SensA: Sensitivity Analysis for Quantitative Change-impact Prediction

Haipeng Cai, Siyuan Jiang, Raul Santelices
University of Notre Dame
Notre Dame, Indiana, USA
{hcai|sjiang1|rsanteli}@nd.edu

Ying-jie Zhang
Tsinghua University
Beijing, China
zhangyj1991@gmail.com

Yiji Zhang
University of Notre Dame
Notre Dame, Indiana, USA
yzhang20@nd.edu

***Abstract*—Sensitivity analysis determines how a system responds to stimuli variations, which can benefit important software-engineering tasks such as change-impact analysis. We present SensA, a novel dynamic-analysis technique and tool that combines sensitivity analysis and execution differencing to estimate the dependencies among statements that occur in practice. In addition to identifying dependencies, SensA quantifies them to estimate *how much* or *how likely* a statement depends on another.**

Quantifying dependencies helps developers prioritize and focus their inspection of code relationships. To assess the benefits of quantifying dependencies with SensA, we applied it to various statements across Java subjects to find and prioritize the potential impacts of changing those statements. We found that SensA predicts the actual impacts of changes to those statements more accurately than static and dynamic forward slicing. Our SensA prototype tool is freely available for download.

I. Introduction

For many software-engineering tasks, developers must isolate the *key* relationships that explain the behavior of a program. In particular, when preparing to change a program, developers must first identify the *main* consequences and risks of modifying the program locations they intend to change before properly designing and propagating those changes. Unfortunately, *change-impact analysis* techniques [1] can be quite imprecise, which hinders their applicability and adoption.

Change-impact analyses usually work at coarse granularities, such as methods and classes (e.g., [2]–[5]). They give a first approximation of code dependencies but do not specify which *statements* in particular create those dependencies. Thus, they can classify as impacted many unaffected methods and classes. Moreover, these techniques can miss higher-level structural relationships [6] and thus can be inaccurate.

At the statement level, the forward version of *program slicing* [7], [8] reports all statements that might depend on a statement or a change in it for all executions (via *static* analysis) or for a set of executions (via *dynamic* analysis). Slicing, however, can also be inaccurate. Static slicing can report too many potentially-affected statements that are not truly affected [9] whereas dynamic slicing [10], [11] gives more compact results but can still suffer from imprecision [12], [13] in addition to missing results.

Researchers have tried various approaches to improve the precision of slicing by producing smaller *slices* (i.e., sets of affected statements), such as combining static slices with runtime data [14]–[17], *relevant* slicing [18], [19], and pruning slices based on various rules [20]–[23]. However, accuracy problems persist as many unaffected statements are still reported as affected and truly affected statements are missed. Improving underlying algorithms such as points-to analyses can reduce this inaccuracy but at increasing costs for decreasing payoffs. Yet, very-large impact sets are still produced [24], [25] by existing techniques, which hinder impact-set inspections.

In this paper, we take a different and complementary direction that focuses on the *degree of influence* of statements in a program. Instead of trying to compute smaller sets of affected statements, our approach separates affected statements by *likelihood* (or *strength*) of being really affected so that users can decide which code to inspect first, which can greatly reduce the costs of inspecting the potentially-affected statements. Our approach provides an under-approximation of the *semantic* dependencies [26], an undecidable problem, by finding a subset of statements *definitely* affected by another statement and the *frequency* at which they are affected.

Static slicing, in contrast, only conservatively approximates such dependencies. Moreover, slicing does not identify the frequency (or strength) of dependencies—it only prescribes a traversal order of dependencies [7]. Our new technique and tool, called SensA, uses dynamic analysis instead, although its results *can be compared with static slicing* because it reports unaffected statements from static slices with a score of 0.

SensA goes beyond the work of Masri and Podgurski [12], who measured information-flow strengths within *existing* executions. Our technique combines *sensitivity analysis* [27] and *execution differencing* [13], [28], [29] to estimate those frequencies or likelihoods. Sensitivity analysis is used in many fields to measure influences among different parts of a system. Thus, SensA is conceptually related with mutation analysis [30], fuzzing [31], and code-deletion approaches [32]–[34]. Execution differencing has been used for program merging and debugging [13], [35]. However, to the best of our knowledge, neither approach, alone or in combination, has been used to *measure* dependencies in code and the potential impacts of changing code on the rest of the program. Unlike [12], SensA focuses on candidate change locations and generates multiple variants of existing executions for comprehensive results.

SensA detects and *quantifies* the impacts that the behavior of a statement or potential changes to it may have on the rest of the program. SensA inputs a program P, a test suite (or execution set) T, and a statement c. For each test case t in T, SensA repeatedly executes t, each time changing *the value* *computed at* c and finding the differences in those executions. For each execution, the differences show which statements change their *behavior* (i.e., states or occurrences) when the value computed at c is modified. Using this information for all test cases in T, SENSA computes the *sensitivity* of each statement s to changes in c as the frequency with which s behaves differently. SENSA reports this frequency as the estimated likelihood (or strength) that c impacts s at runtime. The greater the frequency for s is, the more likely it is that s will be impacted by the behavior of c.

Although SENSA assigns weights to *all* statements in a static slice, it can be applied only to statements executed at least once. However, SENSA typically analyzes a large number of executions that result from modifying values at c many times. Also, dynamic analyses in general are quite useful when executions exist already (e.g., [2], [36]) or can be generated. For SENSA, the goal is to answer questions on the existence and frequency of semantic dependencies for a *specific set* of runtime behaviors—those caused by the provided executions and alternative executions derived automatically from them.

To evaluate SENSA, we empirically compared the effectiveness of SENSA and of static and dynamic forward slicing for *predictive impact analysis* on various candidate change locations across Java subjects. For each location, we ranked statements by impact likelihood according to SENSA and by breadth-first search as proposed by Weiser for slicing [7]. Then, we estimated the effort a developer would spend inspecting the rankings to find all statements impacted by actual changes (*bug fixes*) in those locations for the same test suite. Our results indicate with statistical confidence that SENSA outperforms both forms of slicing at predicting actual impacts of such changes by producing more accurate rankings describing the real dependencies on those locations.

The benefit of this work is the greater effectiveness of *actively* quantifying (via sensitivity analysis) dependencies on code and changes with respect to static and dynamic slicing, which do not address all semantic aspects of dependencies and only distinguish them via breadth-first traversals. With this new kind of information, developers can identify influences and impacts more effectively by focusing on the code most likely or most strongly affected in practice. Thus, many dependence-based software-engineering tasks can benefit from this work.

The main contributions of this paper are:

- The concept of sensitivity-analysis-based dependence *quantification* to prioritize statements in slices.
- The SENSA technique that finds these dependencies and estimates their likelihoods via dynamic analysis.
- An empirical study that compares the accuracy of SENSA and slicing for predicting change impacts.

II. BACKGROUND

This section presents core concepts on which the rest of the paper builds and illustrates them using the example program of Figure 1. Program `prog` in this figure takes an integer `n` and a floating point number `s` as inputs, creates a local variable `g`, initializes `g` to the value of `n`, manipulates the value of `s` based on the value of `g`, and returns the value of `s`.

```
    float prog(int n, float s)
    {
1:    int g = n;
2:    if (g ⩾ 1 && g ⩽ 6) {
3:       s = s + (7-g)*5;
4:       if (g == 6)
5:           s = s * 1.1;
      }
      else {
6:       s = 0;
7:       print g, " is invalid";
      }
8:    return s;
    }
```

Fig. 1: Example program used throughout the paper.

A. Syntactic and Semantic Dependencies

Syntactic program dependencies [26] are derived directly from the program's syntax, which are classified as control or data dependencies. A statement s_1 is *control dependent* [37] on a statement s_2 if a branching decision at s_2 determines whether s_1 necessarily executes. In Figure 1, for example, statement 3 is control dependent on statement 2 because the decision taken at 2 determines whether statement 3 executes or not.

A statement s_1 is *data dependent* [38] on a statement s_2 if a variable v defined at s_2 might be used at s_1 and there is a *definition-clear path* from s_2 to s_1 in the program for v (i.e., a path that does not re-define v). For example, in Figure 1, statement 8 is data dependent on statement 3 because 3 defines `s`, 8 uses `s`, and there is a path $\langle 3,4,8 \rangle$ that does not re-define `s` after 3. The parameters of the example `prog` are inputs and thus are not data dependent on any statement.

Semantic dependencies represent the actual behaviors that the program can exhibit, which syntactic dependencies can only over-approximate. Informally, a statement s is *semantically dependent* on a statement t if there is any change that can be made to t that affects the behavior of s. For example, in Figure 1, statement 5 is semantically dependent on statements 1, 2, 3, and 4 because they could be changed so that the execution of 5 changes (e.g., by not executing anymore after the change) or the state at 5 (variable s) changes. In this case, the semantic dependencies of statement 5 coincide with its direct and transitive syntactic dependencies.

However, in this example, if statement 1 just declares that `g` is an alias of `n` (i.e., it is not an executable statement) and only values of `n` in [1..6] are valid inputs, the condition at 2 is always true and, thus, statement 5 is *not* semantically dependent on 2 despite being transitively syntactically dependent on that statement.

More formally, as defined by Podgurski and Clarke [26], a statement s_1 is semantically dependent on a statement s_2 in a program P if and only if:

1) $\exists i \in I$ where I is the input domain of the program,
2) $\exists c \in C$ where C is the set of all possible changes to the values or conditions computed at s_2, and
3) the occurrences or states of s_1 differ when P runs on input i with and without c applied to s_2.

B. Program Slicing

Program slicing [7] determines which statements in a program may affect or be affected by another statement. *Static* slicing [7], [8] identifies such statements for all possible executions whereas *dynamic* slicing [11] does this for a particular execution. (Joining the dynamic results of multiple executions is called *union slicing* [39].) A (static or dynamic) *forward slice* from statement s is the set containing s and all statements directly or transitively affected by s along (static or dynamic) control and data dependencies. Because slicing is based on the transitive closure of syntactic dependencies, it attempts to (over-)approximate the semantic dependencies in the program.

For example, the *static* forward slice from statement 3 in Figure 1 is the set {3,5,8}. We include statement 3 in the slice as it affects itself. Statements 5 and 8, which use `s`, are in the slice because they are data dependent on the definition of `s` at 3. Another example is the *dynamic* forward slice from statement 1 in `prog` for input $\langle n$=0, s=1$\rangle$, which is {1,6,7,8}. In this case, statement 2 uses `g` to decide that statements 6 and 7 execute next (i.e., 6 and 7 are control dependent on 2) and statement 8 is data dependent on 6.

C. Execution Differencing (DEA)

Differential execution analysis (DEA), or simply *execution differencing*, is designed to identify the runtime *semantic dependencies* [26] of statements on changes. Although finding all semantic dependencies in a program is an undecidable problem, DEA techniques [13], [28], [29], [35] can detect such dependencies on changes when they occur *at runtime* to under-approximate (find a subset of) the set of all semantic dependencies in the program.[1] DEA cannot guarantee 100% recall of semantic dependencies, but it achieves 100% precision. This is usually better than what dynamic slicing achieves [12], [13].

DEA executes a program before and after a change to collect and compare the execution histories of both executions [13]. The *execution history* of a program is the sequence of statements executed and, at each statement, the values computed or branching decisions taken. The differences between two execution histories reveal which statements had their *behavior* (i.e., occurrences and values) altered by a change—the conditions for semantic dependence [26].

To illustrate, consider input $\langle n$=2, s=10$\rangle$ for `prog` in Figure 1 and a change in statement 3 to `s=s`. DEA first executes `prog` *before* the change for an execution history of $\langle$1(2), 2(*true*), 3(35), 4(*false*), 8(35)$\rangle$ where each element $e(V)$ indicates that statement e executed and computed the value set V. DEA then runs `prog` *after* the change, obtaining the execution history $\langle$1(2), 2(*true*), 3(10), 4(*false*), 8(10)$\rangle$. Finally, DEA compares the two histories and reports 3 and 8, whose values changed, as the dynamic semantic dependencies on this change in statement 3 for that input.

III. Technique

The goal of SensA is, for a program P and an input set (e.g., test suite) T, to *detect* and *quantify* the effects on P of the *runtime behavior* of a statement C or any *changes* in C. To this end, SensA combines sensitivity analysis [27] and execution differencing [13].

[1]This is subject to all sources of non-determinism being controlled.

In this section, we first give a detailed overview of SensA and we discuss its applications and scope for dependence-based software engineering. Then, we formally present this technique including its process, and the state-modification strategies that SensA currently offers.

A. Overview and Example

Every statement s has a role in a program. This role is needed, for example, when s is being considered for changes and *predictive* impact analysis must be performed for s. Ideally, to find the role of s, one should identify all statements that *semantically* depend on s [26]. Semantic dependence considers *all possible changes* to the computations performed at s for all possible inputs to represent the *effects* of the behavior of s.

Unfortunately, computing semantic dependence is an undecidable problem. For impact analysis, DEA can tell which statements are dynamically impacted by a change. However, *before* developers can design and apply a change to a statement, they first need to know the *possible effects* of changing that statement.

To both *identify* and *quantify* the actual influences (the role) of a statement s in the program for a set of executions, SensA uses sensitivity analysis on s. SensA repeatedly runs the program while *modifying the values computed* at statement s and identifies in detail, using DEA, the impacted statements for each changed value.[2] Then, SensA computes the *frequency* at which each statement is impacted (i.e., the sensitivity of those statements to s). These frequencies serve as estimates, for the executions and modifications considered, of the *likelihood* and (to some extent) the *strength* of the influences of s.

By design, the modifications made by SensA are constrained to *primitive values* and strings (common objects) computed by statements. However, all statements in a program and their states, including heap objects, are considered by execution differencing in SensA. Also, to determine the sensitivity of the program on other types of statements s, such as one containing a method call c to a data structure, the user can identify the statement(s) of interest in that operation that compute the supported values that make up the data structure and apply SensA to that (those) statement(s) instead of c.[3]

We use the example program `prog` in Figure 1 again to illustrate how SensA works for inputs (2,10) and (4,20). Suppose that a developer asks for the effects of line 1 on the rest of `prog`. SensA instruments line 1 to invoke a state modifier and also instruments the rest of the program to collect the execution histories that DEA needs. The developer also configures SensA to modify variable g in line 1 with values in the "valid" range [1..6].

For each input I, SensA first executes `prog` without changes to provide the *baseline* execution history for DEA. Then, SensA re-executes `prog` on I five times—once for each other value of g in range [1..6]. We list the execution histories for this example and test input (2,10) on Table I.

[2]This process follows the definition of semantic dependence in terms of changes to the values computed—rather than the instruction itself—at s [26].

[3]Naturally, SensA can be extended in the future to modify all non-primitive values—a "higher-order" modification. How to make those changes useful and valid remains to be investigated.

TABLE I: Execution histories for `prog` with input (2, 10)

Run	Execution history
baseline	$\langle$1(2), 2(true), 3(35.0), 4(false), 8(35.0)$\rangle$
modified #1	$\langle$1(1), 2(true), 3(40.0), 4(false), 8(40.0)$\rangle$
modified #2	$\langle$1(3), 2(true), 3(30.0), 4(false), 8(30.0)$\rangle$
modified #3	$\langle$1(4), 2(true), 3(25.0), 4(false), 8(25.0)$\rangle$
modified #4	$\langle$1(5), 2(true), 3(20.0), 4(false), 8(20.0)$\rangle$
modified #5	$\langle$1(6), 2(true), 3(15.0), 4(true), 5(16.5), 8(16.5)$\rangle$

The execution histories for the test input (4,20) are similar to these. Finally, SENSA applies DEA to the execution histories of the baseline and modified runs of each test case and computes the *frequency* (sensitivity) (i.e., the fraction of all executions) at which each line was *impacted* by having its state or occurrences changed.

A developer can use these frequencies directly or through a ranking of affected statements by frequency. In our example, the resulting ranking is $\langle\{1, 3, 8\}, \{4, 5\}, \{2, 6, 7\}\rangle$ where lines 1, 3, and 8 are tied at the top because their states (the values of g and/or s) change in all modified runs and, thus, their sensitivity is 1. Lines 4 and 5 come next with sensitivity 0.2 as line 4's state changes in one modified run and 5 executes for one modification on each input (when g changes to 6) whereas 5 is not reached in the baseline executions. Lines 2, 6, and 7 rank at the bottom because 2 never changes its state and 6 and 7 never execute.

In contrast, as defined by Weiser [7], static forward slicing ranks statements by *dependence distance* from line 1. These distances are found by a breadth-first search (BFS) of dependencies from the slicing criterion [7]. Thus, the result for static slicing is the ranking $\langle\{1\}, \{2, 3, 4, 7\}, \{5, 6, 8\}\rangle$. For dynamic slicing, a BFS of the dynamic dependence graph, which is the same for both inputs, yields the ranking $\langle\{1\}, \{2, 3, 4\}, \{8\}\rangle$.

To illustrate the usefulness of these rankings in this example, consider their application to *predictive* change-impact analysis. Suppose that the developer decides to change line 1 to `g = n + 2`. The *actual* set of impacted statements for this change and inputs is $\{1, 3, 4, 5, 8\}$. This is exactly the set of statements placed at the top two levels of the SENSA ranking. In contrast, static slicing predicts statement 2 as the second most-likely impacted statement, but that statement is not really impacted. Static slicing also predicts statement 5 as one of the least impacted, even though this statement is actually impacted after making this concrete change.

Dynamic slicing, perhaps against intuition, performs even worse than static slicing in this example. The ranking for dynamic slicing misses the actually-impacted statement 5 and predicts statement 2, which is not really impacted, as the second most-impacted. Note that, in the context of this paper, a *forward version* of relevant slicing [18] would not perform better either, although in general it may achieve a higher recall, than forward dynamic slicing. In this example, the forward relevant slice is identical to the dynamic slice.

The usefulness of SENSA depends on the executions and modifications chosen as well as application-specific aspects such as the actual change made to the statement analyzed by SENSA. If, for example, the range [0,8] is used instead of [1,6] to modify g in line 1, the sensitivity of statements 4 and 5 will be higher because they will not execute for some of the modifications. (The sensitivity of statement 3 does not change as it is always affected either by state changes or by not executing when g is not in [1,6].) Also, the sensitivity for 2, 6, and 7 will be non-zero because g can now be outside [1,6]. In this particular case, the SENSA ranking does not change but the frequencies change, and the developer's assessment of the possible impacts of line 1 might rely on those quantities.

Program `prog` is a very simple example that contains only eight statements. This program does not require much effort to identify, quantify, and rank potential impacts, regardless of the approach used. In a realistic case, however, the differences in prediction accuracy between SENSA and both forms of slicing can be substantial, as our study of Section IV indicates.

B. Scope and Limitations

As a dynamic analysis, SENSA requires the existence of at least one test case that covers the analyzed statement. However, test suites do not always achieve 100% coverage of their programs. Therefore, SENSA is only applicable to covered statements or when new covering executions can be created.

As with any dynamic analysis, the results of SENSA are also subject to the quality and representativity of the test cases and, in particular, those covering the analyzed statement C. The more behaviors are exercised for C, the more dependencies SENSA can find and quantify, and the more representative those behaviors are of the real usage of C, the more accurate the quantification will be.

To ensure that any difference observed by execution differencing is indeed caused by the values changed by SENSA, the analyzed executions need to be *deterministic*. If a program has a source of non-determinism that gets executed, it has to be "fixed" by providing the always the same values (e.g., times and random numbers). We use this approach in our study by manually determinizing our subjects. How to automatically determinize programs or adapt execution differencing to such cases is a matter of future work.

The quality of SENSA's predictions is also a function of the change strategies for the values computed at C. Intuitively, the more modifications are made to C, the more effects of C are reflected in the results. Therefore, the user will make these strategies modify C and re-execute the program as many times as it is practical according to that user's budget.

It is important to note that, in a *predictive* change-impact analysis scenario, the test suite for the subject will be used as the input set for SENSA. Developers will use this test suite again, perhaps with a few updates, *after* the changes. Thus, the change effects that the developer will experience will be subject to the same or similar runtime conditions as the one exploited by SENSA for predictive change-impact analysis. This is why we use the same test suite for SENSA and for the ground truth later in our study.

C. Formal Presentation

SENSA is a technique that, for a statement C (e.g., a candidate change location) in a program P with test suite T (or, more generally, an input set) computes for each statement s in P a relevance value between 0 and 1. These values are

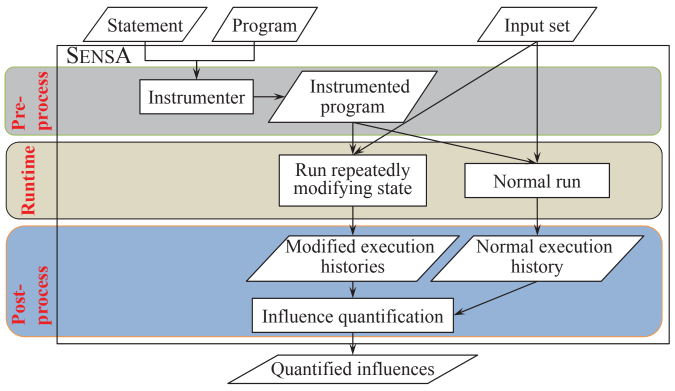

Fig. 2: The SENSA process for impact/slice quantification.

estimates of the frequencies or sizes of the influences of C on each statement of P (or, more precisely, the static forward slice of C). Next, we present SENSA's process and algorithm.

Figure 2 shows the process that SENSA follows to quantify influences in programs. The process logically flows from top to bottom in the diagram and is divided into three stages: (1) *Pre-processing*, (2) *Runtime*, and (3) *Post-processing*.

For the first stage, at the top, the diagram shows that SENSA inputs a *Program* and a *Statement*, which we denote P and C, respectively. In this stage, an *Instrumenter* inserts at C in program P a call to a *Runtime* module (which executes in the second stage) and also instruments P to collect the execution histories of the program for DEA, including the values written to memory [13]. (Branching decisions are implicit in the sequences of statements in execution histories.) The result in the diagram is the *Instrumented program*.

In the second stage, SENSA inputs an *Input set* (test suite) T and runs the program repeatedly modifying the value computed at C (*Run repeatedly modifying state*), for each test case t in T. SENSA also runs the instrumented P without modifications (*Normal run*) as a baseline for comparison. For the modified executions, the runtime module uses a user-specified *strategy* (see Section III-D) and other parameters, such as a value range, to decide which value to use as a replacement at C each time C is reached. Also, for all executions, the DEA instrumentation collects the *Modified execution histories* and the *Normal execution history* per test.

In the last stage, SENSA uses DEA to identify the differences between the *Modified* and *Normal* execution histories and, thus, the statements affected for each test case by each modification made during the *Runtime* stage. In this stage, *Influence quantification* processes the execution history differences and calculates the frequencies as fractions in the range [0,1]. These are the frequencies at which the statements in P were affected by the modifications made at runtime. The result is the set *Quantified influences* of statements influenced by C, including the frequencies that quantify those influences. In addition, SENSA ranks the statements by decreasing frequency.

D. Modification Strategies

SENSA is a generic modifier of program states at given program locations. The technique ensures that each new value picked to modify the original value in a location is *different* to maximize diversity while minimizing bias. When SENSA runs out of possible values for a test case, it stops and moves on to the next test case.

Users can specify parameters such as the number of modifications and modification strategy to use to pick each new value for a statement. For each slicing criterion and strategy, SENSA performs **20 modifications** by default. (We empirically found that, at least for the subjects and changes we studied, more modifications do not increase the cost-effectiveness of SENSA. Also, the choice of values affects the quality of the results of SENSA, so we designed two different (basic) strategies while making it straightforward to add other strategies in the future. The built-in modification strategies are:

1) *Random*: Picks a random value from a specified range. The default range covers all elements of the value's type except for *char*, which only includes readable characters. For some reference types such as *String*, objects with random states are picked. For all other reference types, the strategy currently picks *null*. Instantiating objects with random states is left for future work.
2) *Incremental*: Picks a value that diverges from the original value by increments of i (default is 1.0). For example, for value v, the strategy picks v+i and then picks v–i, v+2i, v–2i, etc. For common non-numeric types, the same idea is used. For example, for string *foo*, the strategy picks *fooo*, *fo*, *foof*, *oo*, etc.

Because some modifications make a program run for a much longer time or run forever, SENSA *skips* those modifications when the running time of the modified program exceeds 10 times the original program. Modifications that cause early terminations do not need special treatment, though, because SENSA can work the same way as with normal executions.

Completeness. Although most heap object values are not *directly* supported at the moment, any supported value within a heap object can be modified by SENSA at the location where that value is computed. Thus, indirectly, SENSA can target any value in memory and track its changes via differencing.

An example of a potential strategy to add is one based on values observed in the past at C that replace the values currently computed at C. First, the strategy would collect all values observed at C for the test suite. Then, the strategy picks iteratively from this pool each new value to replace at C. This strategy would seek values that are more meaningful to the program P because P has computed them at some point.

IV. STUDY: CHANGE-IMPACT PREDICTION

We evaluated SENSA using a specific type of changes, *fault fixes*, which are commonly available for research at the statement level, and the test suites provided with research subjects. We compared the predictions of SENSA with those of *static* and *dynamic* slicing using Weiser's traversal of slices [7].[4] The rationale is that the more closely a technique predicts the impacts that changes will actually have, the more effectively developers will focus their efforts on identifying

[4]Relevant slicing [18], [19] is an option in between for comparison, but a forward version must be developed first. We expect to do this in future work.

TABLE II: Experimental subjects and their characteristics

Subject	Short description	LOC	Tests	Changes
Schedule1	priority scheduler	301	2650	7
NanoXML	XML parser	3521	214	7
XML-security	encryption library	22361	92	7
Ant	project builder	44862	205	7
			Total:	28

and designing the changes that need to be made. Our research questions address effectiveness and efficiency:

RQ1: How accurately does SENSA *predict* the actual impacts that changes will have, compared with slicing?

RQ2: How expensive is it to use SENSA?

A. Usage Scenario

Developers predict change impacts with SENSA as follows:

1) Statement (or set) s is identified as a candidate change location (**the actual change is not yet decided**)
2) SENSA quantifies and ranks the statements in the forward static slice from s by impact likelihood
3) The developer inspects those impacts, helped by the predicted order, and decides what to change and how
4) The developer finally designs and applies changes to s and other impacted statements

B. Experimental Setup

We implemented SENSA in Java as an extension of our dependence analysis and instrumentation system DUA-FORENSICS [40]. As such, SENSA analyzes Java-bytecode programs. The SENSA tool is available to the public for download.[5] DUA-FORENSICS also provides static and dynamic slicing and execution differencing for computing the actual impacts of changes. We run our study on a Linux machine with an 8-core 3.40GHz Intel i7 CPU and 32GB of memory.

We studied four Java subjects of different types and sizes and seven changes (fault fixes) per subject, for a total of 28 changes. Table II describes the subjects. Column *LOC* shows the size of each subject in non-comment non-blank lines of code. Column *Tests* and *Changes* show the number of tests and changes we studied per subject, respectively.

The first subject, Schedule1, is part of the Siemens suite translated from C to Java and is representative of small software modules. NanoXML is a lean XML parser designed for a small memory footprint. XML-security is the signature and encryption component of the Apache project. Ant is a popular build configuration tool for software projects. We *determinized* these subjects where needed.

The faults for all subjects were introduced by other researchers and contributed to the SIR repository [41]. For each fault, the changed program is the *fixed* program, without the fault. Each fault fix modifies, adds, or deletes one to three lines of code. For XML-security, only seven changes are covered by at least one test case (coverage is a minimum requirement for dynamic analysis). Thus, for consistency, we chose the first seven changes provided with the other SIR subjects. For Schedule1, v7 involves two methods so we chose v8 instead. (Future plans include studying SENSA on multiple changes.)

[5]http://nd.edu/~hcai/sensa/html

C. Methodology

Figure 3 shows our *experimental* process for SENSA. The inputs of SENSA are a program, a statement (candidate change location), and the test suite. SENSA quantifies the *runtime influence* of the statement on the rest of the program. The output of SENSA is the set of program statements ranked by decreasing influence. For SENSA, we use the default setting of 20 modifications per slicing criterion and we study each of the two strategies (*Random* and *Incremental*) separately.

For *tied statements* in the ranking, the rank assigned to all of them is the average position in the ranking of those tied statements. To enable a comparison of this ranking with the rankings for *forward* static and dynamic slicing, the SENSA and dynamic-slicing rankings *include at the bottom, tied with score zero, all statements of the static forward slice not found* by each as affected.

To the right of the diagram, an *Actual impact computation* takes the same three inputs and an *actual* change for the given statement. This procedure uses our execution-differencing technique DEA [13] to determine the *exact* set of statements whose behavior differs when running the test suite on the program before and after this change.

The procedure *Ranking comparison* at the bottom of the diagram measures the *predictive effectiveness* of the SENSA ranking for the input statement—when the actual change is not yet known—by comparing this ranking with the set of actually-impacted statements after the change is designed and applied to that statement. The experimental process makes a similar comparison, not shown in the diagram, for the rankings obtained from *breadth-first searches* (BFS) of the dependence graphs for static and dynamic slicing. BFS is the dependence-traversal order of slices defined by Weiser [7]. For dynamic slicing, we join the dynamic slices for all executions that cover the input statement. This is known as a *union slice* [39].

Ranking comparison computes the effectiveness of a ranking at predicting the actually-impacted statements by determining how high in the ranking those statements are on average. The rank of each impacted statement represents the effort a developer would invest to find it when traversing the ranking from the top. The more impacted statements are located near the top of the ranking, the more effective is the ranking at predicting the actual impacts that will occur in practice after making the change. We interpret the average rank of the impacted statements as the average *inspection effort*.

Example. Consider again the example discussed in Section III-A. For SENSA, lines 1, 3, and 8 are tied at the top of the ranking with average rank (1+2+3)/3 = 2 and lines 4 and 5 are tied at average rank 4.5. SENSA does not detect differences in lines 2, 6, and 7, which get tied at the bottom with rank 7. If a change is actually made that impacts lines 1–4 and 8, the *average inspection effort* for SENSA is the average rank of those lines divided by the static slice size, or (2+2+2+4.5+7)/(5×8) = 43.75%. For static and dynamic slicing, the average efforts are computed similarly.

Ideal case. As a basis for comparison of the predictions of the techniques we studied (SENSA and slicing), we computed the inspection cost for the *ideal* (best) scenario for each change. This ideal case corresponds to a ranking in which all statements

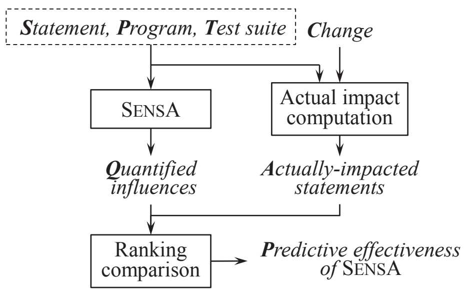
Statement, Program, Test suite
Change
SensA
Actual impact computation
Quantified influences
Actually-impacted statements
Ranking comparison
Predictive effectiveness of SensA

Fig. 3: Experimental process for *predictive* impact analysis, where the location is known but **not the actual change**, and ground truth (actual impacts) to evaluate those predictions.

impacted by the change are placed at the top of that ranking. No other ranking can make a better prediction.

For RQ1, we computed the average inspection costs for the entire rankings for SensA, static slicing, and dynamic slicing. For RQ2, we measured the execution time of the test suite on the program and the time each phase of SensA takes on that program with that test suite.

Test suite choice. By design, the test suites we use to compute the SensA and dynamic-slicing predictions *before* making changes are the same we used to find the actual impacts (the ground truth) *after* making the changes. To the casual reader, using the same test suite to obtain both the prediction and ground truth might seem biased. However, we chose to do so because developers will normally use the entire test suite of the program for SensA *before* they decide their changes and then the same (occasionally updated) test suite to observe the actual impacts *after* defining and applying those changes. *Therefore, using the same test suite before and after is not only appropriate, but necessary for evaluation.*

D. Results and Analysis

1) RQ1: Effectiveness: Table III presents the average inspection efforts per subject (seven changes each) and the average effort and standard deviation for all 28 changes. The units are percentages of the static slice sizes, which comprised up to 63% of the entire subject, on average for all changes in that subject. For the *Ideal* case, the effort is an absolute value representing the minimum possible effort—the best possible prediction. For each of *Static slicing*, *Dynamic slicing*, and SensA with *Random* (SensA-*Rand*) and *Incremental* (SensA-*Inc*) strategies, the table shows the average effort required to find *all* actual impacts in their respective rankings, as explained in Section IV-C.

For example, for XML-security, the best position on average in the ranking for all statements impacted by the changes is 5% of the static forward slice. On top of this, static and dynamic slicing add 26.94% and 40.37% average inspection effort, respectively, for a total of 31.94% and 45.37%. These extra efforts can be seen as the *imprecision* of those techniques. In contrast, SensA-*Rand* and SensA-*Inc* impose 8.15% and 16.49% extra effort, respectively, over the ideal case. This is considerably less than both forms of slicing.

Analysis. The *Ideal case* results indicate that the number of statements impacted in practice by the changes in our study, as a percentage of the total slice size, decreased with the size of the subject from 47.90% in Schedule1 down to 5% or less in the largest subjects. This phenomenon can be explained by two factors. First, the larger subjects contain many loosely coupled modules and the changes, which are thus more scattered, can only affect smaller fractions of the program. The second factor is that static slicing will find connections among those modules that are rarely, if ever, exercised at runtime. For Schedule1, however, slicing is more precise and most of the program executes and is affected by the changes. In fact, an average of 97.8% of the statements in the slices for Schedule1 were impacted. These factors explain the large inspection efforts needed by all techniques for this subject.

TABLE III: Average inspection efforts

Subject	Ideal (best) case	Average effort			
		Static slicing	Dynamic slicing	SensA Rand	SensA Inc
Schedule1	47.90	50.14	48.34	48.01	48.01
NanoXML	8.84	22.71	27.09	20.27	22.37
XML-security	5.00	31.94	45.37	13.15	21.49
Ant	3.21	39.16	41.55	29.84	23.76
average	**16.24**	**35.99**	**40.59**	**27.82**	**28.91**
standard dev.	19.36	13.22	13.08	19.20	20.59
p-value w.r.t. static slicing:				**0.0098**	**0.0237**

Remarkably, for all subjects but Schedule1, dynamic slicing produced worse predictions than static slicing. This counterintuitive result is explained by the divergence of paths taken by executions before and after the changes. Because of these divergences, dynamic slicing missed many impacted statements that were not dynamically dependent on the slicing criterion before the change but became dynamically dependent after the change. SensA did not suffer so much from this problem because its modifications altered some paths to approximate the effects that any change could have.

For Schedule1, the inspection effort for both versions of SensA was, on average, 0.11% more than the ideal, which is very close to the best possible result. For this subject, dynamic slicing also did well at predicting change impacts with only 0.44% extra effort, whereas static slicing was the worst predictor with 2.24% extra effort. Considering the high minimum effort for this small subject, however, these differences in effort are rather small in absolute terms.

For NanoXML, the ideal average effort is much lower at 8.84%, thus impact prediction much more effective there. For this subject, SensA-*Rand* was the best variant of SensA, requiring 2.44% less effort on average than static slicing, while SensA-*Inc* was only slightly better than static slicing.

For XML-security, in contrast, both variants of SensA—especially SensA-*Rand*—were considerably more effective than static slicing, which required more than six times the ideal effort to isolate all impacts, whereas SensA-*Rand* required less than three times the least possible effort.

For Ant, SensA-*Inc* was the best variant with a considerable decrease of 15.4% in effort compared to static slicing.

Importantly, on average for all 28 changes, both versions of SensA outperformed static slicing, with SensA-*Rand* requiring 8.17% less effort than static slicing and 12.77% less than dynamic slicing to capture the actual impacts. The standard deviation for both variants of SensA, however, is greater than for both forms of slicing, which suggests that they are less predictable than slicing.

Statistical significance. To assess how conclusive is the advantage of both variants of SENSA over static slicing—the best slicing type—we applied to our 28 data points a Wilcoxon signed-rank one-tailed test [42] which makes no assumptions about the distribution of the data. Both p-values, listed in the last row of Table III, are less than .05, indicating that the superiority of both types of SENSA—especially SENSA-*Rand*—is statistically significant.

Conclusion. For these subjects and changes, with statistical significance, SENSA is more effective on average than slicing techniques at *predicting* the statements that are, later, truly impacted when changes are made. These results highlight the imprecision of static and dynamic slicing for predicting impacts, contrary to expectations, and the need for a technique like SENSA to better find and quantify dependencies. In all, SENSA can save developers a substantial amount of effort for identifying the potential consequences of changes.

2) RQ2: Computational Costs: To study the cost factor, we collected the time it took SENSA on the 28 changes in our experimental environment (see Section IV-B) using the respective test suites. Table IV first shows the average time in seconds it takes to run the *entire* test suite for each subject without instrumentation (column *Normal run*). The next three columns report the average time in seconds taken by each of the three stages of SENSA per subject.

Analysis. First, the pre-processing stage (column *Static analysis*) performs static slicing, which is needed by our experiment and is also necessary to instrument the program for SENSA, dynamic slicing, and DEA for actual impacts. As expected, this time grows with the size and complexity of the subject, where the two largest subjects (22-45K LOC) dominate. The average costs per subject were less than 16 minutes, which seems acceptable for an unoptimized prototype.

The runtime stage (column *Instrumented run*) of SENSA repeatedly executes 20 times (the default) those test cases that cover the candidate change location. In contrast with the first stage, the cost of the runtime stage is proportional to the number of test cases that cover those locations. The number of test cases available for our subjects is inversely proportional to the subject size (Table II), which explains the cost distribution seen in the table.

The average costs for *Instrumented run* range from 6 to 79 minutes, which might or might not be acceptable for a developer depending on the circumstances. However, it is crucial to note that this stage can be significantly optimized by running multiple modifications and multiple test cases in parallel, which our prototype does not yet support but can be easily added. Another important reduction in this cost can be achieved by using fewer test cases.

Finally, the costs of the third stage for all subjects except Schedule1 (for which a disproportionate total of 2650 test cases exist) are quite small as this stage simply reads the runtime data, computes frequencies, and ranks the semantically-dependent statements.

As for slicing, static slicing took times similar to the static analysis column in Table IV. Dynamic slicing, not shown here, had much greater costs than SENSA because our current implementation derives such slices from more costly information [13]. It remains to be seen how an optimized dynamic slicer compares with SENSA cost-wise.

TABLE IV: Average computational costs in seconds

Subject name	Normal run	Static analysis	Instrumented run	Influence ranking
Schedule1	186.0	6.1	4756.8	1054.1
NanoXML	15.4	16.5	773.1	10.0
XML-security	65.2	179.3	343.9	20.8
Ant	75.2	942.9	439.0	7.0

Conclusion. The observed costs are encouraging for SENSA for three main reasons. First, developers using our unoptimized prototype can in many cases accept to get the impact predictions of SENSA if those are provided within the time budgets they can afford. Second, the cost-benefit ratio of SENSA can be even smaller for inspecting larger impact sets, and the overhead can be more acceptable when the impact sets are too large to be fully inspected. Third, our prototype is *highly parallelizable*. It can be significantly optimized by parallelizing the large number of runs made by SENSA.

E. Threats to Validity

The main *internal* threat to the validity of our study is the potential presence of implementation errors in SENSA for sensitivity analysis and the underlying DUA-FORENSICS [40] for execution differencing and slicing. Although SENSA is a research prototype developed for this work, we have tested and used it for more than one year already. Meanwhile, DUA-FORENSICS has been in development for many years and has matured considerably. Another internal threat is the possibility of procedural errors in our use of SENSA, DUA-FORENSICS, and related scripts in our experimental process. To reduce this risk, we tested, manually checked, and debugged the results of each phase of this process.

The main *external* threat to the validity of our study and conclusions about SENSA is that our set of subjects, changes, and test suites might not represent the effects of similar changes (e.g., bug fixes) in other software. Nevertheless, we chose our subjects to represent a variety of sizes, coding styles, and functionality to achieve as much representativity as possible. The SIR subjects have been used extensively in other experiments conducted by the authors and by many other researchers. Moreover, all subjects but Schedule1 are "real-world" open-source programs.

In particular, our results might not generalize to *all kinds* of changes in software. We only studied changes that represent *bug fixes* and small corrections as a first demonstration of SENSA. These are changes commonly available for experimentation [41]. Also, SENSA is currently applicable to one or a *few* statements at a time but small changes are hard to find in other sources, such as code repositories. Larger changes in those repositories, however, could be broken down into smaller pieces, or SENSA could be adapted for larger changes. We intend to explore this more broadly in future work. In all, this study highlights one of the many potential applications of SENSA—impact analysis for bug fixes.

A *construct* threat can be our choice of *ground truth* (actual impacts of changes) and the method to compute it. We used execution differencing (DEA) to find, for a test suite, which statements behave differently in response to changes in another statement. We also *determinized* our subjects where needed but without affecting their semantics. DEA, like SENSA, works at the *statement* level, unlike repository-mining methods which are coarser and possibly noisier. Also, the actual impacts found via DEA are a subset of *all impacts* a change can have, so we chose subjects for which reasonable test suites exist. Moreover, and importantly, we used the same test suites for SENSA so that its predictions apply to the same runtime profile.

Finally, a *conclusion* threat is the appropriateness of our statistical analysis and our data points. To minimize this threat, we used a well-known test that makes *no assumptions* about the distribution of the data [42]. Another issue could be the use of an equal number of changes per subject. For each of the larger SIR subjects, however, seven was the largest number of faults we could use and we deemed inadequate to study less than seven changes in smaller subjects. Moreover, and importantly, SENSA *outperformed slicing by greater margins for the larger subjects*.

V. RELATED WORK

We previously outlined an early version of SENSA [43] and we showed initial, promising results. In this paper, we expanded our presentation and definition of SENSA, its process, algorithm, and modification strategies. Moreover, we extended our experiments from two to four subjects and included dynamic slicing in our study.

Masri and Podgurski [12] measured information-flow strengths within *existing* executions by correlating the values of variables in those executions. SENSA, instead, directly implements an under-approximation of the definition of semantic dependence by Podgurski and Clarke [26] and analyzes many more executions by systematically modifying existing ones.

A few other techniques discriminate among statements *within slices*. Two of them [23], [44] work on dynamic backward slices to estimate influences on outputs, but do not consider impact influences on the entire program. These techniques could be compared with SENSA if a backward variant of SENSA is developed in the future. Also for backward analysis, thin slicing [22] distinguishes statements in slices by pruning control dependencies and pointer-based data dependencies incrementally as requested by the user. Our technique, instead, can be used to automatically estimate the influence of statements in a static slice in a safe way, *without dropping* any of them, to help users prioritize their inspections.

Program slicing was introduced as a backward analysis for program comprehension and debugging [7]. Static forward slicing [8] was then proposed for identifying the statements affected by other statements, which can be used for change-impact analysis [1]. Unfortunately, static slices are often too big to be useful. Our work alleviates this problem by recognizing that not all statements are equally relevant in a slice and that a dynamic analysis can estimate their relevance to improve the effectiveness of the forward slice. Other forms of slicing have been proposed, such as dynamic slicing [11], union slicing [39], relevant slicing [18], [19], deletion-based slicing approaches [32]–[34], and the already mentioned thin slicing [22], all of which produce smaller backward slices but can miss important statements for many applications. Our technique, in contrast, is designed for forward analysis and does not trim statements from slices but scores them instead.

Dynamic impact analysis techniques [2], [5], [45], which collect execution information to assess the impact of changes, have also been investigated. These techniques, however, work at a coarse granularity level (e.g., methods) and their results are subject strictly to the available executions. Our technique, in contrast, works at the statement level and analyzes the available executions and, in addition, multiple variants of those executions to predict the impacts of changes. Also, our technique is *predictive*, unlike others that are only *descriptive* [5], [13] (i.e., using knowledge of changes *after* they are made).

Mutation testing is a specific form of sensitivity analysis that simulates common programming errors across the entire program [30]. Its purpose is to assess the ability of a test suite to detect errors by producing different outputs. This approach is related to testability-analysis approaches, such as PIE [46], which determine the proneness of code to propagate any errors to the output so they can be detected. A related testing approach is fuzzing [31]. Similar to these approaches, SENSA modifies program points to affect executions but it focuses on points of interest to the user (e.g., candidate change locations) and analyzes not only the influences on outputs but also the influences on all statements.

Many fault-localization approaches (e.g., [47]), although not directly related to SENSA, share a common aspect with our work: they assess their effectiveness in terms of the inspection effort for finding certain targets in the program. For fault localization, those targets are faults, whereas in our work they are the influences of a statement. This effort is often measured as the percentage of the program that must be inspected to reach those targets.

VI. CONCLUSION AND FUTURE WORK

Program slicing is a popular but imprecise analysis technique with a variety of applications. To address this imprecision, we presented a new technique called SENSA for finding and quantifying real dependencies within static slices. Our studies suggest that this approach outperforms static and dynamic forward slicing for tasks such as impact analysis. Rather than pruning statements from slices, SENSA finds and grades statements according to their dependence relevance.

We plan to expand our studies to subjects and changes of other types and sizes to generalize our results and characterize the conditions for the effectiveness of SENSA. We are also developing a visualization for quantified dependencies to improve our understanding of the approach, to enable user studies, and to support other researchers. Using this tool, we will study how developers take advantage in practice of quantified slices.

Slightly farther in the future, we foresee adapting SENSA to quantify dependencies for other key tasks, such as debugging, comprehension, mutation analysis, interaction testing, and information-flow measurement. More generally, we see SENSA's scores as abstractions of program states as well as interactions among such states. These scores can be expanded to multi-dimensional values or reduced to discrete sets, depending on cost-effectiveness needs.

Acknowledgment

This work was partially supported by ONR Award N000141410037 to the University of Notre Dame.

References

[1] S. A. Bohner and R. S. Arnold, *An introduction to software change impact analysis*. Software Change Impact Analysis, IEEE Comp. Soc. Press, pp. 1–26, Jun. 1996.

[2] T. Apiwattanapong, A. Orso, and M. J. Harrold, "Efficient and precise dynamic impact analysis using execute-after sequences," in *Proc. of Int'l Conf. on Softw. Eng.*, May 2005, pp. 432–441.

[3] M. Gethers, B. Dit, H. Kagdi, and D. Poshyvanyk, "Integrated Impact Analysis For Managing Software Changes," in *Proc. of Int'l Conf. on Softw. Eng.*, May 2012, pp. 430–440.

[4] M. Petrenko and V. Rajlich, "Variable Granularity for Improving Precision of Impact Analysis," in *Proc. of IEEE Int'l Conference on Program Comprehension*, May 2009, pp. 10–19.

[5] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley, "Chianti: a tool for change impact analysis of java programs," in *Proc. of ACM Conf. on Obj. Oriented Prog. Syst., Lang., and Appl.*, Oct. 2004, pp. 432–448.

[6] R. Vanciu and V. Rajlich, "Hidden dependencies in software systems," in *IEEE Int'l Conference on Software Maintenance*, 2010, pp. 1–10.

[7] M. Weiser, "Program slicing," *IEEE Trans. on Softw. Eng.*, 10(4):352–357, Jul. 1984.

[8] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Trans. on Prog. Lang. and Systems*, 12(1):26-60, Jan. 1990.

[9] D. Binkley, N. Gold, and M. Harman, "An empirical study of static program slice size," *ACM TOSEM*, vol. 16, no. 2, 2007.

[10] D. Binkley, S. Danicic, T. Gyimóthy, M. Harman, A. Kiss, and B. Korel, "Theoretical foundations of dynamic program slicing," *Theor. Comp. Sci.*, vol. 360, no. 1, pp. 23–41, 2006.

[11] B. Korel and J. Laski, "Dynamic program slicing," *Inf. Process. Lett.*, vol. 29, no. 3, pp. 155–163, 1988.

[12] W. Masri and A. Podgurski, "Measuring the strength of information flows in programs," *ACM Trans. Softw. Eng. Methodol.*, vol. 19, no. 2, pp. 1–33, 2009.

[13] R. Santelices, M. J. Harrold, and A. Orso, "Precisely detecting runtime change interactions for evolving software," in *Proceedings of IEEE International Conference on Software Testing, Verification and Validation*, Apr. 2010, pp. 429–438.

[14] R. Gupta and M. L. Soffa, "Hybrid Slicing: An Approach for Refining Static Slices Using Dynamic Information," in *Proc. of ACM Int'l Symp. on Foundations of Softw. Eng.*, Oct. 1995, pp. 29–40.

[15] S. Horwitz, B. Liblit, and M. Polishchuk, "Better Debugging via Output Tracing and Callstack-sensitive Slicing," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 7–19, 2010.

[16] J. Krinke, "Effects of context on program slicing," *J. Syst. Softw.*, vol. 79, no. 9, pp. 1249–1260, 2006.

[17] M. Mock, D. C. Atkinson, C. Chambers, and S. J. Eggers, "Program Slicing with Dynamic Points-to Sets," *IEEE Transactions on Software Engineering*, vol. 31, no. 8, pp. 657–678, 2005.

[18] H. Agrawal, J. R. Horgan, E. W. Krauser, and S. London, "Incremental Regression Testing," in *Proceedings of IEEE Conference on Software Maintenance*, Sep. 1993, pp. 348–357.

[19] T. Gyimóthy, A. Beszédes, and I. Forgács, "An Efficient Relevant Slicing Method for Debugging," in *Proceedings of joint European Software Engineering Conference and ACM International Symposium on the Foundations of Software Engineering*, Sep. 1999, pp. 303–321.

[20] M. Acharya and B. Robinson, "Practical Change Impact Analysis Based on Static Program Slicing for Industrial Software Systems," in *Proc. of Int'l Conf. on Softw. Eng.*, May 2011, pp. 746–765.

[21] G. Canfora, A. Cimitile, and A. D. Lucia, "Conditioned program slicing," *Information and Software Technology*, vol. 40, no. 11-12, pp. 595–608, Nov. 1998.

[22] M. Sridharan, S. J. Fink, and R. Bodik, "Thin slicing," in *Proceedings of ACM Conference on Programming Language Design and Implementation*, Jun. 2007, pp. 112–122.

[23] X. Zhang, N. Gupta, and R. Gupta, "Pruning dynamic slices with confidence," in *Proceedings of ACM Conference on Programming Language Design and Implementation*, 2006, pp. 169–180.

[24] J. Jasz, L. Schrettner, A. Beszedes, C. Osztrogonac, and T. Gyimothy, "Impact Analysis Using Static Execute After in WebKit," in *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, Mar. 2012, pp. 95–104.

[25] H. Cai, R. Santelices, and T. Xu, "Estimating the Accuracy of Dynamic Change-Impact Analysis using Sensitivity Analysis," in *Int'l Conf. on Softw. Security and Reliability*, 2014, pp. 48–57, IEEE Rel. Society.

[26] A. Podgurski and L. Clarke, "A formal model of program dependences and its implications for software testing, debugging, and maintenance," *IEEE Trans. on Softw. Eng.*, vol. 16, no. 9, pp. 965–979, 1990.

[27] A. Saltelli, K. Chan, and E. M. Scott, *Sentitivity Analysis*. John Wiley & Sons, Mar. 2009.

[28] K. J. Hoffman, P. Eugster, and S. Jagannathan, "Semantics-aware Trace Analysis," in *Proceedings of ACM Conference on Programming Language Design and Implementation*, Jun. 2009, pp. 453–464.

[29] M. K. Ramanathan, A. Grama, and S. Jagannathan, "Sieve: A Tool for Automatically Detecting Variations Across Program Versions," in *Proc. of Int'l Conf. on Automated Softw. Eng.*, 2006, pp. 241–252.

[30] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," vol. 37, no. 5, pp. 649–678, 2011.

[31] P. Oehlert, "Violating Assumptions with Fuzzing," *IEEE Security and Privacy*, vol. 3, no. 2, pp. 58–62, 2005.

[32] D. Binkley, N. Gold, M. Harman, J. Krinke, and S. Yoo, "Observation-Based Slicing," RN/13/13, UCL, Dept. of Comp. Sci., Jun. 2013.

[33] H. Cleve and A. Zeller, "Finding Failure Causes through Automated Testing," in *AADEBUG*, Aug. 2000.

[34] R. A. DeMillo, H. Pan, and E. H. Spafford, "Critical Slicing for Software Fault Localization," in *Proceedings of ACM International Symposium on Software Testing and Analysis*, Jan. 1996, pp. 121–134.

[35] W. N. Sumner and X. Zhang, "Comparative Causality: Explaining the Differences Between Executions," in *Proc. of Int'l Conf. on Softw. Eng.*, 2013, pp. 272–281.

[36] R. Santelices and M. J. Harrold, "Demand-driven propagation-based strategies for testing changes," *Software Testing, Verification and Reliability*, vol. 23, no. 6, pp. 499–528, 2013.

[37] J. Ferrante, K. Ottenstein, and J. Warren, "The program dependence graph and its use in optimization," *ACM Trans. on Prog. Lang. and Systems*, 9(3):319-349, Jul. 1987.

[38] A. V. Aho, M. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques and Tools (2nd Ed.)*, Sep. 2006.

[39] A. Beszedes, C. Farago, Z. Mihaly Szabo, J. Csirik, and T. Gyimothy, "Union Slices For Program Maintenance," in *Proceedings of IEEE International Conference on Software Maintenance*, 2002, pp. 12–21.

[40] R. Santelices, Y. Zhang, H. Cai, and S. Jiang, "DUA-Forensics: A Fine-Grained Dependence Analysis and Instrumentation Framework Based on Soot," in *Proceeding of ACM SIGPLAN International Workshop on the State Of the Art in Java Program Analysis*, Jun. 2013, pp. 13–18.

[41] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Emp. S. Eng.*, vol. 10, no. 4, pp. 405–435, 2005.

[42] R. E. Walpole, R. H. Myers, S. L. Myers, and K. E. Ye, *Probability and Statistics for Engineers and Scientists*. Prentice Hall, Jan. 2011.

[43] R. Santelices, Y. Zhang, S. Jiang, H. Cai, , and Y. jie Zhang, "Quantitative Program Slicing: Separating Statements by Relevance," in *Proceedings of IEEE/ACM ICSE - NIER*, May 2013, pp. 1269–1272.

[44] T. Goradia, "Dynamic Impact Analysis: A Cost-effective Technique to Enforce Error-propagation," in *Proceedings of ACM International Symposium on Software Testing and Analysis*, Jul. 1993, pp. 171–181.

[45] J. Law and G. Rothermel, "Whole program path-based dynamic impact analysis," in *Proc. of Int'l Conf. on Softw. Eng.*, May 2003, pp. 308–318.

[46] J. Voas, "PIE:A Dynamic Failure-Based Technique," *IEEE Trans. on Softw. Eng.*, 18(8):717–727, Aug. 1992.

[47] J. A. Jones and M. J. Harrold, "Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique," in *Proc. of Int'l Conf. on Automated Softw. Eng.*, 2005, pp. 273–282.