size() > 0 && partial.getFieldType(0).equals(DateTimeField-
      Type.monthOfYear()) && fieldIndex == 0) {
            int curMonth0 = partial.getValue(0) - 1;
            int newMonth = ((curMonth0 + (valueToAdd \% 12) + 12) \% 12) + 1;
            return set(partial, 0, values, newMonth);
       }
   ....
}
```

Figure 5.23: Inconsistent code example in JodaTime v1571.

```
String getDisplayName() {
    return getDisplayName(false, LONG, Locale.getDefault());
}
```

Figure 5.24: TimeZone.getDisplayName() implementation in JodaTime v1745.

as it was statically there but never executed. A sample of the call pairs in `sd+` is shown in Figure 5.25.

```
Property.toString() → java.lang.Class.getName()
Property.toString() → java.lang.Object.getClass()
Property.toString() → java.lang.String.charAt(int)
```

Figure 5.25: Sample of call pairs in sd+ in POI v4783.

5.3.4 Nondeterministic Behaviour

While examining call stack partitions, we surprisingly found new call stacks appearing in `stack:d+` in versions where there were unexecuted changes only. Looking into the stacks and related source code, we found that some of the unit tests are nondeterministic, meaning that different program paths are taken with different test runs. Consequently, the number of call stacks in dynamic call graphs becomes inconsistent, as the dynamic call graph is generated by executing the entire test suite of a system.

Since the nondeterministic stacks could affect the quality of our results, we decided to eliminate the sources of indeterminism. We found that the nondeterministic calls appear due to one of three cases: weak references, weak hashmaps, and cache issues. Weak references are used in Java to indicate than an object is eligible for garbage collection if memory resources are needed. In the case of weak references, we found stacks ending with a call to `java.lang.ref.WeakReference.get()`; a sample is shown in Figure 5.26. The second type, weak hash maps, represent a hashtable-based Map implementation with weak keys, meaning that an entry will automatically be removed when its key is no longer in ordinary use. This type also appears at the end of stacks with a call to `java.util.WeakHashMap.get(java.lang.Object)`; a sample is shown in Figure 5.27. Finally, cache issues were the hardest to detect, as they require source code inspection. Developers often store variables in the cache, then try to retrieve the variables later. If the variable exists, a certain program path is taken, if not, a different path is taken. For instance, in JodaTime, we found that in the method `ISOChronlogy.getInstance()`, shown in Figure 5.28, the variable `chrono` is retrieved from cache first. If it exists it will be returned, otherwise, the cache will be locked and the variable will be stored, resulting in a different program path. A sample of a stack with this case is shown in Figure 5.29.

```
WeakReference::get()
GJLocaleSymbols::forLocale(Locale)
GJMonthOfYearDateTimeField::getAsShortText(int, Locale)
BaseDateTimeField::getAsShortText(long, Locale)
...
```

Figure 5.26: Sample of WeakReference call stack.

```
WeakHashMap::get(Object)
GJLocaleSymbols::forLocale(Locale)
GJMonthOfYearDateTimeField::getAsShortText(int, Locale)
BaseDateTimeField::getAsShortText(long, Locale)
...
```

Figure 5.27: Sample of WeakHashmap call stack.

```
ISOChronology getInstance(DateTimeZone zone) {
         ...
        ISOChronology chrono = cFastCache[index];
        if (chrono != null && chrono.getZone() == zone) {
            return chrono;
        }
        synchronized (cCache) {
            chrono = cCache.get(zone);
            if (chrono == null) {
                chrono = new ISOChronology(ZonedChronology.getInstance-
                (INSTANCE_UTC, zone));
                cCache.put(zone, chrono);
            }
        }
        cFastCache[index] = chrono;
        return chrono;
    }
```

Figure 5.28: ISOChronology.getInstance() method in JodaTime.

```
String::hashCode()
DateTimeZone::hashCode()
CachedDateTimeZone::hashCode()
Map::get(java.lang.Object)
ISOChronology::getInstance(DateTimeZone)
ISOChronology::withZone(DateTimeZone)
DateTime::withZone(DateTimeZone)
TestISODateTimeFormat::testFormat_basicDateTimeNoMillis()
```

Figure 5.29: Sample call stack with getInstance() when `chrono` does not exist in the cache.

Chapter 6

Discussion

We found that our polymorphic mapping algorithm and subpartitioning phase resulted in more manageable and meaningful sets of call pairs compared to the previous approach. When examining the results, we found that partitions representing new dynamic behaviour are greater in size than partitions representing old behaviour; indicateing that as developers change systems, they are more likely to introduce new behaviour than eliminating old behaviour. We also found a strong correlation between line insertions and new dynamic behaviour. Surprisingly, there was no correlation between line deletions and elimination of old dynamic behaviour. This implies that developers are more likely to remove unexecuted or dead code than code that is executed dynamically; this can also be inferred from the strong correlation we found between line deletions and the unexecuted partition `s-`.

In terms of unexecuted changes, our results show that developers add unexecuted/dead code more often than removing it. However, for JodaTime and Apache POI, the average size of removed unexecuted code was greater than added unexecuted code. Conversely, in the industrial system, the average size of added unexecuted code was significantly greater than removed unexecuted code. A possible explanation of this is that developers working on libraries are more keen on adding test cases to execute and verify new code, as opposed to developers working on more broad industrial applications. This is also shown in the distribution of partitions in the three systems; the unexecuted partitions in the industrial system are relatively high compared to the open source systems. As a result, the inconsistent partitions are also higher in the industrial system, simply because when a previously unexecuted pair gets executed in a later version, it appears in an inconsistent partition. As expected, the number of consistent calls were high in the three systems indicating that the majority of calls added by developers are statically and dynamically coherent.

When examining different types of changes for the open source systems, we found that unexecuted calls are added most often in adaptive changes. This suggests that when developers add new features or adapt the system to a new environment, they are less willing to add tests for the changes as opposed to corrective or perfective changes. We also found that unexpected calls are lowest in corrective changes in both systems, indicating that developers may be more cautious when fixing defects.

We constantly investigated the usefulness and effectiveness of call stacks in our analysis. It became evident that call stacks provide useful information necessary to understand the actual impact of a change. Call stacks were particularly helpful in cases where inconsistent partitions contained call pairs originating from classes that were not modified by the developer. In these cases, if call stacks were not provided, it becomes challenging for the developer to investigate the behaviour causing these calls to appear. Furthermore, call stacks helped in detecting nondeterministic behaviour that cannot be inferred from call pairs. However, since the sizes of call stack partitions are usually large, as shown in our results, it is insufficient to depend on call stacks only. On the contrary, examining call pairs only oversimplifies the analysis as shown in our qualitative results. Therefore, we believe that call stacks complement call pairs to provide complete behaviour analysis for the developer. We suggest that developers start their analysis by examining call pairs in key partitions then, if any interesting pairs are detected, examine the stacks including the desired pairs.

6.1 Threats to Validity

The major threat to the external validity of our findings is the limited number of systems we evaluated. We used two open source systems that are actively maintained and fairly stable; applying our approach to more systems with different levels of maturity could yield different findings. We also need to evaluate more industrial systems to verify the findings found based on the industrial system in our study. However, to overcome this threat, we aimed to extract a relatively high number of commits to analyze (the most recent 101 source code commits). This allows us to investigate the behaviour of systems over a longer period to cover different kinds of changes.

In terms of threats to internal validity, even though we eliminated a high percentage of mismatched elements compared to the previous approach, it is possible that our results contain some mismatches as well. However, by comparing results before and after our polymorphic mapping algorithm, we found that the sizes of key partitions reduced significantly and became more manageable. Another threat to internal validity is that we classified the changes in the open source systems manually as one of four types: adaptive, corrective, perfective, or preventive. For some changes, the classification could be challenging as the commit description is ambiguous and open to different interpretations. Finally, a risk to the internal validity of our qualitative analysis is relying on our experience and judgement to interpret whether changes appearing in certain partitions are worth investigation.

6.2 Future Work

A possible extension to our work is to perform a longitudinal study considering the entire lifetime of systems. By analyzing the partitions among all versions of a system, we could understand the behavioural changes a system undergoes through different stages of its lifecycle. Additionally, we could visualize the changes in behavioural categories. Another possible extension would be evaluating the effectiveness of developers contributing to a project by extracting their commits and analyzing the impact of their commits on the system. In terms of the evaluation, we could consider a greater number of systems with different levels of maturity and at various stages of development. Considering additional industrial systems will allow us to verify the findings based on the current industrial system results.

Our approach could further be extended by considering all possible static call stacks into our analysis. We could compare the static and dynamic call stacks then determine code coverage in terms of the executed percentage of call stacks. We could also track the changes in call stacks coverage over the lifetime of a system. In addition, we would be able to form complete call stack partitions, similar to the call pairs partitions, by considering all combinations of static and dynamic call stacks from two versions of a system.

Chapter 7

Conclusion

Evolution represents an integral and essential phase of the software life cycle; developers evolve software to fix defects, add new features, and improve performance and design. As developers modify software, some static modifications may lead to unintended changes in dynamic behaviour. In this thesis, we improved the Inconsistency Inspector, a previous approach that combines static and dynamic analysis to categorize behavioural changes. We greatly improved its accuracy by implementing polymorphic mapping and utilizing type hierarchies. We further refined the model by introducing a change-centric state-transition model that captures the flow of call dependencies among different partitions. Furthermore, we extended the approach by incorporating complete dynamic call stacks into the analysis.

We evaluated our enhanced approach over three software systems to analyze dynamic behavioural changes across 100 versions. Our polymorphic mapping algorithm reduced mismatches between static and dynamic analyses by 53%. We found that developers introduce new behaviour more often than eliminating old behaviour. Our results imply that developers are more likely to remove unexecuted code than code that is executed dynamically. In terms of change types, we found that corrective changes encountered the least inconsistent and unexpected behaviour, while adaptive changes experienced the highest unexecuted behaviour. We also found that call stack analysis provides useful information that helps in understanding the actual impact of changes.

Overall, our approach helps developers discover behavioural changes that are not easily identified by regression testing or manual inspection. After making a source code modification, developers can use our approach to check for inconsistent, unexecuted, or unexpected behavioural changes. By examining key partitions, developers will be able to decide whether a behavioural change is intended or problematic, and thus, build more confidence in their modifications. Finally, we suggest that developers use a combination of call pairs and call stack analyses to better understand the behavioural impact of their changes.

APPENDICES

Appendix A

Unexpected Partitions

In the following, we provide sample call pairs in unexpected partitions from JodaTime and Apache POI as represented in our tool.

A.1 JodaTime Sample

```
<partition name="d+" count="2">

  <path s="org.joda.time.TestDateTimeZoneCutover.doTest_getOffsetFromLocal-
  (int, int, int, int, java.lang.String, org.joda.time.DateTimeZone)" t="org-
  .joda.time.base.AbstractInstant.toString()"/>

  <path s="org.joda.time.TestDateTimeZoneCutover.doTest_getOffsetFromLocal-
  (int, int, int, int, java.lang.String, org.joda.time.DateTimeZone)" t="org-
  .joda.time.base.BaseDateTime.getMillis()"/>

</partition>
```

```
<partition name="d+:cbjdk" count="2">

  <path s="java.util.Map.get(java.lang.Object)" t="org.joda.time.MockZone.-
  equals(java.lang.Object)"/>

  <path s="java.util.Map.put(java.lang.Object, java.lang.Object)" t="org.joda.-
  time.MockZone.equals(java.lang.Object)"/>

</partition>
```

A.2 Apache POI Sample

```
<partition name="d-:junit" count="2">

 <path s="junit.framework.Assert.assertEquals(java.lang.String, java.lang.Obj-
 ect,java.lang.Object)" t="org.apache.poi.hpsf.SpecialPropertySet.toString()"/>

  <path s="junit.framework.Assert.assertEquals(java.lang.Object, java.lang.Obj-
  ect)" t="org.apache.poi.hpsf.Property.equals(java.lang.Object)"/>

</partition>

<partition name="d-:cbjdk" count="2">

  <path s="java.lang.StringBuffer.append(java.lang.Object)" t="org.apache.-
  poi.hpsf.ClassID.toString()"/>

  <path s="java.lang.StringBuffer.append(java.lang.Object)" t="org.apache.poi.-
  hpsf.Section.toString()"/>

</partition>
```

Appendix B

Inconsistent Partitions

In the following, we provide sample call pairs in inconsistent partitions from JodaTime and Apache POI as represented in our tool.

B.1 JodaTime Sample

```
<partition name="sd+" count="3">

  <path s="org.joda.time.TestStringConvert.testDays()" t="org.joda.time.-
  Days.toString()"/>

  <path s="org.joda.time.TestStringConvert.testYears()" t="org.joda.time.-
  Years.toString()"/>

  <path s="org.joda.time.base.BaseSingleFieldPeriod.equals(java.lang.-
  Object)" t="org.joda.time.Weeks.getPeriodType()"/>

</partition>
```

```
 <partition name="sd-"  count="3">

  <path s="org.joda.time.tz.DateTimeZoneBuilder.writeTo(java.io.Data-
  Output)" t="org.joda.time.tz.FixedDateTimeZone.getNameKey(long)"/>

  <path s="org.joda.time.tz.DateTimeZoneBuilder.writeTo(java.io.Data-
  Output)" t="org.joda.time.tz.FixedDateTimeZone.getOffset(long)"/>

  <path s="org.joda.time.tz.DateTimeZoneBuilder.writeTo(java.io.Data-
  Output)" t="org.joda.time.tz.FixedDateTimeZone.getStandardOffset(long)"/>

</partition>
```

B.2 Apache POI Sample

```
<partition name="sd+" count="2">

  <path s="org.apache.poi.hpsf.basic.TestWrite.diff(byte[], byte[])"
  t="java.lang.Math.min(int, int)"/>

  <path s="org.apache.poi.hpsf.basic.TestWrite.testVariantTypes()"
  t="java.lang.Double.<init>(double)"/>
</partition>
<partition name="sd-" count="2">

  <path s="org.apache.poi.hpsf.Util.toString(java.lang.Throw-
  able)" t="java.io.StringWriter.close()"/>

  <path s="org.apache.poi.hpsf.Util.toString(java.lang.Throw-
  able)" t="java.io.StringWriter.toString()"/>
</partition>
```

Appendix C

Unexecuted Partitions

In the following, we provide sample call pairs in unexecuted partitions from JodaTime and Apache POI as represented in our tool.

C.1 JodaTime Sample

```
<partition name="s+" count="3">

  <path s="org.joda.time.chrono.TestLenientChronology.test_iso--
  Chrononolgy()" t="org.joda.time.chrono.TestLenientChronology.fail()"/>

  <path s="org.joda.time.field.LenientDateTimeField.set(long, int)"
   t="org.joda.time.field.LenientDateTimeField.add(long, long)"/>

  <path s="org.joda.time.field.LenientDateTimeField.set(long, int)"
  t="org.joda.time.field.LenientDateTimeField.get(long)"/>

</partition>
```

```
<partition name="s-" count="3">

  <path s="org.joda.time.field.LenientDateTimeField.set(long, int)"
   t="org.joda.time.field.LenientDateTimeField.add(long, int)"/>

  <path s="org.joda.time.field.LenientDateTimeField.set(long, int)"
  t="org.joda.time.field.LenientDateTimeField.getMaximumValue(long)"/>

  <path s="org.joda.time.field.LenientDateTimeField.set(long, int)" t="org-
  .joda.time.field.LenientDateTimeField.getMinimumValue(long)"/>

</partition>
```

C.2 Apache POI Sample

```
<partition name="s+" count="2">

  <path s="org.apache.poi.ddf.AbstractEscherOptRecord.fillFields(-
  byte[], int, org.apache.poi.ddf.EscherRecordFactory)" t="org.apache-
  .poi.ddf.AbstractEscherOptRecord.readInstance(byte[], int)"/>

   <path s="org.apache.poi.ddf.EscherOptRecord.setVersion(short)"
   t="org.apache.poi.ddf.AbstractEscherOptRecord.setVersion(short)"/>

</partition>
<partition name="s-" count="1">

  <path s="org.apache.poi.ddf.EscherRecord\$EscherRecordHeader.-
  toString()" t="java.lang.StringBuilder.append(java.lang.String)"/>

</partition>
```

References

[1] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. Efficient and precise dynamic impact analysis using execute-after sequences. In *Proceedings of the International Conference on Software Engineering*, pages 432–441, 2005.

[2] Robert S. Arnold and Shawn A. Bohner. Impact analysis - towards a framework for comparison. In *Proceedings of the Conference on Software Maintenance*, pages 292–301, 1993.

[3] Anton Babenko, Leonardo Mariani, and Fabrizio Pastore. Ava: automated interpretation of dynamically detected anomalies. In *Proceedings of the international symposium on Software testing and analysis*, ISSTA, pages 237–248, 2009.

[4] Linda Badri, Mourad Badri, and Daniel St-Yves. Supporting predictive change impact analysis: A control call graph based technique. In *Proceedings of the Asia-Pacific Software Engineering Conference*, pages 167–175, 2005.

[5] B. Breech, A. Danalis, Stacey Shindo, and Lori Pollock. Online impact analysis via dynamic compilation technology. In *Proceedings of the International Conference on Software Maintenance*, pages 453–457, 2004.

[6] Fred P. Brooks, Jr. The mythical man-month. *SIGPLAN Notices.*, 10(6), April 1975.

[7] Jim Buckley, Tom Mens, Matthias Zenger, Awais Rashid, and Günter Kniesel. Towards a taxonomy of software change: Research articles. *Journal of Software Maintenance and Evolution*, 17(5):309–332, September 2005.

[8] Mark Dowson. The ariane 5 software failure. *SIGSOFT Software Engineering Notes*, 22(2), March 1997.

[9] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Aiding program comprehension by static and dynamic feature analysis. In *Proceedings of the International Conference on Software Maintenance*, pages 602–, 2001.

[10] Andrew David Eisenberg and Kris De Volder. Dynamic feature traces: Finding features in unfamiliar code. In *Proceedings of the International Conference on Software Maintenance*, pages 337–346, 2005.

[11] Emelie Engström, Mats Skoglund, and Per Runeson. Empirical evaluations of regression test selection techniques: a systematic review. In *Proceedings of the international symposium on Empirical software engineering and measurement*, pages 22–31, 2008.

[12] Keith Brian Gallagher and James R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, August 1991.

[13] Micheal Godfrey and Qiang Tu. Evolution in open source software: a case study. In *In Proceedings of the International Conference on Software Maintenance*, pages 131 –142, 2000.

[14] Chetna Gupta, Yogesh Singh, and Durg Singh Chauhan. An efficient dynamic impact analysis using definition and usage information. *Journal of Digital Content Technology and its Applications*, 3(4):112–115, 2009.

[15] Kevin J. Hoffman, Patrick Eugster, and Suresh Jagannathan. Semantics-aware trace analysis. In *Proceedings of the 2009 SIGPLAN conference on Programming language design and implementation*, pages 453–464, 2009.

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

[17] Andrew J. Ko, Robert DeLine, and Gina Venolia. Information needs in collocated software development teams. In *Proceedings of the international conference on Software Engineering*, pages 344–353, 2007.

[18] James Law and Gregg Rothermel. Incremental dynamic impact analysis for evolving software systems. In *Proceedings of the International Symposium on Software Reliability Engineering*, 2003.

[19] James Law and Gregg Rothermel. Whole program path-based dynamic impact analysis. In *Proceedings of the International Conference on Software Engineering*, pages 308–318, 2003.

[20] Steffen Lehnert. A taxonomy for software change impact analysis. In *Proceedings of the International Workshop on Principles of Software Evolution and the ERCIM Workshop on Software Evolution*, pages 41–50, 2011.

[21] Ondrej Lhotak. Comparing call graphs. In *Proceedings of the SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 37–42, 2007.

[22] K. Martersteck and A. Spencer. Introduction to the 5ESS TM switching system. 1985.

[23] A. Mockus, R.T. Fielding, and J. Herbsleb. A case study of open source software development: the apache server. In *In Proceedings of the International Conference on Software Engineering*, pages 263 –272, 2000.

[24] Ranjith Purushothaman and Dewayne E. Perry. Toward understanding the rhetoric of small source code changes. *IEEE Transactions on Software Engineering*, 31(6):511–526, June 2005.

[25] Xiaoxia Ren, Barbara G. Ryder, Maximilian Stoerzer, and Frank Tip. Chianti: a change impact analysis tool for java programs. In *Proceedings of the international conference on Software engineering*, pages 664–665, 2005.

[26] Thomas Reps, Thomas Ball, Manuvir Das, and James Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In

Proceedings of the Software Engineering conference held jointly with the SIGSOFT international symposium on Foundations of Software Engineering, pages 432–449, 1997.

[27] Abhishek Rohatgi, Abdelwahab Hamou-Lhadj, and Juergen Rilling. An approach for mapping features to code based on static and dynamic analysis. In *Proceedings of the International Conference on Program Comprehension*, pages 236–241, 2008.

[28] Gregg Rothermel and Mary Jean Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, August 1996.

[29] Kaitlin Duck Sherwood and Gaill C. Murphy. Reducing code navigation effort with differential code coverage. University of British Columbia, September 2008.

[30] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. Questions programmers ask during software evolution tasks. In *Proceedings of the SIGSOFT international symposium on Foundations of Software Engineering*, pages 23–34, 2006.

[31] E. Burton Swanson. The dimensions of maintenance. In *Proceedings of the International Conference on Software Engineering*, pages 492–497, 1976.

[32] F. van Rysselberghe and S. Demeyer. Studying software evolution information by visualizing the change history. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 328 – 337, Sept. 2004.

[33] Norman Wilde and Michael C. Scully. Software reconnaissance: mapping program features to code. *Journal of Software Maintenance*, 7(1):49–62, January 1995.

[34] Franck Xia and Praveen Srikanth. A change impact dependency measure for predicting the maintainability of source code. In *Proceedings of the Annual International Computer Software and Applications Conference - Workshops and Fast Abstracts - Volume 02*, pages 22–23, 2004.

[35] Zhenchang Xing and Eleni Stroulia. Data-mining in support of detecting class co-evolution. In *Proceedings of the International Conference on Software Engineering and Knowledge Engineering*, pages 123–128, 2004.

[36] Jianjun Zhao, Hongji Yang, Liming Xiang, and Baowen Xu. Change impact analysis to support architectural evolution. *Journal of Software Maintenance*, 14(5):317–333, September 2002.

[37] Thomas Zimmermann, Peter Weisgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *Proceedings of the International Conference on Software Engineering*, pages 563–572, 2004.