

# UC San Diego

## UC San Diego Electronic Theses and Dissertations

### Title

Continuous Semantic Inspection

### Permalink

<https://escholarship.org/uc/item/880242xm>

### Author

Yan, Yan

### Publication Date

2017

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA, SAN DIEGO

**Continuous Semantic Inspection**

A dissertation submitted in partial satisfaction of the requirements for the degree

Doctor of Philosophy

in

Computer Science

by

Yan Yan

Committee in charge:

Professor William G. Griswold, Chair  
Professor William Howden  
Professor Ranjit Jhala  
Professor Sorin Lerner  
Professor Kevin Patrick

2017

Copyright

Yan Yan, 2017

All rights reserved.

The Dissertation of Yan Yan is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

---

---

---

---

---

Chair

University of California, San Diego

2017

## DEDICATION

To my family, and my dearest memory.

## EPIGRAPH

Diligence is the mother of good luck.

—*Benjamin Franklin*

I hear and I forget. I see and I remember. I do and I understand.

—*Xun Zi*

A day is a miniature of eternity.

—*Ralph Waldo Emerson*

## TABLE OF CONTENTS

SIGNATURE PAGE .....	iii
DEDICATION .....	iv
EPIGRAPH .....	v
TABLE OF CONTENTS .....	vi
LIST OF FIGURES .....	x
LIST OF TABLES .....	xii
ACKNOWLEDGEMENTS .....	xiii
VITA .....	xv
ABSTRACT OF THE DISSERTATION .....	xvi
Chapter 1 Introduction .....	1
1.1    Challenges .....	2
1.2    Overview of Continuous Semantic Inspection .....	3
1.3    Roadmap .....	5
Chapter 2 Related Work .....	8
2.1    Patch Comprehension .....	8
2.1.1    Syntax Differencing .....	9
2.1.2    Semantic Differencing .....	9
2.1.3    Interpreting Changes by Natural Language .....	11
2.1.4    Comparisons .....	11
2.2    Invariant Inference and Differencing .....	12
2.2.1    Daikon .....	13
2.2.2    Invariant Differencing .....	15

2.3	Improving Lightweight Code Review .....	15
Chapter 3 Continuous Semantic Inspection .....		18
3.1	“Hello CSI” – A GSON Case Study .....	18
3.2	Concepts .....	25
3.2.1	Invariant Differentials .....	25
3.2.2	Impact Isolation .....	27
3.2.3	Invocation Flows .....	30
3.3	Design and Implementation .....	31
3.3.1	Build Framework .....	31
3.3.2	Fact Extraction .....	33
3.3.3	Dynamic Invariant Inference .....	34
3.3.4	User Interface Generation .....	35
Acknowledgments .....		37
Chapter 4 Performance and Scalability .....		38
4.1	Challenges .....	38
4.2	Methods to Improve Performance .....	39
4.3	Scalability .....	44
Acknowledgments .....		47
Chapter 5 Quantitative Evaluation .....		48
5.1	Finishing the GSON Case Study .....	48
5.2	Reducing Inspection Load .....	55
5.3	Checking Test Sufficiency .....	60
5.4	Checking Semantic Change Consistency .....	63



5.4.1	Incorrect Invariants .....	65
5.4.2	Missing Expected Invariants .....	67
5.5	Threats to Validity .....	68
	Acknowledgments .....	70
Chapter 6 User Study .....		71
6.1	Study Description .....	72
6.1.1	Roles and Participants .....	72
6.1.2	Issue Lifecycle and Reviewer Workflow .....	74
6.1.3	Study Tasks .....	77
6.2	Code Review Process .....	80
6.2.1	Control Group Review Process .....	81
6.2.2	Experimental Group Review Process .....	86
6.3	Reviewer Feedback .....	91
6.4	Limitations .....	94
6.5	Threats to Validity .....	95
6.6	Final Remarks .....	96
	Acknowledgments .....	96
Chapter 7 Future Work .....		97
7.1	Quality of Invariant Differentials .....	97
7.2	Tool Integration .....	99
7.3	More Applications .....	100
Chapter 8 Conclusion .....		102
Appendix I .....		105

Appendix II.....	111
Bibliography .....	113

## LIST OF FIGURES

Figure 1 Likely invariants are a summary of test executions .....	13
Figure 2 The commit history for <code>JsonPrimitive:equals</code> .....	19
Figure 3 A screenshot of GETTY for commit #7 .....	21
Figure 4 Source-impact isolated invariant differentials.....	23
Figure 5 Source code changes to <code>JsonPrimitive:equals</code> .....	24
Figure 6 Test change impact isolation for commit #7, for old source .....	28
Figure 7 Three-layer GETTY Architecture .....	32
Figure 8 Patch Summary Zone .....	35
Figure 9 Invocation Flow Zone.....	35
Figure 10 Semantic Inspection Zone .....	36
Figure 11 Inference Execution Time of 6 Open Source Projects .....	42
Figure 12 Average running time versus number of processors .....	45
Figure 13 Cloud cost of invariant extraction against the number of CPUs used.....	46
Figure 14 Selected dynamic invariant differentials for commits #1 - #3.....	49
Figure 15 The failing test case in commit #10.....	50
Figure 16 New feature to compare between integers in commit #4 .....	50
Figure 17 Commit #4: the invariant differential clearly indicates a bug. ....	51
Figure 18 Test case to confirm the integer-equality bug .....	52
Figure 19 Test case to confirm the decimal-equality bug.....	53
Figure 20 Commit #11: invariant differential points to a similar bug as commit #4.....	53
Figure 21 Fixing the introduced integer-equality bug .....	54
Figure 22 Test Sufficiency Inspection Result.....	61
Figure 23 GSON bug at commit b634804, indicated by the invariant diff.....	65

Figure 24 Collections bug at commit 83226e1, indicated by the invariant diff.....	66
Figure 25 Crypto bug at commit 9faf04e; no invariant diff for exit-point 138 .....	67
Figure 26 Configuration bug at commit c75a72c, lack of invariants for String conversion.....	69
Figure 27 User Study Issue Lifecycle .....	75
Figure 28 Reviewer Interactions .....	76
Figure 29 Control Group Review Process .....	81
Figure 30 Code change in the intValue method.....	82
Figure 31 the intValue method with context.....	83
Figure 32 Experimental Group Review Process .....	86
Figure 33 Invariant and code diff for the longValue method in GETTY .....	87
Figure 34 The test case for the longValue method .....	88
Figure 35 Use test impact isolation to verify the lack of tests .....	89

## LIST OF TABLES

Table 1 Behavior Comparison Strategies .....	29
Table 2 Execution Time in ATAM Mode.....	39
Table 3 Project Information and Execution Time of Dynamic Inference.....	40
Table 4 Execution Time (Hours) of Different Modes (1).....	41
Table 5 Execution Time (Hours) of Different Modes (2).....	43
Table 6 Execution Time (Hours) of PATSC Mode .....	44
Table 7 Results of Invariant Reduction without Impact Isolation .....	56
Table 8 Results of Invariant Reduction under Source Impact Isolation .....	57
Table 9 Results of Invariant Reduction under Test Impact Isolation for Old Source.....	58
Table 10 Results of Invariant Reduction under Test Impact Isolation for New Source .....	58
Table 11 Results of Inspecting Buggy Commits .....	64
Table 12 Participant Labels .....	74

## ACKNOWLEDGEMENTS

I am greatly indebted to my advisor, Professor William G. Griswold, for his guidance and support in my research and other matters. His insightful advice has always turned my seemingly trivial ideas into exciting projects. I can still remember our first research meeting when I discussed my initial idea of using invariants for software engineering. At that early phase of my research, he encouraged me to dig further and wider, and positively fostered all my following research activities. Bill has also been my inspirational example of personal charisma and professional integrity. He has been a great mentor and demonstrated to me his high bar of success in family life without compromising any quality of his professional achievement. I feel it a great honor to have the chance to work with Bill and stay in the Griswold academic tree.

I am very grateful that I had the opportunity to work with Doctor Massimiliano Menarini. He was my colleague and later became my *de facto* co-advisor after he got his PhD degree. Max is a very smart researcher and engineer. He unblocked me from nearly all the technical issues I had encountered in my research. Max met with Bill and me regularly to discuss about this dissertation. Max shared numerous great opinions and new ideas. He was the co-author of all my research papers, and was also actively involved in the acquisition and analysis of the data for Chapter 4, Chapter 5 and Chapter 6 in this dissertation.

Thanks must also go to the professors Sorin Lerner, Ranjit Jhala, and William E. Howden, for sharing their insightful comments during the course of my research. Their input was invaluable. I learnt that not only the research results but also the way to achieve

and present them matter. Special thanks are given to Professor Kevin Patrick for his support of my first half of PhD study. His CYCORE project attracted me into the area of web applications and thus had a deep impact on my career path. Here I would like to thank all my committee members for reading my dissertation and attending my final defense.

Life at UCSD would be dull without all the great people I met during this phase of my life. First thank Professor Ingolf Krueger for recruiting me and becoming my first advisor. I felt home to meet with my office mates: Massimiliano Menarino, Barry Demchak, Celal Ziftci, Filippo Seracini, To-ju Huang and Xiang Zhang, and then Steven Rick, Danilo Gesques Rodrigues and Soohyun Nam. Specially, I would like to thank Professor Nadir Weibel for providing the nice lab for doing my user study for Chapter 6.

Chapter 3 contains the material as it appears in “Mining Software Contracts for Software Evolution”. Yan, Yan; Menarini, Massimiliano; Griswold, William G. IEEE International Conference on Software Maintenance and Evolution, 2014. The dissertation author was the primary investigator and author of this paper.

Chapter 3, Chapter 4, Chapter 5 and Chapter 6, in part, are currently being prepared for submission for publication of the material, Yan, Yan; Menarini, Massimiliano; Griswold, William G. “Mining Code Repositories for Semantics-Assisted Code Review” The dissertation author was the primary investigator and author of this material.

At last, I appreciate the support of my family. This work is dedicated to you, my most important people in this world.

## VITA

- 2006 Bachelor of Science, Tsinghua University
- 2009 Master of Science, Peking University
- 2014 Internship, AppFolio, Inc.
- 2015 Internship, Uber Technologies, Inc.
- 2017 Doctor of Philosophy, University of California, San Diego



ABSTRACT OF THE DISSERTATION

**Continuous Semantic Inspection**

by

Yan Yan

Doctor of Philosophy in Computer Science

University of California, San Diego, 2017

Professor William G. Griswold, Chair

As testing is an incomplete validation of software changes, many developers review code changes before patching the system. Popular source code versioning systems aid review by showing the textual differences between the old and new versions of the

source code. This leaves developers with the difficult task of determining whether the differences produced the desired behavior.

We introduce Continuous Semantic Inspection (CSI), which aids code review with inter-version differential semantic analysis. During inspection of a new commit, a developer is presented with not only code differences, but also changes to behaviors, as expressed by likely invariants inferred from testing. We hypothesize that with the extra semantic information developers can more easily determine whether the code changes produced the desired effect.

This dissertation comprises four parts to demonstrate our hypothesis. (1) We present the design and implementation of GETTY, a highly automated tool to support the concept of CSI. (2) To scale the expensive invariant inferences for practical use, we divide GETTY's analysis into multiple processes distributed to a 16-processor cluster. We achieve substantial performance improvement and show that our approach is feasible for open source projects by enabling a timely repair-compile-review feedback loop. (3) From applying CSI on six open source projects, we found inferred invariants contain the information required to help discover insufficient tests and inconsistent semantic changes at the time of their introduction. (4) We conducted a user study of reviewers using GETTY, which shows that invariant differentials draw attention to information that otherwise often escapes notice, helping developers formulate more focused questions that keep the review moving forward.

# Chapter 1

## Introduction

Code review is a software quality assurance practice of examining changed code for overlooked mistakes. Most code review processes fall into two categories: formal software inspection and lightweight code review [1]. Formal software inspection is the traditional method of review that mandates strict review criteria, including but not limited to specification checklists and in-person meetings [2]. By contrast, lightweight code review requires less overhead and is intended to be more cost-effective. Reviewers peruse the changed code and make judgments based on their own experience, and the form of conducting reviews is flexible.

Nowadays most organizations and companies are adopting lightweight code review practice as part of team shipping processes [3]. For example, Google requires all code changes to be peer reviewed before deploying [4]. In a typical interaction cycle, developers submit a patch of reasonably small size for review [5]; reviewers (often other developers [6]) examine the code changes, check for implemented functionalities and possible mistakes, and either accept the patch or send feedback for a revision. The end result of fast development-review turnarounds is a development history of frequent, small, independent, and complete contributions by both developers and reviewers.

In this dissertation, we are concerned with improving the quality and efficiency of lightweight code review process.

## 1.1 Challenges

Today's popular Version Control Systems (VCS, e.g., *git* [7]) integrate with textual differencing tools (e.g., *git-diff* [8]) to aid lightweight code review. A reviewer can start with a summary of changed code between two versions, then navigate to related code snippets at their own discretion. When developers are fixing a bug, for example, a reviewer expects the code changes to reflect changes to the software's behavior such that the erroneous behavior is gone.

However, the textual, program-level differences from current differencing tools provide only indirect information about the *actual* behavioral impact of code changes. The results of testing provide only a pass/fail view of that behavior, perhaps disguising subtle bugs. Reviewers have to read further into the source code to understand both the syntactic and semantic changes, and examine the related tests to verify that the changes are being properly tested. For certain cases, serious reviewers will have to compile the code, run the tests and manually observe their runtime behaviors to confirm the change delivers the desired result.

This dissertation aims for an effective and efficient method to aid code review by bridging the gap between textual code differences and their actual behavioral impacts. There are existing solutions (as will be discussed in Chapter 2) in three categories: syntax differencing, semantic differencing, and natural language interpretation, with decreasing human efforts and increasing computational efforts. As code review activities are becoming more and more frequent [5], ideal methods should also be efficient for fast development iterations. However, the solutions that require less human efforts are

generally not efficient enough, while the efficient solutions may require more effort from reviewers. Therefore, it remains challenging to design a means that is both effective and efficient.

## 1.2 Overview of Continuous Semantic Inspection

We propose balancing the computational and human efforts so it is both effective for reviewers and efficient for fast software iterations. To this end we advocate inspecting program semantics during code review, with the assurance that crafted tooling will support the inspection in fast development cycles like nightly builds.

To support and encourage semantic review of source code changes, we provide reviewers with direct summaries of the behavioral effects of those changes. The summaries would ideally be concise, comprehensive, presented in a familiar notation, and integrated into the existing reviewing infrastructure. Likewise, the production of the summaries should require little or no effort on the part of the developer or reviewer, just like Continuous Integration today supports effortless testing. We call the resulting infrastructure and process *Continuous Semantic Inspection* (CSI).

In CSI, we suggest differencing semantics in order to produce a focused view of the important behavioral changes. The idea of differential assertion checking [9], [10] is a gesture in the right direction, but the requirement to manually add assertions is both labor-intensive [11] and does not provide a comprehensive view of semantic effects. However, applying this concept to the likely invariants extracted by a tool like Daikon [12] could provide the best of both worlds. Its invariants are reported in the terminology of the program itself at the method and class level, akin to software contracts. Although

Daikon can produce overwhelming volumes of likely invariants, differencing them can eliminate this problem. Because they are reported at the method level, it is easy to attach them to the code difference summaries provided by a tool like *git*.

The approach is not without challenges, however. First, the behavioral effects of changed source code versus changed tests must be distinguished, or the reviewer could be misled. Second, changed code can have widespread effects on behaviors, affecting other parts of the code. Reviewers need help in finding paths where semantic effects propagate. Third, many compute-intensive steps are implied by this approach. For example, inferring invariants from runtime information can be costly because it may have to be done at whole-program mode. Given that more tests are more likely to ensure software quality, they may also deteriorate the performance and invalidate practical use of a tool.

We propose isolating behavioral impacts to one part of the program by fixing the other parts of the program during check-in. For example, by running the same tests on both the old and the new source, all resulting invariant differences can be confidently attributed to the source. To help reviewers reason about behaviors, we build local-area call-graphs consisting of callers, immediate siblings and callees for each method of interest. We hypothesize that all semantic information is organized in a natural way for reviewers to digest: reviewers can isolate behavioral changes of interest, and they can explore the related behavioral changes by following the control flow displayed in the local-area call-graphs.

For efficiency, we suggest parallelizing test executions to improve performance. We identified that the performance bottleneck of our approach is the invariant inference

process, whose massive cost is in turn due to memory pressure. Software testing is embarrassingly parallel, and so is dynamic invariant detection. We can distribute the load of inference processes to multiple processors to reduce the memory pressure. We also found this approach is highly scalable in that dispatching inference processes to a high-performance cluster achieved substantial performance improvement. Therefore, we proved that our approach is efficient for practical use.

### **1.3 Roadmap**

This dissertation presents our approach to aiding code review. We argue that our approach is both effective and efficient. The remainder of this dissertation is structured as follows.

In Chapter 2, we discuss the related work. We discuss the existing differencing techniques that deal with different levels of program information. After comparing their computational and human costs, we discover that semantic differencing is the promising technique that better balances the trade-offs between effectiveness and efficiency. This observation motivated the overall design of our approach. After investigating into the existing, highly usable, invariant detection tool, Daikon [12], we decide to use dynamically inferred invariants to help reviewers better understand semantic changes.

In Chapter 3, we define the concept of Continuous Semantic Inspection (CSI), which is enabled by augmenting code-change summaries with automatically extracted behavior-change summaries. We introduce CSI through a scenario of a reviewer using GETTY, our tool support for CSI, on an open source project. The scenario highlights the three most salient features of CSI: behavioral diffing, impact isolation, and exploration of

invocation flows. We present the design of our GETTY tool. GETTY differences likely invariant dynamically inferred by Daikon, isolates invariant changes due to single source by fixing other parts of the program before inference, and builds the local-area dynamic call-graph to organize gathered invariants. We finish this chapter with a three-layer implementation of GETTY. The infrastructure layer integrates Daikon and existing open source tools for Continuous Integration, like Maven [13] and *git*, which eliminates the human overhead of pulling check-ins and building source code [14]. The application layer follows the design of GETTY to extract likely invariants for diffing. It also computes all the information needed, including static change summary and dynamic invocation flows, for the top layer that generates User Interface (UI) for reviewers.

In Chapter 4, we describe the improved implementation of GETTY for scalability. It automates and parallelizes the production of invariants for Java projects. We introduce and study different modes for the parallelization. We found that dividing inference processes at class level achieved most performance improvement on a single computer, and substantial performance boost when the processed were deployed to multiple cluster nodes. This reveals that although likely invariant inference is expensive in practice, it is possible to leverage the independent nature of test executions to distribute the inference process in the cloud and achieve the desired performance.

We evaluate the effectiveness of GETTY in the next two chapters. In Chapter 5, we present a case study of Google's JSON library (GSON) where we replayed a portion of its revision history, showing that employing CSI provides behavior-change summaries that can reveal bugs and other problems earlier than they were actually discovered. To



generalize our observation, we performed a retrospective quantitative analysis of six open source projects, where CSI as supported by GETTY helped find gaps in testing in 32 of 100 commits. Furthermore, for one known bug in each of six projects, CSI's invariant differentials made the bug evident at its point of introduction in 4 out of 6 cases. In Chapter 6 we evaluate the effectiveness of GETTY from a human perspective. We conducted a six-group user study to observe whether and how GETTY users achieved the desired goals in a code review setting where they inspect real issues and commits from GSON repository. We found that GETTY users generally outperformed the past reviewers of GSON in that they were able to more actively identify specific test inadequacies. The derived review patterns prove effective in discovering bugs at earlier times.

In Chapter 7 we discuss other potential applications of using invariants, alternative inference techniques, and future work. We conclude in Chapter 8.

# Chapter 2

## Related Work

A large body of work has attempted to help reviewers understand the commits, known as patch comprehension. The techniques come with various computational and human costs. Care must be taken to balance the costs for better assisting code review. In this chapter, we will first discuss all the related techniques for patch comprehension in Section 2.1.

Our proposed invariant-based technique depends on the tools we use for invariant inference and differencing. From the many options that exist we choose Daikon and textual differencing for our research. Daikon and the related invariant differencing techniques will be discussed in Section 2.2 of this chapter.

At last, a number of research literatures are dedicated to improve the quality of lightweight code review by utilizing other code analysis tools. This will be discussed in Section 2.3.

### **2.1 Patch Comprehension**

Software patches are mostly implemented and expressed by line-based, textual changes to source code. To help understand a patch, developers created a spectrum of techniques to translate its textual changes into more meaningful representations. Some of these techniques may not be designed for code reviews, but they can potentially bridge the gap between textual code changes to their semantic impact.

### 2.1.1 Syntax Differencing

One step to better comprehend code change is to understand its underlying syntactical changes. GumTree [15] takes as input two versions of source code, computes differences between their Abstract Syntax Trees (ASTs), and derives a sequence of edit actions that well reflects developers' intent. It uses an efficient differencing algorithm so as to scale for large ASTs in open source projects. In addition to element additions and deletions, GumTree is particularly helpful in detecting moved and renamed elements. Kim *et al.* apply structural changes project-wide (*LSdiff* [16]) and introduce the concept, systematic changes (similar, related changes to multiple contexts), into a code review tool, CRITICS [17]. Given a specified change, CRITICS prepares a context-aware AST edit template. As an effort to interact with the tool, reviewers iteratively customize its parameters, let it match against codebase, summarize systemic edits and locate potentially problematic edits. CRITICS scales to industry-scale projects and receives favorable feedback from professional engineers. Furthermore, the set of elements that are relevant or essential (for example, references of a changed method) but not presented directly in the textual code change can be computed to better understand the impact of code change [18], [19].

### 2.1.2 Semantic Differencing

**Static Semantic Differencing.** Many researchers have appreciated the value of differencing static semantic information. Lahiri, Vaswani and Hoare from Microsoft Research discuss differential static analysis [20]. Several promising applications are highlighted, including semantic differencing and differential contract checking. Person *et*

*al.* proposes differential symbolic execution to detect and characterize the effects of program changes in terms of behavioral differences, then use a theorem prover to compare the symbolic summaries for such differences [21]. *iProperty* hits the similar idea [10]. SYMDIFF [9] presents differential assertion checking for comparing different versions of a program with respect to a set of assertions. The approach defines relative correctness: the second program version does not violate assertions the first one satisfies. Although it provides a weaker guarantee than outright correctness, it is more tractable than traditional assertion checking, and is still powerful: the authors of SYMDIFF were able to soundly verify null-pointer dereferencing bugs.

**Dynamic Impact Analysis.** Chianti executes tests on two versions of the code and differentiates their runtime behaviors, then it decomposes the difference into a set of predefined atomic changes like “add a new class”, “remove a method”, “change definition of static initializer”, etc., and then relates those changes to affected tests [22]. Chianti is particularly helpful in isolating changes that lead to a test failure. iDiSE considers dynamic calling context information from inter-procedural analysis to categorizing impact behaviors, and extending notions of test coverage by chang impact information [23]. Although Chianti and iDiSE were designed as debugging tools, their underlying technologies could be applied to aid code reviews.

**Hybrid Differencing Techniques.** Holmes and Notkin take a hybrid static-dynamic approach to the differencing concept [24]. Their approach analyzes invocation dependencies based on their presence in each of four graphs: the static call graph and the dynamic call graph from each of the two versions given. A visualization of the

differences across the cross-product of graphs can reveal anomalies that motivate further inspection. For example, a developer who updated a third-party library and expected their system to behave the same would be surprised to find their control flow has changed at runtime, but in the static call graph.

### 2.1.3 Interpreting Changes by Natural Language

Another very broad area of related work that can help programmers understand semantic changes is program comprehension in natural languages [25], [26]. Recently, Su *et al.* [27] and Devanbu *et al.* [28] used statistic models to study software and buggy code and discovered their naturalness as human artifacts. This makes it possible to understand code changes by using natural languages processing techniques. For example, reviewers often read test cases to understand code changes between two versions [29]. A recent technique, TestDescriber can extract natural language phrases from test cases and generate natural language summaries [30].

### 2.1.4 Comparisons

**Computational cost vs. human effort.** Syntax analysis itself does not imply any computational cost on semantic inference. Its main focus is to help developers understand editing activities. But to understand the resulted patch developers need to take their own efforts to understand the semantics. On the other hand, natural language interpretations can be much easier for human consumption. However, that implies significantly more computational cost on semantic inference and language translation. What is worse, natural languages are by their nature ambiguous so it is nearly impossible to ensure the

preciseness or even correctness of the result. Balancing computational cost and human effort, we favor the idea of semantic differencing.

**Static vs. dynamic techniques.** Despite great potentials, the practical use of static differential analysis is limited because the approach generally requires users to write assertions, intermediate contracts, or worse, proof scripts, all of which impose very high overhead to programmers [20]. By contrast, dynamic differential analysis can be more accessible because it generally does not require extra annotations. Moreover, static analysis can be overly conservative, limiting the value of its inferences. By contrast, dynamic analysis provides insights from runtime information, which helps increase reviewer’s confidence. However, it is worth noting that dynamic semantic analysis could perform in whole-program mode so scalability problems of dynamic analysis tools could be more serious than static analysis tools [31]. Our approach is motivated by dynamic differencing ideas like [23] and [24], but uses differences of source-level likely program invariants rather than differences in the more abstract control flow. We also dealt with performance and scalability issues that could impede its practical use for code reviews.

## 2.2 Invariant Inference and Differencing

Runtime information is often named in the low-level binary code or machine language space. Such information from trace files can be hard for users to understand. In addition, runtime information can be unstable. For example, applications using system clock or random number generator may get different results from them each time. Using runtime information carelessly risks high rate of false positives or false negatives. [31]

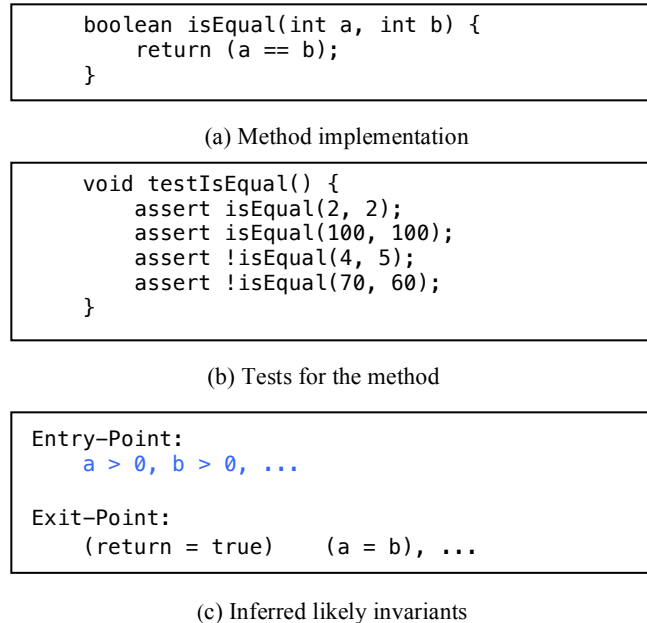


Figure 1 Likely invariants are a summary of test executions

### 2.2.1 Daikon

Invariants, as a summary of runtime data from test execution, can be a powerful tool for reviewing semantic differences. Dynamic invariant inference was pioneered by the Daikon tool [12].

Daikon comprises a front-end and a back-end. The Daikon front-end (also known as an instrumenter or tracer) instruments the target program. During program execution, The Daikon front-end records information about variable values at entry- and exit-program points, and produces data trace files. One can choose to obtain traces for only part of the target program to avoid inundating oneself with output, and can also improve performance. After recording data traces, Daikon passes it to its back-end (invariant detector) as input. The Daikon back-end applies machine-learning techniques to infer likely invariants, according to its set of invariant template library, from the traces.

The inferred likely invariants can be considered a summary of the program executions. As an example, consider the `isEqual` method in Figure 1. Developers added four test cases (four pair of numbers) to verify the correctness of the implementation. From test execution, Daikon deduces that at the entry-point of the `isEqual` method, `a` and `b` are both positive numbers, and that at the exit-point, if the method returns `true` then `a` and `b` must be identical. The expected contract of the `isEqual` method should indicate that it returns `true` if and only if `a` and `b` are identical. Likely invariants are close to the expected contract in that they indicate the method is probably correct when comparing two positive numbers. As a summary of the test executions, the closer they are to the expected contract, the better quality of the test suite it can be.

Daikon’s invariant template library supports a wide range of invariants for up to three scalar dimensions. For example, its `FloatLessThan` is an invariant type representing the “less than” relationship between two double scalars. Daikon is especially supportive for inferring invariants for arrays; for example, it can detect invariants regarding an array’s size, index, and even relationships between internal elements.

Daikon is still under active development and is limited in certain ways. For example, Daikon does not have a strong invariant template for the `String` type. Optimistically, `String` is just an array of characters, so there is potential for improvement of Daikon by taking advantage its excellent array invariant templates. Likewise, currently Daikon does not support invariant inference for program points that throw exceptions. We use an improved version of Daikon that supports exceptional exit invariant inference [32].



### 2.2.2 Invariant Differencing

The number of invariants inferred by Daikon can be overwhelming, while they may not be all useful. However, we need not display all invariants to reviewers. Since our purpose is to help developers understand the change of semantics, we difference the invariants and display the changes only. We postulate the number of changed invariants is smaller than the number of all invariants, and consequently the number of invariants to display can be reduced.

Our differencing method is based on the similarity of invariant texts. We discussed in Section 2.1.2 alternative methods that support some understanding of logic. As an application example, Randoop employs static invariant differencing to determine when random test case generation can halt, that is when adding tests stops improving the invariants [33]. Their differencing method is more sophisticated than our current textual approach. We will discuss our plan on improving our results by comparing logical formulae in Section 7.1.

### 2.3 Improving Lightweight Code Review

Researchers expended significant effort on understanding how developers understand the code and how code review quality can be improved. Y. Tao, *et al.* conducted an exploratory study at Microsoft and revealed the lack of tool support for acquiring information such as a patch's completeness, consistency and risks [29]. O. Kononenko, *et al.* performed a qualitative analysis of a survey of 88 Mozilla developers, and concluded that code review quality is greatly influenced by the thoroughness of reviewers' feedback and their familiarity with the code [5]. On this path, we believe that

tool support for helping developers understand behavioral aspects of the code base is a direct solution for improving code review quality.

Unfortunately, most code review tools were not following the same path. For example, CRITICS [17] (discussed in Section 2.1.1) is designed to be a dedicated code review tool for helping understand developers' editing behaviors, but program's semantic differences. Gerrit, created by Google and integrated with git, supports distributed code review by providing a staging area for changes where they can be reviewed prior to committing to the repository [34]. Phabricator is a platform integrating many tools, including git-diff based code review [35]. Our code review tool, GETTY (to discuss in Section 3.3), generates results as HTML files and thus could be integrated into these existing platforms. However, different from textual-diff based tools, GETTY adds semantic information to aid code review and has its unique advantages in discovering test inadequacies and bugs (Chapter 5).

Additionally, reviewers may use other tools for assessing code quality during review. For example, reviewers of Apache Commons Collections project use Checkstyle, FindBugs and EMMA [36]. Checkstyle is a tool to automate the process of checking whether the code adheres to a style standard [37]. Coding style issues are outside the scope of this paper, but it is more often integrated into the build process of a project so reviewers should mostly see the code clear of style issues at code review phase. FindBugs is a static analysis tool for detecting code patterns that could lead to bugs. It grants reviewers insights about bad code smells, according to a list of known bug patterns, by analyzing the syntactic structure of a program [38]. Such static bug detection technique

has the advantage of not being limited by the quality of tests and can be supplemental to our dynamic approach. EMMA is a Maven plugin for reporting code coverage by the test suite [39]. The quantitative evaluation of our tool on reviewing test cases (Section 5.3) corroborates recent results that popular test coverage tools are not always a good indicator of test suite effectiveness; and the user study (Chapter 6) demonstrates that our tool is stronger than EMMA for evaluating test suite during code review.

## Chapter 3

### Continuous Semantic Inspection

We propose that, during inspection of a new check-in, we present to a developer with not only code differences, but also changes to behaviors. Behavioral changes can be expressed by likely invariants, which can be dynamically inferred from test execution. As a result, developers can more easily determine that the code changes produced the desired effect, or introduced a bug.

In this chapter, we first build a scenario from an open source project and introduce the most salient features of our approach, Continuous Semantic Inspection (CSI). After discussing CSI, we provide details on the design and implementation of its tool support, GETTY. We use GETTY to complete the case study of the open source project.

#### **3.1 “Hello CSI” – A GSON Case Study**

We introduce CSI through a scenario of a reviewer using our GETTY tool for Java on the GSON project [40]. The GSON project is a Google-sponsored open source Java library for conversions between Java Objects and their JSON representations. From the start of the project in 2008 until the writing of this dissertation, GSON has undergone 1,302 commits by 44 contributors, with 33 software releases. All changes were peer reviewed.

Of interest is the change history of the `equals` method in the `JsonPrimitive` class. `JsonPrimitive` represents a JSON primitive value that is a string, a Java primitive

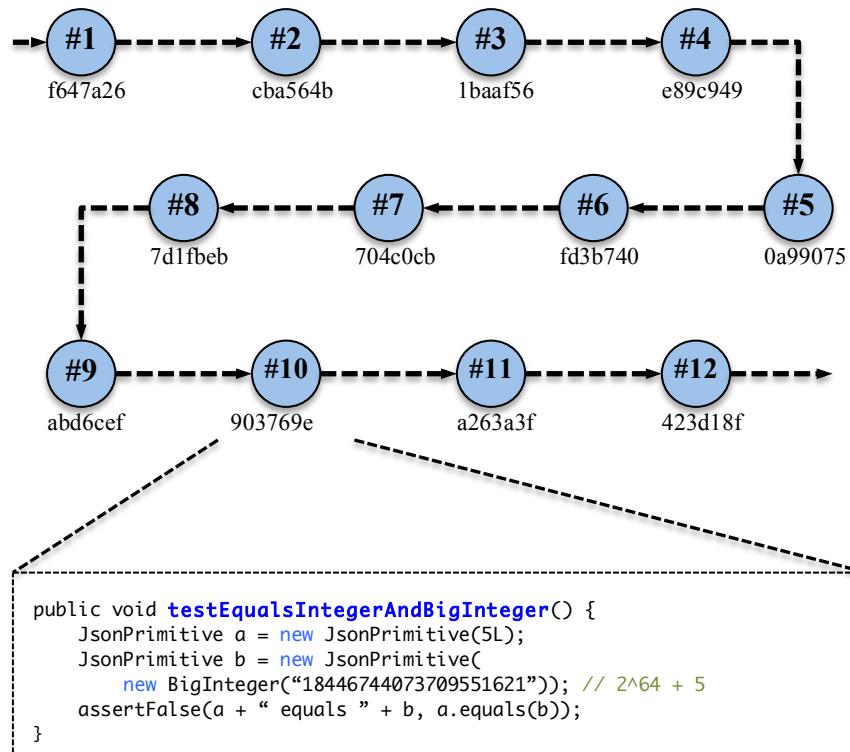


Figure 2 The commit history for `JsonPrimitive:equals`

(e.g., `int`), or a Java primitive wrapper type (e.g., `Integer`). The class has a `value` attribute that stores the value of the primitive. Its `equals` method takes as an argument another `JsonPrimitive` object and returns `true` if and only if both objects contain identical values of the `value` attribute.

Figure 2 shows, in chronological order, all 12 commits in the history that affected the behavior of the `equals` method. To ease our presentation, we refer to a commit by its index number in the circle rather than the commit hash underneath. Our target method was firstly implemented in commit #1. After commit #12 the implementation stabilized and the feature is kept for all future versions. Notice that we highlight commit #10 that introduced a test comparing two JSON primitive integers:  $2^{64}+5$  and 5. As asserted the `equals` method should return false because the two integer values are different. However,

the test failed because our target method returns true instead. Apparently, it is a bug. The bug was never fixed: in the last related commit (#12) developers considered “the price is too much to pay” to fix the bug and manually passed the tests by changing its `assertFalse` to `assertTrue`.

In this case study, a bug was not revealed until commit #10, despite the fact that all prior commits passed their code reviews. We are interested in which of the previous commits actually introduced the bug. We will answer this question in Section 5.1. Here, we “replay” the review of commit #7, supposing that the GSON team had access to GETTY. The reviewer is seeking to confirm that a bug in `equals` for comparing an NaN floating point value (not-a-number) has actually been corrected. The reviewer brings up the default view, comparing commit #7 to the previous commit, in which each version is run on all the test cases in its own commit (Figure 3). This provides a holistic view of what happened between the two commits, but does not distinguish the effects due to source changes versus test changes.

The reviewer first peruses the upper box in the figure, which lists top-to-bottom (a) two commit hashes under compare and the common package name of all methods of the project, (b) the source code methods that have changed (highlighted in blue), (c) the testing methods that have been updated (also highlighted in blue) and (d) the methods for which their invariants have changed compared to previous commit (hidden by default, but highlighted in red in all boxes).

Compare Commits: [16567e2 vs. 704c0cb](#) Common Package: com.google.gson

Updated Source: [JsonArray.equals](#), [JsonArray.hashCode](#), [JsonArray.toString](#), [JsonObject.equals](#), [JsonObject.hashCode](#), [JsonObject.toString](#), [JsonPrimitive.equals](#)

Updated Tests: [JsonArrayTest:testEqualsNonEmptyArray](#), [JsonArrayTest:testEqualsOnEmptyArray](#), [JsonNullTest:testEqualsAndHashCode](#), [JsonObjectTest:testEqualsNonEmptyObject](#), [JsonObjectTest:testEqualsOnEmptyObject](#), [JsonObjectTest:testReadPropertyWithEmptyStringName](#), [JsonPrimitiveTest:testBoolean](#), [JsonPrimitiveTest:testEquals](#), [JsonPrimitiveTest:testEqualsAcrossTypes](#), [JsonPrimitiveTest:testEqualsDoesNotEquateStringAndNonStringTypes](#), [JsonPrimitiveTest:testExponential](#), [JsonPrimitiveTest:testParsingStringAsBoolean](#), [JsonPrimitiveTest:testStringsAndChar](#), [JsonPrimitiveTest:testValidJsonOnToString](#), [common.MoreAsserts:assertEqualsAndHashCode](#)

Methods & Classes with Possible Invariant Changes  HIDE

More Methods  SHOW Tests  SHOW

Legends: code-changed invariant-changed invariant-unchanged ...(#old-calls + #newly-added)

<p><a href="#">JsonPrimitive:equals</a> (0+20)</p> <p><a href="#">JsonObject:remove</a> (1+0) <a href="#">JsonObject:get</a> (1+0)</p> <p><a href="#">JsonPrimitive:&lt;init&gt;</a> (13+4)</p> <p><a href="#">JsonElement:getAsJsonPrimitive</a> (20+0)</p>	<p><a href="#">common.MoreAsserts:assertEqualsAndHashCode</a> (0+40)</p> <p><a href="#">JsonPrimitiveTest:testEquals</a></p> <p><a href="#">JsonPrimitiveTest:testEquals</a> (0+3)</p> <p><a href="#">JsonPrimitiveTest:testEquals</a> (0+3)</p>	<p><a href="#">JsonPrimitive:equals</a> (0+20) <a href="#">JsonPrimitive:hashCode</a> (13+10)</p> <p><a href="#">JsonObject:remove</a> (1+0) <a href="#">JsonObject:has</a> (1+0)</p> <p><a href="#">JsonPrimitive:&lt;init&gt;</a> (0+4) <a href="#">JsonElement:isJsonPrimitive</a> (12+0)</p>
<p><a href="#">JsonPrimitive:isFloatingPoint</a> (20+21)</p> <p><a href="#">JsonPrimitive:isIntegral</a> (57+35)</p> <p><a href="#">JsonPrimitive:getAsNumber</a> (30+26)</p>		

Invariant Changes Due To:  Source & Test Change  Source Change Only  Test Change (for OLD Source)  Test Change (for NEW Source)

compare invariants for ( com.google.gson.JsonPrimitive.equals )

REMOVED	ADDED
<code>com.google.gson.JsonPrimitive.equals(java.lang.Object)::ENTER</code>	<code>com.google.gson.JsonPrimitive.equals(java.lang.Object)::ENTER</code>
<code>obj != null</code>	<code>(return == true) ==&gt;&gt; (orig(obj) != null)</code>
<code>obj.getClass().getName() == com.google.gson.JsonPrimitive.class</code>	<code>(return == true) ==&gt;&gt; (orig(obj).getClass().getName() == com.google.gson.JsonPrimitive.class)</code>
<code>com.google.gson.JsonPrimitive.equals(java.lang.Object)::EXIT</code>	<code>com.google.gson.JsonPrimitive.equals(java.lang.Object)::EXIT</code>
<code>com.google.gson.JsonPrimitive.equals(java.lang.Object)::EXIT346</code>	<code>com.google.gson.JsonPrimitive.equals(java.lang.Object)::EXIT346</code>
<code>com.google.gson.JsonPrimitive.equals(java.lang.Object)::EXIT356</code>	<code>com.google.gson.JsonPrimitive.equals(java.lang.Object)::EXIT349</code>
<code>com.google.gson.JsonPrimitive.equals(java.lang.Object)::EXIT356;condition="not(return == true)"</code>	<code>com.google.gson.JsonPrimitive.equals(java.lang.Object)::EXIT349;condition="not(return == true)"</code>
<code>this.value.getClass().getName() == java.lang.Long.class</code>	<code>return == false</code>
<code>com.google.gson.JsonPrimitive.equals(java.lang.Object)::EXIT356;condition="return == true"</code>	<code>com.google.gson.JsonPrimitive.equals(java.lang.Object)::EXIT356</code>
<code>com.google.gson.JsonPrimitive.equals(java.lang.Object)::EXIT361</code>	<code>orig(obj) != null</code>
<code>com.google.gson.JsonPrimitive.equals(java.lang.Object)::EXIT</code>	<code>orig(obj).getClass().getName() == com.google.gson.JsonPrimitive.class</code>
<code>com.google.gson.JsonPrimitive.equals(java.lang.Object)::EXIT;condition="return == true"</code>	<code>com.google.gson.JsonPrimitive.equals(java.lang.Object)::EXIT361</code>
	<code>orig(obj).getClass().getName() == com.google.gson.JsonPrimitive.class</code>
	<code>com.google.gson.JsonPrimitive.equals(java.lang.Object)::EXIT363</code>
	<code>orig(obj) != null</code>
	<code>orig(obj).getClass().getName() == com.google.gson.JsonPrimitive.class</code>
	<code>com.google.gson.JsonPrimitive.equals(java.lang.Object)::EXIT;condition="return == true"</code>
	<code>orig(obj) != null</code>
	<code>orig(obj).getClass().getName() == com.google.gson.JsonPrimitive.class</code>

More Display Options:  Invariant Diff Only  Complete Invariants  Source Diff  Source Code

List of All Code Changes  HIDE

Generated at Oct. 04, 2016. Getty - Semantful Differentials.

Figure 3 A screenshot of GETTY for commit #7

The reviewer has clicked the `JsonPrimitive:equals` method link (boxed in green) in the “Updated Source” section, causing it to be listed in the middle of the next section of dotted boxes, as well as listing its changed invariants at the bottom.

The dotted boxes are a summary of the invocations closely related to `equals`. Above `equals`, the reviewer sees that there are just three direct callers of `equals`. The caller shown in red is an indication that there were changes to its invariants. The callers shown with underlines indicate that there were changes to its source code, which is consistent to the methods underlined in the above box. Notice that all callers of `equals` are tests. Reviewers can choose to hide them by clicking the toggle button that

immediately follows the “Tests” option. To the left of `equals` are shown methods that one of `equals`’s caller called immediately before calling `equals`, and to the right methods that one of `equals`’s callers called immediately after `equals` returned. Similarly, methods in red indicate change of invariants and methods underlined change of source code. In addition, the methods in gray indicate their invariants remain unchanged after the commit. Reviewers can choose to hide such methods by clicking the toggle button that immediately follows the “More Methods” option. The reviewer does not see anything on the left that suggests that they would affect `equals`’s invariants because they all appear to be queries without any side-effects, so she is not motivated to click any of them to view their invariant or source code differences. The box below shows the methods that are called by the `equals` itself. The red highlighting on the two type tests were not modified for this check-in (they are not underlined, and do not appear in the modified source list above), so the reviewer surmises that other changes in commit #7 – test cases or changes to `equals` itself – have altered their invariants.

The reviewer now turns her attention to the invariants displayed below the invocation summary. Removed invariants are highlighted in red, added invariants highlighted in green, and changed invariants highlighted in yellow (none in this figure), just as text changes are highlighted in *git*. The gray header closely above the removed invariants indicates that they are for the entry point of the method. The reviewer’s immediate observation is that commit #7 is an improvement since it is now testing for the case when `other` is equal to `null`, as well as for other cases that are not only for



Compare Commits: 16567e2 vs. 704c0cb Common Package: com.google.gson

Updated Source: [JsonArray.equals](#), [JsonArray.hashCode](#), [JsonArray.toString](#), [JsonObject.equals](#), [JsonObject.hashCode](#), [JsonObject.toString](#), [JsonPrimitive.equals](#)

Updated Tests: [JsonArrayTest:testEqualsNonEmptyArray](#), [JsonArrayTest:testEqualsOnEmptyArray](#), [JsonNullTest:testEqualsAndHashCode](#), [JsonObjectTest:testEqualsNonEmptyObject](#), [JsonObjectTest:testEqualsOnEmptyObject](#), [JsonObjectTest:testReadPropertyWithEmptyStringName](#), [JsonPrimitiveTest:testBoolean](#), [JsonPrimitiveTest:testEquals](#), [JsonPrimitiveTest:testEqualsAcrossTypes](#), [JsonPrimitiveTest:testEqualsDoesNotEquateStringAndNonStringTypes](#), [JsonPrimitiveTest:testExponential](#), [JsonPrimitiveTest:testParsingStringAsBoolean](#), [JsonPrimitiveTest:testStringsAndChar](#), [JsonPrimitiveTest:testValidJsonOnToString](#), [common.MoreAsserts:assertEqualsAndHashCode](#)

Methods & Classes with Possible Invariant Changes  HIDE

More Methods  Tests

Legends: code-changed invariant-changed invariant-unchanged ...(#old-calls + #newly-added)

<pre> common.MoreAsserts:assertEqualsAndHashCode (0+40)   JsonPrimitiveTest:   testEqualsDoesNotEquateStringAndNonStringTypes (0+3)   JsonPrimitiveTest:testEquals (0+3) </pre>	<pre> [ JsonPrimitive.equals ] </pre>	<pre> JsonPrimitive.equals (0+20)  JsonPrimitive.hashCode (13+10)   JsonObject.remove (1+0)  JsonObject.get (1+0)   JsonPrimitive.&lt;init&gt; (13+6)   JsonElement.getAsJsonPrimitive (20+0) </pre>
<pre> JsonPrimitive.isFloatingPoint (20+21)   JsonPrimitive.isIntegral (57+35)   JsonPrimitive.getAshNumber (38+26) </pre>		

Invariant Changes Due To:

compare invariants for ( com.google.gson.JsonPrimitive:equals )

REMOVED	ADDED
<pre> com.google.gson.JsonPrimitive.equals(java.lang.Object)::EXIT356;condition="not(return == true)" this.value.getClass().getName() == java.lang.Long.class com.google.gson.JsonPrimitive.equals(java.lang.Object)::EXIT359 (return == false) ==&gt; (this.value == java.lang.Float.NaN) (return == false) ==&gt; (obj.value == java.lang.Float.NaN) </pre>	<pre> com.google.gson.JsonPrimitive.equals(java.lang.Object)::EXIT356;condition="not(return == true)" com.google.gson.JsonPrimitive.equals(java.lang.Object)::EXIT361 return == true </pre>

More Display Options:

List of All Code Changes

Generated at Oct. 04, 2016. Getty - Semantifful Differentials.

Figure 4 Source-impact isolated invariant differentials

JsonPrimitives, although she suspects that the latter is only due to checking for the null case.

She might check into that issue later, but what is bothering her right now is that she cannot tell whether the NaN bug has been fixed. She suspects that is because both the code and the test cases have been changed, so she switches the view to a condition in which both commits were run on the same tests, choosing in particular the union of their two test suites (Figure 4). This view can be turned on by choosing from the tab labeled “Source Change Only” that indicates reviewers want to see invariant changes due to nothing but change to the method itself. She knows that some old test cases might not compile on the new code and *vice versa*, but that is fine with that, since a non-compiling (and hence non-running) test case will be useful behavioral information captured by the invariants that distinguishes the source code of the two versions.

When the results of the new testing condition are displayed, the reviewer immediately surmises that `equals` now returns true for the case when `this.value` and `other.value` are both NaN (the added invariant, in green), whereas before it returned false (the removed invariants just above it, in red). Their gray header shows that these invariants apply just to the return statement (i.e., exit point at line 361), which is where the floating-point cases are handled.

The reviewer is pleased, but the fact that this exit point only ever returns true tells her that it is being under-tested. Before firing off a comment to the developer, she scrolls down to view the source code diff for the `equals` method by clicking the tab labeled “Source Diff” of “More Display Options” section. The output (Figure 5, cropped) displays code differentials starting from the first line of code change. Optionally, she could use the side bar to scroll up or down to view the full method body. She notes that it is fairly straightforward and probably correct. She still sends a comment to the developer

Invariant Changes Due To:  Source & Test Change  Source Change Only  Test Change (for OLD Source)  Test Change (for NEW Source)

compare invariants for { com.google.gson.JsonPrimitive.equals }

REMOVED	ADDED
com.google.gson.JsonPrimitive.equals(java.lang.Object)::EXIT356;condition="not(return == true)"	com.google.gson.JsonPrimitive.equals(java.lang.Object)::EXIT356;condition="not(return == true)"
this.value.getClass().getName() == java.lang.Long.class	
com.google.gson.JsonPrimitive.equals(java.lang.Object)::EXIT359	com.google.gson.JsonPrimitive.equals(java.lang.Object)::EXIT361
(return == false) ==> (this.value == java.lang.Float.NaN)	
(return == false) ==> (obj.value == java.lang.Float.NaN)	return == true
com.google.gson.JsonPrimitive.equals(java.lang.Object)::EXIT359;condition="not(return == true)"	com.google.gson.JsonPrimitive.equals(java.lang.Object)::EXIT361;condition="return == true"
this.value.getClass().getName() == java.lang.Float.class	

More Display Options:  Invariant Diff Only  Complete Invariants  Source Diff  Source Code

```

359 .....return getAsNumber().doubleValue() == other.getAsNumber().doubleValue();
360 .....double b = other.getAsNumber().doubleValue();
361 .....return a == b || (Double.isNaN(a) && Double.isNaN(b));
362 .....}
363 .....return value.equals(other.value);
364 .....}
365 .....
366 .....Returns true if the specified number is an integral type
367 .....(Long, Integer, Short, Byte, BigInteger)
368 .....
369 .....private static boolean isIntegral(JsonPrimitive primitive){
370 .....if (primitive.value instanceof Number){
371 .....Number number = (Number) primitive.value;
372 .....return number instanceof BigInteger || number instanceof
Integer
373 .....|| number instanceof Short || number instanceof Byte;
374 .....}
375 .....return false;
376 .....}
377 .....}
378 .....

```

Figure 5 Source code changes to `JsonPrimitive.equals`

asking for at least a failing test case, but frames it in terms of the test being useful for future regression testing.

The above scenario highlights the three most salient features of CSI as embodied by GETTY: invariant differentials, impact isolation, and exploration of invocation flows. The display of just the *changes* in invariants provided the reviewer with a concise and focused view on commit #7, despite the fact that the commit had dozens of invariants. As a result, she could make quick inferences about the commit. The ability of the reviewer to explore different combinations of old and new tests helped her isolate the semantic affects due to the source changes. Finally, the summary of the application's call structure around equals helped her quickly focus on a particular part of the program. In a more complicated case, she might have chosen to click on some of the invocations to further explore changes to invariants. In the following sections, we provide additional details on these three elements.

## **3.2 Concepts**

### **3.2.1 Invariant Differentials**

CSI builds on a summary of a program version's behavior. We use Daikon, but many similar tools could suffice. Key for our purposes, however, is summaries of the input-output behavior of methods. These can naturally be phrased in terms of observed invariants. For a dynamic tool like Daikon, these invariants are not absolute, but depend on executions, which we discuss more in the next subsection.

The number of likely invariants for a method before and after a commit can be numerous, and reasoning about their differences can be mentally challenging. However,

because the behavioral changes between program versions can be quite small, so could the differences in their invariants. This motivates the creation of invariant difference sets between program versions to suppress the common invariants and help the reviewer focus her attention on just what's changed since the last commit. For example, for the `equals` method in commit #7, with no isolation of effects (Figure 3), there were 26 invariants before commit and 32 after. As shown in Figure 3, for just the changed invariants, there were just 12 – 3 removed and 9 added – an 80% reduction. With the isolation of effects to the source, there were 37 and 34, before and after commit, actually more than without isolation. Yet as shown in Figure 4 there are just 5 changed invariants – 4 removed and 1 added – a notable 93% reduction.

For each kind of program point of a method  $m$  – entry, exit, and exceptional exit – GETTY calculates the change in invariants between an older version  $a$  and a newer version  $b$  as two sets, the removed invariants and added invariants. For example, for the entry point invariants, the differences are calculated as:

$$\text{removed}(m, a, b) = I(m, a) - I(m, b) \quad (1)$$

$$\text{added}(m, a, b) = I(m, b) - I(m, a) \quad (2)$$

GETTY, following *git*'s style of code differencing, actually displays added, removed, and *changed* invariants. A changed invariant is merely a presentation of an added invariant paired with a removed invariant based on their overall similarity. For example, a removed invariant  $x < 5$  would be paired with the added invariant  $x < 6$  because they involve the same variable, operator, and value type. GETTY currently uses

the minimality of text differences to infer changed invariants, but better results could be achieved by comparing the logical formulae [33].

### 3.2.2 Impact Isolation

As seen in the GSON equals scenario, when developers modify application source code (*abbrev. source*) they often add test cases (*abbrev. tests*) as well, meaning that invariants can change due to either or both source and test changes. CSI must support a reviewer in isolating behavioral impacts to one or the other.

GETTY can show the invariants for a check-in under a variety of conditions. For example, by running the same test cases on both the old and the new source, any resulting invariant differences can be confidently attributed to the source code. By intersecting the two versions' test suites, they should be guaranteed to compile on both source code bases. However, this explicitly excludes test cases that were intentionally written to demonstrate the behavior of a particular version. The union of all tests, on the other hand, will oftentimes have some test cases that will not compile on one version or the other. As a simple example, if a new method is introduced in the new version and some tests are added to test the new method, then these new tests will not compile with the old source. This creates an asymmetry in which test cases run on which version, which seems to defeat the isolation of effects to the source. However, the failure of compilation is really just an early indicator of a failure to run. If our tool were designed for Python, for example, the compilability distinction would disappear into a runtime condition. The fact that a test case runs on one version but not the other reveals a property of the source code:

```

Entry-Point:
  other ≠ null
  other is a JsonPrimitive object

Exit-Points:
  point-359(361 in new source):
    return = true
    (return = false) → (this.value = NaN)
    (return = false) → (other.value = NaN)

```

Figure 6 Test change impact isolation for commit #7, for old source

the behavior is attributable to the source. Thus, the default condition for isolating source effects is to run the union of the test cases, modulo compilability.

To formalize what tests are actually run in the various conditions, we define the Compatible Set of Tests (*CST*) as a function that takes as input a set  $\Gamma_u$  of all test cases of version  $u$  and a set  $\Sigma_v$  of the source of version  $v$ , and returns all test cases in  $\Gamma_u$  that can compile with the source  $\Sigma_v$ .

$$CST(\Gamma_u, \Sigma_v) = \{ \tau \in \Gamma_u \mid \tau \text{ compiles with } \Sigma_v \} \quad (3)$$

We denote the code of version  $v$  a pair  $(\Sigma_v, \Gamma_v)$ , where the first element is the source and the second element the set of test cases. The second row in Table 1 lists the comparison strategy for exploring the effects of source changes. (A more conservative approach would be to use test cases that compile on *both* sources, i.e.,  $CST(\Gamma_{new}, \Sigma_{old}) \cup CST(\Gamma_{old}, \Sigma_{new})$ ).

When we are interested in the impact due to changes in the test cases, we fix the source and execute different versions of the test suites. Since the source code bases cannot be unioned, there are two possible conditions here: running the two test suites on the old source, and running the two test suites on the new source. Figure 6 shows an example of test impact isolation for commit #7, using the old source. The invariant

Table 1 Behavior Comparison Strategies

View Effects of	<i>old</i> version: $(\Sigma_{old}, \Gamma_{old})$ vs. <i>new</i> version: $(\Sigma_{new}, \Gamma_{new})$			
	src	tests	src	tests
whole check-in	$\Sigma_{old}$	$\Gamma_{old}$	$\Sigma_{new}$	$\Gamma_{new}$
source only	$\Sigma_{old}$	$CST(\Gamma_{new}, \Sigma_{old}) \cup \Gamma_{old}$	$\Sigma_{new}$	$CST(\Gamma_{old}, \Sigma_{new}) \cup \Gamma_{new}$
tests for <i>old</i> src	$\Sigma_{old}$	$\Gamma_{old}$	$\Sigma_{old}$	$CST(\Gamma_{new}, \Sigma_{old})$
tests for <i>new</i> src	$\Sigma_{new}$	$CST(\Gamma_{old}, \Sigma_{new})$	$\Sigma_{new}$	$\Gamma_{new}$

differentials convey two pieces of information. First, at the entry-point the `other` object is not always `non-null`, so the reviewer can conclude that the new tests are indeed bringing in the corner case of testing equality of `null`. Second, after introducing the new test cases, at exit-point 359 (361 in the new source), the `old equals` does not always return `true`, but returns `false` on the new testcase when both `this.value` and `other.value` are `NaN`. The developers are expecting to see it fail here, since the passing behavior was introduced by the new source. The observation of a new test case’s impact on the old source demonstrates that the new test is not simply passing all the time, but also capable of revealing incorrect behavior.

As a parting remark, we note that the performance implications of computing all these conditions on every check-in need not be overwhelming. Although we list four conditions, there is substantial overlap among the test runs, allowing for reuse of the logging data. For example, the source-only condition uses the same sources and all the test cases from the whole-check-in condition. Thus, the source-only condition requires new runs for only the compatible new tests on the old code and the compatible old tests on the new code. Moreover, as shown in commit #7, the change in test suites from version to version tends to be small, so the additional computation is minimal.

### 3.2.3 Invocation Flows

A change of one method in the source code can have widespread effects on behavior, affecting numerous methods. This is the primary motivation for providing behavior change summaries, as the summaries directly articulate those widespread effects. Still, a reviewer needs help in finding her way around. Semantic effects are propagated directly by the control flow in the application: a variable or field is set in one method, and then its value is passed to another method, where it is used, set, returned, and so forth. Thus, a natural way for a reviewer to explore a source code base is to navigate its call graph, from caller to callee, from callee to caller, and so forth [29], [41].

Building on this insight, GETTY provides a local-area call-graph, as seen in the dotted boxes in Figure 3 and exhibited in the scenario at the beginning of this section. Only callers, immediate siblings, and callees whose invariants have changed are necessarily displayed. As screen space allows more local neighbors are displayed (in gray, to indicate their invariants were not affected by the commit). Clicking any method in the displayed local call-graph puts that method in the center and displays its callers, immediate siblings, and callees around it. In this way, it is possible to explore all the invariant changes through the program version's control flow.

GETTY computes all invocation flows from execution traces during testing. Because dynamic flows are the flows that actually occurred, a reviewer can compare the dynamic flows with the expected flows based on the code changes to identify problematic or unexpected results [24].



### 3.3 Design and Implementation

Central to CSI is the idea of semantic differencing. GETTY as the tool for CSI takes as input two versions of the target software project and delivers the semantic information to developers. For this purpose, GETTY needs to resolve project dependencies and build it, extract necessary static and dynamic information, collect invariants and incorporate their differentials into the display for developers.

Currently, GETTY is a three-layer system (shown in Figure 7) implementing the above functionalities for analyzing Java projects. The bottom-layer provides infrastructural support for checking out different versions and building them. On top of it is the middle-layer for gathering information: *villa* and *agent* are two application components to extract static and dynamic facts, respectively, from the target project; *center* is the application component to infer invariants for the interested methods at runtime. All information gathered from middle-layer is sent to the top-layer, *gallery*, to be processed and presented to users. We provide more details for each layer in the following sections.

#### 3.3.1 Build Framework

Build framework is the infrastructure layer of GETTY. It is a set of Python scripts that integrates Maven [13] for dependency and build management, and interacts with *git* [42] for version control.

Maven is a software project management and comprehension tool for Java projects. All library dependencies in a Maven project are explicitly declared in the Project Object Model (POM). Build framework uses Maven to automatically retrieve all the

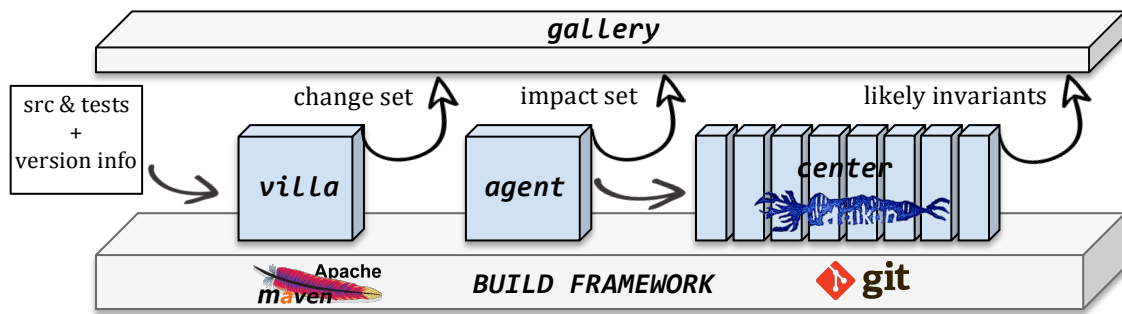


Figure 7 Three-layer GETTY Architecture

dependencies either from online repositories or from local cache, then integrate them into developers' local environment and configure the project as is specified in ".pom" files. When everything is ready the build framework can automatically compile and test the project.

*git* is a popular choice of VCS to store and manage versions. The finest unit of versions is a commit object with a distinguished SHA-1 hash value. Except for the initial commit, each commit documents all changes from its immediate previous commit. Build framework interacts with a project's *git* repository for the following three tasks:

1. Checkout the commits that GETTY needs to analyze and compare;
2. Get file-based, program-level, textual change details via *git-diff* [8];
3. Use *git-stash* [43] to back up developers' current work so after GETTY analysis all work will be restored;
4. Retrieve developers' comments by *git-log* [44].

The typical workflow by the build framework is: back up current work, checkout a specific commit of the project, download and/or integrate all dependent libraries, configure and build the project, send source code and binaries to upper layer for further

processing, clean up and restore developers' work. All processes in the workflow are automatic after user specifies the project and the version commit hash.

### 3.3.2 Fact Extraction

The middle-layer is an application layer on top of the build framework. It collects all information needed for CSI, including static facts, dynamic facts and invariants. In this subsection we provide details for getting static and dynamic facts.

*villa* is a Java component that extracts static facts from source code. Taking as input a textual patch file (output of *git-diff*), *villa* parses the differential information and records the updated source files and changed line numbers of the two versions. From the path of the changed files, build framework tells *villa* whether a change is made to source or tests. Then *villa* fetches the corresponding source code, perform analysis on its AST and correlates changed line numbers to specific methods or test cases. Repeating the above procedure, *villa* computes the set of all methods and test cases that have been changed. We define the computed set to be *change set* of the current patch. In addition, *villa* parses all source code of the project and gets the set of all methods and the set of all test cases for statistics purpose, the union of which is defined as *project set*.

*agent* is a Java component that extracts dynamic invocation information from test executions for invocation flows. *agent* acquires information by Java instrumentation: it adds logging methods into byte codes right before/after each method invocation/return. Each log entry is one line specifying whether it is right before a method call or after. In order to support multi-threading programs, each log entry also records the thread ID of the current invocation so that each thread can be analyzed separately. During analysis,

*agent* maintains one call stack for each thread. It pushes a method onto the stack when it reads the logged method entry, and pops it out when it reads the logged method exit. *agent* counts the number of invocations and computes all invocation relations based on the stack activities.

### 3.3.3 Dynamic Invariant Inference

*center* is a Python component responsible for inferring invariants from test execution. It checks out two code versions and executes tests for invariants. Since a method can be exercised by any tests, directly or indirectly, we always execute the whole test suite so that no runtime data is missed. This implies a great performance overhead. *center* takes two measures to improve the performance.

First, we need not infer invariants for all methods in *project set*. Notice that the invariants of a method may impact its neighbors (callers, callees, methods immediately called by callers before/after calling this method), so the interested targets include methods in *change set* and all their neighbors. We define the set of all changed methods and their neighbors (to a certain depth) to be the *impact set* of current patch. *center* takes as input the invocation flow information and computes such *impact set*.

Second, notice that software testing is embarrassingly parallel, and so is invariant detection. *center* forks a set of processes to run Daikon and infer invariants from testing. Depending on the available computational resources the number of processes may vary. We will further evaluate factors we considered for penalization and their effects in Chapter 4.

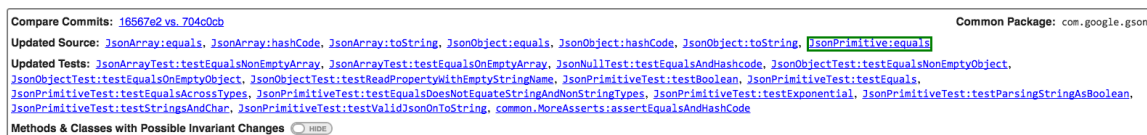


Figure 8 Patch Summary Zone

For the purpose of impact isolation (Section 3.2.2), *center* checks out different (combinations of) versions, builds the project, runs all tests and infers invariants. After inferring invariants, *center* differences the result for future use.

### 3.3.4 User Interface Generation

The top-layer, *gallery*, synthesizes all information from middle-layer components and creates a user interface (UI) for reviewers. We discussed how a reviewer would interact with the UI in Section 3.1. In this section we provide details on how the UI is constructed.

As shown in Figure 3, there are three zones in the user interface, displayed as three boxes from top to bottom:

- Patch Summary Zone (Figure 8). The top zone is a change summary between two commits. The “Compare Commits” shows the two commits under compare. Clicking into the link reviewers will see all developers’ comments documented by *git-log*. “Common Package” to the upper right corner displays the common

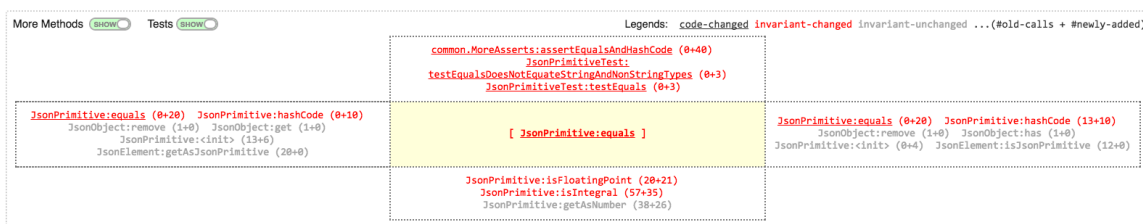


Figure 9 Invocation Flow Zone

Invariant Changes Due To:	
<a href="#">Source &amp; Test Change</a>	<a href="#">Source Change Only</a>
<a href="#">Test Change (for OLD Source)</a>	<a href="#">Test Change (for NEW Source)</a>
compare invariants for ( com.google.gson.JsonPrimitive.equals )	
REMOVED	ADDED
com.google.gson.JsonPrimitive.equals(java.lang.Object)::ENTER	com.google.gson.JsonPrimitive.equals(java.lang.Object)::ENTER
obj != null	(return == true) ==> (orig(obj) != null)
obj.getClass().getName() == com.google.gson.JsonPrimitive.class	(return == true) ==> (orig(obj).getClass().getName() == com.google.gson.JsonPrimitive.class)
com.google.gson.JsonPrimitive.equals(java.lang.Object)::EXIT	com.google.gson.JsonPrimitive.equals(java.lang.Object)::EXIT
com.google.gson.JsonPrimitive.equals(java.lang.Object)::EXIT346	com.google.gson.JsonPrimitive.equals(java.lang.Object)::EXIT346
com.google.gson.JsonPrimitive.equals(java.lang.Object)::EXIT356	orig(obj).getClass().getName() == com.google.gson.JsonPrimitive.class
com.google.gson.JsonPrimitive.equals(java.lang.Object)::EXIT356;condition="not(return == true)"	com.google.gson.JsonPrimitive.equals(java.lang.Object)::EXIT349
this.value.getClass().getName() == java.lang.Long.class	orig(obj).getClass().getName() == java.lang.Object.class
com.google.gson.JsonPrimitive.equals(java.lang.Object)::EXIT356;condition="return == true"	return == false
com.google.gson.JsonPrimitive.equals(java.lang.Object)::EXIT361	com.google.gson.JsonPrimitive.equals(java.lang.Object)::EXIT356
com.google.gson.JsonPrimitive.equals(java.lang.Object)::EXIT;condition="return == true"	orig(obj) != null
	orig(obj).getClass().getName() == com.google.gson.JsonPrimitive.class
	com.google.gson.JsonPrimitive.equals(java.lang.Object)::EXIT361
	orig(obj).getClass().getName() == com.google.gson.JsonPrimitive.class
	com.google.gson.JsonPrimitive.equals(java.lang.Object)::EXIT363
	orig(obj) != null
	orig(obj).getClass().getName() == com.google.gson.JsonPrimitive.class
	com.google.gson.JsonPrimitive.equals(java.lang.Object)::EXIT;condition="return == true"
	orig(obj) != null
	orig(obj).getClass().getName() == com.google.gson.JsonPrimitive.class

More Display Options: [Invariant Diff Only](#) [Complete Invariants](#) [Source Diff](#) [Source Code](#)

Figure 10 Semantic Inspection Zone

package name of all the methods and test cases displayed in the UI. “Updated Source” and “Updated Tests” lists the methods and test cases in *change set*, computed by *villa*. At last, reviewers have the option to display all methods and classes whose invariants might have been changed after the code change.

- Invocation Flow Zone (Figure 9). The middle zone displays the invocation flow information computed by *agent* component. The zone is initially empty. After a reviewer clicks one of the methods listed in the Patch Summary Zone, that method will be placed in the center of Invocation Flow Zone with its four neighbors updated. *gallery* places two toggle buttons to the upper left corner, serving as two display options: clicking the “More Methods” toggle button shows/hides the methods with no invariant change; clicking the “Tests” toggle button shows/hides test cases. If the reviewer clicks any of the neighboring methods, that method will be placed in the middle with all of its four neighbors updated.

- Semantic Inspection Zone (Figure 10). The bottom zone displays invariant differentials (computed by *center*) for the method placed in the middle of Invocation Flow Zone. On top of the display, *gallery* lists all the isolation options, depending on which *gallery* loads and displays the corresponding invariant differentials. At the bottom are more display options, in case the reviewer needs to view the complete texts of invariants, or the source code.

### **Acknowledgments**

Section 3.3 contains the material as it appears in “Mining Software Contracts for Software Evolution”. Yan, Yan; Menarini, Massimiliano; Griswold, William G. IEEE International Conference on Software Maintenance and Evolution, 2014. The dissertation author was the primary investigator and author of this paper.

This chapter, in part, is currently being prepared for submission for publication of the material, Yan, Yan; Menarini, Massimiliano; Griswold, William G. “Mining Code Repositories for Semantics-Assisted Code Review” The dissertation author was the primary investigator and author of this material.

# Chapter 4

## Performance and Scalability

We have discussed the design and implementation of our CSI tool, GETTY, and used it to complete one case study of GSON project. In this chapter, we discuss and evaluate our solution to scale GETTY for more projects.

### 4.1 Challenges

Like Continuous Integration (CI), CSI depends on heavy lifting by the back-end to support developers' and reviewers' work. Indeed, CSI not only requires the same compilation and testing support of CI, but also adds the often massive cost of inferring likely invariants with Daikon, even for a reduced set of targets (impact set, Section 3.3.3). More specifically, consider GSON project at the commit #4 of our previous case study. There were 676 methods and 707 test cases in the project. Under the environment of a Macbook Pro (Intel 2.53 GHz Dual-Core CPU, 4GB DDR3 RAM, Mac OS X Yosemite), one pass of the test suite execution, without any optimization or parallelization, takes over 10 seconds to complete. When we execute all tests for each method, one after another, the total execution time is nearly 7,121 seconds, nearly 2 hours. The overhead rises to over 5 hours after running Daikon instrumented tests and inferring invariants. Another open source project, Apache Commons Collections [36], contains more than 3,000 methods with more than 1,500 tests. Under the same environment, running the



Table 2 Execution Time in ATAM Mode

<b>GETTY components</b>	<i>villa</i>	<i>agent</i>	<i>center</i>	<i>gallery</i>
Avg. Timespan (seconds)	78.86	250.14	9716.01	89.74
Avg. Percentage	0.78%	2.47%	95.86%	0.89%

Environment: Intel 2.53 GHz Dual-Core, 4GB DDR3 RAM, Mac OS X Yosemite

same analysis above for a random pair of commits in Collections project takes more than 4 days to complete.

However, an ideal CSI infrastructure like GETTY should support an overnight build. For example, a review of current practices at Apache and Mozilla show that developers typically perform code reviews 5 to 21 times per week [5]. Therefore, review tools that take over a day to complete invalidate their practical use for nightly builds. As a heuristic we estimate the analysis be completed in 12 hours.

## 4.2 Methods to Improve Performance

The unoptimized analysis in the previous section always executes the whole test suite for all methods at once. For reference, we call it the All-Test-All-Methods mode (**ATAM**). Notice that we only need the invariants that are relevant to the commit in question, so we only need to let GETTY analyze the *impactset*. We selected 10 random commit pairs from GSON, ran GETTY analysis in ATAM mode, collected performance data of the four components and summarized the average timespan for each component in Table 2. Because of the reduced inference workload, the total time span is reduced. However, the average time span is still over 3 hours. Of all the components, *center* takes over 95% of the total execution time, even when just focusing on *impactset*, so dynamic invariant inference is the performance bottleneck.

Table 3 Project Information and Execution Time of Dynamic Inference

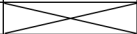
Projects	Commits	KLoC	Methods	Tests	TTE	TIE (ATAM)
GSON <sup>G</sup>	1,294	36.6	582	1,276	13.3s	3.22h
CLI <sup>A</sup>	827	8.6	212	206	18.0s	5.98h
Codec <sup>A</sup>	1,608	14.1	272	334	23.3s	11.64h
Crypto <sup>A</sup>	550	9.9	183	20	49.5s	23.20h
Collections <sup>A</sup>	2,881	100.1	3,177	1388	86.9s	55.51h
Configuration <sup>A</sup>	2,730	92.3	2,121	1,962	127.5s	

G: Google Project; A: Apache Commons Project. **TTE**: total time for testing; **TIE**: total time for inference  
 Environment: Intel 2.53 GHz Dual-Core, 4GB DDR3 RAM, Mac OS X Yosemite

To further study how we can improve performance of dynamic inference, we randomly selected five more projects from Apache Commons repositories [45] under directory “C”. We took 10 random commits of each project, executed the *center* analysis on them, and take the truncated mean (discarding two side outliers) of each group of data. Their characteristics and the results are summarized in Table 3. The total time taken by the inference process (last column) increases with the growing execution time of executing test suites. The cost of inference has no evident relationship with the other factors. The size of a commit will influence the size of the target set and hence the inference time, but we controlled for this by taking the truncated average of the 10 commits per project. The cost is far too high for the last three projects. Various bottlenecks are in play, but the most persistent one is that inferring all the invariants at once requires far more memory than available RAM, causing Daikon to thrash in virtual memory.

To reduce memory pressure, we used a mode of Daikon in which the traces are piped directly to Daikon (less cost from disk I/O), and introduced two modes that track fewer methods at a time:

Table 4 Execution Time (Hours) of Different Modes (1)

Projects	ATAM	ATSM	ATSC	PATSC
GSON	3.22	3.94	2.70	2.32
CLI	5.98	5.38	3.10	2.79
Codec	11.64	14.90	3.67	3.52
Crypto	23.20	39.53	9.43	7.99
Collections	55.51	79.85	25.36	23.83
Configuration		95.69	54.81	49.96

Environment: Intel 2.53 GHz Dual-Core, 4GB RAM, Mac OS X Yosemite

- All-Tests-Single-Method (**ATSM**) mode – executing the whole test suite for a single method each time;
- All-Tests-Single-Class (**ATSC**) mode – executing the whole test suite for a single class each time.

The rationale behind the two modes is to release Daikon from processing too many data points at a time, though it could result in many more test suite executions. Theoretically, ATSM processes data points for one single method so the memory pressure each time should be minimal. ATSC processes all data points for a class (i.e., a group of methods), which imposes more memory pressure than one single method; however, it reduces the total number of repeated test executions. Moreover, since a method program point is dependent on its parent records (typically the containing class program points) [46], each method program point data processing involves its class program point data processing. ATSC processes class program point data once for all methods of that class; therefore it minimizes the repeated data processing for class program points.

Table 4 records the average execution time of using different modes in the six open source projects, under the environment of a dual-core Macbook Pro with 4GB

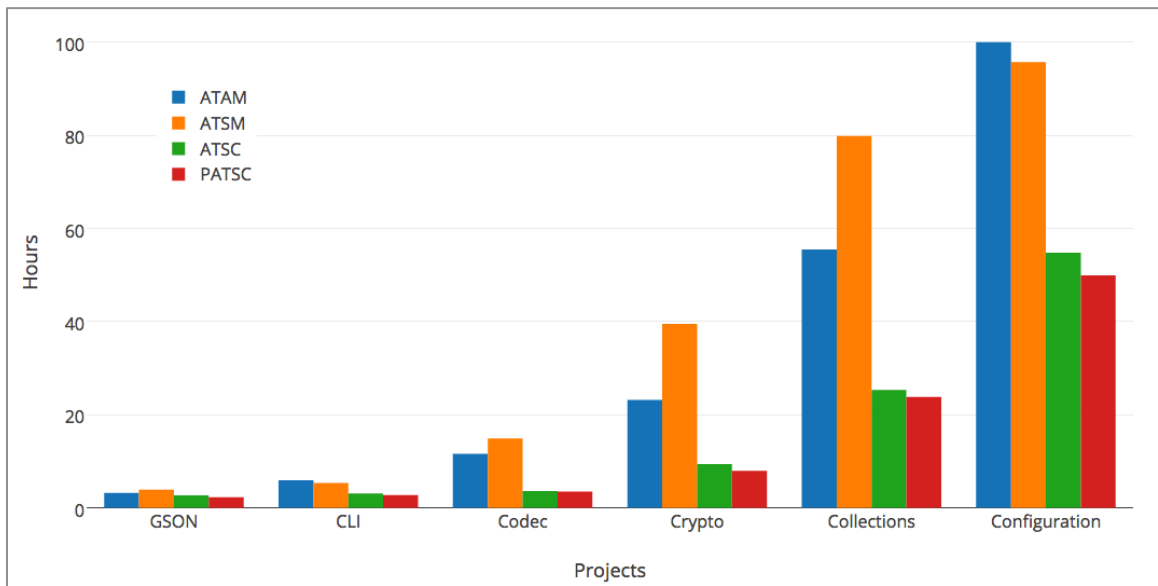


Figure 11 Inference Execution Time of 6 Open Source Projects

RAM. The first three columns compare the performance impact among the three modes. All data in Table 4 is visualized in Figure 11 for comparison purpose. Compared to the unoptimized ATAM mode, ATSM mode got invariants for the Configuration project, but it also increased the total time of inference for most projects because the test suite was executed many more times repeatedly. ATSC results in an overall improvement, but Collections and Configuration are still processed too slowly.

Noticing that invariant detection is embarrassingly parallel, our final improvement was to introduce the Parallel-ATSC mode (**PATSC**), which exploits the fact that most tests are naturally independent from each other. We partition each project's class-granularity inference processes into as many groups as there are cores on the machine, and distribute them to all cores. Each group for a project is executed concurrently on a separate core, and on each core the processes of the group are executed sequentially in batch. After all the groups finish, *center* merges all the invariants collected, a trivial step.

Table 5 Execution Time (Hours) of Different Modes (2)

<b>Projects</b>	<b>ATSC</b>	<b>PATSC</b>
GSON	2.14	1.09
CLI	2.27	1.21
Codec	3.02	1.75
Crypto	7.85	3.02
Collections	20.19	11.89
Configuration	43.66	25.40

Environment: Intel 2.66 GHz Octa-Core, 16GB RAM, Ubuntu Server 16.04

The last column of Table 4 lists the execution time of PATSC mode. From Figure 11, PATSC mode improved the performance in general; however, its average improvement was approximately only 10%, notably less than expected for dual-core environments. This confirms that the memory pressure is the key: it not only limits each single process of inference, but also impedes the potential speedup from parallelism.

Fortunately, higher-performance hardware is reasonably common and affordable in today's cloud or cluster environment; for example, as of 2016, the price of renting an Octa-core, 32GB RAM instance at Amazon EC2 is about 50 cents per hour. To evaluate the performance improvement by PATSC over plain ATSC, we moved our experiments to a higher-performance machine with an Octa-core processor, whose 2.66GHz clock speed is 5% higher than the 2.53GHz on the Macbook Pro used in the prior experiments, and has four times the RAM, 16GB. In this new environment, we executed the same *center* analysis in ATSC and PATSC modes, and recorded the time in Table 5. It shows that PATSC mode (parallelizing on 8 cores of one single node) achieved nearly 100% improvement over the plain ATSC condition. Its suboptimal speed-up is still due to the cores' contention on the memory bus to the shared memory. However, compared to the previous environment (about 10% improvement with 4GB RAM), the speed-up by

Table 6 Execution Time (Hours) of PATSC Mode

Projects	2 processors (1 node)	4 processors (2 nodes)	8 processors (4 nodes)	16 processors (8 nodes)
GSON	1.09	0.50	0.26	0.16
CLI	1.21	0.70	0.34	0.20
Codec	1.75	0.78	0.41	0.23
Crypto	3.02	1.74	0.90	0.51
Collections	11.89	8.75	4.94	1.89
Configuration	25.40	11.65	6.14	3.03

Environment: Intel 2.66 GHz 4 Dual-Core, 16GB RAM, Ubuntu Server 16.04

PATSC mode with more RAM is much more notable, demonstrating its potentially more performance improvement when the memory pressure is further released.

### 4.3 Scalability

While the ATSC mode is appropriate for a desktop environment, the highly parallel PATSC mode is more appropriate to a high-performance cluster. For evaluating PATSC's scalability, we moved to a cluster with eight high-performance nodes (each with the same hardware and software environments as is in the previous section), typical of what can be found in a cloud deployment today.

We conducted experiments for the six projects on 2, 4, 8 and 16 processors, and summarized their results in Table 6. The last column indicates that running on all eight nodes provides 6.8x average speedup over single-node computation. The first four projects, GSON, CLI, Codec and Crypto, all completed their analysis within 1 hour. Collections project completed in 2 hours. Configuration project took the longest time to complete; however, it still completed with approximately 3 hours, over 8x speedup over single-node analysis, and over 14x improvement over the plain ATSC mode.

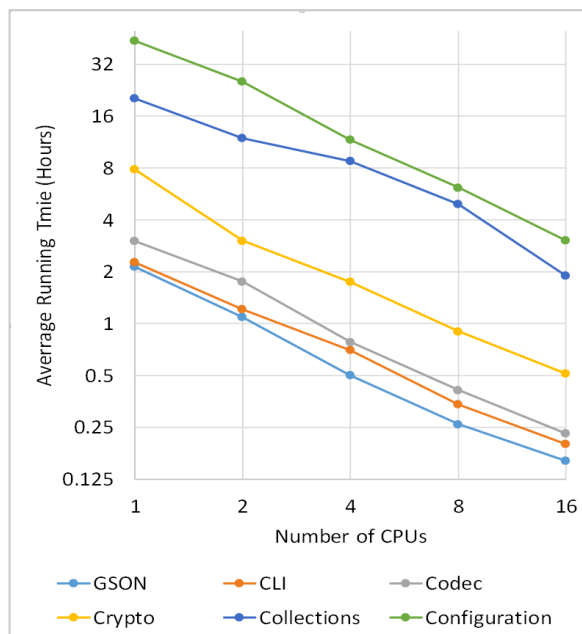


Figure 12 Average running time versus number of processors

Figure 12 plots running time against the number of processors used. Overall the completion time decreases with increasing number of nodes used for each project. The single-processor condition runs plain ATSC, the rest PATSC. ATSC runs 1.2x faster on this machines processor compared to the Macbook Pro. PATSC running on the same single node (using both processors) achieved 1.8x speedup and 89% efficiency. Its suboptimal speed-up is due to memory contention. Running on all 16 processors (eight nodes) provides a total speedup is 13.1 with 82% efficiency, with a nearly linear speedup across the range, implying high scalability of invariant inference for CSR.

Related is the cost of computing GETTY's invariants in the cloud, say as part of an existing continuous integration process. Figure 13 plots the estimated additional CPU cost on Amazon EC2. The costs are modest, tracking project size. The high efficiency of parallelization modestly increases the baseline, average 24%.

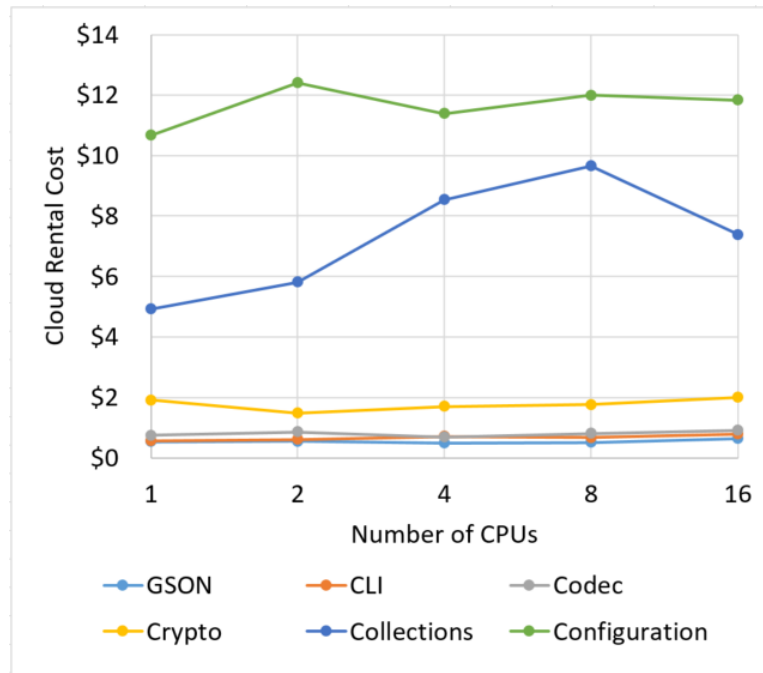


Figure 13 Cloud cost of invariant extraction against the number of CPUs used

The scalability of PATSC is bounded by the number of classes in the impact set. For the 60 commits examined here, the average number of classes is 142, with a standard deviation of 148, suggesting generally ample parallelism. At the low end there are a few commits that contain just 12 classes, for example commit c241318 in Collections. Its times for invariant inference are 0.57, 0.31, 0.23, and 0.21 hours, on 2, 4, 8, and 16 processors, respectively. The overall speedup from 2 to 16 processors is 2.7x, with no discernable speedup from 8 to 16 processors since the maximum expected speedup is 12, and the net time is bounded by the longest running class. However, all the small commits also have short running times that don't demand high levels of parallelization.

In summary, by partitioning the traces sent to Daikon it is possible to reduce memory pressure and achieve high parallelism in inference, providing high scalability of CSI, sufficient for supporting its use as part of a Continuous Integration process.



**Acknowledgments**

This chapter, in part, is currently being prepared for submission for publication of the material, Yan, Yan; Menarini, Massimiliano; Griswold, William G. “Mining Code Repositories for Semantics-Assisted Code Review” The dissertation author was the primary investigator and author of this material.

# Chapter 5

## Quantitative Evaluation

In Section 3.1 we replayed part of the history of the `JsonPrimitive:equals` method in the GSON project. In this chapter, we finish the introduced case study and demonstrate that CSI with GETTY can help reveal insufficiency of testing and find bugs. For all 12 commits related to the target methods, we compare the number of changed invariants with the total number of invariants and evaluate the percentage of the invariant reduction which creates a more focused view for reviewers. By looking at a larger number of projects and commits, we further address the question of whether CSI is similarly effective on a regular basis. In addition to GSON, we studied the five open source projects introduced in the previous section: CLI, Codec, Crypto, Collections, and Configuration (Table 3). We first examine 100 of their commits to assess CSI's ability to make judgments on test sufficiency. We then investigate whether CSI aids in finding bugs in the six projects.

### 5.1 Finishing the GSON Case Study

From Section 3.1, we have already seen that CSI with GETTY can both quickly reveal inadequate testing and successful revisions in GSON. Replaying the history of `equals` from the beginning repeatedly shows this ability (See Figure 14), so for the sake of brevity we focus here on whether CSI can aid in finding the introduction of the bugs.

```

Entry-Point:
  other has only one instance
  other = this

Exit-Point:
  return = true

```

(a) Commit #1: equals method added, tested on just one value.

```

Entry-Point:
  other has only one instance
  other ≠ null

Exit-Point:
  return = true

```

(b) Commit #2: more tests improve the invariant differentials, but there is no testing for a null input.

```

Exit-Points: (AFTER commit)
point-362:
  this.value = null
  (return = true)  (other.value = null)
point-364:
  this.value ≠ null
  (return = true)  (this.value = other.value)

```

(c) Commit #3: branch refactoring of equals; invariant differentials indicate that it is behaving as expected.

- For presentation purposes, fully qualified names have been shortened.

Figure 14 Selected dynamic invariant differentials for commits #1 - #3

We highlight commit #10 in Figure 15, which introduced a test comparing two JSON primitive integers:  $2^{64}+5$  and 5. As asserted, the equals method should return false because the two values are different. However, the test failed, revealing a bug. The bug was never fixed. (In the last commit, #12, the developers considered “the price is too much to pay” to fix the bug and overrode the test failure by changing its “assertFalse” to “assertTrue”.)

It is interesting to note that the bug was not revealed until commit #10, despite all previous commits passing their code reviews. The question, then, is whether CSI could have aided in finding this bug sooner, preferably at the point of introduction, were it in use by this project at the time.

```

public void testEqualsIntegerAndBigInteger() {
    JsonPrimitive a = new JsonPrimitive(5L);
    JsonPrimitive b = new JsonPrimitive(new BigInteger("18446744073709551621")); // 2^64 + 5
    assertFalse(a + " equals " + b, a.equals(b));
}

```

Figure 15 The failing test case in commit #10

In commit #4, developers introduced new features such that: (1) integers of different types (Byte, Short, Integer, Long, and BigInteger) are comparable to each other in equals, and (2) floating-point numbers of different types (Float, Double, BigDecimal) are comparable to each other.

Figure 16 shows the two new if-branches added in equals to compare integers and floating-points. The predicate method `isIntegral` checks whether a `JsonPrimitive` object represents an integer, i.e., the type of value attribute is Byte, Short, Integer, Long, or BigInteger. If both this and other represent integers, the first branch is executed and both value attributes are converted to Long to compare for equality. Similarly, the predicate `isFloatingPoint` checks whether a `JsonPrimitive` object represents a decimal, i.e., the type of value attribute is Float, Double, or BigDecimal. If both this and other store decimals, the second branch is executed and value attributes are converted to Double to compare for equality. The two branches are

```

374+ if(isIntegral(this) && isIntegral(other)) {
375+     return getAsNumber().longValue()== other.getAsNumber().longValue();
376+ }
377+ if(isFloatingPoint(this) && isFloatingPoint(other)) {
378+     return getAsNumber().doubleValue()== other.getAsNumber().doubleValue();
379+ }

```

Figure 16 New feature to compare between integers in commit #4

```

Exit-Points:
point-375:
    (return = false) → (this.value is Long)

point-378:
    return = true

```

Figure 17 Commit #4: the invariant differential clearly indicates a bug.

independent from each other since the two predicates separate all input numbers into two disjoint sets.

We first study the branch from line 374 to line 376 for integers. As reviewers we expect the branch returns `true` when both `value` attributes store the same integer values, regardless of the specific integer types.

We examine dynamic invariant differentials in Figure 17. At exit-point of line 375, where the new integer-comparison branch returns, `this.value` must be `Long` if the return value is `false`. This is surely an incorrect invariant because the type of `this.value` being `Long` is not a necessary condition for `equals` to return `false`. The new branch can return `false` as long as both `value` attributes are quantitatively unequal, even when one is or both are not `Long`. In this regard, we suspect a bug was introduced that created an incorrect dependency between the result of comparison and the types of `value` attributes.

To verify if the bug actually exists, we need to add a test. The incorrect invariant grants the insight on how to write the test. We challenge the incorrect `Long` type dependency by creating and comparing `BigInteger` objects that cannot be precisely converted to `Long`. Java's `Long` type is 64-bit signed integer so any integer representation over 64 bits will be truncated when converting to `Long`. For example,  $2^{64}+1$  that takes (at least) 65 bits is converted to 1 because all bits beyond the 64<sup>th</sup> are lost after conversion.

```

public void testEqualsForBigIntegers() {
    BigInteger limit = new BigInteger("18446744073709551616"); // 2^64
    JsonPrimitive one = new JsonPrimitive(1L);
    JsonPrimitive lp1 = new JsonPrimitive(limit.add(new BigInteger("1"))); // limit + 1
    JsonPrimitive lp1c = new JsonPrimitive(limit.add(new BigInteger("1"))); // another limit + 1
    JsonPrimitive lp2 = new JsonPrimitive(limit.add(new BigInteger("2"))); // limit + 2
    // compare 1, limit + 1, limit + 2, etc.
    assertFalse("limit + 1 = 1", lp1.equals(one));
    assertFalse("1 = limit + 1", one.equals(lp1));
    assertFalse("limit + 1 = limit + 2", lp1.equals(lp2));
    assertTrue("limit + 1 = limit + 1", lp1.equals(lp1c));
}

```

Figure 18 Test case to confirm the integer-equality bug

We therefore create four `JsonPrimitive` objects: `one` for 1, `lp1` for  $2^{64}+1$ , `lp1c` for  $2^{64}+1$  (a different object), and `lp2` for  $2^{64}+2$ , and assert that none of them are equal except for the pair of `lp1` and `lp1c`. Our test in Figure 18 is stronger than the test in Figure 15 because that test only considers the case where `other.value` is `BigInteger`, but ours considers `this.value` being `BigInteger` as well.

Since our test fails, we conclude that the bug discussed was introduced in commit #4. This is a typical case where a bug was introduced after the developers added a new feature. Notice that commit #4 is dated Sep 23, 2009, but commit #10 discovered the bug on Sep 9, 2011, nearly 2 years after the bug was introduced. Reviewers could have found the bug much earlier if they had been able to examine dynamic invariant differentials.

Additionally, consider the branch from line 377 to line 379 in Figure 16 that deals with floating-point comparisons. Similarly to the previous integer branch, we expect this branch to return `true` when both represent the same decimal values, regardless of the specific decimal type of `value` attribute.

```

public void testUnequalDecimals() {
    JsonPrimitive smaller = new JsonPrimitive(1.0);
    JsonPrimitive larger = new JsonPrimitive(2.0);
    assertFalse("smaller = larger", smaller.equals(larger));
    BigDecimal dmax = BigDecimal.valueOf(Double.MAX_VALUE);
    JsonPrimitive smallBD = new JsonPrimitive(dmax.add(new BigDecimal("100.0"))); //dmax + 100.0
    JsonPrimitive largeBD = new JsonPrimitive(dmax.add(new BigDecimal("200.0"))); //dmax + 200.0
    assertFalse("smallBD = largeBD", smallBD.equals(largeBD));
}

```

Figure 19 Test case to confirm the decimal-equality bug

In Figure 17, the dynamic invariant at exit-point 378 says the return value is always true, i.e., for all tests so far this branch returns true only. This indicates that either there is a lack of testing for unequal decimals, or unequal decimals are compared but there is a bug.

Consequently, we add a test case (Figure 19) to compare unequal decimals to check for a bug. Building on previous experience, we consider not only unequal `Double` numbers but also unequal `BigDecimal` numbers. Our test case passes the first assertion but fails the second one. Passing the first assertion implies that `equals` behaves correctly given two small unequal decimals, confirming our hypothesis that the wrong invariant was due to lack of testing. The failure of the second assertion reveals a new bug when comparing large unequal numbers. This bug was never found or discussed in GSON project. In commit #11 (right after developers discovered the integer comparison bug in commit #10), the developers further modified the same decimal comparison branch; but

```

Exit-Points:
point-348:
(return = false) → (this.value is LazilyParsedNumber)

```

Figure 20 Commit #11: invariant differential points to a similar bug as commit #4

```

316  if(isIntegral(this) && isIntegral(other)) {
317-   return getAsNumber().longValue() == other.getAsNumber().longValue();
318  }

                                     Before bug-fix

316  if(isIntegral(this) && isIntegral(other)) {
317+   return getAsBigInteger().equals(other.getAsBigInteger());
318  }

                                     After bug-fix

```

(a) Suggested fixing patch

```

Exit-Points:
point-317: {return = false} → {this.value is Long}

```

(b) Incorrect invariant disappeared after bug-fix

Figure 21 Fixing the introduced integer-equality bug

the invariant differential in Figure 20 shows that, similar to exit-point 375 in Figure 17, at the exit-point of the decimal branch the return value is incorrectly correlated to the specific type of `value` attribute. Up to the date of this paper’s writing, the developers did not add any tests to reveal this bug. As a byproduct of our research, we submitted the new bug report, along with the test in Figure 19, to the GSON project.

Many of the problems identified here could in principle have been identified through test coverage reports. However, although standard test coverage tools confirm that the conditions of a branch were tested, they do not reveal coverage of the domain and range of methods. Invariant differentials directly state the anomalous properties of the input (e.g., `other` is never `null`) or output, pointing to what kinds of tests need to be added (inputs that include `null`). And coverage tools do not help in identifying bugs, just areas of the code that are insufficiently tested. For example, in commit #4 there were 65 executions of `equals` for numeric equalities. Among them, 16 tested integer equalities



and 30 tested decimal equalities. We did not have to examine all test executions to identify the missing test cases, but we only examined dynamic invariant differentials and compared them with our expectation. We concluded not only were they insufficiently tested, but also both branches have bugs.

Since the loss of precision in conversion is identified as the root cause of the integer comparison bug, we can fix it by converting each integral value attribute to `BigInteger` before comparing for equality. Figure 21-a shows our suggested fix. We reviewed our own commit using CSI and confirmed the incorrect invariant disappeared after the bug fix (Figure 21-b). Our real fix [47] submitted to GSON project was more comprehensive. To avoid introducing bugs into other parts of the program, we inspected more related invariant differentials and invocation flows. For example, we further examined the semantics of `isIntegral` and found that the only other method using this predicate was `hashCode`. We concluded this case study by adding tests to ensure `hashCode` was not affected by the bug fix.

## 5.2 Reducing Inspection Load

We surmise that, in most cases, not all invariants are changed after a patch. Invariant differentials are a subset of invariants that are changed. In the finished GSON case study, it was the invariant differentials, not the full list of invariants, that we needed to inspect for a lighter inspection load. In this section we study whether the size of invariant differentials is related to the size of the code diff, and how much invariant differentials shrink compared to the full list of invariants.

Table 7 Results of Invariant Reduction without Impact Isolation

$$\text{Diff Correlation ( (Added Invs + Deleted Invs), (RSL + RTL) )} = -0.169$$

Commit	RSL	RTL	Old Invs	New Invs	Added Invs	Deleted Invs	Reduction
#2	0	25	23	25	2	0	95.83%
#3	6	0	25	23	0	2	95.83%
#4	15	65	23	26	3	0	93.87%
#6	38	11	35	35	0	0	100%
#7	3	25	26	34	12	4	73.33%
#8	12	5	35	35	0	0	100%
#9	1	41	35	35	0	0	100%
#10	0	4	41	41	0	0	100%
#11	7	0	41	42	2	1	96.38%
#12	0	1	42	42	0	0	100%
<b>Average</b>	8.2	17.7	32.6	33.8	1.9	0.7	95.52%

RSL: # related *source* Logical Lines of Code changed; RTL: # related *test* Logical Lines of Code changed

Among the 12 commits related to the target method, commit #1 is the one introducing the method so its invariant differential is exactly the full list of invariants inferred; commit #5 is the only commit that does not compile, so there are no invariants inferred and hence there are no invariant differentials. For our evaluation we exclude these two commits and study the other 10 commits for the more general case when a patch modifies part of the existing method and the patch compiles.

In Table 7 we summarize the results of the 10 commits without impact isolation. For each commit, we count the number of related logical lines that were changed. For the source files, the related lines are the lines for the `equals` method, and all methods that the `equals` method calls, for example, the `isIntegral` method. For the test files, the related lines are the lines of test cases that invoked the `equals` method, either directly or indirectly. On average, at one commit developers updated 8.2 lines of code for `equals` method, and meanwhile updated 17.7 lines of code (3.6 test caess) for testing it. Similarly, comparing the invariants, we can observe some old invariants were removed in

Table 8 Results of Invariant Reduction under Source Impact Isolation

Diff Correlation ( (Added Invs + Deleted Invs), RSL) = -0.201

Commit	RSL	Old Invs	New Invs	Added Invs	Deleted Invs	Reduction
#2	0	23	25	2	0	95.83%
#3	6	25	23	0	2	95.83%
#4	15	23	26	2	0	95.91%
#6	38	35	35	0	0	100%
#7	3	37	34	3	4	90.14%
#8	12	35	35	0	0	100%
#9	1	35	35	0	0	100%
#10	0	41	41	0	0	100%
#11	7	41	42	2	1	96.38%
#12	0	42	42	0	0	100%
<b>Average</b>	8.2	33.7	33.8	0.9	0.7	97.41%

RSL: # related *source* Logical Lines of Code changed; RTL: # related *test* Logical Lines of Code changed

the newer version, while some new invariants were introduced in the newer version. We thus compute the reduction of invariants by each invariant differential as follows:

$$\text{Reduction} = 100\% - (\text{Added Invs} + \text{Deleted Invs}) / (\text{Old Invs} + \text{New Invs}) \quad (4)$$

For the 10 commits, we achieved on average 95.52% reduction without impact isolation, indicating that reviewers would need to inspect about every 20 invariants under this isolation condition, a evident reduction. Underneath Table 7 we also computed the correlation between the size of code diffs and that of invariant differentials. The correlation result of -0.169 indicates that the size of invariant differentials is not directly related to the size of a patch.

We further study the 10 commits under different isolation conditions. Table 8 shows the result under source impact isolation. We achieved on average 97.41% invariant reduction for inspection, a slightly higher reduction compared to the condition without impact isolation. The correlation between the size of source diffs and that of invariant

differentials is -0.201, which shows the size of invariant differentials under source impact isolation condition is not directly related to the size of source change.

The results under test impact isolation condition are listed in Table 9 (for old source) and Table 10 (for new source). We achieved on average 97.30% and 97.13%

Table 9 Results of Invariant Reduction under Test Impact Isolation for Old Source

Diff Correlation ( (Added Invs + Deleted Invs), RTL ) = 0.118

Diff Correlation ( (Added Invs + Deleted Invs), RTC ) = -0.049

<b>Commit</b>	<b>RTL</b>	<b>RTC</b>	<b>Old Invs</b>	<b>New Invs</b>	<b>Added Invs</b>	<b>Deleted Invs</b>	<b>Reduction</b>
#2	25	4	23	23	0	0	100%
#3	0	0	25	25	0	0	100%
#4	65	13	23	23	0	0	100%
#6	11	3	35	35	0	0	100%
#7	25	3	26	37	14	3	73.01%
#8	5	2	35	35	0	0	100%
#9	41	9	35	35	0	0	100%
#10	4	1	41	41	0	0	100%
#11	0	0	41	41	0	0	100%
#12	1	1	42	41	0	0	100%
<b>Average</b>	<b>17.7</b>	<b>3.6</b>	<b>32.6</b>	<b>33.6</b>	<b>1.4</b>	<b>0.3</b>	<b>97.30%</b>

RTL: # related test Logical Lines of Code changed; RTC: # of related test cases updated

Table 10 Results of Invariant Reduction under Test Impact Isolation for New Source

Diff Correlation ( (Added Invs + Deleted Invs), RTL ) = 0.167

Diff Correlation ( (Added Invs + Deleted Invs), RTC ) = -0.001

<b>Commit</b>	<b>RTL</b>	<b>RTC</b>	<b>Old Invs</b>	<b>New Invs</b>	<b>Added Invs</b>	<b>Deleted Invs</b>	<b>Reduction</b>
#2	25	4	25	25	0	0	100%
#3	0	0	23	23	0	0	100%
#4	65	13	25	26	1	0	98.03%
#6	11	3	35	35	0	0	100%
#7	25	3	26	34	12	4	73.33%
#8	5	2	35	35	0	0	100%
#9	41	9	35	35	0	0	100%
#10	4	1	41	41	0	0	100%
#11	0	0	42	42	0	0	100%
#12	1	1	42	42	0	0	100%
<b>Average</b>	<b>17.7</b>	<b>3.6</b>	<b>32.9</b>	<b>33.8</b>	<b>1.3</b>	<b>0.4</b>	<b>97.13%</b>

RTL: # related test Logical Lines of Code changed; RTC: # of related test cases updated

reduction of invariants for inspection, respectively. However, the correlations between the size of test change and that of invariant differentials are still low (0.118 and 0.167). Additionally, we attempted to correlate the number of updated test cases with the size of invariant differentials, but the results, -0.049 and -0.001, still show that the size of invariant differentials is not directly related to the size of test changes.

As a summary, we make the following conclusions. First, there are not any evident correlations between the size of code diff and the size of invariant differentials, but it is not a surprising result. Developers update code for different purposes: adding new features, fixing bugs, refactoring code for readability, or more. Change of code does not necessarily lead to change of semantics. For example, a developer could break down a large method body into a set of smaller sub-procedures without changing its semantics. Therefore the size of code change is not strongly related to the semantic change, and hence the size of invariant differentials. Second, when there are invariant differentials, they usually are a great reduction of the full list of invariants. In our study, invariant differentials gave a minimum average reduction of 95.52%, which was under the condition without any impact isolation. This means a reviewer would only need to inspect every 20 invariants, a much smaller inspection load. The worst reduction (73.01% ) took place on commit #7 under the condition of test impact isolation for old source, where new tests were added for the previously under-tested branches of the `equals` method. It immediately improved the quality of invariants with more likely invariants inferred that are added to the full list. However, as we pointed out in Section 3.2.1, under the source

impact isolation condition shown in Figure 4, there were only 5 changed invariants – 4 removed and 1 added – still a notable 93% reduction.

### 5.3 Checking Test Sufficiency

Using GETTY, we applied CSI on 100 test-only commits randomly selected from the 6 projects. Depending on the project size and history length, the number of commits chosen from each project varies (Appendix I). We then inspected the invariant differences of each commit to identify inadequacies in the testing of the methods under test in the commit. The question is whether or not the invariant differences were able to expose insufficiency, and why or why not.

As a simple metric of insufficiency, we consider tests insufficient for a method if they do not cover all combinations of *types* that result in different behaviors of the method. More specifically, this means different classes for possible input Java objects or different value combinations that could lead to different outputs. For example, when testing `equals` for two integers (See Section 5.1), we want to see test cases for all combinations of regular and big integers, of equal and unequal value; and, if testing for a method doing modulo operation, we want to see test cases for different multiples of the divider with different remainders. Although this may overlook corner cases (i.e., it lowers our success rate), it is a straightforward, repeatable metric. We determined ground truth, then, by exhaustively inspecting the tests and source after making the first determination with GETTY.

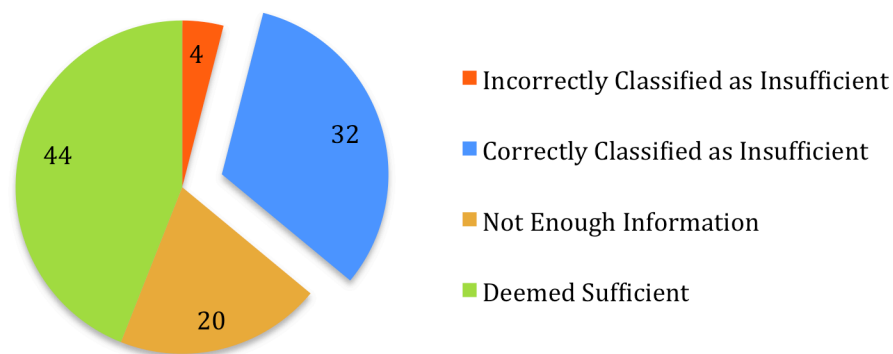


Figure 22 Test Sufficiency Inspection Result

We attach the detailed inspection result at Appendix I, and summarize the statistic result in Figure 22. Cumulatively, of the 100 testing commits, 32 were identified as being insufficient. For example, the commit #10 we discussed (Figure 2) was one of the testing commits we inspected. At the exit point of the integer comparison branch in `JsonPrimitive>equals` method, we observed the invariant “`result = false → this.value is Integer`”. This indicates that the testing commit, though captured the bug, was still inadequate because it only considered the unequal cases for when `this.value` is `integer`, but it is also important to test the unequal case when comparing two numbers that are both `BigInteger`. Another example from GSON project was the commit `7d7680f`, where developers added tests to ensure any array of nulls are deserialized correctly [47]. The array deserializer branch implied that “`size(arr) <= 1`”, indicating that developers tested the input JSON string like “`[]`” and “`[null]`”. However, we regarded this as insufficient because an array of more than one `null` will

introduce separator(s) into the string, like “[null,null,null]”, which could possibly alter the behaviors of the deserializer.

CSI led us to incorrectly classify 4 commits as insufficiently tested, due to invariants derived from coincidental correlations in the data. For example, the commit f63e8e9 of the Codec project added a number of tests for Base-64 decoding [49]. The decode method returns an array of bytes as result. However, at the exit-point an invariant, “size(return) % 2 = 0”, indicates that the size of the return array always contains even number of bytes. This could be a bug, but for this commit this result is completely accidental. As a known issue, Daikon could over-generalize invariants since its inference is based on machine learning techniques, which could not guarantee 100% accuracy of the results. Incorrect could mislead reviewers to make wrong judgment, but reviewer experience might aid in better spotting these.

We still consider 44 commits sufficiently tested. For example, the commit 62e69cf of Collections project specifically tested the implemented map for setting null values for certain keys. The put method of the map takes the first argument key as the key and the second argument value as the value. The invariant differentials indicated that a previous entry-point invariant, “value ≠ null”, was removed after commit. Since this shows the effect of added tests is exactly for setting null values, we consider this commit sufficiently tested.

20% of the commits did not produce sufficient invariant differentials for the task, mostly because the invariants did not provide useful information to reviewers. This is especially the case for testing multi-threaded programs. For example, commit ae90d56 of



Configuration project introduced a number of tests to prove a race condition was fixed; however the invariants did not change, and thus the invariant differential did not provide any useful information to reviewers. Furthermore, it was not unusual for Daikon to infer the same invariants despite more tests are added, and in those cases Getty delivers little semantic information to reviewers.

Given the simple nature of our insufficiency metric, these results are quite positive. It is notable that all of the chosen testing commits were considered 100% covered by the branch coverage report obtained by Maven EMMA plugin [39]. Our results confirm recent results that popular test coverage tools are not always good indicator of test suite effectiveness [50].

As an aside, during this study we found two simple invariants that often gave away test insufficiency, both featured in the scenario in Section 5.1. One relates to the failure to test for null as in input value (e.g., `other ≠ null`). The other is a Boolean return value always being `true` or `false`. In addition, exceptional exit-point invariants were helpful in confirming behavior after a failure intentionally induced by a test case.

#### **5.4 Checking Semantic Change Consistency**

We randomly selected one confirmed bug from each project (including a new one from GSON), found the commit where the bug was introduced, and applied CSI to check whether the bugs could have been found at the time of their introduction. As ground truth, we checked out the commit confirming the existence of the bug, executed the failed test(s), traced the buggy method's behaviors using Eclipse JDT Remote Debugger [51], and studied the root cause of the bug. Then we used *git-blame* [52] to trace back to the

Table 11 Results of Inspecting Buggy Commits

Projects	Introduction Commit	Discovery Commit	Time Between Commits	Lends Insights?
GSON	b634804	60ef777	1 day	Yes
CLI	9b2b803	6c740e7	72 months	No
Codec	2405423	b9cab09	159 days	No
Crypto	9faf04e	ee2136e	6 days	Yes
Collections	83226e1	9dbf838	19 months	Yes
Configuration	c75a72c	821ccfa	60 months	Yes

Y/N: Whether CSI lent insights to discover the bug at the time of its introduction

editing history and find the commit that made the buggy edit. Finally, we applied CSI to inspect the semantic changes of the commit that introduced the bug. The results are summarized in Table 11.

The commit from the CLI project fixed the bug documented in issue CLI-252 [53], where the command line parser threw an exception when parsing an option that is the prefix of another. The bug was introduced when developers added partial matching to the parser. GETTY failed to identify this bug because Daikon did not infer useful invariants for the options. The bug in the Codec project regards the `colognePhonetic` method in the `ColognePhonetic` class. It takes a string and returns an encoded string, using the Kölner Phonetik algorithm [54]. The bug concerns an encoding with sequentially repeated digits. Both bugs went unrecognized, at the very least, because Daikon's current `String` invariant templates do not consider string contents beyond equivalence.

CSI granted insights for discovering the other four bugs listed for GSON, Crypto, Collections, and Configuration, between 1 day and 60 months before the bug was explicitly discovered and fixed. We discovered two typical patterns for deciding whether a commit is buggy: incorrect invariants, and missing expected invariants.

```

177 private JsonElement findAndInvokeCustomerSerializer(ObjectTypePair objTypePair) {
178     Pair<JsonSerializer, ObjectTypePair> pair = objTypePair.getMatchingSerializer(serializers);
179     If (pair == null) {
180         return null;
181     }
182     JsonSerializer serializer = pair.getFirst();
183     objTypePair = pair.getSecond();
184     start(objTypePair);
185     try {
186         return serializer.serialize(objTypePair.getObject(), objTypePair.getType(), context);
187     } finally {
188         end(objTypePair);
189     }
190 }

```

```

Exit-Points:
point-186:
(return = null) → (objTypePair = orig(objTypePair))

```

Figure 23 GSON bug at commit b634804, indicated by the invariant diff

### 5.4.1 Incorrect Invariants

For the first pattern, incorrect invariants, we take the example from GSON project at commit b634804. The method of interest is `findAndInvokeCustomerSerializer`, shown in Figure 23. At line 186, it is expected that the return statement will always return a `JsonElement` object as the serialized result. From the invariant differential, however, we found that at this exit point it is possible for the method to return `null`. Java's `null` is not a `JsonElement` object, and the fact that this exit point can return `null` indicates that this is likely a bug, which could cause more future bugs like null-pointer exceptions if the result were referenced and used by other parts of the program. This was actually the corner case developers had overlooked: when the serialized result is “null”, it should return `JsonNull` object (`JsonNull` is a subclass of `JsonElement`), instead of the Java's

```

1121 TrieEntry<K, V> subtree(final K prefix, final int offsetInBits, final int lengthInBits) {
    // implementation of the algorithm ...

```

```

Entry-Point:
    offsetInBits < lengthInBits

```

Figure 24 Collections bug at commit 83226e1, indicated by the invariant diff

`null`. The developers did not find the bug until a later commit, 60ef777, where they specifically handled the case when return value is `null`, and returns a `JsonNull` object instead.

Another incorrect invariant we found is for the bug from the Collections project at commit 83226e1. In this commit developers introduced `AbstractPatriciaTrie`, an abstract class that encapsulates Patricia Algorithm for tries [55]. Starting from line 1121 is the method of interest, `subtree`, which takes as input a `prefix` and additional parameters, and finds the subtree that contains the `prefix` as prefix. Figure 24 shows the full signature for `subtree`. As the two additional parameters, `offsetInBits` tells the method to start checking from this given offset in bits, and `lengthInBits` tells the method where to stop. For this initial implementation, the entry point invariant says that `offsetInBits` is always less than `lengthInBits`, indicating that the method will always start checking somewhere before it ends at a later position. However, this should be considered a bug because in reality it is quite possible to start and end at the same index. The developers did not find this bug until over a year later when they find the Patricia Tries they implemented could sometimes mismatch [56]. They fixed at commit 9dbf838, 19 months after the bug was introduced.

```

115 public static CryptoCipher getInstance(String transformation,
116                                     Properties props) throws GeneralSecurityException {
... // implementation ...
120   StringBuilder errorMessage = new StringBuilder("CryptoCipher ");
... // implementation ...
137   if (errorMessage.length() == 0) {
138       throw new IllegalArgumentException("No classname(s) provided");
139   } finally {
... // implementation ...

```

Figure 25 Crypto bug at commit 9faf04e; no invariant diff for exit-point 138

In the above two examples, invariant differentials are superior to test coverage metrics for catching bugs. Though the two interested methods were 100% tested in terms of branch coverage, the problems in the two buggy commits lie in either the domain (Collections example) or the range (GSON example) of the methods. Invariant differentials deliver more detailed information to help developers review the semantic impact of a commit.

## 5.4.2 Missing Expected Invariants

Sometimes, we inspected invariant differentials but failed to find the expected invariants, and we regarded that as being a sign of bug. We take the example from Crypto project at commit 9faf04e. Developers refactored the implementation of `getInstance` method for a clearer logical code structure. We inspected the invariant differentials, but did not find any invariants inferred for the exceptional exit point at line 138 (Figure 25). This turned out to be a bug. The branch covering line 138 would be executed only if the length of `errorMessage` is 0; however, earlier at line 120, `errorMessage` is initialized with a length greater than 0, and none of the code between line 120 and 137 would

decrease the length of `errorMessage`. Therefore line 138 resides in a dead branch. Developers did not find this problem until the commit `ee2136e`.

The dead branch in the Crypto example could in principle be found by reading test coverage reports. Here we have another example from Configuration project at commit `c75a72c`, whose bug cannot be found by test coverage reports. In this commit, developers implemented the `to` method to convert a specified value (`value`) to a target class (`cls`) with additional parameters (`params`) to assist this conversion (Figure 26). The `to` method contains one dedicated branch for each possible target class, and developers intended to cover all primitive types, including `Integer`, `Boolean`, etc.. We inspected invariants inferred for each type's conversion branch, but we did not see any invariants inferred for any handling `String` type, which is one of Java's primitive types. We concluded that the developers forgot to handle the `String` type and the `to` method contains a bug. This bug was not fixed until 5 years later when developers found the `to` method cannot handle "a trivial conversion" [57, p. 48]. In this case, developers overlooked a specific branch. Test coverage report measures how the existing branches are covered, but it cannot lend any insights on overlooked branches. We were able to identify the overlooked branch because invariant differentials delivered the semantic information for us to better understand the intent of the program.

## 5.5 Threats to Validity

The author of this dissertation, who has over 10 years of programming experience, studied the cases discussed in this chapter. However, the validity of the data is still threatened by the researcher's subjective judgments. The researcher is familiar

```

87 static Object to(Class cls, Object value, Object[] params) throws ConversionException {
88     if (Boolean.class.equals(cls) || Boolean.TYPE.equals(cls))
89     {
90         return PropertyConverter.toBoolean(value);
91     }
92     else if (Number.class.isAssignableFrom(cls) || cls.isPrimitive())
93     {
94         ... // more conversions
95
96         ... // more else-if branches
97
98     }
99     else if (Color.class.equals(cls))
100    {
101        return PropertyConverter.toColor(value);
102    }
103    else
104    {
105        throw new ConversionException("The value '" + value + "' (" + value.getClass() + ")
106            can't be converted to a '" + cls.getName() + "' object");
107    }
108 }

```

```

Exit-Points:
point-102:
    return's class is java.lang.Boolean, ... // more invariants
point-108:
    return's class is java.lang.Integer, ... // more invariants
... // more exit-points
point-146:
    return's class is java.awt.Color, ... // more invariants

```

Figure 26 Configuration bug at commit c75a72c, lack of invariants for String conversion

with invariants, which may be an unusual situation for reviewers in industry. The results in this chapter could therefore be overly optimistic. To obtain more objective opinions from realistic reviewers, we conducted a user study. We will discuss our user study in the next chapter.

Another threat is the diversity of the selected projects for our study. Although the projects were randomly selected for research purposes, they may not be very representative. One of the six projects is from Google, while the other five are all from

the Apache Foundation. Limited community cultures could lead to limited diversity, which might jeopardize the generality of our conclusions in this chapter.

### **Acknowledgments**

This chapter, in part, is currently being prepared for submission for publication of the material, Yan, Yan; Menarini, Massimiliano; Griswold, William G. “Mining Code Repositories for Semantics-Assisted Code Review” The dissertation author was the primary investigator and author of this material.



# Chapter 6

## User Study

This chapter describes how six pairs of Java programmers performed review tasks using GETTY for CSI. We sought to answer the following research questions:

- How do invariant differentials affect the review process?
- What are reviewers' attitudes and insights about Continuous Semantic Inspection and GETTY?

From our analysis of the programmer's work, we derived the following observations:

- Invariant differentials altered the reviewers' process. Generally, reviewers using GETTY used invariant differentials to generate a hypothesis about a patch. Then they read the code to verify the hypothesis they proposed, and use the hypothesis to further understand requirements. Because reviewers asked more focused questions during their review, the end result was the production of generally more focused review comments left for the developer.
- Most of our reviewers had positive attitudes about CSI. They acknowledged that inspecting semantic changes is necessary and helpful to code review tasks, and that they would like to incorporate similar

procedures into their daily code review process. Additionally, they observed opportunities to improve GETTY.

This chapter is organized as follows. We first describe our study design (Section 6.1). Then, we analyze participants' behaviors and study how the process of reviewers using GETTY was different from the reviewers not using it (Section 6.2). We discuss participants' feedback (Section 6.3) on GETTY and CSI for their overall experiences and suggestions. After discussing threats to validity (Section 6.5) of the study, we conclude this chapter.

## **6.1 Study Description**

To gain insights on patterns of using GETTY, we conduct a lab study to document the reviewers' experience using GETTY and answer the research questions of this chapter. Our expectation was that CSI reviewers in our study could identify problems in inadequately tested commits from the changed invariants and code, and that they could interact with the developers in a consistent manner until the issues are fully resolved.

### **6.1.1 Roles and Participants**

There were three roles in this study: developer, reviewer, and internet helper. All roles work together for resolving issues. We discuss the roles and participants we enrolled in this section, and will further describe issue workflow in Section 6.1.2.

The developer is the programmer who implements the functionality or fixes the bug, and update the issue. It is typical that the developer creates an issue, constantly updates the issue, and closes the issue when it is resolved, so for simplicity we use one role, developer, to represent not only the programmer, but also the issue creator, reporter

and owner. A reviewer is a separate engineer who reviews the code changes and provides comments to the developer. It is also the role whose behaviors we study in this chapter. An internet helper plays the role of search engines (e.g., Google [58]), online Q&A communities (e.g., StackOverflow [59]), and more for GETTY, since the tool, being a prototype, has no online presence. In the study, our participants are the reviewers, while we play all the other roles.

We enrolled 18 anonymous participants with 1 to 16 years of programming experience in academia or industry. All of our participants volunteered to fill out a questionnaire (Appendix II) regarding their programming background, so we can better interpret the results. We rate their experiences by three levels: novice, experienced, and advanced. Novice programmers are beginners who are new to programming. For example, two of our participants are undergraduate students with less than 2 years of programming experiences (all for their coursework), so we rate them as novice programmers. All other 16 participants are either graduate students or professional developers from industry, and they all have at least 4 years of programming experience. We consider participants with less than 10 years of experience as experienced, and those with 10+ years of experience as advanced. Based on this rating system, we have 8 experienced and 8 advanced participants.

We further divided the 18 participants into 6 study groups and 3 control groups, each of which comprises 2 participants of similar experience level (Table 12). The two participants in the same group performed the review tasks together. We set the pair-programming style to avoid the negative impact of similar approaches like the Think-

Table 12 Participant Labels

	E1	E2	E3	E4	E5	E6	C1	C2	C3
Left	LA	LA	LN	LE	LE	LE	LA	LA	LE
Right	RA	RA	RN	RE	RE	RE	RA	RA	RE

Label: Group Number (1-9) + Seat Position (L, R) + Experience Level (A, E, N)

Aloud Protocol [60], where researchers may unintentionally influence what participants say and do [61].

For presentation purposes, we label each participant by his or her group number (experimental groups are from E1 to E6, and control groups are from C1 to C3), seat position (L for left, or R for right), and experience level (N for novice programmer; E for experienced programmer, or A for advanced programmer). For example, the reviewer of experimental group #5 sitting to the left is referred to as E5-LE, and the reviewer of control group #1 sitting to the right is referred to as C1-RA. Table 12 lists all of the participants for future reference.

### 6.1.2 Issue Lifecycle and Reviewer Workflow

Each issue in the study is in one of the three states: open, pending review, or closed. The state chart is shown in Figure 27. An issue is initially in an open state. When the first implementation is available, it becomes pending review. Reviewers will review the issue. Unless all reviewers are satisfied with the implementation, they will leave comments and move the issue back to open status. Developers have to continue to work on the issue based on reviewers' comments. When a successive fix is available, again the

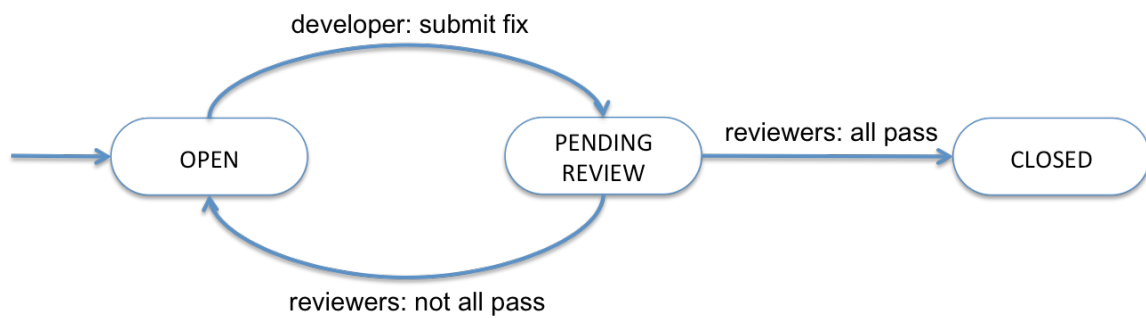


Figure 27 User Study Issue Lifecycle

issue will be pending review and it will be reviewers' turn to review the changes. The issue will be closed only if all reviewers agree to pass it.

All of the issues reviewed in this study were not good enough to pass the initial review. We set up a two-phase review process for the bounded duration of our study. In the first phase, the reviewers review the original issue with the original patch. After reviewers finish their review, we assume that there were other reviewers, independent from the current ones, that also finished their code review and the issue owner will synthesize all review comments and propose a new patch. In the second phase, the reviewers review the new patch. If they are satisfied we close the issue; otherwise, we comment that the new suggestion for improvement will be moved to another issue and then we still close the issue

For each review phase, to resolve an issue reviewers not only need to read the issue report and code base, but also may interact with related persons (developer, project manager, etc.) and other sources (search engine, web forums, etc.). For example, when reviewers are confused by some programming tricks used by developers, they can ask the developer for explanations; when they lack certain background knowledge, like integral

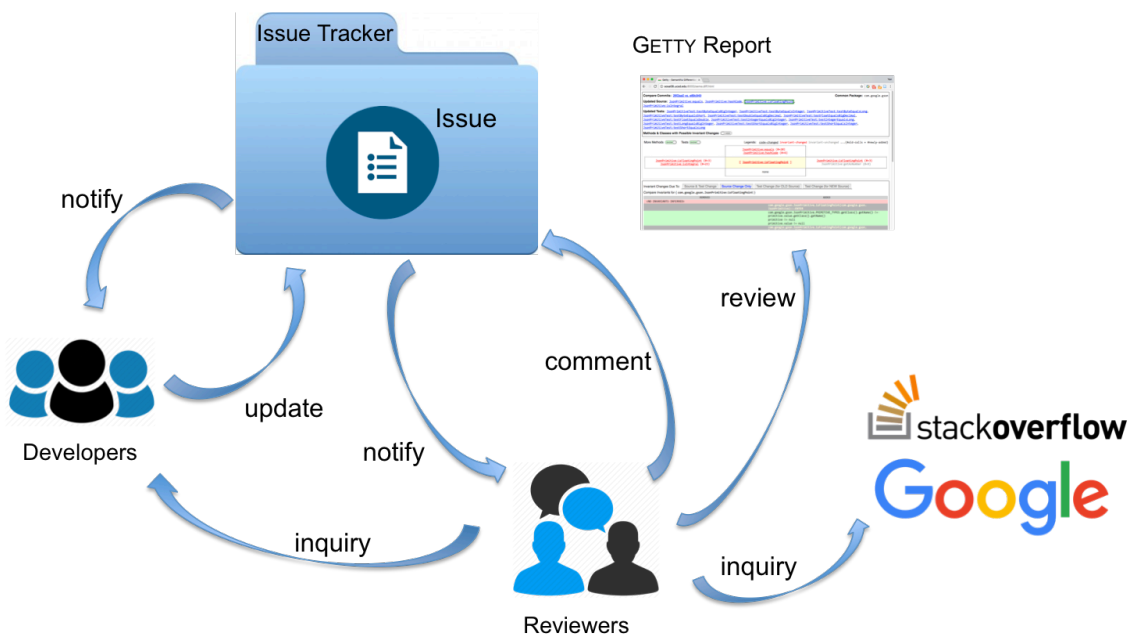


Figure 28 Reviewer Interactions

precision definitions for different integer types in Java, they can search Google or post questions to [stackoverflow.com](http://stackoverflow.com) for answers.

Figure 28 presents the workflow from reviewers' standpoint. After the reviewers are notified to review an issue, they will read the issue report and review the report generated by GETTY. When any part of the program confuses the reviewers, they can ask the developer for an explanation. For any other related questions, reviewers may ask [stackoverflow.com](http://stackoverflow.com) or Google; but in our study they cannot wait for the replies. Therefore, the experimental investigator plays the role of the web forums or search engine. Instead of issuing inquires online, reviewers can directly ask the experimental investigator questions and they can expect answers to be given promptly and correctly. The reviewers will gather any problems they find during review, and leave their comments in the issue tracking system. Developers will be notified of the new comments

and update the code and issue report accordingly. After each update a new GETTY report will be generated for the reviewers. The reviewers will repeat their reviewing process until they are all satisfied with the update.

### 6.1.3 Study Tasks

We chose three real issues from the GSON project. All of the chosen issues had passed their original code review process, but, in fact, they either were insufficiently tested or suffered from undiscovered bugs. The three issues, their symptoms and our expected fixes are highlighted below:

1. Issue-#1 [62]: The patch for this issue targets is to fix two methods, `intValue` and `longValue`, of `LazilyParsedNumber` class. The class contains a `String` attribute, `value`, which can be interpreted as a number. The two methods interpret `value` and output the corresponding number value as `int` and `long`, respectively. The issue was to fix the conversion failure when converting decimal number strings. It is expected that decimal numbers be converted to integers ignoring all digits after decimal points. Before the issue developers always used `BigInteger` to convert the `value`, but it would fail the conversion when `value` is a decimal number because the conversion method of `BigInteger` will raise a `NumberFormatException`. For this issue developers replaced `BigInteger` with `BigDecimal` so the two methods can handle floating-point numbers without failures. The developers added only one simple test case, converting “1.0” to 1, for this fix, but did not consider other trickier conversion situations. We expect reviewers to react on this problem and give developers

some suggestions on testing; specifically, we believe the developers should at least consider the cases when `value` is a number that overflows the range of `int` or `long`, and when `value` is a decimal number whose digits after decimal point are not all `0`'s.

2. Issue-#2 [63]: The issue is to implement two new methods, `equals` and `hashCode`, of the same `LazilyParsedNumber` class. `equals` compares another `LazilyParsedNumber` and returns `true` if their `value` attributes are equal numbers. `hashCode` computes a hash value of `value` and returns the result as the Hash value of the `LazilyParsedNumber` class. In this issue the developers implemented both methods and added tests for them. For testing the `equals` method, developers added a test case that a `LazilyParsedNumber` object whose `value` is "1" is equal to another `LazilyParsedNumber` object whose `value` is also "1"; for testing the `hashCode` method they added the test case that the above two objects have identical Hash codes. Ideally, reviewers should find both tests were inadequate. For example, for `equals` developers should have considered the case when the two `LazilyParsedNumber` objects contain unequal `value` values. In addition, there is a subtle bug in the patch. According to Java's specification, developers should test that the `equals` and the `hashCode` methods behave in a consistent way; i.e., two equal `LazilyParsedNumber` objects should have identical Hash values, but that was not accounted for in the patch.



3. Issue-#3 [64]: This issue contains two parts. In the first part, developers updated the application logic of the `JsonNumber` method in `JsonParser` class. `JsonNumber` is a helper method for the parser. Its purpose is to return a `JsonPrimitive` object (more specifically, a number primitive) if the object being parsed can be interpreted that way. During interpretation, developers used `BigInteger` to parse all integers to avoid precision loss, and used `BigDecimal` to keep all zero's at the end of decimal point. This involves some semantic changes at the library methods, `getAsBigInteger` and `getAsInt`, in the `JsonPrimitive` class. In order to study whether reviewers can find unimplemented requirements, we added the second part that specifies the specification of the two methods, `getAsInt` and `getAsBigInteger`. More specifically, `getAsInt` should only accept strings that can be interpreted as integer values; for other strings, it should raise exceptions. We expect reviewers to problems, either testing issues or bugs, in the patch.

In our study, each issue is reported in the issue tracking system, BitBucket [65]. The issues are initially pending review. We use the implementation (or fix) from the original issue as the developer's first response. The reviewers will then decide whether to pass or fail the code review. We will close the issue if all reviewers are satisfied, or if they already find all problems we expect them to find. If the reviewers are not satisfied, they will leave comments in the issue tracking system with specific reasons (and suggestions, optionally), and set the issue state to be open. We will discuss about the comments, fix them, update the issue promptly and set its state to be pending review

again. Reviewers continue to perform the review task on the updated commit. In our study, the above procedure repeated at most twice per issue.

We simulated the real code review environment. Each group of participants was arranged in the same quiet lab room for their review tasks. We prepared two computers, both Apple's 27-inch iMac (2016 Model). Reviewers used one of the machines for their code review tasks, while the experimental investigator used the other one to reply to reviewers comments, reset issue states and all other related tasks. We video-taped all reviewers' behaviors during their code review. After they finished, we interviewed all participants for their experiences using GETTY and requested suggestions for improvement.

## **6.2 Code Review Process**

We will answer the first research question in this section. How do invariant differentials affect the review process?

As a baseline, the control groups used the Github code-diff page for their review. They did not have access to any extra semantic information. The experimental groups used GETTY, so they could not only view the same code-diff page displayed for control groups, but also the added semantic information like invariant differentials. To get a general understanding of how code review is performed without added semantic information, we first summarize the control groups' behaviors and results. We then analyze the behaviors of the groups using GETTY. We compare the experimental groups' review process and results with the control groups and study how the added semantic information altered their review process.

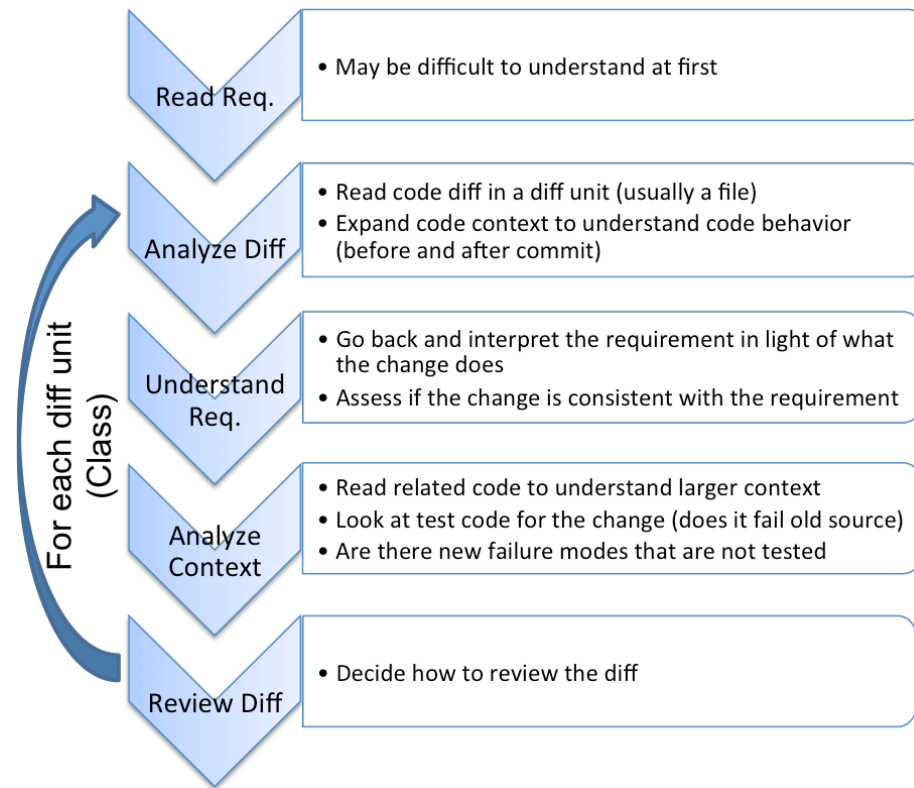


Figure 29 Control Group Review Process

### 6.2.1 Control Group Review Process

Figure 29 shows the typical review process of the control groups. Reviewers start by reading the requirements. Most requirements are vaguely written, which is expected. After roughly understanding the requirement, they open one file in code-diff page and try to understand the code change. The reviewers may need to expand the code context to further understand the code. Based on their understanding of the code change, they go back to the issue description and check if the code change is consistent with the requirement, as they understand it. Reviewers further examine related code, including tests, to check if the changed code is properly tested until they are fully satisfied with the changed file. The reviewers repeat the above process for each file listed in the code-diff

```

@@ -39,7 +39,7 @@ public int intValue() {
39      39      try {
40      40      return (int) Long.parseLong(value);
41      41      } catch (NumberFormatException nfe) {
42      42      - return new BigInteger(value).intValue();
42      42      + return new BigDecimal(value).intValue();
43      43      }
44      44      }
45      45      }

```

Figure 30 Code change in the intValue method

page. After they finish reviewing all files, they discuss about the issue and write down review comments.

Take the issue #1 for example. As the first step, the reviewers read the following issue description:

*“Use BigDecimal to parse number string when requesting it as Integer – LasilyParsedNumber has the value of a string that can be interpreted as a number. Use BigDecimal to parse the number string to avoid precision loss in general. However, when requesting as an integer, it ignores all digits after decimal point if any, and ignores all bits that overflow the range of requested integer type.”*

The requirement as indicated in the description was vague to reviewers at first. For example, C1-LA read and complained about the requirement, “... ignores all bits that overflow the range of requested integer type ... Oh, god that is not a good way of saying that!” C1-RA interpreted the requirement of precision as “truncate to reasonable integer length”.

```

16 16 package com.google.gson.internal;
17 17
18 18 import java.io.ObjectStreamException;
19 19 import java.math.BigDecimal;
20 20 import java.math.BigInteger;
21 21
22 22 /**
23 23  * This class holds a number value that is lazily converted to a specific number type
24 24  *
25 25  * @author Inderjeet Singh
26 26  */
27 27 public final class LazilyParsedNumber extends Number {
28 28     private final String value;
29 29
30 30     public LazilyParsedNumber(String value) {
31 31         this.value = value;
32 32     }
33 33
34 34     @Override
35 35     public int intValue() {
36 36         try {
37 37             return Integer.parseInt(value);
38 38         } catch (NumberFormatException e) {
39 39             try {
40 40                 return (int) Long.parseLong(value);
41 41             } catch (NumberFormatException nfe) {
42 42 -         return new BigInteger(value).intValue();
43 43 +         return new BigDecimal(value).intValue();
44 44     }
45 45 }

```

Figure 31 the intValue method with context

To understand the requirement, the reviewers started to review the actual code change. They noticed that more than one file was changed, so they chose to review the first file and jumped directly to the changed lines (Figure 30). Most often, the reviewers could not fully understand the changed code immediately. Like C1-RA mentioned, “why would we do a big decimal inside an int value? Wasn’t the whole point of the LazilyParsedNumber that we are using an integer type in the integer case?” So they needed to understand more code context by viewing the full method body in the context of its class (Figure 31). By reading the code, most reviewers felt the requirement clearer to them, and came up with specific questions. For example, conversion between C3-LE

and C3-RE was "... the value is a double. In that case, would converting it into a BigInteger cause any exception? If a value is a double and if you convert it into a BigInteger instead of a BigDecimal, would it cause any exceptions or would it just round it up?" This was an excellent question because it was exactly the problem this issue was to address.

With specific questions, the reviewers went back to the changed code and verified whether the code correctly addressed the issue, and whether the code is properly tested. For example, C1-LA found the tests were insufficient, and said the "test case does not exercise any of the interesting edge cases and frankly, you know, if someone actually submitted this for a project that I was working on, I wouldn't say a ton more than that. I'd be like come on man!" This was eventually left as one of the review comments left by this group.

The reviewers gradually understood the requirement, and kept on verifying their understanding of the changed code. For example, the issue report said to use `BigDecimal` to parse the number string to avoid precision loss "in general", but never said what it meant. This raised C1-LA's attention, and he asked whether it was necessary to "maintain decimal if it has it". In addition to the `intValue` method and the `longValue` method, there were other similar methods, `floatValue`, `doubleValue`, etc., in the same class; but then C1-RA found developers "didn't touch that code". They were convinced that by "in general", the developer meant the precision problem for integral numbers only, and believed that the precision problem for floating-point numbers was not part of the requirement.

The reviewers examined other changed files in a similar manner. As the last step, they discussed and wrote down their final review comments. We cite some of the review comments for issue #1 below:

- “The test case does not cover either the new functionality or most potential edge cases. Please add additional test cases covering truncating, large values, and the intValue() code paths” (C1-LA)
- “Functionality looks good. Consider removing import of BigInteger library.” (C2-LA)
- “Instead of json = 1.0, ... consider json = 1, ...” (C3-LE)

It is worth noting that the control groups ran into various additional problems with issue #1 while they were reviewing the code change. For example, group C2 identified that the BigInteger library import could have been removed after the developer made the change. This kind of style issue can be automatically addressed and reported by existing style check tools instead of wasting valuable reviewers’ time. In the third review comment, the reviewers suggested writing one test case, but that, in fact, was an incorrect suggestion. Because the reviewers in control groups did not focus on the semantic impact of the code change, they developed random questions regarding code quality and that led to random review comments. We observed the similar review process and found the qualities of review comments for issue #2 and #3 also varied, and most of them were not useful to the issue. This corroborates with the conclusion drawn from Microsoft’s 2015 survey on the current code review process that less than half of review comments are deemed useful by the author of a change [66].

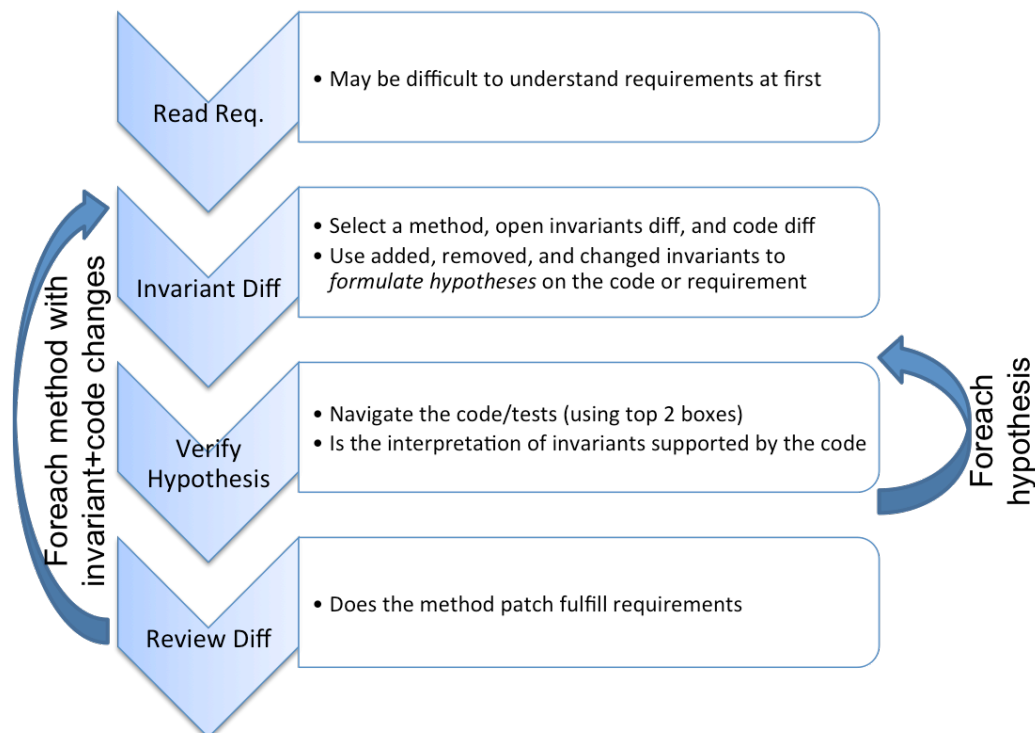


Figure 32 Experimental Group Review Process

## 6.2.2 Experimental Group Review Process

Next, we analyze experimental groups' behaviors and study how GETTY altered their review process and what the consequences were from the change.

Figure 32 shows the general process of the experimental groups. They begin by reading the requirements, which is the same with control groups. The requirements are vague, so they review the changed code to aid their understanding. However, in this process, they inspected the patch on a per-method basis, instead of files. While reviewing a method, they examined its invariant differentials and formulated hypothesis about the code or requirement. To verify each hypothesis, reviewers read related code and check if their interpretation of the invariants is supported by the code. Reviewers repeated the



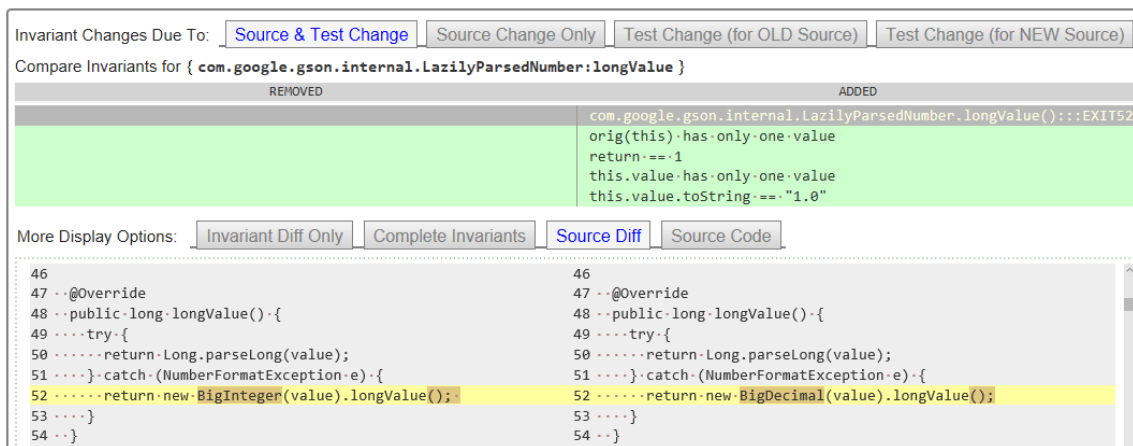


Figure 33 Invariant and code diff for the longValue method in GETTY

above until all changed methods were inspected, then they leave review comments in the issue tracking system.

To ease the comparison, we again take issue #1 as an example. After reading the vague issue description, reviewers opened the GETTY tool and started to review each changed method to better understand the requirements. Generally, reviewers tended to start their focus on a method with both syntactic and semantic changes. For example, E2-LA suggested that they need to “agree on what has changed”, both the code and the invariants; then group E2 selected the longValue method to begin with because it was one of the changed methods whose invariants were also updated.

Because of the way GETTY’s UI is designed, it was easier for reviewers to view invariant differentials before the code change (Figure 33), and most reviewers did so. They developed hypotheses from the invariants. For example, after reading the added invariants for the longValue method (Figure 33), E2-RA said “what this means is that the exception case is tested with only one test ... and that also seems not necessarily great, right? You want at least a few tests for all branches. A couple of tests, at least, for

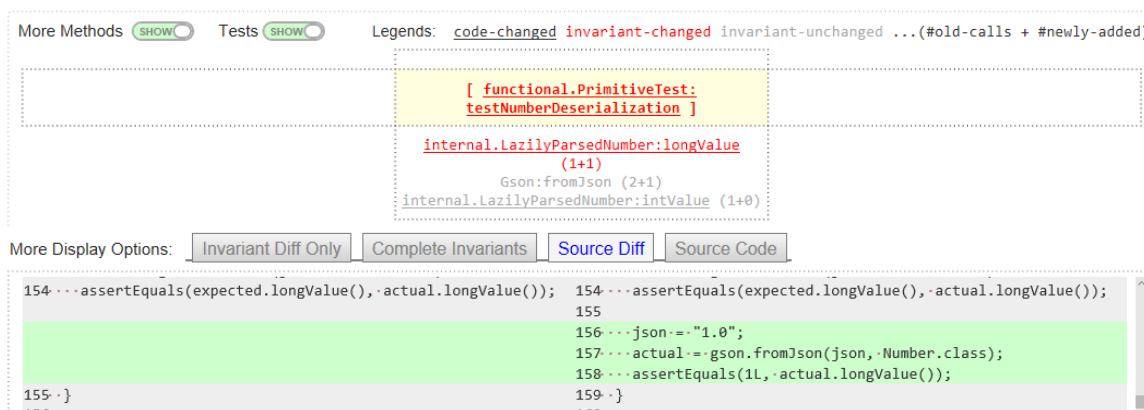


Figure 34 The test case for the longValue method

all branches.” In this hypothesis he raised his concern about the tests, and he surmised that the tests might be inadequate. The same hypothesis was developed by the other experimental groups as well.

The reviewers began to review the code change to verify their hypothesis. Using GETTY, reviewers found their way to the related methods or test cases. For example, for the same hypothesis (test inadequacy) developed by group E1, the reviewers took advantage of the invocation flow information in GETTY (Figure 34), and found one modified caller of longValue, which was the new test case introduced in the commit. After reading the updated tests, E1-L1 said to his partner “the key problem is exactly as you said. This type of test is not very, you know, capture the problem right.” As another example for the same hypothesis, group E6 used the impact isolation feature of GETTY (Figure 35) to verify it. Instead of locating the test case for the longValue method, they clicked the “Test Change (for Old Source)” tab and read the added invariants for the old source when running new tests. From the added invariants at exceptional program points, E6-LE realized that new tests failed the old source, as expected, and there was only one

More Methods  Tests  Legends: code-changed invariant-changed invariant-unchanged ...(#old-calls + #newly-added)

JsonPrimitive:getAsLong (16+0)  
functionalPrimitiveTest:  
testNumberDeserialization (1+1)

Gson:fromJson (1+1)  
JsonPrimitive:getAsNumber (16+0)     [ internal.LazilyParsedNumber:longValue ]     Gson:fromJson (0+1)

none

---

Invariant Changes Due To:

Compare Invariants for { `com.google.gson.internal.LazilyParsedNumber:longValue` }

REMOVED	ADDED
	<code>com.google.gson.internal.LazilyParsedNumber.longValue()::: THROWSCOMBINED</code>
	<code>exception-has-only-one-value</code> <code>exception.getClass().getName() == java.lang.NumberFormatException.class</code> <code>orig(this)-has-only-one-value</code> <code>this == orig(this)</code> <code>this.value == orig(this.value)</code> <code>this.value-has-only-one-value</code> <code>this.value.toString == "1.0"</code> <code>this.value.toString == orig(this.value.toString)</code>

Figure 35 Use test impact isolation to verify the lack of tests

input value (“1.0”) introduced for that purpose. They took notes for the confirmed hypothesis for later use.

Reviewers repeatedly verified each hypothesis they developed. The verified hypotheses helped them better understand the requirements. Reviewers still switched back to the issue description page and discussed if anything was missing. Similar to the control groups, the experimental groups were unclear on why they should use `BigDecimal` to parse the number string to avoid precision loss “in general”. But when they switched back to GETTY they found `intValue` and `longValue` were the only two methods changed in the Patch Summary Zone, so this requirement became clearer to them that the precision concern was for integral numbers only.

Because the reviewers generated their hypotheses based on invariant differentials, their attention was focused on the semantic aspect of the code change. As a consequence,

the experimental groups left more consistent review comments. We cite some of them for issue #1 as follows:

- “I think you can add more tests that would show how the code behaves with a larger variety of decimal inputs, ...”
- “Would you please add test cases for `longValue` and `intValue` called with ‘overflow’ values? ...”
- “Please add more tests using more than one number as input ...”

In the comments above, the reviewers pointed out that the tests were inadequate, and they suggested ways to improve the quality of tests, including testing corner cases like decimal inputs and overflowing values. The review comments uniformly focused on the semantic aspect of the code, and they clearly suggested to developers on how to improve their tests.

The process was similar for issues #2 and #3. For example, in issue #2, group E2 found at the exit-point of the `equals` method, an invariant indicates that its return value was always `true`. They developed the hypothesis that only equal values were tested for the method. There were two overloaded `equals` methods and one was the helper method for the other. They read the added test case and verified their hypothesis. Given that the true return was for equal values, they confirmed that the requirement of the `equals` method was to perform comparisons based on values, not references. At the end, E2 left review comments for more test cases for unequal number comparisons, just like the other experimental group did.

In summary, the new process is hypothesis-driven in that: (1) Reviewers use invariants to generate hypotheses; (2) Reviewers use code and other information provided by GETTY to verify their hypotheses; and,(3) Reviewers use the verified hypotheses to better understand the requirements. Compared to the review comments left by the control groups, the experimental groups captured more of the semantic aspects of the patch and their review comments are more consistent and informative. This result shows that the extra semantic information provided by GETTY can positively alter reviewers' processes.

### **6.3 Reviewer Feedback**

In this section, we will answer the second research question. **What are reviewers' attitudes and insights about Continuous Semantic Inspection and GETTY?**

After the study we interviewed all participants about their experiences. Compared to the control groups, all six groups using GETTY were excited when they learnt they discovered the problems that were missed by the original project reviewers. In addition, our participants commented on the helpfulness of invariant differentials and shared with us several points for improving the tool design.

All Groups except for E5 explicitly expressed favorable opinions of GETTY and would like to integrate CSI into their code review process. Group #1 liked the tool because it “finds the errors using invariants not expected to see”, and believed that “invariant change is a plus and saves a lot of time in code review”. Group E2 said the tool is “overall pretty helpful” and they “enjoyed using the tool”. Group E3 believed reviewing invariant differentials are “necessary since it provided a way to view your code

and remind you if there is something lost”, and it is especially true “for others who may not be familiar with your code”. Group E3 also acknowledged that GETTY “helped find the lack of tests”. Group E4 said the tool was “very handy for reasoning about exceptions” and sometimes, they “did not have to look at all the code (to confirm their expectations)”, but just need to review the semantic changes. They had seen code review tools with syntax changes, but they felt excited to see the tool with semantic changes in the study, and was impressed that our experiments demonstrated such a tool being “approachable”. Group E6 was happy that the tool “provided unique information to help code review” and it is “necessary to view semantic changes in addition to textual changes” during code review. Group E5 liked the idea of inspecting semantic changes, but they pointed out that “invariant diff is not the actual semantic diff”. On the other hand, they still gave credit to GETTY with the comment that “invariant diff may give some insights”, because by using GETTY they were able to perform better than the original reviewers.

The participants suggested a number of ways to improve GETTY. The suggestions fall into two categories: logical presentation and UI improvement.

For the first category, reviewers would like the tool to logically understand the invariant differential information. Specifically, reviewers were interested in two kinds of information. The first is how/where a particular invariant was generated. Group E2, E4, and E6 wanted to know which test(s) generated a particular invariant. During the review, they frequently checked the callers of a method and were trying to find the tests that might be helpful to reason about the invariant differentials. In this scenario it could be

handy if, from each invariant, there were a way to tell which tests were related to it. In addition, the reviewers were interested in the logical difference between two versions of invariants. Currently, GETTY computes and displays only textual difference between two versions of invariants. Not only did this cause reviewers to expend more efforts (Group E3 and E5), but also it led to confusing differential information, especially when invariants of overloaded methods were displayed together (Group E2). Ideally, the tool should present reviewers logical differentials, and link each invariant differential to the particular set of tests that generated it.

Another set of suggestions is for UI improvement. Group E3 and E5 would like a friendlier UI so that the invariants are easier to understand for beginners. This could be solved by clearer invariant syntax as Group E6 suggested. Group E2 liked the integrated layout of GETTY, but wanted invariants shown near the code that generated them so that there were less clicks and moves by reviewers. Moreover, since GETTY did not present invariants separately for overloaded methods, the invariant differentials were clustered for all versions of a method, causing confusion. Group E1 and E6 suggested improving the UI for invocation flows, where they would like interactions between finer granularities of code snippets other than methods. Last, Group E4 wanted useful components of UI to be more accessible. For example, the last group of methods displayed in Patch Summary Zone of GETTY shows all methods and classes whose invariants have changed after commit. This group could be handy for reviewers to quickly jump to the methods of interest without extensive reading, so ideally it should be displayed by default.

As a summary, most reviewers shared their favorable opinions on GETTY and CSI. Meanwhile we received a number of suggestions for improvement. At the time of this dissertation's writing, we improved the UI of GETTY for clearer views. For example, the older version of GETTY was unable to differentiate invariants for overloaded methods; in the newer version, methods with the same name but different signature are treated as different semantic units and their invariants are inferred, differenced and displayed separately.

#### **6.4 Limitations**

From the study we also identified some limitations of GETTY. First is that GETTY failed to help reviewers find the subtle bug in issue #2, where the return values of the `hashCode` method of two `LasilyParsedNumber` objects should be identical as long as the `equals` method returns `true` when comparing the two objects (Section 6.1.3). Ideally, a class-level invariant correlating queries (methods without side-effects) could discover such bugs. Unfortunately, unlike Eiffel [67], Java does not distinguish queries and commands (methods with side-effects) at the language level, and therefore Daikon was not able to infer invariants of this kind. This also shows that CSI is highly dependent on the tools it uses and the languages supported, and it inherits their limitations as well.

Another limitation is the limited exploratory space in invocation flows. As discussed in Section 3.3.2 and 4.2, we compute the local-area call graph for each method, and infer invariants only for the methods within a fixed number of steps away from the target method. While this reduced the load of inference process, it also shrank the exploratory space during inspection. For example, when a reviewer wanted to see all the



test cases testing an interested method, she might have to locate them in the source code. Care needs to be taken to balance the performance impact and the usability of the tool.

## **6.5 Threats to Validity**

There are a number of factors in our study design that could threaten the validity of our results. In this section we talk about these threats to validity.

The first threat is that our participants were not familiar with our tool and the code base they were reviewing. As a contrast, in real life reviewers are usually familiar with their toolset and the projects they review. Likewise, some of our participants (Group #1, #3, #5 and #6) were not even familiar with their own partners. This added an extra level of difficulty to our participants when they were working together. Therefore, all these in overall could lead to more pessimistic results in a real code review environment.

The next threat is the representativeness of the selected issues. We aimed to choose three issues that are “problematic but moderately difficult”, which means that the issues were either under-tested, or have bugs that were not found by the original developers. The third issue was a combination of two issues for assessing unimplemented features. Albeit a common case in most software projects, it was not a realistic issue.

Third, the participants we selected may not be representative because they are mostly graduate students and 4 of them have a background in programming language research. They are very likely to be more proficient in using the information like invariant differentials, than the average reviewers. Therefore, our results could be overly optimistic, perhaps offsetting the pessimistic impact of the first threat.

Finally, our study was conducted in a simulated code review environment. That is to say, the setting may be close to real code review, but it was not real. The issue tracker is less complicated than the popular Phabricator [35]. The issue life cycle is simplified. The two reviewers in each group were working in pairs, using one computer to review each issue assigned. Additionally, the experimental investigator played the role of internet helper and answered reviewers' questions nearly instantly. This is not likely to occur in realistic code review, and moreover, the experimental investigators could deliver incorrect information and negatively impact the quality of the study. All of the above was a trade-off for performing the user study in a bounded duration, and they together could have indefinite impact to the validity of our results.

## **6.6 Final Remarks**

This chapter demonstrated that CSI is a successful way of doing code review from the perspective of a user. With extra semantic information for code review, GETTY positively altered reviewers' processes and led to more consistent and informative review comments. Our participants shared constructive comments to improve our tool. They liked GETTY and would like to integrate CSI into their daily code review process.

## **Acknowledgments**

This chapter, in part, is currently being prepared for submission for publication of the material, Yan, Yan; Menarini, Massimiliano; Griswold, William G. "Mining Code Repositories for Semantics-Assisted Code Review" The dissertation author was the primary investigator and author of this material.

# Chapter 7

## Future Work

In previous chapters we evaluated GETTY and proved it to be helpful in code review tasks. On the other hand, there are limitations of our approach because of how it is designed and implemented. In this chapter, we will suggest ways to improve GETTY and explore more applications for future work.

### 7.1 Quality of Invariant Differentials

The core of our approach is to difference inferred invariants before and after commits to indicate how behaviors change. In this regard, we have concerns on the quality of the inferred invariants and the way we use them. GETTY uses a dynamic approach to infer invariants from test executions. Therefore, the quality of invariant differentials relies heavily on the test suite, the tool we used for inference, and the way we difference and use invariants.

First of all, we can improve the test suite so that Daikon can infer better invariants. In our experiments (Chapter 5 and Chapter 6), we have observed that open source projects generally suffered from a lack of tests. Notably, developers paid great attention to the branch coverage of the code. For example, a method with one single branch (sequential flow) will be claimed tested with 100% code coverage as long as developers write one test case that executes the entire method body. In this case, at the beginning and ending program points we can observe at most one pair of data points.

However, invariant inference at each program point depends on sufficiently many data points, which can hardly be guaranteed by 100% branch-covered test suite. A potential solution is to use random test generator (e.g., Randoop [33]) to populate test suite with automatically generated tests, in the hope that the added tests may increase the diversity of data points and Daikon may therefore infer more accurate invariants. Additionally, we can improve Daikon by improving its inference algorithms and adding more template support, so that added data points can be better used during the inference process of Daikon.

Invariants are not the only way to imply semantic change of a commit. For example, presenting trace differences can be simpler and more straightforward when Daikon has insufficient inputs for inferring invariants. Further analysis of the traces can add more potential value to GETTY. For example, by adding extra instructions to the Daikon front-end, it can track which test(s) generated the data for each program point, and thus for each invariant for that program point. This way it can make it easier for reviewers to track and reason about the invariants.

Alternatively, changing the inference method to the one using static techniques could help without the hassle of coping with test insufficiency. Tools like Houdini [68] can help in this case, but most of them require extra annotation support and add extra burden for developers. Moreover, it could change the way CSI works because the invariant inference is no longer dependent on tests. We need to further investigate its applicability before carrying out this plan.

Last, we plan to present logical invariant differentials to reviewers. Daikon maintains its internal data structures for formal specification and has the capability to print invariants in flexible formats [69]. For example, we can add the Z3 format [70] so the invariants can be interpreted and differenced by the Z3 theorem prover.

## 7.2 Tool Integration

For now GETTY is integrated with a version control tool (*git*), build automation tool (Maven) and dynamic invariant detector (Daikon). As a prototype tool, GETTY is stand-alone for research purposes. However, for future interaction with more software tools it would be beneficial to integrate GETTY's UI into an existing tool suite.

For desktop environments we suggest creating GETTY components for existing IDEs. Specifically, we could create a GETTY plugin for Eclipse [71], one of the most popular Java IDEs. Eclipse's open marketplace provides a myriad of software engineering tools. The integration makes it easier for GETTY to interact with those existing tools. For example, GETTY is currently unable to identify method renaming and infers invariants for the renamed method as if it were newly created; working with Eclipse's refactoring tool could help GETTY resolve this issue.

For web-based environments we suggest integrating GETTY into DCPs, like Phabricator [35], which includes a suite of online software tools for developer collaborations like issue tracker, code review tool, Continuous Integration (CI) portal, and more. For each commit, Phabricator's code review tool displays not only its textual code difference but also the associated issue ID and testing results from CI. This also provides the opportunity to display GETTY's result. After submitting each commit, we

can run GETTY offline, and when the result is available we display it in the code review page to assist reviewers to apply CSI.

### 7.3 More Applications

GETTY's success uncovered the power of invariants, and we believe there are more potential applications as a result of future research on invariants. We envision the following two applications as promising for future work:

- **Mining invariant database for software maintenance.** As software project evolves, so does the invariants implied for each component. CSI compares invariants between a pair of commits; one further step is to infer invariants for all commits and observe how invariants evolve. Our hypothesis is that each time invariants of a method change, it should correspond to updates of either semantics (due to change of requirements) or the test suite; other cases can be considered a sign of introducing bugs. Mining backwards (into past commits) is a way to discover undetected bugs, while mining forwards (for the new commit) is to predict vulnerability of the new patch.
- **Mining invariant database for software reuse.** We postulate that similar software components should have similar invariants. By studying enough projects and their invariants, we can cluster similar components based on their invariants. Machine learning techniques may help during the process. We propose to abstract the clustered, similar components so they become reusable coding template and reduce the cost of software development.

Still, the applications proposed above are speculative. We plan a small user study to investigate the feasibility of each listed application before development.

# Chapter 8

## Conclusion

As a widely agreed-upon practice in software engineering, reviewers manually assess software code before it is merged into version repository or deployed to production. Aside from coding styles, etc., reviewers are interested in how the modified code affects software behaviors. The written tests may shed some light on it, but a gap still exists between the textual difference of two versions of the code and its actual impact.

Software patch comprehension focuses on tools to shorten that gap. Existing tools assist reviewers at syntax, semantic, and natural language levels. Syntax differences require the least cost to compute, but reviewers are left with the most efforts for understanding the resulted semantic impact. In contrast, understanding the patch in natural languages reduces the reviewer's burden, but it requires the highest computational cost and the result can hardly be accurate. As a balance of the costs we advocate presenting semantic differences to reviewers.

In this dissertation we proposed Continuous Semantic Inspection for code review. The core idea behind CSI is to infer invariants from concrete executions, and use the difference between two versions of the invariants to indicate behavioral changes. We implemented the tool, GETTY, to support our concept of CSI and applied it to Java open source projects. Because invariants may change due to various parts of the program, we isolate their impact to help reviewers better understand the cause of the changes. We infer



invariants with the method-level granularity. Reviewers may check invariant differentials of related methods, following the control flow in the pre-computed local-area call graph. All information is integrated in the interactive user interface of GETTY.

Our approach relied heavily on the costly dynamic invariant inference. For practical use, we scaled our approach by parallelizing test executions to relieve the memory pressure of invariant inference. The substantial performance improvement achieved enabled us to conduct a series of studies to evaluate the effectiveness of CSI by GETTY. We reached the following conclusions:

1. CSI is a feasible approach. With careful implementation the computational cost can be managed for practical use. The presence of an implementation like GETTY (Chapter 3) shows an example of a CSI tool for Java open source projects. The performance and scalability assessment of GETTY in our cluster (Chapter 4) demonstrates that CSI is especially suitable for internet-based computing environments, like clouds, for the sake of increased computational power.
2. From our quantitative study of applying GETTY on open source projects for test sufficiency and bug discovery (Chapter 5), we found invariant differentials are a compact behavioral summary with useful information for reviewers to identify problems. First, we were able to use GETTY and fail approximately one third of the inadequately tested commits, which had otherwise passed original reviewers' screening process. Second, we tracked back to the history of several randomly chosen bug-fix issues and applied CSI for each of the associated

commits, and were able to find most bugs as early as they were introduced. The experiment shows the realization of CSI is powerful in that it can grant insights for finding inadequately tested and/or buggy commits during review tasks.

3. CSI is a successful way of doing code review from reviewers' perspective. We proved this by showing how reviewers using GETTY perform their review tasks for inadequately tested commits (Chapter 6). Our user study of 12 participants using GETTY reviewing 3 realistic issues shows that GETTY positively altered reviewers' process and that most reviewers were able to leave consistent and insightful review comments.

Overall, we proposed and argued that inspecting semantic differences during code review is both helpful and cost-effective. Our approach shows an example of the concept, CSI, by differencing dynamically inferred invariants to indicate how semantic changes. We contributed GETTY, an extendable framework and implementation of CSI. GETTY alters reviewers' process so they can ask more focused questions during review and leave quality comments for developers. GETTY's success also shows that CSI has the potential for more applications than a code review tool.

# Appendix I

## Inspection Result of 100 Testing Commits

SUF = Deemed sufficient; INS = Deem insufficient; NEI = Not Enough Information  
INS (FN) = Deemed insufficient, but was incorrect

### Google GSON

Commit	Result	Commit
ef2f731	SUF	drawback: we can do for parameterized token
58dc987	INS	1/2 invalid bug - more integer types expected
7d7680f	INS	not considered array of more than one nulls
e8477b7	SUF	no change of entry-point invariants
27f9716	SUF	enough demensions
657688c	NEI	
ea6f779	INS	only tested one single digit case
903769e	INS	our case study
881ee54	INS (FN)	expect invariant changes but no. Daikon did not find it
a137944	NEI	
1bf627c	SUF	It should fail in the try block
8b852fe	SUF	TypeToken.get() can be null or not null, both tested
52179a3	SUF	yes, double deserialization
752522b	SUF	trivial case
5911ac4	INS	test one class with one specific setup
fe55a8c	NEI	
eb583ca	SUF	Daikon failed to provide more information for multi-D arraies

## Apache Commons Crypto

Commit	Result	Commit
99cae98	INS	just one input for enums is not enough
aef15f4	INS	no logging called
e4156da	SUF	expected exception
0fa9f0a	NEI	multithreading supported badly
d4c6b9f	SUF	minor change no semantic impact
8e3c24e	NEI	more positions possible, we did not handle inner class well enough
8d41191	INS	doFinal needs more multiples of 8
e692c56	INS	lack of more multiples of 8
6b6c35a	SUF	move all tests for recognition, too large to fail
2bdd4f7	INS	getCryptoInputStream impacted
58ab6e7	INS	writeChannel only false case tested
7bed857	SUF	lack of testing this.padding==0, but whether that should be the case needs more domain knowledge

## Apache Commons CLI

Commit	Result	Commit
535beb1	INS	did not consider option with - or --
d89e42a	INS	Forgot argument name=null
1042ba3	NEI	no useful invariants
0f964c6	SUF	Minor one, but interesting to know what will change/unchange
6c740e7	SUF	the same purpose had been verified already
4745ade	INS	Does getting rid of unnecessary tests change anything -- what if this.numberOfArgs=0
4141904	NEI	Did not consider same width with screen, but that was a tough
eed2561	INS	what if arguments are empty
94f50c0	SUF	When option list is long a partial option matching is tested. From CSI reviews may notice that the callee (flatten) of parse method as well as the hasOptionalArg method of OptionBuilder both changed invariants, and their invariants discovered the partially matched options, "--ver" and "verbose", respectively.
0cebfb4	SUF	indeed considered more than one -D flags, good job
f588f55	INS	how about this.args != []
83770d8	SUF	refactoring: break one test into smaller ones
0cbe335	SUF	trivial case. No change. suppress warnings - interesting if there are any semantic changes
1fcf87d	SUF	expected
22576c1	SUF	side effects of tests -- not caught by invariants
f6af623	SUF	annotation changes of tests -- no semantic changes as expected

## Apache Commons Collections

Commit	Result	Commit
5d83e4d	SUF	all tested
3b69171	SUF	tested
c33d396	SUF	basically all methods of FluentIterable class
c241318	SUF	moving tests
9e8b370	SUF	considered both non-null and null cases
11ddae0	SUF	they considered empty, non-empty, full queues, basically, all of them...
62e69cf	SUF	both non-null and null cases considered for ListOrderedMap.put
fb81a3	INS	less tests executed after commit
15ee56b	INS	did not consider underflow or overflow
80e9621	SUF	trivial case
7d2532b	NEI	no useful invariants inferred
bb684e9	SUF	all tested
7c55e29	NEI	no interesting invariants inferred
13c8e44	NEI	no tests for deterministic order, but we don't have the hint from invariants
07d84c5	SUF	existed tests no change of invariants
04af9bc	INS	HashMap:clear some old tests tested null at max index, which was gone after deletion
a06a726	SUF	tested more than expected

## Apache Commons Configuration

Commit	Result	Commit
7e92e57	NEI	no invariants inferred
227c59b	NEI	no invariants inferred
9c726bf	NEI	no invariants inferred
66690eb	NEI	no invariants inferred, but added methods are not tested
39e0246	NEI	no invariants inferred, but no test cases were added for it
d2274ba	INS	Configuration:size() this.throwExceptionMissing = true missing tests
e22e2c1	INS	missing testing default config file not exist or null case
e3cfba3	INS	lack of tesint this.result != null for configuration builder
aef15bf	SUF	reverse of the above
c88169a	INS	system default line separator varies. This should be mocked and tested.
d496d7c	NEI	get test file can be null one, which was not tested, however, we do not have its invariants
0db44b7	INS (FN)	The only this.config is good enough for the purpose, in fact. We just can't see why from invariants.
e2b5ec7	INS	did not consider outfile abnormaly
77d022e	INS	QueryResult>equals forgot the case null=null
60e232b	SUF	our invariants did not show for equals, but it was under tested
a6c3230	SUF	BeanHelper initialized more instances, good
ae90d56	NEI	our tool did not discover interesting invariants for race conditions

## Apache Commons Codec

Commit	Result	Commit
0f1e8c3	SUF	4 cases
3b8cd11	SUF	more than needed, but we cannot tell from invariants
d7b0185	INS	index $\geq 0$ is correct, and the only metaPhone is indeed necessary
cbe33f0	SUF	overkill but to the point. However it is a bad coding practice
9535a94	SUF	with one <code>_random</code> may not be fine as it is used across different test cases
f63e8e9	INS (FN)	Base64:decodeBase64 return[] over-generalized incorrect invariants
03d0f6c	INS (FN)	incorrect Daikon invariants
758111e	NEI	no interesting invariants
b1561e9	SUF	data correction seen in invariants
1e81451	SUF	ditto.
2d76aa8	NEI	bad invariants that hint nothing
54f7ca3	INS	bmpm could be null
26951aa	INS	BeiderMorseEncoder:encode could face null input
805103c	SUF	After both src and test are added previously
bc1c22b	SUF	lack of Long.MAX
2cb3bbd	INS	encoder input arg can be null
ab25ca7	INS	B64:b64from24bit: b0 = 0 case
1e531f7	NEI	In fact not fully tested
4a6c364	INS	tests not executed
2101593	SUF	Long skipped
59c42b1	INS	Base32 emptyTest with chunkSize of 32



# Appendix II

## User Study Questionnaire

The following questionnaire is intended to give us an idea of your programming background, so that we can better interpret the results. Feel free to write in the margins to explain your answers, if necessary.

1. What is your current job title (if student, indicate so here)?
2. How many years have you been programming?
3. Over the last year, about how many hours per week would you say you spend programming, on average?
4. When programming, do you typically use debugging tools? ( Y / N )  
If Y, which tool(s) do you use?

5. When programming, do you typically use testing frameworks? ( Y / N )  
If Y, which framework(s) do you use? If N, can you tell me the reason(s) why you don't use testing frameworks?

6. Did you do code review, or read others' code? ( Y / N )  
If Y, how often do you perform review tasks, or read other' code?

For 7 – 10, on a scale from 1 to 5 (1 = not at all, to 5 = very familiar):

7. How familiar are you with Java?

1      2      3      4      5

8. How familiar are you with JUnit?

1      2      3      4      5

9. How familiar are you with program invariants?

1      2      3      4      5

10. How familiar are you with Design by Contract?

1      2      3      4      5

## Bibliography

- [1] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, “The Impact of Code Review Coverage and Code Review Participation on Software Quality: A Case Study of the Qt, VTK, and ITK Projects,” in *Proceedings of the 11th Working Conference on Mining Software Repositories*, New York, NY, USA, 2014, pp. 192–201.
- [2] M. E. Fagan, “Advances in Software Inspections,” in *Pioneers and Their Contributions to Software Engineering*, M. Broy and E. Denert, Eds. Springer Berlin Heidelberg, 2001, pp. 335–360.
- [3] A. Bacchelli and C. Bird, “Expectations, Outcomes, and Challenges of Modern Code Review,” in *Proceedings of the 2013 International Conference on Software Engineering*, Piscataway, NJ, USA, 2013, pp. 712–721.
- [4] C. Sadowski, “Developer Workflow at Google (Showcase),” in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, New York, NY, USA, 2016, pp. 26–26.
- [5] Oleksii Kononenko, Olga Baysal, and Michael W. Godfrey, “Code Review Quality: How Developers See It,” in *IEEE/ACM 38th IEEE International Conference on Software Engineering*, Austin, TX, USA, 2016.
- [6] Patanamon Thongtanunam, Shane McIntosh, Ahmed E. Hassan, and Hajimu Iida, “Revisiting Code Ownership and its Relationship with Software Quality in the Scope of Modern Code Review,” in *IEEE/ACM 38th IEEE International Conference on Software Engineering*, Austin, TX, USA, 2016.
- [7] “Git.” [Online]. Available: <https://git-scm.com/>.
- [8] “Git - git-diff Documentation.” [Online]. Available: <https://git-scm.com/docs/git-diff>.
- [9] S. K. Lahiri, K. L. McMillan, R. Sharma, and C. Hawblitzel, “Differential Assertion Checking,” in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, New York, NY, USA, 2013, pp. 345–355.
- [10] G. Yang, S. Khurshid, S. Person, and N. Rungta, “Property Differencing for Incremental Checking,” in *Proceedings of the 36th International Conference on Software Engineering*, New York, NY, USA, 2014, pp. 1059–1070.

- [11] Y. Wei, C. A. Furia, N. Kazmin, and B. Meyer, “Inferring better contracts,” in *Proceedings of the 33rd International Conference on Software Engineering*, New York, NY, USA, 2011, pp. 191–200.
- [12] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, “Dynamically discovering likely program invariants to support program evolution,” *IEEE Transactions on Software Engineering*, vol. 27, no. 2, pp. 99–123, 2001.
- [13] “Apache Maven Project.” [Online]. Available: <http://maven.apache.org/>.
- [14] Y. Yan, M. Menarini, and W. Griswold, “Mining Software Contracts for Software Evolution,” in *2014 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2014, pp. 471–475.
- [15] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, “Fine-grained and Accurate Source Code Differencing,” in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, New York, NY, USA, 2014, pp. 313–324.
- [16] M. Kim and D. Notkin, “Discovering and Representing Systematic Code Changes,” in *Proceedings of the 31st International Conference on Software Engineering*, Washington, DC, USA, 2009, pp. 309–319.
- [17] T. Zhang, M. Song, J. Pinedo, and M. Kim, “Interactive Code Review for Systematic Changes,” in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, Piscataway, NJ, USA, 2015, pp. 111–122.
- [18] D. Kawrykow and M. P. Robillard, “Non-essential Changes in Version Histories,” in *Proceedings of the 33rd International Conference on Software Engineering*, New York, NY, USA, 2011, pp. 351–360.
- [19] M. Barnett, C. Bird, J. Brunet, and S. K. Lahiri, “Helping Developers Help Themselves: Automatic Decomposition of Code Review Changesets,” in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, 2015, vol. 1, pp. 134–144.
- [20] S. Lahiri, K. Vaswani, and T. Hoare, “Differential Static Analysis: Opportunities, Applications, and Challenges,” *Microsoft Research*, Nov. 2010.
- [21] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu, “Differential Symbolic Execution,” in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, New York, NY, USA, 2008, pp. 226–237.

- [22] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley, “Chianti: A Tool for Change Impact Analysis of Java Programs,” in *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, New York, NY, USA, 2004, pp. 432–448.
- [23] N. Rungta, S. Person, and J. Branchaud, “A Change Impact Analysis to Characterize Evolving Program Behaviors,” in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, 2012, pp. 109–118.
- [24] R. Holmes and D. Notkin, “Identifying Program, Test, and Environmental Changes That Affect Behaviour,” in *Proceedings of the 33rd International Conference on Software Engineering*, New York, NY, USA, 2011, pp. 371–380.
- [25] W. Maalej, R. Tiarks, T. Roehm, and R. Koschke, “On the Comprehension of Program Comprehension,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 23, no. 4, p. 31:1–31:37, Sep. 2014.
- [26] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke, “A Systematic Survey of Program Comprehension through Dynamic Analysis,” *IEEE Transactions on Software Engineering*, vol. 35, no. 5, pp. 684–702, 2009.
- [27] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, “On the naturalness of software,” in *Proceedings of the 2012 International Conference on Software Engineering*, Piscataway, NJ, USA, 2012, pp. 837–847.
- [28] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu, “On the ‘Naturalness’ of Buggy Code,” in *Proceedings of the 38th International Conference on Software Engineering*, New York, NY, USA, 2016, pp. 428–439.
- [29] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim, “How Do Software Engineers Understand Code Changes?: An Exploratory Study in Industry,” in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, New York, NY, USA, 2012, p. 51:1–51:11.
- [30] S. Panichella, A. Panichella, M. Beller, A. Zaidman, and H. C. Gall, “The Impact of Test Case Summaries on Bug Fixing Performance: An Empirical Investigation,” in *Proceedings of the 38th International Conference on Software Engineering*, New York, NY, USA, 2016, pp. 547–558.
- [31] T. Xie and D. Notkin, “An Empirical Study of Java Dynamic Call Graph Extractors,” University of Washington Department of Computer Science and Engineering, Seattle, WA, USA, UW-CSE-02-12-03, Dec. 2002.

- [32] Philipp Hirsch, “Automatic inference of JML-based security specifications with exception handling,” Master Thesis, Universität Bremen, 2016.
- [33] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, “Feedback-Directed Random Test Generation,” in *29th International Conference on Software Engineering (ICSE’07)*, 2007, pp. 75–84.
- [34] “Gerrit Code Review.” [Online]. Available: <https://www.gerritcodereview.com/>.
- [35] “Phacility - Phabricator.” [Online]. Available: <https://www.phacility.com/phabricator/>.
- [36] “apache/commons-collections,” *GitHub*. [Online]. Available: <https://github.com/apache/commons-collections>.
- [37] “checkstyle - Checkstyle 7.5.1.” [Online]. Available: <http://checkstyle.sourceforge.net/>.
- [38] D. Hovemeyer and W. Pugh, “Finding Bugs is Easy,” *SIGPLAN Not.*, vol. 39, no. 12, pp. 92–106, Dec. 2004.
- [39] “Maven - Maven EMMA plugin.” [Online]. Available: <http://emma.sourceforge.net/maven-emma-plugin/>.
- [40] “google/gson,” *GitHub*. [Online]. Available: <https://github.com/google/gson>.
- [41] G. C. Murphy, W. G. Griswold, M. P. Robillard, J. Hannemann, and W. Leong, “Design Recommendations for Concern Elaboration Tools,” in *Aspect-Oriented Software Development*, R. Filman, T. Elrad, S. Clarke, and M. Aksit, Eds. Addison-Wesley, 2004, p. pp.507-530.
- [42] “Git.” [Online]. Available: <http://git-scm.com/>. [Accessed: 06-Aug-2014].
- [43] “Git - git-stash Documentation.” [Online]. Available: <https://git-scm.com/docs/git-stash>.
- [44] “Git - git-log Documentation.” [Online]. Available: <https://git-scm.com/docs/git-log>.
- [45] “Apache Commons - Apache Commons.” [Online]. Available: <https://commons.apache.org/>.
- [46] J. H. Perkins and M. D. Ernst, “Efficient Incremental Algorithms for Dynamic Detection of Likely Invariants,” in *Proceedings of the 12th ACM SIGSOFT Twelfth*

*International Symposium on Foundations of Software Engineering*, New York, NY, USA, 2004, pp. 23–32.

- [47] “Fix BigInteger equals bug, and its tests. by ybank · Pull Request #903 · google/gson,” *GitHub*. [Online]. Available: <https://github.com/google/gson/pull/903>.
- [48] “Added tests for issue 249 · google/gson@7d7680f,” *GitHub*. [Online]. Available: <https://github.com/google/gson/commit/7d7680fd2f1e63e28cbf1b844831b91a953ac420>. [Accessed: 06-Apr-2017].
- [49] “New test methods testRfc4648 based on RFC 4648 that show our inconsis... · apache/commons-codec@f63e8e9,” *GitHub*. [Online]. Available: <https://github.com/apache/commons-codec/commit/f63e8e9a4cd509e3073a2db90e39e985ac7bcf67>. [Accessed: 06-Apr-2017].
- [50] L. Inozemtseva and R. Holmes, “Coverage Is Not Strongly Correlated with Test Suite Effectiveness,” in *ICSE '14*, Hyderabad, India, 2014.
- [51] D. Team, “Eclipse Debug Project.” [Online]. Available: <https://www.eclipse.org/eclipse/debug/>.
- [52] “Git - git-blame Documentation.” [Online]. Available: <https://git-scm.com/docs/git-blame>.
- [53] “[CLI-252] LongOpt falsely detected as ambiguous - ASF JIRA.” [Online]. Available: <https://issues.apache.org/jira/browse/CLI-252>.
- [54] H. J. Postel, “Die Kölner Phonetik—Ein Verfahren zur Identifizierung von Personennamen auf der Grundlage der Gestaltanalyse,” *IBM-Nachrichten*, vol. 19, pp. 925–931, 1969.
- [55] D. R. Morrison, “PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric,” *J. ACM*, vol. 15, no. 4, pp. 514–534, Oct. 1968.
- [56] “[COLLECTIONS-525] PatriciaTrie - ASF JIRA.” [Online]. Available: <https://issues.apache.org/jira/browse/COLLECTIONS-525>. [Accessed: 05-Apr-2017].
- [57] “[CONFIGURATION-487] DataConfiguration.get() cannot handle a trivial conversion - ASF JIRA.” [Online]. Available: <https://issues.apache.org/jira/browse/CONFIGURATION-487>. [Accessed: 05-Apr-2017].

- [58] “Google Search Engine.” [Online]. Available: <https://www.google.com/>.
- [59] “Stack Overflow.” [Online]. Available: <http://stackoverflow.com/>.
- [60] A. H. JØRGENSEN, “Thinking-aloud in user interface design: a method promoting cognitive ergonomics,” *Ergonomics*, vol. 33, no. 4, pp. 501–507, Apr. 1990.
- [61] N. Miyake, “Constructive Interaction and the Iterative Process of Understanding,” *Cognitive Science*, vol. 10, no. 2, pp. 151–177, Apr. 1986.
- [62] “google/gson@e450822,” *GitHub*. [Online]. Available: <https://github.com/google/gson/commit/e4508227c53749b48318366c1272119031851887>.
- [63] “Issue #627 · google/gson,” *GitHub*. [Online]. Available: <https://github.com/google/gson/issues/627>. [Accessed: 05-Jan-2017].
- [64] “google/gson@d5319d9,” *GitHub*. [Online]. Available: <https://github.com/google/gson/commit/d5319d9e840b2c7237ca435f50c50ffbe7dce507>.
- [65] “Bitbucket.” [Online]. Available: <https://bitbucket.org/>.
- [66] J. Czerwonka, M. Greiler, and J. Tilford, “Code Reviews Do Not Find Bugs: How the Current Code Review Best Practice Slows Us Down,” in *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, Piscataway, NJ, USA, 2015, pp. 27–28.
- [67] B. Meyer, “Applying ‘design by contract,’” *Computer*, vol. 25, no. 10, pp. 40–51, Oct. 1992.
- [68] J. W. Nimmer and M. D. Ernst, “Invariant Inference for Static Checking:,” in *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, New York, NY, USA, 2002, pp. 11–20.
- [69] “The Daikon Invariant Detector Developer Manual.” [Online]. Available: <https://plse.cs.washington.edu/daikon/download/doc/developer.html>.
- [70] L. de Moura and N. Bjørner, “Z3: An Efficient SMT Solver,” in *Tools and Algorithms for the Construction and Analysis of Systems*, 2008, pp. 337–340.



[71] E. F. Inc, "Eclipse - The Eclipse Foundation open source community website."  
[Online]. Available: <https://eclipse.org/>.