Semantics-Assisted Code Review

An Efficient Toolchain and a User Study

Massimiliano Menarini, Yan Yan, and William G. Griswold
Department of Computer Science and Engineering
University of California at San Diego
La Jolla, CA, USA
mmenarini@ucsd.edu, yayan@cs.ucsd.edu, wgg@cs.ucsd.edu

***Abstract*—Code changes are often reviewed before they are deployed. Popular source control systems aid code review by presenting textual differences between old and new versions of the code, leaving developers with the difficult task of determining whether the differences actually produced the desired behavior. Fortunately, we can mine such information from code repositories. We propose aiding code review with inter-version semantic differential analysis. During review of a new commit, a developer is presented with summaries of both code differences and behavioral differences, which are expressed as diffs of likely invariants extracted by running the system's test cases. As a result, developers can more easily determine that the code changes produced the desired effect. We created an invariant-mining tool chain, Getty, to support our concept of semantically-assisted code review. To validate our approach, 1) we applied Getty to the commits of 6 popular open source projects, 2) we assessed the performance and cost of running Getty in different configurations, and 3) we performed a comparative user study with 18 developers. Our results demonstrate that semantically-assisted code review is feasible, effective, and that real programmers can leverage it to improve the quality of their reviews.**

***Index Terms*—Software behavior, mining software repository, code review, likely invariants, dynamic impact analysis, scalability, software testing**

I. Introduction

To aid code review [1], today's popular Version Control Systems (VCS's, e.g., *git* [2]) integrate with textual differencing tools (e.g., *git-diff* [3]). A reviewer can start with a summary of changed code between two versions, then navigate to related code snippets at her own discretion. However, the textual, program-level differences from current differencing tools provide only indirect information about the behavioral impact of code changes. The results of testing provide only a pass/fail view of that behavior, perhaps disguising subtle bugs. A reviewer should 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.

We propose that reviewers would benefit from summaries of the behavioral effects of those changes. The summaries should be concise, comprehensive, presented in a familiar notation, and complement existing review information. Likewise, the production of summaries should require little effort on the part of the developer or reviewer, just like Continuous Integration today supports effortless regression testing. We call the resulting infrastructure and process *Semantic-Assisted Code Review* (SCR).

The idea of differential assertion checking [4], [5] is a gesture in the right direction, but these works require manual effort and do not provide a comprehensive view of semantic effects. However, applying this concept to behavioral summaries mined from code repositories could provide the best of both worlds. Existing tools like Daikon can infer likely invariants and report them in the terminology of the program itself [6]. Likely invariants are properties that summarize the traces of runs fed to Daikon. (Following the terminology used by Daikon, we use the term likely invariant or invariant to refer to these properties.) We propose that performing inter-version differencing of the (voluminous) invariants extracted by Daikon will provide a concise "behavioral diff". Because Daikon invariants are reported at the method level, it would be easy to attach them to the code difference summaries provided by a tool like *git*.

Our approach, realized in a tool called GETTY, builds on Daikon and provides an infrastructure for automatically pulling, building, testing, and analyzing multiple commits of a program from its code repository. Making this approach practical requires solving three critical problems. 1) Because several data- and compute-intensive steps are required we need to address performance (and cost) issues. 2) Because there are two potential causes of change to Daikon's dynamically inferred invariants – changes to source code and changes to tests – we must identify how to extract and compare invariants to highlight the contributions of each of these changes. And 3) because of the novelty and possible complexity of running code reviews based on behavioral summaries, we need to verify that real programmers can effectively use them to improve the quality of their code reviews. The following contributions address these problems:

1. We introduce three complementary techniques for effectively mining behavioral information (in our case, Daikon invariants) for code review (Section III.B). The techniques, applied in GETTY, include (a) using impact analysis to scope invariant mining to the methods related to the current commit, (b) mitigating memory pressure by tracing one class at a time, and (c) parallelizing trace gathering and invariant extraction. Our experiments on 6 open source projects demonstrate that these techniques enable nearly arbitrary reductions in running time by parallelizing analyses.

978-1-5386-2684-9/17$15.00 © 2017 IEEE

ASE 2017, Urbana-Champaign, IL, USA
Technical Research

Moreover, renting processors in the cloud makes running GETTY for SCR acceptable both in terms of time and cost.

2. To help the reviewer understand the effects of changed source code versus changed test cases, our approach extracts likely invariants and presents their differences for various combinations of source code versions and test suite versions. We show how SCR behavior-change summaries can reveal bugs and other problems earlier than they were actually discovered by the original reviewers. To this end, we applied GETTY on a portion of Google's GSON revision history (Sections II and IV). We discovered two bugs that were previously unreported. Moreover, in a retrospective analysis of GSON and five other open source projects, we show that invariant differentials helped find testing gaps in 32 of 100 selected commits (Section V). We also analyzed six randomly chosen previously known bugs (one in each project); GETTY's invariant differentials made the bug evident at its point of introduction in 4 out of 6 cases.
3. To ensure that the results we achieved in our experiments with GETTY can be achieved by programmers in general, and to assess the benefits of SCR compared to traditional lightweight review processes, we ran a comparative user study with 18 participants. We divided the participants in 9 teams of 2 and assigned 3 review tasks on real commits of the GSON open source project. Six teams used the GETTY toolchain for their review while three teams used the tools available in Github. The results of this study demonstrate that SCR as implemented by GETTY changes how reviewers perform their task and can improve the quality of the feedback they produce.

Before diving into GETTY's design and evaluation, we motivate and define Semantic-Assisted Code Review. Before closing, we discuss related work in Section VII.

II. SEMANTICS-ASSISTED CODE REVIEW WITH GETTY

We illustrate how to perform SCR with GETTY by following the review of a real commit taken from the GSON project. The review follows the work of real reviewers that participated in a user study further discussed in Section VI. GSON is a Google-sponsored open source Java library for performing conversions between Java Objects and their JSON representations [7]. When we ran our study, GSON had undergone 1,322 commits by 52 contributors, with 34 software releases since the project's start in 2008. All changes were peer reviewed using what's called *lightweight code review*, which attempts to achieve the benefits of formal code review with lower overhead and delay [8].

We support our scenario and the analysis of how the review is carried out by quoting study participants. The case study presented in this section does not follow the review of a single study participant; instead, it summarizes the process followed by different teams and is simplified to demonstrate the key elements of a continuous semantic review with GETTY (e.g., we show the review of a single method).

We review GSON commit #e450822, which modified the class LazilyParsedNumber. This class parses strings into numeric values; it does this lazily, meaning that the class maintains a string representation of the number and converts it to a numeric

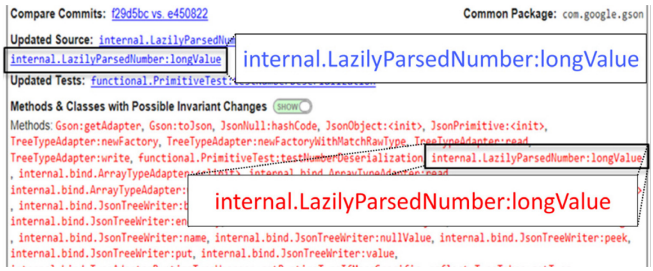

Fig. 1. GETTY methods and classes with changed invariants

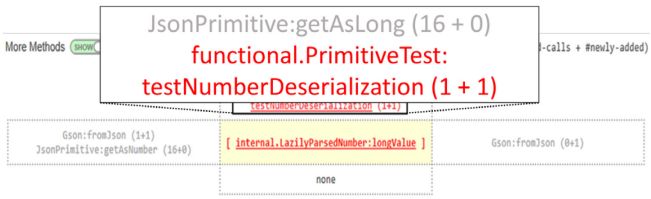

Fig. 2. GETTY navigation of the call graph

value only when a method returning the number is invoked. The patch changes two methods of this class: intValue and longValue which return the number represented as an int and long respectively. The changes implemented in the patch use the BigDecimal class to parse the string even when an integer value is requested. Therefore, no exception is thrown when the string has decimals or if the number is very large. The patch adds one test.

The reviewer first looks at the commit message and possibly at the referenced issues that the commit addresses. In this case the message is: "*Use BigDecimal to parse number string when requesting it as integer. LazilyParsedNumber 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*".

This description is not clear; unfortunately, unclear commit messages and issue reports are commonplace. One of our reviewers commented: "*... ignores all bits that overflow the range of requested integer type. ... that is not a good way of saying that!*" [C1LA] (See Table V for decoding C1LA).

Next, the reviewer opens the GETTY and peruses the upper box in the page (Fig. 1), which lists top-to-bottom (a) the source code methods that have changed (highlighted in blue), (b) the testing methods that have been updated (also highlighted in blue) and (c) the methods whose invariants have changed (highlighted in red). The reviewer notices that both the code of method longValue and its invariants changed (boxed in the picture).

The reviewer clicks the LazilyParsedNumber:longValue method link (boxed in Fig. 1), causing it to be listed in the middle of the next section of dotted boxes (Fig. 2), as well as listing its changed invariants below (Fig. 3).

The dotted boxes summarize the invocations closely related to longValue. Above longValue, the reviewer sees that longValue has two direct callers. Being shown in red means they have changed invariants, underlined means changed source. In this case, a test case calling longValue has changed. To the left of longValue are methods that one of longValue's callers called immediately before calling longValue, and to the right methods called immediately

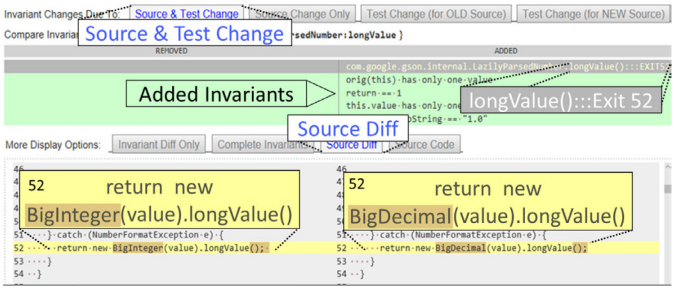

Fig. 3. GETTY invariants diff for longValue

```
156 ···json·=·"1.0";
157 ···actual·=·gson.fromJson(json,·Number.class);
158 ···assertEquals(1L,·actual.longValue());
```

Fig. 4. GETTY right side of diff for testNumberDeserialization (added code)

after longValue returned. The gray text color indicates no changes to the method's source or invariants. The reviewer does not see anything on the left suggesting they would affect longValue's invariants, so she is not motivated to click any of them. The box below shows the methods that longValue itself called, in this case no method is called.

The reviewer now turns her attention to the invariants displayed below the invocation summary (Fig. 3). Removed invariants would be highlighted in red, added invariants in green, and changed invariants in yellow (the figure shows only added invariants), just as text changes are highlighted in *git*. The gray header above the invariants indicates that they are for the exit point of the method in line 52 (the return inside the catch block in yellow). The reviewer observes immediately that it appears that only one value is parsed using BigDecimal in the modified code. In the reviewer's own words: "*So I think 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 all branches.*" [E2RA] This is a working hypothesis the reviewers can then validate by browsing through the code.

She then clicks on testNumberDeserialization in the top box of Fig. 2, opens the Source Diff view (Fig. 4) and confirms that indeed only one test was added; a simple test parsing the string "1.0", thus confirming the hypothesis.

Before firing off a comment to the developer, however, she uses impact analysis to confirm that the existing test works as expected. Likely invariants are computed by running each project's test cases. In this example, both the main source code and the test code have been modified; there are two potential causes for Daikon's likely invariants to change. GETTY isolates the impact of these changes by computing the invariants for different combinations of source and test code versions. For example., the GETTY invariant diff for longValue in the older commit code running the old and new test suites shows a THROWSCOMBINED section on the right side (new test suite). This shows that the old code throws an exception when tested with the value "1.0". One reviewer in our user study stated: "*Okay, so the old source is throwing an exception, presumably because it might have been called as something it wasn't supposed to be called*" [E6RE].

This scenario highlights three features of SCR with GETTY: invariant differentials, impact isolation, and invocation flows. The display of just the *changes* in invariants provided the reviewer a concise behavioral view of the commit, enabling quick creation of working hypotheses about the commit despite the dozens of underlying invariants. Her ability to explore different combinations of old and new tests helped her isolate the behavioral effects due to the source changes. Finally, the summary of the application's call structure around longValue helped her quickly focus on a particular part of the program. We next elaborate on these three elements.

A. Diff'ing Behavior

SCR depends on having 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 commits can be quite small, so could the differences in their invariants. This motivates the creation of invariant difference sets between program commits 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 longValue method in example of the previous section, with no isolation of effects (Fig. 3), there were 2 invariants before commit and 6 after. As shown in the figure, for just the changed invariants, there were just 4 added – a 50% reduction. Other examples we encountered in our study showed reductions of more than 90%.

For each kind of program point of a method *m* – entry, exit, and exceptional exit – GETTY calculates the change in invariants between an older commit and a newer commit as two sets, the removed invariants and the added invariants. Following *git*'s style of code differencing, GETTY displays added, removed, and *changed* invariants. A changed invariant is just 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 contain the same variable, operator, and value type. Taking advantage of Daikon's consistent invariant formatting, GETTY is able to use the minimality of text differences to infer changed invariants. Comparing the logical formulae would achieve better results in some cases [9].

B. Impact Isolation

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

By running the same test cases on both the old and the new source, any resulting invariant differences can be confidently attributed to the changes to the source code. Likewise, by running the old and new test cases on the same source, any resulting invariant differences can be attributed to the changes to the test

TABLE I. BEHAVIOR ISOLATION STRATEGIES

Effects of	"Old" src/test combo		"New" src/test combo	
	src	**tests**	**Src**	**tests**
entire commit	*old*	*old*	*New*	*new*
source only	*old*	*old* ∪ *new*	*New*	*old* ∪ *new*
tests for *old* src	*old*	*old*	*Old*	*new*
tests for *new* src	*new*	*old*	*New*	*new*

cases. Because the separation of source code and tests is standardized, it is possible for GETTY to automatically extract and show the invariants for a commit under four different combinations of source and tests, shown in Table I.

The first variant provides no isolation, showing the full effects of a new commit by simply running the older commit's tests on the old source, and the newer commit's tests on the new source.

The second isolates source effects by running each of the old and new source on a common set of tests, the *union* of the old and new tests. We motivate this variant by first considering an alternative, the *intersection* of the two commits' test suites. The resulting test suite would be guaranteed to compile and run on both source code bases. However, this excludes test cases that were intentionally written to demonstrate the behavior of a particular commit, so it would often produce less useful invariants than desired. The union of all tests, on the other hand, will include those unique tests, but oftentimes some won't run or compile on the commit for which it wasn't written. As a simple example, if a new method is introduced in the new commit and some tests are added to test this method, then these new tests will not compile with the old source. This creates an asymmetry in which test cases run on which commit, 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. The fact that a test case runs on one commit but not the other reveals a *behavioral* property of the source code. Thus, the default condition for isolating source effects is to run the union of the test cases, modulo compilability.

When the reviewer is interested in effects due to changes in the test cases, GETTY executes the test suites defined in the two commits on the same source code. Since the source code bases cannot be unioned like the tests, there are two included variants (last two rows of Table I): running the two test suites on the old source, and running the two test suites on the new source. Fully understanding the effects of the changes to the test cases might require spending time looking at the results of both variants. As discussed in Section II regarding GSON commit #e450822, the test impact isolation running new tests on old code reports an exception, showing that the new test is not simply passing all the time, but is also capable of revealing incorrect behavior.

C. Invocation Flows

A change of one method in the source code can have widespread effects on the behavior of numerous methods. This is a motivation for providing behavior change summaries, that directly articulate those wide-spread effects. Still, a reviewer needs help in finding her way around. Semantic effects are propagated directly by the application's control flow: a field is set in one method, and then its value is passed to another, 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 [10], [11].

GETTY provides a local-area call-graph, as seen in the dotted boxes in Fig. 2. Only callers, immediate siblings, and callees whose invariants have changed are necessarily displayed. As screen space allows, more 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's control flow.

GETTY computes the invocation flows from execution traces during testing. Because the flows actually occurred, a reviewer can compare them with the expected flows to identify problematic or unexpected results [12].

III. IMPLEMENTING AND SCALING SCR

Like Continuous Integration (CI), SCR depends on heavy lifting in the back-end to support developers' and reviewers' work. SCR requires not only the same compilation and testing support of CI, but also adds the often-massive cost of mining likely invariants with Daikon. Both human costs and computational costs must be minimized. We discuss each in turn.

A. Automated Invariant Differential Extraction

Similar to Yan *et al.* [13], we leverage existing open source tools commonly used in Continuous Integration, like the build tool Maven [14] and the testing framework JUnit [15]. Specifically, we implement the GETTY tool chain in four main components:

1) Static Source Diff Analyzer (*villa*) – this component takes the textual code differentials from *git* and determines which methods and test cases have changed. These constitute the "change set".

2) Dynamic Callgraph Analyzer (*agent*) – Next, the old and new source are run on their test suites to extract their dynamic call graphs and acquire the change set's local invocation flows. *Agent* supports multi-threading and exceptions. The resulting "impact set" will be an expanded change set containing all methods and test cases for which we calculate invariants on the two commits.

3) Invariant Detector (*center*) – this component checks out two commits and, using Daikon, infers invariants for all methods in the impact set, in all combinations of tests and source specified in Table I. (*Center* uses a version of Daikon that supports exceptional invariant detection *[16]*.) Since test methods can depend on each other, *center* always executes the whole test suite. To minimize disk I/O, we pipe traces directly to Daikon.

4) Semantic Differential Viewer (*gallery*) – Next, *gallery* assigns invariant changes to methods and creates the user interface described in Section II.

In our use of GETTY on GSON, *center* takes over 95% of the total execution time (with *agent* next at under 3%), even when

just focusing on the impact set. Even for the small GSON app, center requires over 3 hours to run. The next subsection focuses on *center*'s performance.

B. Scalable Invariant Differential Extraction

Ideally, invariant extraction times would be commensurate with build and testing costs, enabling a timely repair-compile-review feedback loop. To achieve such performance, we have three advantages to exploit.

- We only need to trace, and extract invariants for, the methods that are relevant to the current pair of commits, the impact set. Thus, we apply dynamic impact analysis to restrict tracing and invariant inference to relevant methods.
- Much of the performance cost is due to memory pressure. As long as the full test suite is run, the Daikon trace for a class will be the same regardless of which other classes are being traced. To this end, we trace one class at a time, rather than all relevant methods at once.
- Invariant detection is embarrassingly parallel, allowing us to parallelize inference in the cloud. 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 the cores. Each group for a project is executed on a separate core, concurrent with the other groups. After all the groups finish, *center* merges all the invariants collected, a trivial step. Because of the class granularity, the limit to parallelization is the number of classes in the impact set.

To evaluate these techniques, we used GSON and five other randomly selected projects from Apache Commons [17]. We took 10 random commits of each project, executed the *center* analysis on them, and took the truncated mean for each project (throwing out the low and high times). Their characteristics are summarized in Table II. The total time taken by the Daikon inference process running on all methods in the impact set, shown in the second to last column (for reference, we call this the *all-tests-all-methods* mode or ATAM), increases with the running time of the test suite. The cost of inference has no evident relationship with the other factors. The size of a commit will influence the size of the change set and hence the inference time, but we controlled for this by averaging over the 10 commits. While limiting inference to the impact set makes invariant extraction tractable, the cost is far too high for the last three projects, requiring a day or more to run (the last was killed after more than three days).

Reducing memory pressure by tracing one class at a time (*all-tests-single-class,* or ATSC) cuts run times by about a factor of two, as shown in Table II, "ATSC".

We evaluated the parallel mode, called *Parallel-ATSC* or PATSC, on a cluster with eight 2-processor nodes, typical of what can be found in a cloud deployment today. Each node has 2.66GHz CPUs and 16GB of RAM. Fig. 5 plots running time against the number of processors used. The single-processor condition runs plain ATSC, the rest PATSC. PATSC running on the same single node (using both processors) achieved 1.8 speedup and 89% efficiency. Its suboptimal speed-up is due to

TABLE II. PROJECT CHARACTERISTICS

Projects	Commits	KLoC	Methods	Tests	TTE	TIE (ATAM)	TIE (ATSC)
GSON	1,294	36.6	582	1,276	13.3s	3.22h	2.70h
CLI	827	8.6	212	206	18.0s	5.98h	3.10h
Codec	1,608	14.1	272	334	23.3s	11.64h	3.67h
Crypto	550	9.9	183	20	49.5s	23.20h	9.43h
Collections	2,881	100.1	3,177	1388	86.9s	55.51h	25.36h
Configuration	2,730	92.3	2,121	1,962	127.5s		54.81h

TTE: total time for testing; **TIE**: total time for inference
Environment: Intel 2.53 GHz Dual-Core, 4GB DDR3 RAM, Mac OS X Yosemite

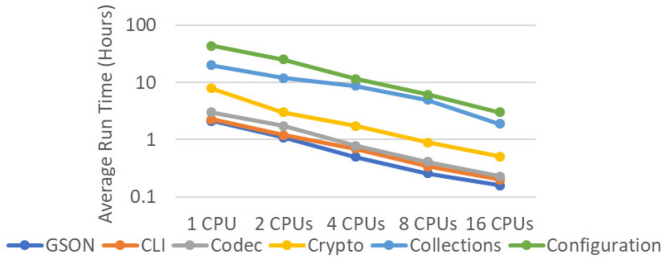

Fig. 5. Average running time versus number of CPUs. The single-processor mode is ATSC; others are PATSC. Environment: Intel 2.66 GHz Dual Processor, Quad-Core, 16GB RAM, Ubuntu Server 16.04.

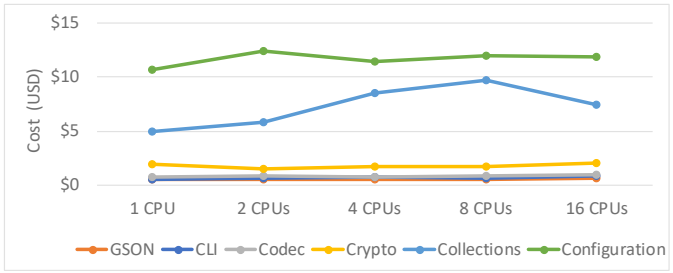

Fig. 6. Cloud cost of Getty's invariant extraction plotted against the number of CPUs used. Each quad-core processor is priced as one independent Amazon EC2 t2.xlarge instance (4 vCPU up to 30GHz, 16GB RAM, $0.244/hour for North California, Feb 7, 2017).

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 SCR.

Related is the cost of computing GETTY's invariants in the cloud, say as part of an existing continuous integration process. Fig. 6 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%.

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 have short running times that don't demand high levels of parallelization.

IV. How to Identify Problems with Getty

In Section II we saw that SCR with Getty can both quickly reveal inadequate testing and help users to successfully review a GSON commit [18]. In this section, we show how Getty supports SCR in finding bugs.

We highlight commit #10 (903769e) in Fig. 8, which introduced a test comparing two JSON primitive integers: 2^{64}+5 and 5. 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 we analyzed, #12 (423d18f), the developers assessed that "the price is too much to pay" to fix the bug and overrode the test's 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 SCR could have aided in finding this bug sooner, preferably at the point of introduction, were it in use by this project at the time.

In commit #4 (e89c949), 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.

Fig. 7 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 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. We expect that the branch returns true when both value attributes store the same integer values, regardless of the specific integer types. The invariant differentials are shown in Fig. 10a. At the 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 should return false if 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 is informative. 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 a 64-bit signed integer so any integer representation over 64 bits will be masked when converting to Long. For example, 2^{64}+1 takes 65 bits, and is converted to 1 after conversion. We therefore create four JsonPrimitive objects: one for 1, lp1 for 2^{64}+1, lp1c for 2^{64}+1, and lp2 for 2^{64}+2, and assert that none of them are equal except for the pair of lp1 and lp1c. Our test (Fig. 9) is stronger than Fig. 8's, because it only considers the case where other.value is BigInteger, but ours considers this.value being BigInteger as well.

```
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+ }
```

Fig. 7. New feature to compare between integers in commit # e89c949

```
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));}
```

Fig. 8. The failing test case in commit #10

```
public void testEqualsForBigIntegers() {
 BigInteger limit =    // 2^64
  new BigInteger("18446744073709551616");
 JsonPrimitive one = new JsonPrimitive(1L);
 JsonPrimitive lp1 =   // limit + 1
  new JsonPrimitive(limit.add(new BigInteger("1")));
 JsonPrimitive lp1c =   // limit + 1, a different object
  new JsonPrimitive(limit.add(new BigInteger("1")));
 JsonPrimitive lp2 =   // limit + 2
  new JsonPrimitive(limit.add(new BigInteger("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));}
```

Fig. 9. Test case to confirm the integer-equality bug

Since our test fails, we conclude that the bug 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. A reviewer could have found the bug much earlier if they had been able to examine invariant differentials.

Additionally, consider the branch from line 377 to line 379 that deals with floating-point comparisons. Similar 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.

In Fig. 10a, the dynamic invariant at exit-point 378 says the return value is always true, i.e., for all tests so far this branch has only returned true. 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 (Fig. 11) to compare unequal decimals. 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 cor-

```
Exit-Points:
 point-375:
  (return = false) → (this.value is Long)
 point-378:
  return = true
```

(a) *Commit #4: the invariant differential clearly indicates a bug.*

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

(b) *Commit #11: invariant differential points to bug like commit #4's.*

Fig. 10. Selected dynamic invariant differentials across commits. For presentation purposes, fully qualified names have been shortened.

```
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 =      // dmax + 100.0
     new JsonPrimitive(dmax.add(new BigDecimal("100.0")));
   JsonPrimitive largeBD =      // dmax + 200.0
     new JsonPrimitive(dmax.add(new BigDecimal("200.0")));
   assertFalse("smallBD = largeBD", smallBD.equals(largeBD));
```

Fig. 11. Test case to confirm the decimal-equality bug.

rectly 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 the GSON project. In commit #11 (commit hash a263a3f, right after developers discovered the integer comparison bug in commit #10), the developers further modified the same decimal comparison branch; but the invariant differential in Fig. 10b shows that, similar to exit-point 375 in Fig. 10a, at the exit-point of the decimal branch the return value is incorrectly correlated to the specific type of value attribute. This is a new bug, discovered with GETTY. We submitted a bug report to the GSON project, along with the test case in Fig. 11, and it awaits action.

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 don't 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; we only examined dynamic invariant differentials. We concluded not only were they insufficiently tested, but also both branches have bugs.

We also notice that most of the information on test quality is a result of the impact isolation analysis (Section IIB) that reviewers need to perform to understand the source of behavioral changes when looking for bugs. Therefore, while other tools exist that can provide similar information on test quality, detecting test problems is a convenient side effect of looking for bugs with semantics-assisted code review.

V. EFFECTIVENESS OF INVARIANT DIFFERENTIALS

In Sections II and IV we replayed part of the history of GSON, to demonstrate that SCR with GETTY can help reveal insufficient testing and find bugs. Using the six open source projects studied in Section III (Table II), this section addresses the question of whether SCR is effective on a regular basis.

A. Identifying Test Insufficiency

Using GETTY, we applied SCR on 100 test-only commits randomly selected from the 6 projects. The number of commits for each project varies according to the project size and history length, (See Table III). The Maven EMMA plugin [19] reports 100% branch test coverage for the chosen testing commits. We 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.

As a simple metric, we consider tests sufficient for a method if they cover all combinations of *types* that result in different behaviors of the method. For example, when testing equals for two integers (See Section IV), we want to see test cases for all combinations of regular and big integers, each with equal and unequal values. Although this overlooks corner cases (i.e., it lowers the experiment's success rate), it is a straightforward, repeatable metric. We determined ground truth by exhaustively inspecting the tests and source after making the first determination with GETTY.

We summarize our code review results in Table III. Cumulatively, of the 100 testing commits, 32 were identified as being insufficient. GETTY led us to incorrectly classify 4 commits as

TABLE III. RESULTS OF TEST SUFFICIENCY INSPECTION

Projects	Commits	Insuff	False Insuff	Little Data
GSON	17	5	1	3
CLI	16	5	0	2
Codec	21	7	2	3
Crypto	12	6	0	2
Collections	17	3	0	3
Configuration	17	6	1	7
Cumulative	100	32	4	20

Insuff: Test insufficiency evident; **False Insuff.**: Incorrectly classified as insufficient;
Little Data: too few invariants;

TABLE IV. RESULTS OF BUGGY COMMIT INSPECTION

Projects	Introduction Commit	Discovery Commit	Time Between Commits	Y/N
GSON	b634804	60ef777	1 day	Y
CLI	e366a69	085a153	72 months	N
Codec	2405423	b9cab09	159 days	N
Crypto	f3c5416	a983a2c	67 days	Y
Collections	eced882	59c6e94	158 days	Y
Configuration	f59158e	d6c3900	85 days	Y

Y/N: Whether CSR lended insights to discover the bug at the time of its introduction

TABLE V. 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 (N, E, A)
N = novice (under two years), E = experienced (under 10 years), and A = advanced (10+ years).

insufficiently tested, due to invariants derived from coincidental correlations in the data. Of the remaining 68 commits, 20 had too few invariants generated to support a judgment. Reviewer experience might aid in improving these numbers. One fifth of the commits did not produce sufficient invariant differentials for the task. Given the simple nature of our insufficiency metric, these results are quite positive. These results corroborate results that popular test coverage tools are not always a good indicator of test suite effectiveness [20].

During this study, we observed two simple invariants that often exposed inadequate testing, both seen in the scenario in Section II. One is the failure to test for null as an input value (e.g., other ≠ null). The other is a Boolean return value always being true or false. Additionally, invariants on exception exits were helpful in confirming behavior after a failure was intentionally induced by a test case.

B. Finding Bugs

We randomly selected one confirmed bug from each project (including a new one from GSON), found the commit that introduced the bug, and applied SCR to check whether the bugs could have been found when introduced. 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 the Eclipse JDT remote debugger [21], and studied the root cause of the bug. Then we used *git-blame* [22] to trace back through the editing history to find the commit that made the buggy edit. Finally, we used GETTY to review the semantic changes of the commit that introduced the bug. See Table IV.

The differentials aided discovery of four of the six bugs, between 1 and 158 days before the bug was reported. For the two that failed, Daikon's lack of invariants over the contents of strings was the cause. For the bug in issue CLI-252 [23], the command line parser threw an exception when parsing an option that is the prefix of another. The bug was introduced when developers added prefix matching to the parser. The bug in Codec concerns a string encoding using the Kölner Phonetik algorithm [24] that contains sequentially repeated digits.

VI. A COMPARATIVE USER STUDY

In this section, we describe a comparative user study that demonstrates that real reviewers can effectively use GETTY to improve the quality of their code reviews.

A. Study Set-up

We arranged the study to simulate a lightweight code review on 3 commits of the GSON project. We identified three roles: developer, reviewer, and internet helper. The developer is the programmer who implements the functionality or fixes the bug, and updates the issue (played by investigators). A reviewer is a separate engineer who reviews the code changes and provides comments to the developer (played by our study subjects). The internet helper plays the role of search engines (e.g., Google), online Q&A communities (e.g., StackOverflow), and more for GETTY, since the tool, being a prototype, has no online presence (role played by the investigators).

We enrolled 18 anonymous participants with 1 to 16 years of programming experience in academia or industry (Table V). All our participants fill out a questionnaire and we rated their programming experience as: novice (under two years), experienced (under 10 years), and advanced (10+ years).

We grouped our participants into 9 two-member review teams of comparable experience. Six performed reviews with GETTY (experimental groups). The other three used the code diff tools available in Github (control groups). We set up pair-programming style reviews to avoid the negative impact of approaches like the Think-Aloud Protocol [25], where researchers may unintentionally influence what participants say and do by asking them about their work [26].

We simulated a real code review environment. All teams performed their reviews in the same quiet lab. We used two Apple 27-inch iMac's: one used for the code review, the other used by the experimental investigator to reply to reviewers' comments through the issue tracker, reset issue states, and perform related tasks. We video recorded all sessions, and interviewed all participants afterwards.

We set up a two-phase review process; once reviewers gave us feedback we would provide the fixes suggested and ask for a second review. We bound the experiment's duration by limiting the number of iterations to two and by preparing a comprehensive set of fixes that would fix all issues reported by previous reviewers, plus all the ones we had found using GETTY.

We chose three issues and commits from the GSON project. All commits passed their original code review process, but were later found to be insufficiently tested or suffered from undiscovered bugs. Issue-#1 [27] was described in Section II.

Issue-#2 [28] implements two new methods, equals and hashCode in class LazilyParsedNumber. These methods are part of the Java Object interface and need to correctly compare the class values. LazilyParsedNumber is complex because it keeps a string representation of numbers. So just comparing strings can give the wrong results for different representations of the same number. In this patch, the developers implemented both methods and tested them. However, most tests are trivial. For example, both equals and hashCode are tested comparing two objects whose string value is "1". Reviewers should find these tests inadequate. In addition, there is a subtle bug. According to the Java specification, developers should ensure that equals and hashCode are consistent; i.e., two LazilyParsedNumber objects where equals returns true must have identical hashCode values. This is not the case in this patch.

Issue-#3 [29] contains patches to JsonParser. Developers updated the application logic of JsonNumber, a helper method used by the parser to return a JsonPrimitive object if the value string can be parsed as a number. To avoid overflow and precision information loss during parsing, they used BigInteger to parse all integers, and BigDecimal to parse decimal numbers. Later, it was decided that this change unintentionally masked overflow errors in getAsBigInteger and getAsInt in class JsonPrimitive.

B. Initial Observations

Overall, the experimental teams liked using GETTY. They acknowledged that inspecting semantic changes is necessary and helpful to code review, and that they would like to incorporate

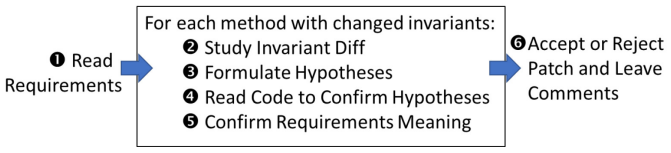

Fig. 12. Review process with GETTY

similar procedures into their daily code review process. They provided useful suggestions for improving GETTY, e.g. linking a method's invariants to the test cases that contributed to its trace.

We also obtained anecdotical evidence that GETTY helps reviewers produce higher quality reviews. For example, all the experimental teams discovered missing tests in the first two issues and suggest good tests. On the other hand, two of the teams that did not have access to GETTY struggled with it. In Issue #1, for example, the reviewers using GETTY found that tests were missing and gave pointed suggestion on how to improve testing. Some of these suggestions are: test for "*big numbers that might overflow*" [E1A], "*negative numbers*" [E1A], examples such as "*1.23*" [E1A] or "*1.99*" [E1A] that cannot be represented as integer values, and "*invalid inputs, e.g. a123*" [E1A]. Reviewers using GETTY noted that only longValue is tested with decimals but that "*BigDecimal is not being tested in intValue()*" [E4E]. On the other hand, only one of the three control teams correctly identified the missing tests. One of the control teams missed the problems completely. They reported "*Functionality looks good*" [C2A]. C3E misinterpreted the code and suggested that the existing test for value 1.0 was wrong and to replace string "1.0" with "1".

We also validated our implicit hypothesis that a good understanding of invariants is required to successfully use GETTY. Our team of novices struggled with issue 3 because they could not decipher what some of the invariants meant. None of the other teams experienced such problems.

C. Effects of Tools on Review Process

The main finding from our study is that GETTY substantially modified the review process of our participants. As exhibited in Section II, GETTY led to a hypothesis-driven process in which reviewers (1) used the invariant diffs to develop an initial understanding of the effects of code changes, and then (2) used GETTY to navigate the code and validate their hypotheses. In simple cases, they sometimes skipped the validation step and just added a comment to their review.

While there are small variations in how the experimental teams performed their reviews, SCR using GETTY followed the general process highlighted in Fig. 12. (1) Reviewers started by looking at the goal of the commit (expressed as an issue in the bug-tracking system or in the commit message). (2) Next, reviewers focused on the methods where invariants had changed. For each such method, they analyzed the invariant diffs presented by GETTY (as described in Section II for Fig. 3), and (3) by looking at these invariants they generated hypotheses on what had been changed (or not). These hypotheses ranged from suspecting missing tests or potential bugs, to clarifications on the meaning of the goal described for the commit (e.g. in Section II E2RA hypothesizes that there was a single test for the new code). (4) Once the reviewer has some working hypothesis for what is going on, he or she read or searched the code to verify the hypothesis. This activity is specifically directed at confirming or disproving the hypothesis, and the code was efficiently navigated using GETTY's navigation features (e.g. the call graph in Fig. 2). (5) Next, the reviewers usually confirmed that requirements were correctly understood by the reviewer considering the code and invariant changes observed. (6) Finally, once all the methods with changed invariants have been studied, the reviewers decided to either accept the patch or suggest changes.

This process contrasts with the control teams. Their primary modality was wide-scale code reading in git-diff, in an effort to understand the behavior of the old and new code. This process consistently induced fatigue, leading to loss of focus, mistakes, and sometimes giving up, when the semantic changes were not evident from the code diffs. In Issue #3, C1LA gave up and assumed that the behavior of getAsBigInteger and getAsInt must be correct as their code was unchanged: "*It's totally plausible to be the getAsBigInteger, getAsInt, currently implement that functionality already, given a number of n. But, I don't know.*"

D. Limitations and Threats to Validity

Limitations. GETTY uses Daikon to extract invariants from code. This means that it inherits some limitation that affect its ability to discover some issues. For example, the second issue in the user study of Section VI contains a subtle bug. This bug arises from the incorrect interplay of the equals and hashCode methods of the LasilyParsedNumber class. Unfortunately, to expose this bug GETTY should provide class invariants using these two query-methods. However, Daikon is not able to detect which methods are queries (do not modify the state) and to use them in invariants. Another limitation arises from the optimizations needed to make GETTY's performance acceptable. Instead of extracting invariants for each method, we compute an impact set as described in Section III.A. While this works well in practice, there could be cases where some important invariant is not computed and presented to reviewers.

Threats to internal validity. We identified multiple factors in our experiments that could potentially affect their validity. 1) Our experiment participants were not familiar with GETTY nor with the projects they reviewed. Yet, the control participants were familiar with their tools, perhaps pessimizing our results. 2) In code review, it's subjective whether one test or another is required, and so our bar for a successful review was subjective on this count. We addressed this by analyzing how GETTY was used instead of fixing an arbitrary bar for successful review.

Threats to external validity. While we endeavored to maximize external validity, there are a few limitations in our experiments. 1) We had a limited number of participants in our study, and although they are all members of population of reviewers, they may not be *representative* of the population. We mitigated this risk by selecting students with distinct experience levels, including several with industry experience. Still our participants may not be representative because they are mostly graduate students and four have a background in programming languages. 2) A second threat is the representativeness of the issues we used in our experiments. We randomly chose problems from 6 open source projects. Still, there may be a bias because we had to choose (a) projects that were relatively well tested and (b) commits that were not too complex to bound the length of the study. 3) Because GETTY is a prototype and does not have manuals or

forums that can be Googled for answers on its use, the investigator first trained the users and then answered questions about the tool and its use during the experiment. This could have influenced the result either positively or negatively. 4) Finally, our study was conducted in a simulated code review environment, with an accelerated timeline (reviewers would get feedback on their reviews immediately). Consequently, in the second review phase the issue was still fresh in the minds of the reviewers.

VII. Related Work

Many researchers have appreciated the value of differencing static semantic information. Lahiri, Vaswani and Hoare from Microsoft Research discuss differential static analysis [30]. 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 [31]. *iProperty* hits the similar idea [4]. SymDiff [5] 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. However, the practical use of differential static 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 [30]. Moreover, static analysis can be overly conservative, limiting the value of its inferences.

Directly related is 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 [32]. Chianti is particularly helpful in isolating changes that lead to a test failure. iDiSE considers dynamic calling context information from inter-procedural analysis to categorizes impact behaviors, and extends notions of test coverage by change impact information [33]. Although Chianti and iDiSE are debugging tools, their underlying technologies could be applied to code review. Our work follows the spirit of Software Change Contracts, which define a formal language to summarize how patches change code behavior. Change Contracts are used to specify the intent of code changes and can be verified at runtime [34]. Preliminary work explored extracting these contracts by tracing the execution of test cases [35]. Instead of specifying expected changes, Getty helps reviewers discover how code changes affect program behavior; we use Daikon to extract generalized likely invariants, provide impact analysis, and support navigation via the call graph.

Holmes and Notkin take a hybrid static-dynamic approach to semantic differencing [12]. 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 in the cross-product of the graphs can reveal anomalies that motivate further inspection. For example, a developer updating a third-party library and expecting the system to behave the same would be surprised to find the control flow has changed at runtime, yet not in the static call graph. Differences in static and dynamic call-graphs are less detailed and more removed from the code than invariant differences. For example, commit #4 discussed in Section IV would have fallen into the "consistent" s^+d^+ partition, because the changes to `equals` produced consonant static and dynamic call graph changes. However, Getty's invariant differentials told the reviewer that the commit actually contains a bug.

Randoop employs invariant differencing to determine when random test case generation can halt, that is when adding tests stops improving the invariants [9]. Their differencing method that supports some understanding of logic is more sophisticated than our current textual approach. Randoop and other approaches (such as Palikareva et al.'s method of performing concolic testing on a unified version of two versions of the code to automatically generate test inputs for new program paths introduced in the code [36]) can be used to automatically generate the tests needed to run Getty.

Gerrit supports distributed code review by providing a staging area for changes where they can be reviewed prior to committing to the repository [37]. Phabricator is a platform integrating many tools, including git-diff based code review [38]. Since our Getty tool generates results as HTML files, they could be integrated into these existing platforms.

VIII. Conclusion

Traditional code review is semantically poor. A summary of *behavioral* differences, can enable the reviewer to see what behaviors have changed since the last commit. The Daikon dynamic invariant extractor can be used to mine behavioral differentials. To isolate effects due to changes in source versus tests, we must run Daikon on different combinations of sources and tests, placing extreme demands on performance. Running Daikon on just the methods of interest reduces run time. Also, partitioning Daikon's inference process on a per-class basis reduces memory pressure and parallelizes well.

Our studies of six open source projects shows that Getty consistently aided in uncovering insufficient testing and finding bugs at their point of introduction. Our case study further shows that Getty changes reviewers' process from reading all the code to generating hypotheses that focus their analysis, ultimately producing better review comments.

Acknowledgment

This research was supported in part by NSF Grant CCF-1719155. We thank Sorin Lerner for his advice on static program analysis, Michael Ernst and his team for their help with Daikon, Philipp Hirch for his contribution to implementing exception handling in Chicory, the participants of our user study, and the anonymous reviewers for their insightful comments.

References

[1] O. Kononenko, O. Baysal, and M. W. Godfrey, "Code Review Quality: How Developers See It," in *Proceedings of the 38th International Conference on Software Engineering*, New York, NY, USA, 2016, pp. 1028–1038.

[2] "Git." [Online]. Available: https://git-scm.com/.

[3] "Git - git-diff Documentation." [Online]. Available: https://git-scm.com/docs/git-diff.

[4] 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.

[5] 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.

[6] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE Trans. Softw. Eng.*, vol. 27, no. 2, pp. 99–123, Feb. 2001.

[7] "google/gson," *GitHub*. [Online]. Available: https://github.com/google/gson. [Accessed: 11-May-2017].

[8] 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.

[9] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-Directed Random Test Generation," in *Proceedings of the 29th International Conference on Software Engineering*, Washington, DC, USA, 2007, pp. 75–84.

[10] G. C. Murphy, W. G. Griswold, M. P. Robillard, J. Hannemann, and W. Leong, "Design recommendations for concern elaboration tools," *Asp.-Oriented Softw. Dev.*, pp. 507–530, 2005.

[11] 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.

[12] R. Holmes and D. Notkin, "Identifying program, test, and environmental changes that affect behaviour," in *2011 33rd International Conference on Software Engineering (ICSE)*, 2011, pp. 371–380.

[13] Y. Yan, M. Menarini, and W. Griswold, "Mining Software Contracts for Software Evolution," in *2014 IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 471–475.

[14] "Apache Maven Project." [Online]. Available: http://maven.apache.org/. [Accessed: 11-May-2017].

[15] "JUnit 4," *GitHub*. [Online]. Available: https://github.com/junit-team/junit4. [Accessed: 11-May-2017].

[16] P. Hirch, "Automatic inference of JML-based security specifications with exception handling," Master Thesis, Universität Bremen, 2016.

[17] "Apache Commons – Apache Commons." [Online]. Available: https://commons.apache.org/. [Accessed: 11-May-2017].

[18] Y. Yan, "Getty Main Repository," *Getty - Semantiful Differentials*. [Online]. Available: https://github.com/mmenarini/semantiful-differentials-getty. [Accessed: 06-Sep-2017].

[19] "Maven - Maven EMMA plugin." [Online]. Available: http://emma.sourceforge.net/maven-emma-plugin/. [Accessed: 11-May-2017].

[20] L. Inozemtseva and R. Holmes, "Coverage is Not Strongly Correlated with Test Suite Effectiveness," in *Proceedings of the 36th International Conference on Software Engineering*, New York, NY, USA, 2014, pp. 435–445.

[21] Debug Team, "Eclipse Debug Project." [Online]. Available: https://www.eclipse.org/eclipse/debug/. [Accessed: 11-May-2017].

[22] "Git - git-blame Documentation." [Online]. Available: https://git-scm.com/docs/git-blame. [Accessed: 11-May-2017].

[23] "[CLI-252] LongOpt falsely detected as ambiguous - ASF JIRA." [Online]. Available: https://issues.apache.org/jira/browse/CLI-252. [Accessed: 11-May-2017].

[24] 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.

[25] A. H. JØRGENSEN, "Thinking-aloud in user interface design: a method promoting cognitive ergonomics," *Ergonomics*, vol. 33, no. 4, pp. 501–507, Apr. 1990.

[26] N. Miyake, "Constructive Interaction and the Iterative Process of Understanding," *Cogn. Sci.*, vol. 10, no. 2, pp. 151–177, Apr. 1986.

[27] "Added test to use BigDecimal to parse number when requesting it as a … · google/gson@e450822," *GitHub*. [Online]. Available: https://github.com/google/gson/commit/e4508227c53749b48318366c1272119031851887. [Accessed: 11-May-2017].

[28] "LazilyParsedNumber does not implement eqals and hashCode methods · Issue #627 · google/gson," *GitHub*. [Online]. Available: https://github.com/google/gson/issues/627. [Accessed: 11-May-2017].

[29] "Parse all JSON numbers as either BigDecimal or BigInteger. From ther… · google/gson@d5319d9," *GitHub*. [Online]. Available: https://github.com/google/gson/commit/d5319d9e840b2c7237ca435f50c50ffbe7dce507. [Accessed: 11-May-2017].

[30] S. K. Lahiri, K. Vaswani, and C. A. R. Hoare, "Differential Static Analysis: Opportunities, Applications, and Challenges," in *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, New York, NY, USA, 2010, pp. 201–204.

[31] 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.

[32] J. Branchaud, S. Person, and N. Rungta, "A Change Impact Analysis to Characterize Evolving Program Behaviors," in *Proceedings of the 2012 IEEE International Conference on Software Maintenance (ICSM)*, Washington, DC, USA, 2012, pp. 109–118.

[33] 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.

[34] D. Qi, J. Yi, and A. Roychoudhury, "Software Change Contracts," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, New York, NY, USA, 2012, p. 22:1–22:4.

[35] T. D. B. Le, J. Yi, D. Lo, F. Thung, and A. Roychoudhury, "Dynamic Inference of Change Contracts," in *2014 IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 451–455.

[36] H. Palikareva, T. Kuchta, and C. Cadar, "Shadow of a Doubt: Testing for Divergences Between Software Versions," in *Proceedings of the 38th International Conference on Software Engineering*, New York, NY, USA, 2016, pp. 1181–1192.

[37] "Gerrit Code Review." [Online]. Available: https://www.gerritcodereview.com/. [Accessed: 16-Nov-2016].

[38] "Phacility - Phabricator." [Online]. Available: https://www.phacility.com/phabricator/. [Accessed: 11-May-2017].