Aiding Code Change Understanding with Semantic Change Impact Analysis

Quinn Hanam
Electrical and Computer Engineering
University of British Columbia
Vancouver, Canada
qhanam@ece.ubc.ca

Ali Mesbah
Electrical and Computer Engineering
University of British Columbia
Vancouver, Canada
amesbah@ece.ubc.ca

Reid Holmes
Computer Science
University of British Columbia
Vancouver, Canada
rtholmes@cs.ubc.ca

***Abstract*—Code reviews are often used as a means for developers to manually examine source code changes to ensure the behavioural effects of a change are well understood. Unfortunately, the behavioural impact of a change can include parts of the system outside of the area syntactically affected by the change. In the context of code reviews this can be problematic, as the impact of a change can extend beyond the diff that is presented to the reviewer. Change impact analysis is a promising technique which could potentially assist developers by helping surface parts of the code not present in the diff but that could be affected by the change. In this work we investigate the utility of change impact analysis as a tool for assisting developers understand the effects of code changes. While we find that traditional techniques may not benefit developers, more precise techniques may reduce time and increase accuracy. Specifically, we propose and study a novel technique which extracts semantic, rather than syntactic, change impact relations from JavaScript commits. We (1) define four novel semantic change impact relations and (2) implement an analysis tool called SEMCIA that interprets structural changes over partial JavaScript programs to extract these relations. In a study of 2,000 commits from the version history of three popular NodeJS applications, SEMCIA reduced false positives by 9–37% and further reduced the size of change impact sets by 19–91% by splitting up unrelated semantic relations, compared to change impact sets computed with Unix diff and control and data dependencies. Additionally, through a user study in which developers performed code review tasks with SEMCIA, we found that reducing false positives and providing stronger semantics had a meaningful impact on their ability to find defects within code change diffs.**

I. INTRODUCTION

Code reviews, a part of modern software engineering practice, require developers to understand how atomic code changes (commonly called commits) affect program behaviour [20]. Research has shown that developers desire tool support for understanding the effects of code changes during code review [7], [33]. Automatically tracing the effects of code changes is the domain of change impact analysis. Broadly defined as the practice of "identifying the potential consequences of a change..." [6], change impact analysis is a technique which provides additional context that can help developers understand the effects of code changes. More specifically, *static* change impact analysis is commonly used to detect potential changes to program behaviour by using program slicing to compute control and data dependencies for statements touched by a change (e.g., Gethers et. al. [14]).

In this work, **we investigate the efficacy of using static change impact analysis to help developers understand the effects of code changes**. First, we investigate whether traditional change impact analysis (i.e., techniques based on data and control dependencies) can help developers understand code changes. We conduct a user study where developers perform simple code review tasks relating to code navigation and bug finding. **We found no evidence that traditional change impact analysis increased code change understanding performance.**

Second, **we theorize that traditional change impact analysis did not increase performance because of various sources of noise in our change impact analysis tool**. Specifically, (1) change impact analysis suffered from high false positive rates due to imprecise syntactic change information and (2) multiple semantic relations were grouped together inside syntactic data-dependency and control-dependency relations, which obscured relevant semantic information.

Third, **we address the problem of noise in static change impact analysis by presenting a novel change impact analysis tool** which computes semantic, rather than syntactic, change impact relations. We introduce novel semantics for four change impact analyses, which show relationships between structural changes and changes to program behaviour. We implement our analyses in tool called SEMCIA, which identifies these semantic relationships for JavaScript. SEMCIA is optimized for ease-of-use by performing a partial (intra-file) analysis which shows local results robustly without needing a complete system, and which could be directly integrated into existing code review tooling. SEMCIA is available online [3]. We found that **using AST diff instead of Unix diff reduced false positives by 29-53%, and that using semantic relations reduced the size of the change impact sets by 20-90%.**

Fourth, we investigate whether our semantics-based change impact analysis implementation can help developers understand code changes. Using the same user study setup as our initial investigation, we provide empirical evidence that reducing false positives and providing stronger semantics can have a meaningful impact on tasks which require understanding the effects of code changes. We found that while traditional static change impact analysis is likely unsuited for helping developers understand code changes, **semantic change impact**

analysis reduced the completion time of targeted tasks by 30-90%.

Our main contributions in this work include the following:

1) A controlled user study which evaluates the utility of traditional change impact analysis techniques on code change understanding tasks.
2) Four novel semantic change impact relations, aimed at supporting developers performing specific code change understanding tasks.
3) A novel semantic change impact analysis tool for JavaScript, which removes sources of noise (i.e., false positives and syntactic relations) from traditional change impact analysis techniques.
4) A controlled user study which evaluates the utility of semantic change impact analysis techniques on code change understanding tasks.

II. Background

When Bacchelli and Bird interviewed code review practitioners at Microsoft, they found that code reviewers struggle to understand the effects of changes. One developer summarized the problem by stating *"...big-picture impact analysis requires contextual understanding. When reviewing a small, unfamiliar change, it is often necessary to read through much more code than that being reviewed"* [7]. More specific information needs related to change impact have also been identified separately by Ko et al. (i.e. *"How have resources I depend on changed?"*) [20] and Tao et al. (i.e. *"How does this change alter the program's... behaviour?"* and *"Who references the changed classes/methods/fields?"*) [33].

These information needs often generalize to code tracing problems, where a developer must trace through code to infer semantic information. However, developers have difficulty inferring semantic information from low levels of abstraction because of failures and limitations of human memory [30], and require tool support to do so accurately and efficiently.

The most common tool for viewing source code changes, the Unix diff utility, displays line-level edit operations [28] that transform one version of source code to another. Because Unix diff only displays information about syntactic changes, using Unix diff to infer the effects of code changes requires developers to manually trace control or data flow beginning with lines that contain syntactic changes. This lack of tool support makes it difficult to understand the effects of code changes during tasks such as code review.

Change impact analysis provides one potential solution to providing support for understanding the effects of code changes. Roughly speaking, change impact analysis is the process of determining what regions of code are impacted by a change. Because of the wide variety of applications for which change impact analysis is used, many different change impact analysis techniques have been developed [14]. A task like code review requires precisely tracking behavioural changes in source code as it is evolved, and we therefore focus on the change impact analysis technique that uses static analysis, or *static change impact analysis*. Other change impact analysis techniques are only loosely related to source code (e.g., by mining bug reports or by tracking meta information about file changes) and do not provide information about changes to runtime behaviour.

Prior approaches to static change impact analysis (e.g., [14], [8], [34], [4], [11]) have almost exclusively used a technique where Unix diff is used to identify a slicing criterion (i.e., variables in modified lines), and data and control dependencies are computed for all variables in the criterion. We begin by conducting a preliminary study to determine whether or not this form of change impact analysis can improve the performance of developers performing code change understanding tasks.

III. Preliminary Study

It is unclear whether or not traditional static change impact analysis can aid developers performing code change understanding tasks, such as code review. To gain insight into this, we performe a user study in which we evaluate change impact analysis as a tool for assisting with code review tasks. Specifically, our goal is to answer the following research question:

RQ1: Are there code review tasks for which traditional change impact analysis (i.e., one that computes control and data dependencies) can improve speed or accuracy over a typical (i.e., Unix diff) diff utility?

For our study, we ask software developers to perform code review tasks with two different tools, which replicate (1) the common functionality available in diff utilities, and (2) a diff utility augmented with traditional change impact analysis information.

A. Tools Under Evaluation

We implemented the following two (web-based) change impact analysis tools for the study:

UnixDiff is modelled after the functionality of the diff utility used by the popular version control host GitHub[1]. This tool shows a Unix diff in *split view*, where the original file and the new file are shown side by side and aligned according to the Unix diff. Deleted lines are highlighted in red and inserted lines are highlighted in green. Two unchanged lines adjacent to inserted or deleted lines are shown for context, while all other unchanged lines are hidden but can be expanded through a context menu.

SynCIA is implemented on top of UnixDiff, and provides the results of a traditional change impact analysis. This change impact analysis displays program slices containing data and control dependencies, where the slicing criterion is everything inside Unix diff. The slices are shown when selected from a context menu. For every line containing a criterion and dependency that is part of the slice, two lines surrounding that line are shown for context while all other lines are hidden. SynCIA also provides basic code navigation by highlighting definitions and uses of selected values.

[1] https://github.com/

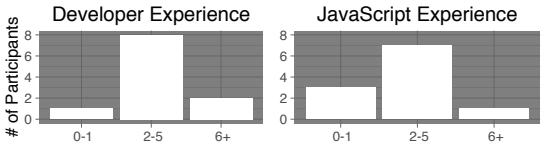

Fig. 1: Experience of user study participants in years.

TABLE I: Subject commits

Subject	GitHub Project	Commit	Lines	Time (s)
Karma	karma-runner/karma	82f1c1	162	<1
PM2	Unitech/pm2	d0cc50	2,389	39
Popcorn	popcorn-official/popcorn-desktop	db90cb	392	<1

B. Subjects

Participants. We recruited 11 participants (eight graduate students and three industrial developers) to participate in our study. Figure 1 shows our participants' experience as developers and with JavaScript. All our participants had experience using at least one diff tool.

Commits. Our study participants were not familiar with the the source code for any of the projects used in the study. To find non-trivial, but not overwhelming, commits we mined 134 Node.js projects and randomly selected 10 commits that met two requirements: (1) exactly one file was modified, and (2) between 10 and 30 lines were labelled as inserted by Unix diff. Among these 10 commits, we then randomly selected three for use in the study, that contained four types of behavioural changes: changes to variables, (function) values, conditions and callsites. Table I shows the details of these three commits.

C. Tasks

We created code review tasks that captured common bug patterns identified by Hanam et. al. [16] or code navigation tasks identified by Murphy et. al. [27]. The goal of the code review tasks was to force the participants to understand the code changes with respect to different semantic domains. The four bugs involved in the code review tasks were:

Scope Conflict Bug. We introduced one scope conflict, where a new variable hid another variable declared at a higher scope, in each of the subject diffs and asked participants to locate it.

Incorrect Condition Bug. We injected one bug into a branch condition in each of the subject diffs and asked the participants to locate it. The expected behaviour of the conditions were either obvious (threw a null dereference exception), or explained by code comments inserted into the diff.

Incorrect Arguments Bug. We introduced one bug into the arguments of one callsite in each of the subject diffs and asked participants to locate it. The bug was either a missing argument or an incorrect argument order, which was obvious from comparing argument values at the callsite to parameter names.

Callsites of Modified Functions. We asked participants to identify all callsites (within the file only) of functions whose behaviour was modified by the change. Participants were not told what functions changed or how many callsites there were. In the Karma and Popcorn diffs, only one callsite called a modified function. In the PM2 diff, six callsites called a modified function, with four of them being callback functions.

TABLE II: User study results. Search times are expressed relative to UNIXDIFF. Negative times mean participants were faster than with UNIXDIFF. Success rates are expressed as the percent of available bugs or callsites found. The best result(s) for a measure is highlighted.

Task	Subject	Mean Search Time (s)		Success Rate (%)	
		UNIXDIFF	SYNCIA	UNIXDIFF	SYNCIA
Scope	Karma	420	-122	0	50
	PM2	360	+60	60	0
	Popcorn	136	+77	100	100
Condition	Karma	147	-39	75	100
	PM2	136	+77	100	100
	Popcorn	90	+111	100	100
Arguments	Karma	290	-24	67	100
	PM2	284	-33	50	67
	Popcorn	146	-48	100	100
Callsites	Karma	272	-54	100	100
	PM2	349	+14	29	42
	Popcorn	296	-141	100	100

Each participant performed six reviews: three tasks on each of the two subject diffs. The diffs were presented in the same order for each task (Karma, PM2, Popcorn). Task order and the diff tool used for each task was randomly selected. Before each of the nine reviews, participants completed a short tutorial. During this tutorial, the participant learned how to use the selected diff tool and performed the task on a small training commit. The tutorial ensured that participants were familiar with both the code pattern they were looking for and how to use the diff tool. For bug identification tasks, participants were instructed to first identify and explain the bug to the researcher conducting the session. If correct, participants indicated the location of the bug on the web page and triggered a timer which logged the time taken to identify the bug. If the participant had not completed a review after seven minutes, they were stopped and the search time was recorded as seven minutes.

A total of 64 reviews were performed by the participants; one participant performed four reviews instead of six. At least two reviews and at most five reviews were performed for each {*diff, tool*} pair. II shows the results of the study. Columns 3–4 show the mean time participants spent searching for bugs or callsites. The mean search times for SYNCIA (column 4) are shown relative to the mean search time of UNIXDIFF. A negative value means that the mean search time was less than for UNIXDIFF. Columns 5–6 show the percent of bugs or callsites that were successfully found during the reviews.

D. Summary of Findings (RQ1)

The use of SYNCIA did not show a statistically significant benefit over Unix diff. This preliminary study suggest that

Listing 1: Old version of the running example.

```
1  function pythagorean(a, b) {
2    var c;
3    if(a == null || b == null)
4      return undefined;
5    a = Math.pow(a, 2);
6    b = Math.pow(b, 2);
7    c = Math.sqrt(a + b);
8    return c;
9  }
10 pythagorean(3, 4);
11 ;
```

Listing 2: New version of the running example.

```
1  function hypLength(a, b) {
2    var c;
3    if(!a || !b)
4      return undefined;
5    a = Math.pow(a, 2);
6    b = Math.pow(b, 2);
7    c = Math.sqrt(a + b);
8    return c;
9  }
10 hypLength(3, 4);
11 hypLength(6, 8);
```

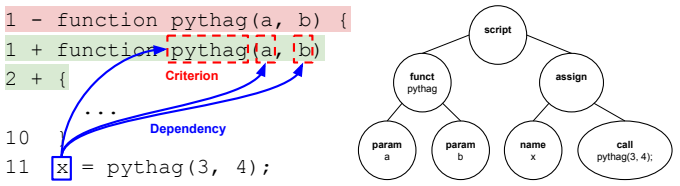

Fig. 2: Left – an example of a false dependency caused by a whitespace change and an imprecise diff utility. A change impact analysis might incorrectly infer that the value of `x` has changed based on the information provided by Unix diff that the definition of `pythag` has changed. Right – the AST diff for the same code (no change).

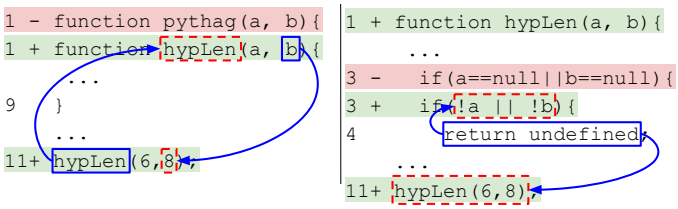

Fig. 3: Syntactic dependencies which contain multiple semantic relationships. Left – data dependencies, which include a variable rename and a new argument value. Right – control dependencies, which include a new callsite and a modified branch condition.

traditional change impact analysis may be unsuited to helping developers understand the effects of code changes.

Next, we investigate reasons why traditional change impact analysis may be unsuited to code comprehension tasks. Specifically, we theorize that the imprecision of these techniques causes noise in analysis results that obscures relevant information and negates any potential benefit.

IV. Noise in CIA

As discussed in Section II, traditional static change impact analysis uses a technique where Unix diff is used to identify a slicing criterion (i.e., variables referenced in modified lines), and data and control dependencies are computed for all variables in the criterion. This approach to static change impact analysis has two major limitations: (1) it suffers from high false positive rates, and (2) it does not provide semantic relationships between code changes and changes to program behaviour. We characterize these limitations as adding noise to sets of semantic change impact relations. The following two sections describe these two sources of noise in greater detail.

1) False Positives in Change Impact Analysis: False positives in change impact analysis are reported differences in behaviour where no differences exist. False positives are pernicious in a code review context as they represent additional information added to a diff that has no value to the developer.

Existing static change impact analysis tools use character-based changes provided by Unix diff to determine the slicing criterion. Such changes select each statement with one or more character changes as a slicing criterion. These include changes which do not modify the Abstract Syntax Tree (AST), such as whitespace changes. It is trivial to prove the semantic equivalence of unchanged parts of the code by parsing each version of the source code into an AST and checking subtree equivalence.

Consider Listings 1 and 2, which show the old (P_{old}) and new (P_{new}) versions of a program which computes the length of the hypotenuse of right angle triangles. We will use this change as a running example.

Figure 2, shows a method rename refactoring, which is part of the change made in our running example. A newline character has been added before the left brace at line 1. If we use a Unix diff tool, the function definition for `pythag` is selected as the criterion. Not knowing which elements of the function declaration was changed, the analysis must assume that `pythag` returns a new value. It therefore records a dependency relation between the value of `x` and the change to `pythag`.

We can eliminate these types of false positives simply by parsing each version of the source code into an AST and computing a set of transformations to the AST rather than the text file. Furthermore, because AST diff is more precise than Unix diff, fewer AST nodes will be included in the list of structural changes, which allows the change impact analysis to ignore larger portions of the program.

2) Syntactic vs. Semantic Relations: Existing static change impact analysis tools use data and control dependencies to compute an over-approximation of locations in code that contain modified states. Because control and data dependencies represent syntactic rather than semantic relations [26], multiple semantic relations end up grouped together. This has a similar effect to that of false positives, in that it makes it more difficult for developers to find the semantic relations they are interested in.

Consider the left hand side of Figure 3, which shows two data dependencies on expressions that were modified in our running example (since JavaScript has first order functions, `hypLen` is a variable that points to a function object). These two data dependencies have different semantics (i.e., they changes the program's state in different ways). One dependency is caused by a variable renaming and does not affect program behaviour, while the other dependency is caused by a new value being passed to the function, which causes a number

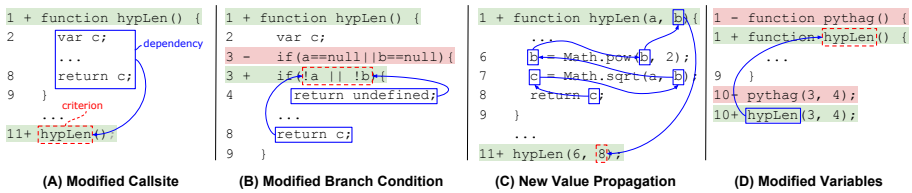

Fig. 4: (A) The execution of `hypLen` and consequently the statement at lines 2-8, are affected by the new callsite at line 11. (B) The execution of the statements at lines 4-8 may be affected by the change to the condition at line 3. (C) The value of `b` inside function `hypLen` is affected by the new integer literal at line 11. (D) The renamed variable `hypLen` at line 1, is used at line 10.

of variables to point to new values in memory.

Similarly, consider the right hand side of Figure 3, which shows two control dependencies on expressions that were modified by a change (not shown) to the program in our running example. These two control dependencies also have different semantics. One change is caused by a call stack being added, which causes new statement executions, while the other is caused by a change to the branch condition, which causes statements to be executed under different conditions.

V. Semantic Change Impact Analysis

Our solution to noise in change impact analysis involves two components. First, using AST diff rather than Unix diff to select the criterion (ie. the syntactic changes that impact behaviour) is a simple change that uses existing technology. Second, developing a semantics-based change impact analysis is a novel solution that requires us to define our desired semantics. In this section, we provide a sketch of these semantics. A more detailed and formal presentation can be found in our tech report [1].

The first step in comparing the behaviour of two programs is to decide which points in the programs should be compared. To make clear what we want to compare, we define a program point as *one or more executions of a single statement which occur in an equivalent context*. Consider again our running example (Listings 1 and 2). The statement at line 8 is executed once for each callsite of the function declared at line 1; once in P_{old} and twice in P_{new}. This yields the following two program points and their states:

Program Point	Statement in P_{old}	Value of `c` in P_{old}	Statement in P_{new}	Value of `c` in P_{new}
8_1	`return c;`	5	`return c;`	5
8_2	-	-	`return c;`	10

Because programs can have large state spaces, it is impractical to keep track of the state after each possible execution. Static analysis tools solve this problem by merging program points that are executed within equivalent contexts. For simplicity, in this work we merge all program points with the same statement. In static analysis terms, this is known as a flow sensitive, context insensitive analysis. In our running example, this strategy yields the following program points and their states at lines 5-8:

TABLE III: Interleaved program points of P_{old} and P_{new}.

PP	Statement in P_{old}	Values in P_{old}	Statement in P_{new}	Values in P_{new}
5	`a = pow(a, 2)`	$a:\{9\}$	`a = pow(a, 2)`	$a:\{9,36\}$
6	`b = pow(b, 2)`	$b:\{16\}$	`b = pow(b, 2)`	$b:\{16,64\}$
7	`c = sqrt(a+b)`	$a:\{9\}$ $b:\{16\}$ $c:\{5\}$	`c = sqrt(a+b)`	$a:\{9,36\}$ $b:\{16,64\}$ $c:\{5,10\}$
8	`return c`	$c:\{5\}$	`return c`	$c:\{5,10\}$

While merging program points makes program analysis tractable, the cost is a loss of precision. For example, while we know that at line 8, the value of `c` is either 5 or 10, we no longer know for which callsite (i.e. line 10 or 11) each value holds.

To compare the behaviour of two programs, we must first decide which program points should be compared. We refer to the process of aligning the program points of two programs as interleaving, where *two program points are interleaved iff they are judged to occur at the same point in time in an execution*.

Table III gives a basic example of an interleaving, where program points are interleaved with each other if they occur at the same line number. Computing a good interleaving is an active research area. The chosen interleaving can affect the precision and soundness of the change impact analysis. Prior approaches for computing an execution interleaving range from doing it manually (e.g., [21]) to the method proposed by Partush and Yahav [31] which uses speculative correlation to approximately minimize the differences between the abstract states in both versions. A third approach, is to interleave statements which are matched by the structural diff utility. This approach is unsound (it does not maintain temporal order) but is automated and fast.

A. Semantic Relations for Static Change Impact Analysis

We propose and implement four semantic relations for static change impact analysis that can provide additional information about the impact of a code change while addressing the problems with false positives identified previously. The four relations we identify are by no means exhaustive, and represent a first step towards identifying meaningful semantic elements that may be relevant to a code evolution task.

The semantic relations we define should provide developers support for navigating the impact of code changes. To identify

potential semantic relations, we take inspiration from popular IDE navigation features in Eclipse, such as those identified by Murphy et. al. [27]. These include *search for reference*, *open declaration*, *highlight variables* and *expand block*. Such features may help developers answer questions about changes. While a full specification of each relation is available in our tech report [1], we give brief examples of each relation here.

Modified Callsites The *open declaration* navigation feature suggests the need to view new or modified function calls and the definitions/bodies of those function calls. An example of such an analysis is shown in Figure 4.A. The execution of `hypLen` and consequently the statements at lines 2–8, are affected by the new callsite at line 11.

Modified Branch Conditions The *expand block* navigation feature suggests the need to identify what branch conditions have changed and what statements are affected by these changes. An example of such an analysis is shown in Figure 4.B. The execution of the statements at lines 4–8 may be affected by the change to the condition at line 3.

New Value Propagation The *search for reference* navigation feature suggest the need to track how new values (including modified functions) are propagated or used throughout the program. An example of such an analysis is shown in Figure 4.C. The value of `b` inside function `hypLen` is affected by the new integer literal at line 11.

Modified Variables The *highlight variables* navigation feature suggests the need to track where new variables are used in the program. An example of such an analysis is shown in Figure 4.D. The variable `hypLen` is renamed at line 1 and used at line 11 is defined at line 1. Even though this relation is unrelated to program behaviour (it is a refactoring), from a slicing perspective it can be specified as a criterion/dependency relationship in the same way as the other relations.

B. Semantic Change Impact Analysis by Interpreting Structural Changes

The approach we selected for our analysis uses abstract interpretation to interpret and track the effects of structural changes. While more complete specification of our analysis is available in our tech report [1], for clarity we provide an example-oriented description here.

Modified Callsites Consider the two versions (P_{old} and P_{new}) of a program in Table IV. There is a structural change at line 8, where a new callsite to `bar` has been added. When the analysis reaches line 8, the AST diff tells it that the call site has been added. When the analysis proceeds to line 5, it pushes a new stack frame, and its modification state (**changed**) onto the abstract call stack. The stack frame is popped when control flows out of `bar`.

Modified Branch Conditions Consider the two versions (P_{old} and P_{new}) of the program in Table V. There is a structural change at line 3, where the branch condition of the `if` statement has changed. When the analysis reaches line 3, the AST diff tells the interpreter that the branch condition has changed. When the analysis proceeds to line 4, it pushes the condition, and its modification state (**changed**) onto the branch condition stack. The condition is popped from the stack when control flows out of the condition's block.

TABLE IV: Example of a our modified callsite analysis.

	Interleaved Statement from P_{old}	Interleaved Statement from P_{new}	Concrete Call Stack P_{old},P_{new}	Abstract Call Stack in P
1	`funct foo(){`	`funct foo() {`		
2	`  return;`	`  return;`	[7],[7]	[7:**unchanged**]
3	`}`	`}`		
4	`funct bar(){`	`funct bar() {`		
5	`  return;`	`  return;`	[],[8]	[8:**changed**]
6	`}`	`}`		
7	`foo();`	`foo();`	[],[]	[]
8	`;`	`bar();`	[],[]	[]

TABLE V: Example of modified branch condition analysis.

	Interleaved Statement from P_{old}	Interleaved Statement from P_{new}	Concrete Condition P_{old},P_{new}	Abstract Condition in P
1	`if(x) {`	`if(x) {`		
2	`  log('a');`	`  log('a';`	$[x]$,$[x]$	[x:**unchanged**]
3	`} else if(y){`	`} else if(z){`		
4	`  log('b');`	`  log('b');`	$[!x, y]$, $[!x, z]$	[$!x$:**unchanged**, y:**changed**]
5	`}`	`}`		
5	`return;`	`return;`	[], []	[]

New Value Propagation Consider the two versions ($P_{\text{old}}$and P_{new}) of the program in Table VI. There is a structural change at line 2, where the integer literal is changed from 0 to 1. When the analysis reaches line 2, the AST diff tells the interpreter that the integer literal has changed and the analysis updates the value of `y` to **changed** in the abstract store. When the analysis reaches line 4, the analysis interprets the result of `x * y` as **changed**, and updates the value of `z` to **changed** in the abstract store.

Modified Variables Consider the two versions ($P_{\text{old}}$and P_{new}) of the program in Table VII. There is a structural change at line 2, where the variable is renamed from `y` to `z`. During variable hoisting, the AST diff tells the interpreter that the variable name has changed. The analysis places `z`, and its modification state (**changed**) into the abstract environment.

TABLE VI: Example of new value propagation analysis.

	Interleaved Statement from P_{old}	Interleaved Statement from P_{new}	Concrete Store P_{old},P_{new}	Abstract Store in P
1	`x = 1;`	`x = 1;`	$x:1, x:1$	$\{x:$ **unchanged**$\}$
2	`y = 0;`	`y = 1;`	$y:0, y:1$	$\{y:$ **changed**$\}$
3	`w = x;`	`w = x;`	$\{w:1,x:1\}$, $\{w:1,x:1\}$	$\{w:$ **unchanged**, $x:$ **unchanged**$\}$
4	`z = x * y;`	`z = x * y;`	$\{x:1,y:0,$ $z:0\}\{x:1,$ $y:1,z:1\}$	$\{x:$ **unchanged**, $y:$ **changed**, $z:$ **changed**$\}$

TABLE VII: Example of modified variable analysis.

	Interleaved Statement from P_{old}	Interleaved Statement from P_{new}	Concrete Environment P_{old},P_{new}	Abstract Environment in P
1	`var x;`	`var x;`	x, x	{x, **unchanged**}
2	`var y;`	`var z;`	y, z	{z, **changed**}
3	`log(x);`	`log(x);`	x, x	{x, **unchanged**}
4	`log(y);`	`log(z);`	y, z	{z, **changed**}

C. Implementation

We implemented our semantic change impact analysis as a tool named SEMCIA, which is implemented on top of the CommitMiner [3] static analysis framework. CommitMiner is an abstract interpreter for JavaScript, similar to the formally specified JSAI [18], but which (1) enables change impact analysis by providing user-specified analyses with information from diff utilities (ie. Unix diff or AST diff), and (2) enables partial program analysis by discovering entry points and recovering type and control flow information (in a similar fashion to Dagenais and Henderson [10]). We configured CommitMiner to perform a flow sensitive, context insensitive analysis.

VI. FALSE POSITIVE STUDY

Section IV described two sources of noise present in change impact analysis. These sources of noise can be measured by the number of false or semantically unrelated dependencies created by a change impact analysis. Recall that in semantic change impact analysis, a criterion is an AST node that has some syntactic change (ie. inserted, removed or updated) applied to it, and a dependency is a program element (e.g., a variable or statement) whose accessible state has changed because of the criterion. By answering the following research questions, we evaluate our technique's ability to reduce noise in change impact analysis:

RQ2: How many false dependencies are eliminated by computing the criterion with AST diff rather than Unix diff?

RQ3: How are control and data dependencies partitioned into our four semantic dependencies?

To answer these, we analyzed the commit histories of three open source Node.js projects: MediacenterJS[2], PM2[3] and Karma[4]. These applications are medium size JavaScript projects selected for their diversity and relatively long commit histories.

SEMCIA successfully analyzed 444 file changes from 299 MediacenterJS commits, 958 file changes from 594 PM2 commits and 1,572 file changes from 1,127 Karma commits. Files were ignored if they were minimized (i.e. library code), or did not have a `.js` extension. Merge commits were ignored, because their changes are already included in the commit history. SEMCIA was not able to parse some files, either because they used non-JavaScript 1.6 syntax or because they had syntax errors. In terms of execution durations, nearly all analyses completed in under one second, with only outliers running for more than one second and no analysis took longer than one minute.

[2]https://github.com/jansmolders86/mediacenterjs

[3]https://github.com/Unitech/pm2

[4]https://github.com/karma-runner/karma

A. RQ2: Unix diff vs AST diff.

We first investigate the affect of using AST diff instead of Unix diff to compute structural changes (i.e. the criterion), since Unix diff has traditionally been used inside static change impact analysis tools. As demonstrated by Falleri et. al. [12], AST diff is substantially more precise than Unix diff when used to compute an AST transformation. We can therefore safely use AST diff as ground truth, since the criterion it creates is almost always a subset of the criterion created by Unix diff.

For this experiment, the structural diff utility (i.e. Unix diff or AST diff) is the independent variable. The number of dependencies created are independent variables. The flow analysis (SEMCIA) is a control variable and behaves the same for both diff utilities.

The first column of Table VIII shows the number of AST nodes in the criterion computed by AST diff. The second column shows the number of dependencies computed by SEMCIA using AST diff. The third column shows the number of AST nodes in the criterion computed by Unix diff. The fourth column shows by what % the size of the criterion set increased. The fourth column shows the number of dependencies computed by SEMCIA using Unix diff. The fifth column shows by what % the number of dependencies increased.

These results show that a large number of false positive dependencies (23–49% of the total annotations) were created when Unix diff was used instead of AST diff. This suggests that change impact analysis utilities can improve their precision significantly by basing their analysis on a criterion computed by AST diff rather than Unix diff.

B. RQ3: Syntactic vs Semantic Relations

We now investigate how SEMCIA partitions the syntactic dependencies created by control and data dependency analysis into separate semantic dependencies. To compute data and control dependencies, we implemented a tool called **SYNCIA**, which is the same as SEMCIA but computes syntactic (data and control) dependencies.

For our experiment, the flow analysis (i.e., SEMCIA or SYNCIA) is the independent variable. The number of dependencies added (to the criterion and dependency sets) is the dependant variable. The structural diff utility (AST diff) is a control variable and behaves the same for both change impact analyses.

We compare the number of dependencies computed by SEMCIA to the number of dependencies computed by SYNCIA. Variable and value dependencies are compared to data dependencies, while call and condition dependencies are compared to control dependencies. Note that while variable dependencies are a subset of data dependencies, value dependencies are not a subset of data dependencies. Data dependency analysis uses all variables and values which are labelled

TABLE VIII: Noise eliminated in the commit histories of MediacenterJS, Karma and PM2. The second column group shows noise caused by using Unix diff as the source of structural change information. The third column group shows noise caused by using syntactic (control and data dependency) relations as a proxy for semantic relations.

Task	SemCIA + AST diff		SemCIA + Unix diff				Data & Control Dependencies				
	Criterion	Deps	Criterion	Increase	Deps	Increase		Criterion	Increase	Deps	Increase
Variable	2,627	5,234	4,699	44%	10,190	49%	Data	29,968	1,041%	16,211	210%
Value	14,414	17,910	26,409	29%	24,863	28%			108%		-9%
Call	6,794	509	9,295	27%	696	27%	Control	8,398	24%	4,318	748%
Condition	1,604	3,809	2,194	27%	4,932	23%			424%		13%

as changed as the slicing criterion, while value dependency analysis uses only values which are labelled as changed as the slicing criterion (semantically, this means *there is a new value in memory*). The condition dependencies and call dependencies are both subsets of control dependencies.

The fourth column group of Table VIII shows the results of this experiment. The largest increase in dependencies occurs for modified call site dependencies, where the number of dependencies is increased by 748% respectively.

This occurs because the number of modified call site dependencies is high relative to the number of branch condition dependencies. For example, if we wanted to look at functions were called because of a change, if we used SynCIA rather than SemCIA, we might expect most of the dependencies we are shown to be caused by changes to branch conditions, rather than changes to call sites.

Regarding the modified value dependencies, the number of dependencies actually decreases by 9% when using SynCIA. This occurs because many of the dependencies that SynCIA considers to be part of the criterion, SemCIA considers to be dependencies. This is caused by the fact that in data dependency analysis, variables and values are considered part of the criterion, whereas in modified value dependency analysis, only values are considered part of the criterion.

C. Summary of Findings (RQ2 and RQ3)

Regarding RQ2, our results show that interpreting Unix diff introduces a significant number of false positives over AST diff. Regarding RQ3, our results show that using data dependencies and control dependencies as a proxy for semantic dependencies introduces a significant amount of noise.

VII. User Study

Ultimately, we believe the information generated by SemCIA about code changes can help developers better understand the implications of a change during code review. While the previous section demonstrated that SemCIA can reduce noise in static change impact analysis, we next need to see if developers could benefit from using SemCIA during code review tasks. Specifically, our goal is to answer the following research question:

RQ4: Are there code review tasks for which SemCIA can improve speed or accuracy over (1) a typical diff utility or (2) control and data dependencies?

TABLE IX: User study results. Search times are expressed relative to UnixDiff. Negative times mean participants were faster than with UnixDiff. Success rates are expressed as the percent of available bugs or callsites found. The best result(s) for a measure is highlighted.

Task	Subject	Mean Search Time (s)			Success Rate (%)		
		UnixDiff	SynCIA	SemCIA	UnixDiff	SynCIA	SemCIA
Scope	Karma	420	-122	-192	0	50	80
	PM2	360	+60	-269	60	0	100
	Popcorn	136	+77	-80	100	100	100
Condition	Karma	147	-39	+1	75	100	100
	PM2	136	+77	-80	100	100	100
	Popcorn	90	+111	+10	100	100	100
Arguments	Karma	290	-24	-221	67	100	100
	PM2	284	-33	-239	50	67	100
	Popcorn	146	-48	-111	100	100	100
Callsites	Karma	272	-54	-80	100	100	100
	PM2	349	+14	-180	29	42	100
	Popcorn	296	-141	-174	100	100	100

To answer this, we extend our user study from Section III to include SemCIA, where each participant was given three additional reviews to complete using SemCIA.

We implemented an additional (web-based) change impact analysis tool: **SemCIA** is implemented on top of UnixDiff, and provides the results of SemCIA change impact analysis. This change impact provides variable, value, call and condition slices as defined in Section V. The slices are shown when selected from a context menu. For every line containing a criterion and dependency that is part of the slice, two lines surrounding that line are shown for context while all other lines are hidden. SemCIA also provides basic code navigation by highlighting definitions and uses of selected values.

Figure 5 shows a screenshot of this tool displaying a slice of modified callsites. The slicing criterion is annotated in red, while dependencies are annotated in blue. Criterion and dependency annotations are highlighted differently depending on the slice and the AST node being annotated. For example, the 'function' keyword of a function declaration is highlighted in the *value* slice, while the entire function (usually spanning multiple lines) is highlighted in the *call* slice.

Each participant performed three additional reviews: three tasks using SemCIA. The conditions of the study are the same as in the preliminary study (and were, in fact, completed at the same time).

A. Results

A total of 32 additional reviews were performed by the participants; one participant performed two additional reviews

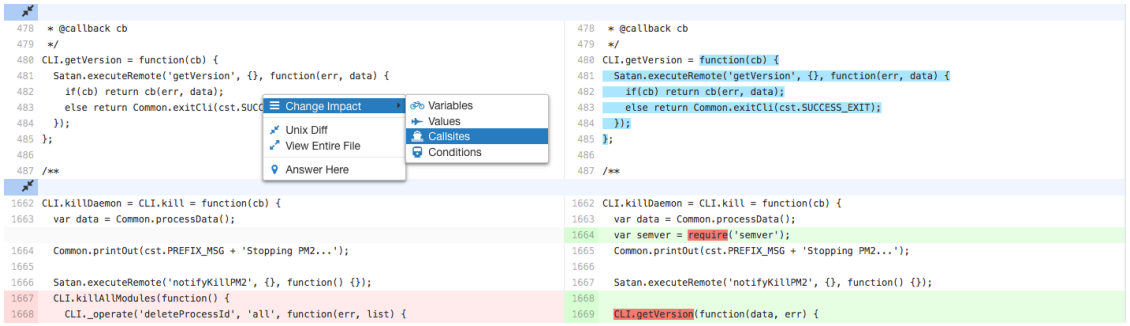

Fig. 5: A screen capture of the SEMCIA tool used in the user study. The file is sliced to show modified callsites. The slicing criterion (modified callsites) is annotated in red; dependencies (functions called by modified callsites) are annotated in blue.

instead of three. IX shows the results of the study. Columns 3–5 show the mean time participants spent searching for bugs or callsites. The mean search times for SYNCIA (column 4) and SEMCIA (column 5) are shown relative to the mean search time of UNIXDIFF. A negative value means that the mean search time was less than for UNIXDIFF. Columns 6–8 show the percent of bugs or callsites that were successfully found during the reviews.

We test statistical significance with a two-tailed, homoscedastic variance t-test. Our null hypothesis is that *there was no difference in task completion time between those who used* SEMCIA *and those who used either* SYNCIA *or* UNIXDIFF.

Scope Conflict Bug. Participants using SEMCIA outperformed UNIXDIFF and SYNCIA in both search time and success rate. Participants using SEMCIA spent 46%–75% less time (statistically significant; with null hypothesis $p = 0.013$) on average searching for scope conflicts. Participants using SEMCIA successfully found the scope conflicts more often for two out of the three files. Using the variable slice provided by SEMCIA, participants immediately identified which variables could be problematic and were able to see all uses of those variables inside the slice without scrolling. This allowed participants to focus on solving the problem rather than navigating the code.

Incorrect Condition Bug. For this bug, there is no clear evidence that participants using either SYNCIA or SEMCIA outperformed UNIXDIFF for search time (null hypothesis $p = 0.378$) or success rate. This may be because there was little code navigation required to find the modified conditions or diagnose the problem unlike the other three tasks. The modified branch conditions and the statements which they controlled were also relatively easy to find by inspecting the line changes provided by UNIXDIFF. Some participants mentioned that the *control* slice provided by SEMCIA made the task more difficult because it reduced the amount context surrounding each modified condition.

Incorrect Arguments Bug. Participants using SEMCIA outperformed UNIXDIFF and SYNCIA in both search time and success rate. Participants using SEMCIA spent 76%–84% less time (statistically significant; with null hypothesis $p = 0.01$) on average searching for incorrect arguments. Participants using SEMCIA successfully found the incorrect arguments more often for two out of the three files. Using the *call* slice provided by SEMCIA, participants immediately identified which callsites could be problematic and were able to see the function declarations of the callees inside the slice. This seems to have allowed the participants to focus on solving the problem rather than navigating the code.

Callsites of Modified Functions. Participants using SEMCIA outperformed UNIXDIFF and SYNCIA in both search time and success rate. Participants using SEMCIA spent 29%–59% less time (statistically significant; with null hypothesis $p = 0.015$) on average searching for callsites of modified functions. Participants using SEMCIA found more callsites for one of the three files. Using the *value* slice provided by SEMCIA, participants immediately identified the functions that modified behaviour and were able to see all callsites of those functions within the slice. The file where success rate was improved over UNIXDIFF was PM2, the longest file with the most (six) callsites to find. Four callsites called functions as callbacks. In these cases the declarations of these modified functions were nested inside callsites, and the variables pointing to these functions at their callsites were aliases of the function. This made it especially difficult for participants to figure out what functions had changed without the aid of the code navigation feature included with SEMCIA and SYNCIA.

B. Summary of Findings (RQ4)

This study suggest that SEMCIA can help developers better understand how code has evolved during code review tasks. Specifically, for code reviewing tasks that require code navigation, reduced noise in change impact slices can help reviewers identify and quickly navigate to relevant parts of the program. Furthermore, the study provides evidence that the reduction in noise that SEMCIA provides makes it easier to understand the effects of code changes.

VIII. Discussion

Threats to validity. Our analysis framework currently handles JavaScript 1.6 syntax. Like most flow analysis frameworks [24], ours has sources of unsoundness. It does not support dynamic code evaluation (eval), reflection, or the JavaScript event loop, and does not model the behaviour of many JavaScript API functions. These limitations typically manifest as dependencies which are missing from the change impact set. Since all of the change impact tools we study use the same analysis framework, the results are internally consistent.

In terms of generalizability, other JavaScript projects may have different commit sizes, file sizes, design patterns and control flows that may affect the accuracy reported in Section VI. Additionally, our approach may work differently for non-JavaScript languages. While the analysis will likely have better soundness and precision on less dynamic languages, other languages may have different commit sizes, file sizes, design patterns and control flows that may change the results for those languages.

Our user study was limited in that we only had 11 participants, eight of which were graduate students. The code review tasks they were performing were also on codebases they were not familiar with, which is not the usual case when one is performing code reviews. That said, the level of familiarity was equivalent with all treatments evaluated in the user study.

Applications. While the primary motivation for this work was to help developers understand the evolution of their systems, our approach can also be used to analyze code changes automatically without requiring fully-buildable systems. Given the improvements our tool as in terms of false positives, and the design decisions we made in favour of performance, we believe the approach could be used to automatically generate large-scale datasets for automated analyses that require broad collections of semantic change information.

Artifacts. We have made the artifacts created during this work openly accessible. The following are available in our downloadable companion [2]: (1) our formal specification of semantic change impact analysis, (2) the datasets from our mining study, and (3) the diff utility used in our user study. The code for SemCIA is publicly available on GitHub [3].

IX. Related Work

Change Impact Analysis. Prior work that uses program slicing for change impact analysis has almost exclusively used syntactic relationships and data and control dependencies [14]. As our empirical study has shown, this type of change impact analysis can yield imprecise results with unclear semantics. While some applications of change impact analysis (e.g., test selection) may tolerate imprecision and noise, human code comprehension [17], [23] tasks are less tolerant of imprecision and noise.

One notable exception is the work by Gyori et. al. [15], which uses symbolic execution to reduce false positives in C/C++ change impact analysis. They use their symbolic equivalence checking tool, SymDiff, to check small sections of modified code for semantic equivalence during dataflow analysis. When two sections are proved semantically equivalent (e.g., x = y and x = y + 0), the dataflow analysis ignores the change. While equivalence checking with symbolic execution subsumes abstract interpretation as a technique for eliminating false positive impacts, relative to abstract interpretation symbolic execution is time consuming. Symbolic execution also requires a symbolic execution engine with adequate language support, which does not yet exist for JavaScript. Finally, Gyori et. al. do not address the problem of specifying or separating different semantic domains, which we do using program slicing.

Data and Control Dependencies. JDiff [5] uses the structure of object oriented programs to compute changes to control flow graphs, which can be used to detect changes to control and data dependencies. Techniques for decomposing unrelated code changes leverage change impact analysis based on data and control dependencies, such as the work by Barnett et al. [8] and the work by Tao and Kim [34]. These techniques may achieve more precise results with the improvements proposed in this work.

Behavioural Equivalence. Equivalence checking is the process of determining whether the output of two pieces of code are always the same given the same input. In the context of the semantics of changes, equivalence checking can either verify that the behaviour of a function does not differ between versions or label the points in the program where values can differ between versions. The SymDiff [21], [22] tool checks for behavioural equivalence between program versions by using a constraint solver to symbolically compute output values and check where output values differ. Because runtimes for single methods range from a few seconds to over one hour, equivalence checking is generally limited to critical code where formal verification is needed.

Higher Level Semantics. Higher level semantics than changes to data and control dependencies, and changes to symbolic values have also been proposed for generating useful information about code changes. Various approaches have been proposed that summarize structural or behavioural code changes as higher level semantics [19], [32], [13], [29], [25], [9].

X. Conclusion

In this paper, we defined four new semantic relations for change impact analysis and implemented a tool called SemCIA that extracts these semantic relations from code changes. SemCIA reduced false positive annotations by 23–49%, and reduced annotations with unrelated semantics by 19–91%. The reductions in noise provided by SemCIA helped developers perform code review tasks more quickly and accurately. Ultimately, we believe that semantic change impact analysis could help developers better understand how their systems are evolving.

REFERENCES

[1] Appendix of "Aiding Code Change Understanding with Semantic Change Impact Analysis". https://ece.ubc.ca/~qhanam/icsme2019/hanam_semcia_tr_2019.pdf, 2019.

[2] Artifacts of "Aiding Code Change Understanding with Semantic Change Impact Analysis". https://ece.ubc.ca/~qhanam/icsme2019/artifacts.tar.gz, 2019.

[3] CommitMiner. https://github.com/qhanam/CommitMiner, 2019.

[4] Mithun Acharya and Brian Robinson. Practical change impact analysis based on static program slicing for industrial software systems. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 746–755, 2011.

[5] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. A differencing algorithm for object-oriented programs. *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 2–13, 2004.

[6] Robert S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, 1996.

[7] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 712–721, 2013.

[8] Mike Barnett, Christian Bird, João Brunet, and Shuvendu K. Lahiri. Helping developers help themselves: Automatic decomposition of code review changesets. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 134–144, 2015.

[9] Raymond P.L. Buse and Westley R. Weimer. Automatically documenting program changes. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 33–42, 2010.

[10] Barthélémy Dagenais and Laurie Hendren. Enabling static analysis for partial java programs. In *Proceedings of the International Conference on Object-oriented Programming Systems Languages and Applications (OOPSLA)*, pages 313–328, 2008.

[11] Nurit Dor, Tal Lev-Ami, Shay Litvak, Mooly Sagiv, and Dror Weiss. Customization change impact analysis for erp professionals via program slicing. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 97–108, 2008.

[12] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 313–324, 2014.

[13] B. Fluri, E. Giger, and H. C. Gall. Discovering patterns of change types. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 463–466, 2008.

[14] Malcom Gethers, Bogdan Dit, Huzefa Kagdi, and Denys Poshyvanyk. Integrated impact analysis for managing software changes. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 430–440, 2012.

[15] Alex Gyori, Shuvendu K. Lahiri, and Nimrod Partush. Refining interprocedural change-impact analysis using equivalence relations. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 318–328, 2017.

[16] Quinn Hanam, Fernando S de M Brito, and Ali Mesbah. Discovering bug patterns in JavaScript. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, pages 144–156, 2016.

[17] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 672–681, 2013.

[18] Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. JSAI: A static analysis platform for JavaScript. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, pages 121–132, 2014.

[19] Miryung Kim and David Notkin. Discovering and representing systematic code changes. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 309–319, 2009.

[20] Andrew J. Ko, Robert DeLine, and Gina Venolia. Information needs in collocated software development teams. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 344–353, 2007.

[21] Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. Symdiff: A language-agnostic semantic diff tool for imperative programs. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, pages 712–717, 2012.

[22] Shuvendu K. Lahiri, Kenneth L. McMillan, Rahul Sharma, and Chris Hawblitzel. Differential assertion checking. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 345–355, 2013.

[23] Chris Lewis, Zhongpeng Lin, Caitlin Sadowski, Xiaoyan Zhu, Rong Ou, and E. James Whitehead Jr. Does bug prediction support human developers? Findings from a Google case study. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 372–381, 2013.

[24] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: A manifesto. *Communications of the ACM*, 58(2):44–46, 2015.

[25] Benjamin Livshits and Thomas Zimmermann. Dynamine: Finding common error patterns by mining software revision histories. In *Proceedings of the Joint International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 296–305, 2005.

[26] Isabella Mastroeni and Damiano Zanardini. Data dependencies and program slicing: From syntax to abstract semantics. In *Proceedings of the Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM)*, pages 125–134, 2008.

[27] Gail C. Murphy, Mik Kersten, and Leah Findlater. How are Java software developers using the Eclipse IDE? *IEEE Software*, 23(4):76–83, July 2006.

[28] Eugene W. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1(1):251–266, 1986.

[29] Stas Negara, Mihai Codoban, Danny Dig, and Ralph E. Johnson. Mining fine-grained code changes to detect unknown change patterns. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 803–813, 2014.

[30] C. Parnin and S. Rugaber. Programmer information needs after memory failure. In *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, pages 123–132, June 2012.

[31] Nimrod Partush and Eran Yahav. Abstract semantic differencing via speculative correlation. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, pages 811–828, 2014.

[32] Kyle Prete, Napol Rachatasumrit, Nikita Sudan, and Miryung Kim. Template-based reconstruction of complex refactorings. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 1–10, 2010.

[33] Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang, and Sunghun Kim. How do software engineers understand code changes?: An exploratory study in industry. In *Proceedings of the International Symposium on the Foundations of Software Engineering (FSE)*, pages 51:1–51:11, 2012.

[34] Yida Tao and Sunghun Kim. Partitioning composite code changes to facilitate code review. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, pages 180–190, 2015.