SENSA: Sensitivity Analysis for Quantitative Change-impact Prediction

Haipeng Cai, Siyuan Jiang, Ying-jie Zhang[†], Yiji Zhang, and Raul Santelices
University of Notre Dame, Indiana, USA
[†]Tsinghua University, Beijing, China
email: {hcai|sjiang1|yzhang20|rsanteli}@nd.edu, [†]zhangyj1991@gmail.com

ABSTRACT

Sensitivity analysis is used in many fields to determine how different parts of a system respond to variations in stimuli. Software-development tasks, such as change-impact analysis, can also benefit from sensitivity analysis. In this paper, we present SENSA, a novel dynamic-analysis technique and tool that uses sensitivity analysis and execution differencing for estimating the influence of program statements on the rest of the program. SENSA not only identifies the statements that can be affected by another statement, but also *quantifies* those effects. Quantifying effects in programs can greatly increase the effectiveness of tasks such as change-impact analysis by helping developers prioritize and focus their inspection of those effects. Our studies on four Java subjects indicate that SENSA can predict the impact of changes much more accurately than the state of the art: forward slicing. The SENSA prototype tool is freely available to the community for download.

1. INTRODUCTION

Modern software is increasingly complex and changes constantly. Therefore, it is crucial to provide automated and effective support to analyze the interactions among its components and, in particular, the effects of changes. Developers must understand the consequences and risks of modifying each part of a software system even before they can properly design and test their changes. Unfortunately, existing techniques for analyzing the effects of code, such as those for *change-impact analysis* [7], are quite imprecise, which hinders their applicability and limits their adoption in practice.

Many change-impact analyses operate at coarse levels such as methods and classes (e.g., [20, 26]). Although these techniques provide a first approximation of the effects of changes, they do not distinguish which particular statements play a role in those effects and, thus, can classify as impacted many methods and classes that are not really impacted (false positives). Moreover, these techniques can miss code-level dependencies not captured by higher-level structural relationships [36] (false negatives).

At the code level (statements), the forward version of *program slicing* [16, 38] reports all statements that might be impacted by a change in a statement for any execution (via *static* forward slicing) or for a set of executions (via *dynamic* forward slicing). Slicing, however, is also imprecise. Static slicing usually reports many potentially-affected statements that are not really affected [6, 23], whereas dynamic slicing [2,5,18] reduces the size of the results but still produces false positives [21,30] in addition to false negatives.

To increase the precision of slicing by reducing the size of the resulting *slices* (the sets of affected statements), researchers have tried combining static slices with execution data [13,15,19,23] and pruning slices based on some criteria [1,8,33,39]. However, those methods still suffer from false positives and false negatives. Moreover, the number of statements reported as "affected" can still be too large. Optimizing underlying algorithms can reduce this imprecision [19,22,24] but at great costs for small payoffs.

In this paper, we take a different approach to measuring the influence of statements in programs. Instead of trying to further reduce the number of affected statements, our approach distinguishes statements by *likelihood* of impact so that users can focus on the most likely impacts first. Our new technique and tool, SENSA, uses *sensitivity analysis* [28] and *execution differencing* [30,34,40] to estimate those likelihoods. Sensitivity analysis is used in many fields to measure relationships among different parts of a system. In software engineering, sensitivity analysis has been used to analyze requirements and components [14,27] and, in a restricted way, for *mutation analysis* [9, 17, 37]. Meanwhile, execution differencing has been used for program integration and debugging. However, neither approach, alone or combined, has been used for forward slicing and change-impact analysis, as our new technique does.

SENSA quantifies the effect that the behavior of a statement or potential changes in it may have on the rest of the program. SENSA inputs a program P, a test suite T, and a statement c. For each test case t in T, SENSA repeatedly executes t, each time changing the value computed by c to a different value and finding the differences in those executions. For each execution, the differences show which statements change their *behavior* (i.e., state or occurrences) when 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 that c affects (or impacts) s in future runs of the program. The greater the frequency for s is, the more likely it is that s will be impacted by the behavior of c.

To evaluate the effectiveness of SENSA for estimating the influence of statements, we empirically compared the ability of SENSA and static and dynamic forward slicing for predicting the impacts of changes in those statements. For each of a number of potential change locations in four Java subjects, we ranked all statements in the subject by impact likelihood according to SENSA and by dependence distance as originally proposed by Weiser for slicing [38]. Then, we computed the average effort a developer would spend inspecting each ranking to find all statements impacted when applying a real change to each such location and executing the corresponding test suite. Our results show that, for these subjects, SENSA outperforms both forms of slicing at predicting change impacts, reducing slice-inspection efforts to small fractions.

We also performed three case studies manually on these subjects to investigate how well SENSA and program slicing highlight the cause-effect chains that explain how bugs actually propagate to failing points and, thus, how bug fixes can be designed. To achieve this goal, first, we manually identified the statements that are impacted by each bug and also propagate the erroneous state to a failing point. Then, we computed how well SENSA and slicing isolate those event chains. Although we cannot make general conclusions, we found again that SENSA was better than slicing for isolating the specific ways in which buggy statements make a program fail.

The main benefit of this work is the greater usefulness of quantified influences, as computed by SENSA, over simple static and dynamic program slices. With this kind of information, developers can use slices more effectively, especially large slices, by focusing on the statements most likely to be affected in practice. Moreover, quantified impacts can be used not only for change-related tasks but also, potentially, to improve other applications, such as testability analysis [37], execution hijacking [35], and mutation testing [9].

The main contributions of this paper are:

- The concept of impact quantification, which, unlike slicing, distinguishes statements by estimated likelihood of impact.
- A new technique and tool, SENSA, that estimates these likelihoods and is available to the public.
- Studies that measure and illustrate the effectiveness of SENSA for predicting the effects and impacts of changes.

2. BACKGROUND

This section presents core concepts necessary for understanding the rest of the paper and illustrates these concepts 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`.

2.1 Program Dependencies

Control and data dependences are the building blocks of program slicing [16, 38]. A statement s_1 is *control dependent* [11] on a statement s_2 if a branching decision taken at s_2 determines whether s_1 is necessarily executed. In Figure 1, an example is statement 3, which is control dependent on statement 2 because the decision taken at 2 determines whether statement 3 executes or not. This dependence is *intra-procedural* because both statements are in the same procedure. Control dependences can also be *inter-procedural* (across procedures) [32].

A statement s_1 is *data dependent* [3] on a statement s_2 if a variable v defined (written to) at s_2 is used (read) at s_1 and there is a *definition-clear path* in the program for v (i.e., a path that does

```
 float prog(int n, float s) {
      int g;
1:    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;               }
```

Figure 1: Example program used throughout the paper.

not re-define v) from s_2 to s_1. 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 (3,4,8) that does not re-define `s` after 3. Data dependencies can also be classified as intra-procedural (all definition-use paths are in the same procedure) or inter-procedural (some definition-use path crosses procedure boundaries). The formal parameters of `prog`, however, are inputs—they are not data dependent on any other statement.

2.2 Program Slicing

Program slicing [16,38], also called static slicing, determines which statements of the program may affect or be affected by another statement. A *static forward slice* (or, simply, forward slice) from statement s is the set containing s and all statements transitively affected by s along control and data dependences. (To avoid invalid inter-procedural paths, the solution by Horwitz, Reps, and Binkley [16] can be used.) For example, the 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 forward slice from statement 1 in `prog`, which is {1,2,3,4,5,6,7,8}. Statement 2 uses `g`, so it is data dependent on statement 1. Statements 3, 4, 5, 6, and 7 are control dependent on statement 2, so they are also in the forward slice. Finally, statement 8 depends on `g` at 1 transitively because it depends on `s` at 3, 5 and 6, all of which have definition-clear paths to 8.

The size of a static slice can vary according to the precision of the analysis. For example, the point-to analysis can affect the size of a static slice. If we use a coarse points-to analysis in which a pointer can be any memory address, a forward slice from a statement s that defines a pointer p would include any statement that uses or dereferences any pointer (which may or may not point to the same address as p) if the statement is reachable from s.

3. EXECUTION DIFFERENCING

Differential execution analysis (DEA) is designed specifically to identify the runtime *semantic dependencies* [25] of statements on changes. Semantic dependencies tell which statements are truly affected by other statements or changes. Data and control dependencies and, therefore, slicing, only provide necessary but not sufficient conditions for semantic dependence. This is the main reason for the imprecision of slicing.

Although finding semantic dependencies is an undecidable problem, DEA detects such dependencies on changes when they occur at runtime to under-approximate the set of semantic dependencies in the program. Therefore, DEA cannot guarantee 100% recall of semantic dependencies but it achieves 100% precision. This is much better than what dynamic slicing usually achieves [21,30].

DEA executes a program before and after a change to collect the execution history (including states) [30] of each execution and then compare both executions. The execution history of a program is the sequence of statements executed and the values computed at each statement. 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 [25].

4. TECHNIQUE

The goal of SENSA is, for a program P and its test suite T, to quantify the potential impacts of changing a statement s in P by applying sensitivity analysis [28] and execution differencing [30]. In Section 4.1, we first give an overview of this new technique using an example program to illustrate it. Then, in Section 4.2, we give a precise description of our technique including its process and its algorithm. Finally, in Section 4.3, we describe the three state-modification strategies that SENSA offers.

4.1 Overview

When a change is made to a program statement, DEA tells developers which other statements in the program are impacted. However, developers first need to know the possible impacts of changing a statement *before* deciding to change it and before designing and applying a change in that location. Also, in general, developers often need to understand the influences that specific statements exert on the rest of the program and, thus, determine their role in it.

To identify and *quantify* those influences, SENSA uses sensitivity analysis on a program statement s that is a candidate for changes or whose role in the program must be measured. Based on the test suite for the program, SENSA repeatedly runs the program while modifying the state of statement s and identifies in detail, using DEA, the impacted statements for each modification. Then, SENSA computes the frequency with which each statement is impacted—the sensitivity of those statements to s. These sensitivities are estimates, based on the test suite and the modifications made, of the *strength* or *likelihood* of each influence of s.

We use the example program `prog` in Figure 1 again to illustrate how SENSA works using two test inputs: (2, 82.5) and (3, 94.7). Suppose that a developer asks for the effect 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 q with values in its "valid" range of [1..6]. For each test case, SENSA first executes `prog` without changes to provide the baseline execution history for DEA. Then SENSA re-executes the test case five times—once for each other value of q in range [1..6]. Finally SENSA applies DEA to the execution histories of the baseline and modified runs of each test case and computes the *frequency* (i.e., the fraction of all executions) with which every other line was *impacted* (i.e., changed its state or occurrences).

In the example, the result is the ranking ({1,2,3,4,8}, {5}, {6,7}), where lines 1, 2, 3, 4, and 8 are tied at the top because their states (the values of q and/or s) change in all modified runs and, thus, their sensitivity is 1.0. Line 5 comes next with sensitivity 0.2 as it executes for one modification of each test case (when q changes to 6) whereas, in the baseline executions, it is not reached. Lines 6 and 7 rank at the bottom because they never execute.

In contrast, static forward slicing ranks the program statements by dependence distance from line 1. These distances are obtained through a breadth-first search (BFS) of the dependence graph—the inspection order suggested originally by Weiser [38]. Thus, the result for static slicing is the ranking ({1}, {2,3,4,7}, {5,6}, {8}). For dynamic slicing, a BFS of the dynamic dependence graph, which is the same for both test cases, yields ranking ({1}, {2,3,4}, {8}).

To measure the predictive power of these rankings, suppose that the developer decides to change line 1 to `g = n + 2`. The *actual* set of impacted statements for this change and test suite is {1,2,3,4,8}—all of them are impacted for both test cases. This is exactly the set of statements placed at the top of the ranking by SENSA. In contrast, static slicing predicts that statement 1 will be the most impacted, followed by 2, 3, and 4, and then statement 8 as the least impacted. This prediction is less accurate than that of SENSA, especially because static slicing also places the unaffected statement 7 at the same level as 2, 3, and 4, and predicts 5 and 6 above 8.

Dynamic slicing, against intuition, performs even worse than static slicing with respect to SENSA in this example. This approach ranks the actually-impacted statements 4 and 8 at or near the bottom and misses statement 5 altogether. Thus, dynamic slicing can be imprecise *and* produce false negatives for impact prediction.

Naturally, the predictions of SENSA depend on the test suite and the modifications chosen. The quality of the predictions depends also on which change we are making predictions for. If, for example, n in the first test case is 5 instead of 2 and the range [0..10] is specified instead of [1..6] to modify line 1, SENSA would produce the ranking ({1,2,8}, {3,4,6,7}, {5}) by using 20 modified runs. The static- and dynamic-slicing rankings remain the same.

If, in this scenario, the developer changes line 1 to `g = n × 2`, the actual impact set is the entire program because, after the change, lines 5, 6, and 7 will execute for this test suite. In this case, SENSA still does a better job than static slicing, albeit to a lesser extent, by including 2 and 8 at the top of the ranking and including 3, 4, 6, and 7 in second place, whereas static slicing places 6 and 8 at or near the end. SENSA only does worse than static slicing for line 5. Dynamic slicing still does a poor job by missing lines 5, 6, and 7.

Note that `prog` is a very simple program that contains eight statements only. This program does not require much effort to identify and rank potential impacts, regardless of the approach used. In a more realistic case, however, the differences in prediction accuracy between SENSA and both forms of slicing can be substantial, as the studies we present in Sections 5 and 6 strongly suggest.

4.2 Description

SENSA is a dynamic analysis that, for a statement C (e.g., a candidate change location) in a program P with test suite T, assigns to each statement s in a program P a value between 0 and 1. This value is an estimate of the "size" or frequency of the influence of C on each s in P. Next, we present SENSA's process and algorithm.

4.2.1 Process

Figure 2 shows the diagram of the process that SENSA follows to quantify influences in programs. The process logically flows from the top to the bottom of the diagram and is divided into three stages, each one highlighted with a different background color: (1) *Pre-process*, (2) *Runtime*, and (3) *Post-process*.

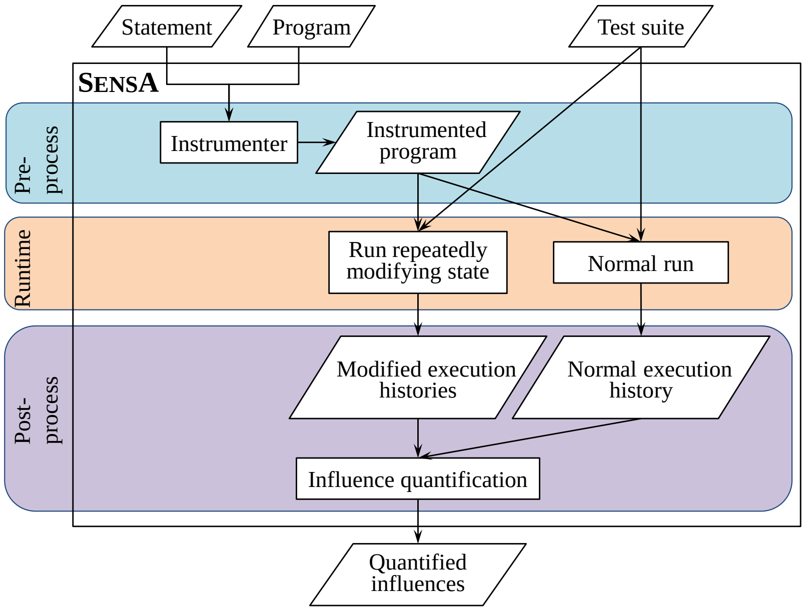

Figure 2: Process used by SENSA for influence quantification.

For the first stage, on the top left, the diagram shows that SENSA inputs a *Program* and a *Statement*, which we denote as 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 uses DEA to instrument P to collect the execution history of the program, including state updates [30]. The result, shown in the diagram, is the *Instrumented program*.

In the second stage, SENSA inputs a *Test suite*, T, and runs the program repeatedly with modifications (*Run repeatedly modifying state*) of the value produced by C, 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 4.3) 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 case.

In the third and 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 *Post-process* stage, *Influence quantification* processes the execution history differences and calculates the frequencies as fractions in the range [0..1]. These are the frequencies with which the statements in P were "impacted" 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. As part of this output, SENSA also ranks the statements by decreasing influence.

4.2.2 Algorithm

Algorithm 1 formally describes how SENSA quantifies the influences of a statement C in a program P. The statements ranked by SENSA are those in the forward static slice of C in P because those are the ones that can be influenced by C. Therefore, to begin with the first stage, the algorithm starts by computing this static slice (line 1). At lines 2–5, SENSA initializes the map *influence* that associates to all statements in the slice a frequency and a rank. Initially, these two values are 0 for all statements. Then, at line 6, SENSA instruments P to let SENSA modify at runtime the values at C and to collect the execution histories for DEA.

For the second stage, the loop at lines 7–16 determines, for all test cases t, how many times each statement is impacted by the modifications made by SENSA. At line 8, SENSA executes t without modifications to obtain the baseline execution history. Then, line 10 executes this same test for the number of times indicated by the parameter given by SENSA-MODS(). Each time, SENSA modifies the value or branching decision at C with a different value.

For the third stage, line 11 asks DEA for the differences between each modified run and the baseline run. In lines 12–14, the differences found are used to increment the *frequency* counter for each statement affected by the modification. Then, the loop at lines 17–19 divides the influence counter for each statement by the total number of modified runs performed, which normalizes this counter to obtain its influence frequency in range [0..1].

4.3 Modification Strategies

SENSA is a generic modifier of program states at given program locations. The technique ensures that each new value picked to replace in a location is unique to maximize diversity while minimizing bias. Whenever 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 modification strategy to use to pick each new value for a statement. The choice of values affects the quality of the results of SENSA, so we designed three different strategies while making it straightforward to add other strategies in the future. The built-in modification strategies are:

Algorithm 1 : SENSA(program P, statement C, test suite T)

```
    // Stage 1: Pre-process
 1: slice = STATICSLICE($P$, $C$)
 2: influence = $\emptyset$   // map statement$\rightarrow$(frequency,rank)
 3: for each statement $s$ in slice do
 4:     influence[$s$] = (0, 0)
 5: end for
 6: $P'$ = SENSA-INSTRUMENT($P$, $C$)
    // Stage 2: Runtime
 7: for each test case $t$ in $T$ do
 8:     exHistBaseline = SENSA-RUNNORMAL($P'$, $t$)
 9:     for $i = 1$ to SENSA-MODS() do
10:         exHistModified = SENSA-RUNMODIFIED($P'$, $t$)
            // Stage 3: Post-process
11:         affected = DEA-DIFF(exHistBaseline, exHistModified)
12:         for each statement $s$ in affected do
13:             influence[$s$].frequency++
14:         end for
15:     end for
16: end for
    // Stage 3: Post-process (continued)
17: for each statement $s$ in influence do
18:     influence[$s$].frequency /= SENSA-MODS()$\times|T|$
19: end for
20: RANKBYFREQUENCY(influence)
21: return influence   // frequency and rank per statement
```

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*.[1]
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.
3. *Observed*: First, the strategy collects all values that C computes for the entire test suite. Then, the strategy picks iteratively from this pool each new value to replace at C. The goal is to ensure that the chosen values are meaningful to the program.

5. EVALUATION

To evaluate SENSA, we studied its ability to predict the impacts of changes in typical operational conditions. We compared these predictions with those of the state-of-the-art—static and dynamic slicing. Our rationale is that the more closely a technique approximates the actual impacts that changes will have, the more effectively developers will maintain and evolve their software. To this end, we formulated three research questions:

RQ1: How good is SENSA overall at predicting the statements impacted by changes?

RQ2: How good are subsets of the SENSA rankings, if time constraints exist, at predicting impacted statements?

RQ3: How expensive is it to use SENSA?

[1]Instantiating objects with random states is in our future plans.

Table 1: Experimental subjects and their statistics

Subject	Short description	LOC	Tests	Changes
Schedule1	priority scheduler	301	2650	7
NanoXML	XML parser	3521	214	7
XML-security	encryption library	25220[3]	92	7
JMeter	performance tester	39153[3]	79	7

The first and second questions address the benefits of SENSA overall and per ranking-inspection effort—when developers can only inspect a portion of the predicted ranking in practice. The third question targets the practicality of this technique.

5.1 Experimental Setup

We implemented SENSA in Java as an extension of our dependence analysis and instrumentation toolkit DUA-FORENSICS [29]. As such, SENSA works on Java bytecode programs. The SENSA tool is available to the public for download.[2] For our experiments, we run SENSA on a dedicated Linux desktop with a quad-core 3.10GHz Intel i5-2400 CPU and 8GB of memory.

We studied four Java programs of various types and sizes obtained from the SIR repository [10], including test suites and changes. Table 1 lists these subjects. Column *LOC* shows the size of each subject in lines of code. Column *Tests* shows the number of tests for each subject. Column *Changes* shows the number of changes used for each subject. We limited this number to seven because, for some of the subjects, this is the maximum available number. For each subject, we picked the first seven changes that come with it. All changes are fixes for bugs in these subjects.

The first subject, Schedule1, is part of the Siemens suite that we translated from C to Java. This program can be considered as representative of small software modules. NanoXML is a lean XML parser designed for a small memory footprint. We used version *v1* of this program as published on the SIR. XML-security is the XML signature and encryption component of the Apache project. JMeter is also an Apache application for load-testing functional behavior and measuring the performance of software.[3]

5.2 Methodology

Figure 3 shows our experimental process. On the top left, the inputs for SENSA include a program, a statement (e.g., a candidate change location), and a test suite. SENSA quantifies the influence of the input statement on each statement of the program for the test suite of the program. This technique outputs those statements ranked by decreasing influence of the input statement. For tied statements in the ranking, the *rank* assigned to all of them is the average position of these statements in the ranking. To enable a comparison of this ranking with the rankings for slicing, the SENSA ranking includes at the bottom, tied with influence zero, those statements in the static forward slice not found to be affected during analysis.

On the right of the diagram, an *Actual impact computation* procedure takes the same inputs and also a change for the input statement. This procedure uses the execution-differencing technique DEA [30] to determine the *exact* set of statements whose behavior

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

[3]Some subjects contain non-Java code. The analyzed subsets in Java are 22361 LOC for XML-security and 35547 LOC for JMeter.

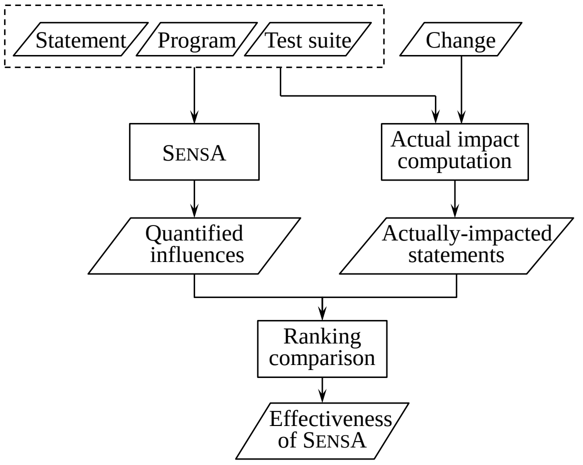

Figure 3: Experimental process.

actually changes 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 *effectiveness* (i.e., prediction accuracy) of the SENSA ranking by comparing this ranking with the set of actually-impacted statements. The process makes a similar comparison, not shown in the diagram, for the rankings obtained from breadth-first searches of the dependence graphs for static and dynamic slicing (the traversal order suggested originally by Weiser [38]).

Ranking comparison computes the effectiveness of a ranking with respect to a set of impacted statements by determining how high in the ranking those statements are located.[4] For each impacted statement, its rank value in the ranking represents the effort that a developer would spend to find that statement 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 isolating and predicting the actual impacts that are likely to occur in practice (after the change is made). The process computes the average rank of the impacted statements to represent their *inspection cost*. The *effectiveness* of the ranking is the inverse of that cost.

To provide a basis of comparison of the quality of the predictions of the techniques we studied, we also computed the inspection cost for the *ideal* scenario for each change. This ideal case corresponds to a ranking in which the top $|S|$ statements are exactly those in the set S of impacted statements for that change. This is the best possible ranking for predicting all impacts of the change.

For RQ1, we computed the average inspection costs for the entire rankings for SENSA, static slicing, and dynamic slicing. For RQ2, we computed, for each ranking, the percentage of impacted statements found in each fraction from the top of the ranking—for fractions $\frac{1}{N}$ to $\frac{N}{N}$, where N is the size of the ranking. For RQ3, we measured the time it takes to run the test suite on the original (non-instrumented) program and on the instrumented versions of the program for SENSA and dynamic slicing. We also collected the times SENSA and DUA-FORENSICS take for static slicing and for pre- and post-processing in SENSA.

[4]The top statement in a ranking has a rank value of 1.

Table 2: Average inspection costs for all actual impacts

Subject	Average cost for all impacts and changes (%)					
	Ideal case	Static slicing	Dynamic slicing	SENSA -Rand	SENSA -Inc	SENSA -Obs
Schedule1	39.4	49.3	47.8	40.2	40.2	40.2
NanoXML	7.3	27.3	22.2	9.2	9.9	-
XML-security	5.6	36.4	39.9	11.6	12.5	-
JMeter	0.2	12.5	37.7	7.7	3.0	-

It is important to note that the test suites used to compute the SENSA rankings are the same that we used to find the actual impacts when applying the changes. Therefore, the SENSA predictions might appear biased on the surface. However, there is a strong reason to use the same test suite for both the technique and the "ground truth" (the actual impacts): developers will use the same, existing test suite to run SENSA and then to observe the actual impacts of their changes. For that reason, we decided to use the same test suite for both parts of the experimental process.

5.3 Results and Analysis

5.3.1 RQ1: Overall Effectiveness

Table 2 presents, for all four subjects, the average inspection costs per subject for the seven changes in that subject for the *Ideal* scenario (best possible ranking) and for a number of techniques. The units are percentages of the static slice sizes, which range between 55–79% of the entire program. The techniques are *Static slicing*, *Dynamic slicing*, and three versions of SENSA: SENSA-*Rand*, SENSA-*Inc*, and SENSA-*Obs* for strategies *Random*, *Incremental*, and *Observed*, respectively.

For some changes in NanoXML, XML-security, and JMeter, SENSA-*Obs* observed only one value at the input statement for the entire test suite. As a consequence, this strategy, by definition, could not make any modification of that value and, thus, could not be applied to those changes. Therefore, we omitted the average cost result for this version of SENSA for those subjects in Table 2.

We first observe, from the *Ideal case* results, 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 39.4% in Schedule1 down to 0.2% in JMeter. This phenomenon can be explained by two factors. First, the larger subjects in this study consist of a variety of loosely coupled modules and the changes, which are thus more scattered, can only affect smaller fractions of the program. The second factor is that, nevertheless, slicing will find connections among those modules that are rarely, if ever, exercised at runtime. Also, aliasing in those subjects decreases the precision of slicing. For Schedule1, however, there is little use of pointers and most of the program executes and is impacted for every test case and by every change. In fact, an average of 78.5% of the statements in the slices for Schedule1 were impacted. These factors explain the much greater inspection costs for the ideal case and all techniques in this subject.

For Schedule1, the inspection cost for all three versions of SENSA was 40.2% on average for all changes in that subject. This cost was, remarkably, only 0.8 percentage points greater than the ideal case. This is a remarkable result, especially because static and dynamic slicing did worse than the ideal case by 8.4 or more percentage points. Our data for individual changes (not shown in the table) for this subject show a consistent trend in which SENSA cost no more than 1.4 percentage points more than the ideal case. Also for individual changes, the differences among the costs of the three versions of SENSA were very small—less than 1% in all cases. In consequence, for this subject, all three strategies were equally good alternatives, and all of them were better than both types of slicing.

The average results for the changes in NanoXML show an even greater gap in effectiveness than in Schedule1 between the two applicable versions of SENSA and static and dynamic slicing. The worst-performing version of our technique, SENSA-*Inc*, cost only 2.2 percentage points more than the ideal case on average (9.9% vs. 7.3% of all statements that can possibly be affected), whereas the breadth-first traversals for static and dynamic slicing cost at least 14.9% more than the ideal ranking.

For XML-security, the results follow the same trend as in the first two subjects, in which SENSA provided better predictions than both static and dynamic slicing at increasing rates. The two applicable versions of SENSA incurred an average cost of no more than 6.9 percentage points over the ideal cost, whereas slicing cost at least 30.8 points more than the ideal.

Remarkably, for XML-security, unlike Schedule1 and NanoXML, dynamic slicing produced worse predictions than static slicing. This apparently counter-intuitive result is explained by the divergence in the paths taken by the executions before and after the actual changes. Because of these divergences, dynamic slicing actually missed many impacted statements that were not dynamically dependent on the input statement that are, however, statically dependent on that statement. SENSA did not suffer from this problem because its modifications were able to alter those paths to approximate the effects that the actual changes would have later.

For the largest subject, JMeter, the results again indicate a superiority of SENSA over slicing, at least for these changes and this test suite. On average, SENSA cost 7.5 and 2.8 percentage points than the ideal cost for the *Random* and *Incremental* strategies, respectively. With respect to slicing, SENSA cost at least 4.8 and 9.4 points less, respectively. No technique, however, was able to get close to the considerably small ideal ranking (0.2%). There are two remarkable observations for this subject. First, dynamic slices was much worse than static slicing—by a much greater than for XML-security—which is, again, explained by its inability to consider alternative execution paths. Second, for this subject, we observed the greatest difference between the two applicable SENSA versions, where, for the first and only time, SENSA-*Inc* was superior. This superiority suggests that the actual changes we studied in JMeter had effects observable mostly in the vicinity of the values computed in the unchanged program, as SENSA-*Inc* simulates, rather than at random different locations in the value range.

In sum, for RQ1 and for these subjects and changes, the results indicate that SENSA is considerably better than slicing techniques at predicting which statements will be impacted when these changes are actually made. In particular, these results highlight the imprecision of both static and dynamic slicing, at least with respect to changes. More importantly, our observations for SENSA are reminiscent of similar conclusions observed for mutation analysis [4] in which small modifications appear to be representative of faults in general—or, as in this experiment, fault fixes. These results suggest that developers who use SENSA can save a substantial amount of effort for understanding the potential consequences of changes.

5.3.2 RQ2: Effectiveness per Inspection Effort

Because program slices are often very large, developers cannot examine in practice all statements in each ranking produced by SENSA or slicing. Instead, they will likely inspect a fraction of all potential impacts and will focus on the most likely impacts first. With this prioritization, developers can maximize the cost-effectiveness of their inspection by analyzing as many highly-ranked impacts as possible within their budget. To understand the effects of such prioritization, we studied the effectiveness of each portion of each ranking produced by the studied techniques.

Figures 4–7 show the cost-effectiveness curves for the best possible ranking-examination order and for the studied techniques SENSA (two or three applicable versions) and static and dynamic slicing. For each graph, each point in the horizontal axis represents the fraction of the ranking examined from the top of that ranking, whereas the vertical axis corresponds to the percentage of actually-impacted statements (on average for all changes in the respective subject) found within that fraction of the ranking. Note that the result presented in Table 2 for each ranking is the average of the Y values for that entire ranking in the corresponding graph.

The *Ideal* curves in these graphs provide detailed insights on how cost-effective impact prediction techniques can aspire to be. For all subjects but Schedule1, this curve rises sharply within the first 10% of the ranking. These curves are not straight lines because they correspond to the average curves for all changes in per subject and, in general, the actual impacts for these changes (which define *Ideal*) vary in size. Only for Schedule1, the *Ideal* curve is mostly straight because all slices have almost the same size and the impacted fractions of the slices have similar sizes.

For Schedule1, because of the high baseline (ideal) costs, all curves are relatively close to each other. However, there are two distinctive groups: the SENSA versions near the ideal curve and the slicing curves below them. Interestingly, the SENSA curves overlap with the ideal curve until about 70% of the ranking. Therefore, a user of SENSA can find as many impacts as it is possible if the inspection budget is 70% or less. At 70%, 90% of the impacts are predicted by SENSA, in contrast with 70–75% for slicing. Also, SENSA-*Obs*, which only applies to this subject, has virtually the same cost-effectiveness as the two other versions of SENSA.

The results for the three other subjects, which represent more modern software than Schedule1, indicate even stronger cost-effectiveness benefits for the two applicable SENSA versions. The curves in Figures 5, 6, and 7 show that SENSA was not only better than slicing overall at predicting impacts, but also that this superiority is even greater in comparison for fractions of the resulting rankings. The curves for either SENSA-*Rand* or SENSA-*Inc*, or both, grow much faster at the beginning than those for slicing. In other words, these results suggest that users can benefit even more from choosing SENSA over slicing when they are on a budget.

The only case in which one of the slicing approaches seems to be competitive with SENSA is for JMeter. For this subject, on average for all seven changes, static slicing overcomes SENSA-*Rand* at about 25% of the inspection effort. However, after that point, static slicing maintains only a small advantage over this version of SENSA, and this difference disappears somewhere between 80–

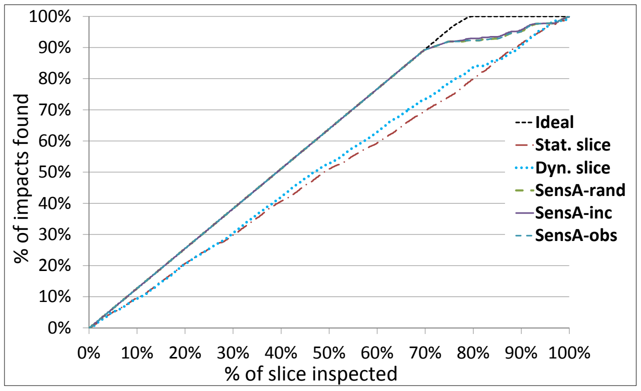

Figure 4: Impacted statements found versus cost for Schedule1

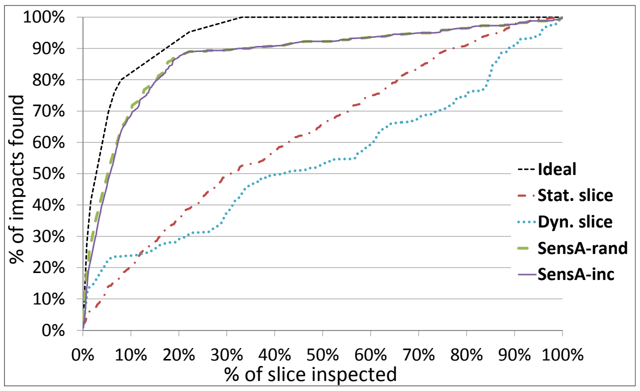

Figure 6: Impacted statements found versus cost for XML-security

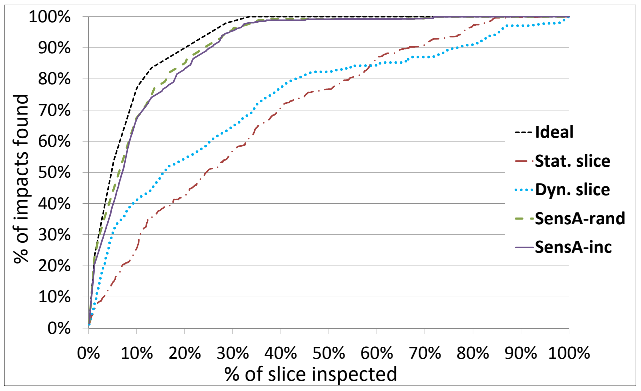

Figure 5: Impacted statements found versus cost for NanoXML

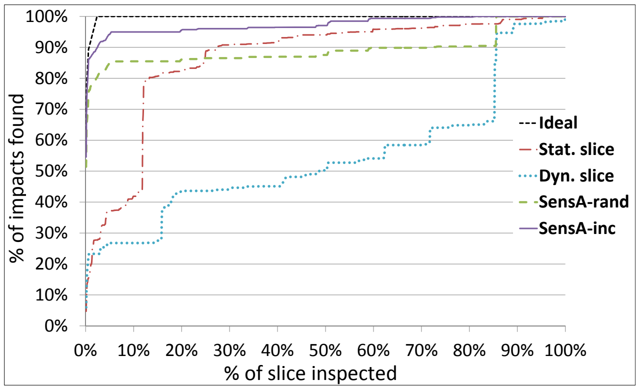

Figure 7: Impacted statements found versus cost for JMeter

90% of the inspection. More importantly, SENSA-*Rand* is considerably better than static slicing before the 25% point, which explains the overall advantage of 4.8 percentage points for the former technique as reported in Table 2. The main highlight for this subject, however, is SENSA-*Inc*, which performs better than both techniques at all points.

In sum, for RQ2, the results indicate that, for these subjects, test suites, and changes, SENSA makes not only better predictions than slicing to understand the potential impacts of changes, but it is also better by even greater amounts for discovering those impacts early in the resulting rankings. This makes SENSA an even more attractive choice when users are on a budget, as it is likely to happen in practice. Also, among all versions of SENSA, and despite performing slightly worse overall than SENSA-*Rand* for NanoXML and XML-security, SENSA-*Inc* seems the best choice based on its cost-effectiveness for JMeter.

5.3.3 RQ3: Computational Costs

For RQ3, we analyze the practicality of SENSA. Table 3 shows the time in seconds it takes to run the test suite for each subject, without instrumentation (column *Normal run*), using the machine and environment described in Section 5.1. These times help put in perspective the analysis times taken by SENSA and slicing. Perhaps against intuition, the longest running time is for Schedule1 because, despite being the smallest subject, its test suite is at least an order of magnitude larger than for the other subjects (see Table 1).

The next three columns report the time in seconds taken by each the three stages of SENSA for each subject. Each time reported here is the average for all changes for that subject. 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 dynamic slicing, DEA, and SENSA. As expected, this time grows with the size of the subjects. Interestingly, despite the plenty of room for optimization of the implementation of DUA-FORENSICS and SENSA, the static analysis for SENSA does not consume more than an average of ten minutes.

For the second stage, as expected due to the varying ratios of subject size to test-suite size, the greatest amount of time is spent by JMeter and the second greatest by Schedule1. In both cases, the cost in time is approximately five minutes. For the third and last stage, which processes the runtime data to compute the SENSA rankings, the time costs were less than the runtime costs. This means that, overall, no individual phase takes an unreasonable amount of time to produce impact predictions. In the worst case, for JMeter, the total cost is below 20 minutes.

In all, these computational-cost results are encouraging for the practicality of SENSA for three reasons. First, we believe that, in many cases, developers can make use of the predictions of SENSA if

Table 3: Average computational costs of techniques, in seconds

Subject name	Normal run	Static analysis	Instrumented run	Influence ranking
Schedule1	169.0	5.4	290.6	19.8
NanoXML	13.4	13.7	35.8	1.8
XML-security	58.2	153.5	67.0	63.2
JMeter	37.5	594.6	328.3	238.9

they are provided within 20 minutes for a program such as JMeter and less than 10 minutes for smaller programs. Second, the machine and environment used in our experiment correspond to a mid-level setup that developers typically use nowadays. Third, many of the steps performed by our research toolset not only can be optimized for individual operations, but can also undergo algorithmic improvements. Moreover, the iterative nature of SENSA provides opportunities for significant speed-ups via parallelization.

5.4 Threats to Validity

The main internal threat to the validity of our studies is the potential presence of implementation errors in SENSA. SENSA is a research prototype developed for this work. However, SENSA is built on top of DUA-FORENSICS, whose development started in 2006 and has matured considerably over the time. Also, this SENSA layer built on top DUA-FORENSICS has been tested, manually examined, and improved over more than half a year.

Another internal threat to our experiment 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, inspected, debugged, and manually verified the results of each phase of this process.

The main external threat to the validity of our study and conclusions about SENSA is that we studied only a limited number and variety of subjects and changes (bug fixes) using test suites that may not represent all possible behaviors of those subjects. Nevertheless, we chose these four subjects to represent a variety of sizes, coding styles, and functionality to achieve as much representativeness as possible of software in general. In addition, these subjects have been used extensively in experiments conducted in the past by the authors and/or other researchers around the world. Moreover, the three largest subjects are real-world open-source programs.

6. CASE STUDIES

To understand in detail the comparative benefits of SENSA and slicing for a more specific task, we investigated in detail how these techniques could help a developer isolate the concrete effects of faulty code to help understand and fix that fault. To that end, we conducted three case studies of fault-effects analysis. When a (candidate) faulty statement is identified, a developer must decide how to fix it. This decision requires understanding the effects that this statement has on the *failing point*—a failed assertion, a bad output, or the location of a crash. However, not all effects of a faulty statement are necessarily erroneous. The interesting behavior of a fault is the chain of events from that fault to the failing point.

For diversity, we performed the case studies on different subjects. For representativeness, we picked the three largest subjects that we already set up for our previous study: NanoXML, XML-security, and JMeter. For each case study, we chose the first change location (a bug to fix) provided with the subject and we identified the first failing point (typically, the only one) where the bug is manifested. Given the faulty statement, which a developer could identify with a fault-localization technique, we manually identified the sequence of all statements that propagate the fault to the failing point. We discarded affected statements that did not participate in this propagation to the failing point. All statements are Java bytecode instructions in a readable representation.

Given a chain of events—the set of propagating statements—and the bug fix that is provided as a change with the subject, we computed how high the chain is in the rankings computed by SENSA and by static and dynamic slicing from the fault location. Specifically, we calculated the average rank of the statements in this chain in each ranking to determine how well those rankings highlight the effects of the fault that actually cause the failure. For SENSA, we used the *Incremental* strategy, which performed best overall in the study of Section 5.

Naturally, three case studies are insufficient to draw definitive conclusions, but, nevertheless, these cases shed light on the workings of the three techniques that we are studying for a particular application that requires an extensive manual effort to investigate. Next, we present our results and analysis for these case studies.[5]

6.1 NanoXML

In this case for NanoXML, the fault is located in a condition for a *while* loop that processes the characters of the DTD of the input XML document. The execution of this fault by some test cases triggers a failure by failing to completely read the input, which then causes an unhandled exception to be thrown when parsing the next section of the document. The bug and its propagation mechanism are not easy to understand because the exception is thrown from a statement located far away from the fault. After an exhaustive inspection, we manually identified the 30 statements that constitute the entire cause-effect sequence that causes the failure.

All 30 statements that cause the failure—and, thus, will change with the bug fix—are placed by SENSA-*Inc* in the top 11.5% of the ranking, whereas static slicing places them in the top 28.7% and dynamic slicing puts them in the top 73.5%. Also, the average inspection cost, computed with the method of Section 5.2, is 6.8% for SENSA-*Inc*, 9.3% for static slicing, and 24.4% for dynamic slicing. Similar to our findings for many changes in Section 5, dynamic slicing was much less effective as well in this case because of the changes in the execution path that the bug fix causes after it is applied (which singles out the statements that propagated the fault).

We also wanted to understand how well the techniques detect the statements whose behavior changes only because their state and not their execution occurrences. Such statements seem more likely to be highlighted by slicing. Thus, we run the predictions for the 23 statements in this propagation chain that execute in both the buggy and fixed versions of the program. SENSA-*Inc* places these statements at an average rank of 5.33%, whereas static slicing puts them at 8.05% on average and dynamic slicing predicts them at 9.30%, also on average. All techniques showed improvements for this subset of statements, especially dynamic slicing. However, dynamic slicing still performed worse that static slicing, possibly caused by long dynamic dependence distances to the impacted statements.

[5]For full details, see http://nd.edu/~hcai/sensa/casestudies

6.2 XML-security

The bug provided with XML-security is only revealed by one of the 92 unit tests for this subject. This unit test fails in the the buggy version because of an assertion failure in that test due to an unexpected result. Manually tracing the execution backwards from that assertion to the fault location reveals that the fault caused an incorrect signature on the input file via a complex combination of control and data flow. The complete sequence of events for the failure trace contains more than 200 bytecode-like statements. Many of those statements, however, belong to helper functions that, for all practical purposes, work as atomic operations. Therefore, we skipped those functions to obtain a more manageable and focused cause-effect chain that be more easily identified and understood.

For the resulting chain of 55 statements, SENSA-*Inc* places 36 of them in the top 1% of its ranking. A total of 86.3% of those top 1% statements are, in fact, in the chain. In sharp contrast, for the top 4% of its ranking, static slicing only locates 9 of those statements. The cost of inspecting the entire sequence using SENSA is 6.6% of the slice whereas static slicing requires inspecting 33.15% of the slice and dynamic slicing needs 17.9%.

The entire forward static slice consists of 18,926 statements. Thus, users would cover the full sequence of events when reaching 1,255 statements in the ranking of SENSA. With static and dynamic slicing, instead users would have to visit 6,274 and 3,388 statements, respectively. Thus, for this fault, a developer can find the chain of events that makes the assertion fail much faster than using slicing.

6.3 JMeter

For this case study, we chose again the first buggy version provided with JMeter and we picked, from among all 79 unit tests, the test that makes the program fail. The failing point is an assertion check by the end of that unit test. Despite the much larger size of both the subject and the forward slice from this fault, the fault-propagation sequence consists of only 4 statements.

Static slicing ranks two of those statements—half of the sequence—in the top 1% and the other two statements farther away. SENSA, in contrast, places the entire sequence in its top 1%, making it much easier to distinguish the effect from all other possible impacts of the fault. The inspection of the entire failure sequence using static slicing would require a developer to go through 2.6% of the forward slice, or 848 statements. For SENSA, this cost would be only 0.1%, or 32 statements.

As we have frequently observed for dynamic slicing, for this fault, and considering the effects of fixing it, dynamic slicing would cost much more than SENSA and static slicing to identify the fault-propagation sequence. In find all four statements in the sequence, users would have to traverse 12.1% of the slice, which corresponds to 4,094 statements. Once again, this case gives SENSA an advantage over static slicing, and especially over dynamic slicing, in assisting with the location of the failure sequence.

7. RELATED WORK

In preliminary work [31], we described at a high level an early version of SENSA and we showed initial, promising results for it when compared with the predictions from breadth-first traversals of static slices [33,38]. In this paper, we expand our presentation of SENSA, its process, algorithm, and modification strategies. Moreover, we expand our comprehensive study to four Java subjects, we add dynamic slicing to our comparisons, and we present three case studies of cause-effects identification using SENSA on program failures.

A few other techniques discriminate among statements within slices. Two of them [12,39] 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 [33] distinguish statements in slices by pruning control dependencies and pointer-based data dependencies incrementally as requested by the user. Our technique, instead, keeps all statements from the static slice (which is a safe approach) and automatically estimates their influence to help users prioritize their inspections.

Program slicing was introduced as a backward analysis for program comprehension and debugging [38]. Static forward slicing [16] was then proposed for identifying the statements affected by other statements, which can be used for change-impact analysis [7]. 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 static analysis can estimate their relevance to improve the effectiveness of the forward slice. Other forms of slicing have been proposed, such as dynamic slicing [18] and thin slicing [33], that produce smaller backward slices but can miss important statements for many applications. Our technique, in contrast, is designed for forward slicing and does not drop statements but scores them instead.

Dynamic impact analysis techniques [20,26], 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 to the executions observed. Our technique, in contrast, works at the statement level and analyzes the program statically to predict the impacts of changes for any execution, whether those impacts have been observed yet or not. In other words, our technique is predictive, whereas dynamic techniques are descriptive. Yet, in general, static and dynamic techniques complement each other; we intend to investigate that synergy in the future.

8. 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 and tool called SENSA for quantifying statements in slices. This quantification improves plain forward static slices by increasing their effectiveness for change-impact and cause-effects prediction. Rather than pruning statements from slices, SENSA grades statements according to their relevance in a slice.

We plan to extend our studies to more subjects and changes. We are also developing a visualization for quantified slices to improve our understanding of the approach, to enable user studies, and to help 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 slices for other important 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 and data structures to further annotate slices. Such values can also be simplified to discrete sets as needed to improve performance.

9. REFERENCES

[1] M. Acharya and B. Robinson. Practical Change Impact Analysis Based on Static Program Slicing for Industrial Software Systems. In *icseseip*, pages 746–765, May 2011.

[2] H. Agrawal and J. R. Horgan. Dynamic Program Slicing. In *Proc. of ACM Conference on Programming Language Design and Implementation*, pages 246–256, June 1990.

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

[4] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proc. of IEEE/ACM Int'l Conf. on Softw. Eng.*, pages 402–411, May 2005.

[5] D. Binkley, S. Danicic, T. Gyimóthy, M. Harman, A. Kiss, and B. Korel. Theoretical foundations of dynamic program slicing. *Theor. Comp. Sci.*, 360(1):23–41, 2006.

[6] D. Binkley, N. Gold, and M. Harman. An empirical study of static program slice size. *ACM TOSEM*, 16(2), 2007.

[7] 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, June 1996.

[8] G. Canfora, A. Cimitile, and A. D. Lucia. Conditioned program slicing. *Information and Software Technology*, 40(11-12):595–608, Nov. 1998.

[9] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, Apr. 1978.

[10] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Emp. S. Eng.*, 10(4):405–435, 2005.

[11] 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, July 1987.

[12] T. Goradia. Dynamic Impact Analysis: A Cost-effective Technique to Enforce Error-propagation. In *Proc. of ACM Int'l Symposium on Software Testing and Analysis*, pages 171–181, July 1993.

[13] R. Gupta and M. L. Soffa. Hybrid Slicing: An Approach for Refining Static Slices Using Dynamic Information. In *Proc. of ACM Int'l Symposium on Foundations of Software Engineering*, pages 29–40, Oct. 1995.

[14] M. Harman, J. Krinke, J. Ren, and S. Yoo. Search based data sensitivity analysis applied to requirement engineering. In *Proc. of Annual Conference on Genetic and Evolutionary Computation*, GECCO '09, pages 1681–1688, July 2009.

[15] S. Horwitz, B. Liblit, and M. Polishchuk. Better Debugging via Output Tracing and Callstack-sensitive Slicing. *IEEE Transactions on Software Engineering*, 36(1):7–19, 2010.

[16] 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.

[17] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Trans. Softw. Eng.*, 37(5):649–678, 2011.

[18] B. Korel and J. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163, 1988.

[19] J. Krinke. Effects of context on program slicing. *J. Syst. Softw.*, 79(9):1249–1260, 2006.

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

[21] W. Masri and A. Podgurski. Measuring the strength of information flows in programs. *ACM Trans. Softw. Eng. Methodol.*, 19(2):1–33, 2009.

[22] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM TOSEM*, 14(1):1–41, 2005.

[23] M. Mock, D. C. Atkinson, C. Chambers, and S. J. Eggers. Program Slicing with Dynamic Points-to Sets. *IEEE Transactions on Software Engineering*, 31(8):657–678, 2005.

[24] H. Pande, W. Landi, and B. Ryder. Interprocedural def-use associations for C systems with single level pointers. *IEEE TSE*, 20(5):385–403, 1994.

[25] A. Podgurski and L. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Softw. Eng.*, 16(9):965–979, 1990.

[26] 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.*, pages 432–448, Oct. 2004.

[27] G. N. Rodrigues, D. S. Rosenblum, and S. Uchitel. Sensitivity analysis for a scenario-based reliability prediction model. In *Works. Arch. Dep. Sys.*, pages 1–5, May 2005.

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

[29] R. Santelices and M. J. Harrold. Efficiently monitoring data-flow test coverage. In *Proc. of Int'l Conf. on Automated Softw. Eng.*, pages 343–352, Nov. 2007.

[30] R. Santelices, M. J. Harrold, and A. Orso. Precisely detecting runtime change interactions for evolving software. In *Proc. of Third IEEE Int'l Conf. on Softw. Testing, Verification and Validation*, pages 429–438, Apr. 2010.

[31] R. Santelices, Y. Zhang, S. Jiang, H. Cai, and Y. jie Zhang. Quantitative Program Slicing: Separating Statements by Relevance. In *Proc. of IEEE/ACM Int'l Conf. on Softw. Eng., New Ideas and Emerging Results*, May 2013. To appear.

[32] S. Sinha, M. J. Harrold, and G. Rothermel. Interprocedural control dependence. *ACM Trans. Softw. Eng. Method.*, 10(2):209–254, 2001.

[33] M. Sridharan, S. J. Fink, and R. Bodik. Thin slicing. In *Proc. of PLDI*, pages 112–122, June 2007.

[34] W. N. Sumner, T. Bao, and X. Zhang. Selecting peers for execution comparison. In *Proc. of ACM Int'l Symposium on Software Testing and Analysis*, pages 309–319, July 2011.

[35] P. Tsankov, W. Jin, A. Orso, and S. Sinha. Execution hijacking: Improving dynamic analysis by flying off course. In *Proc. of IEEE Int'l Conf. on Software Testing, Verification and Validation*, ICST '11, pages 200–209, Mar. 2011.

[36] R. Vanciu and V. Rajlich. Hidden dependencies in software systems. In *IEEE Int'l Conference on Software Maintenance*, pages 1–10, Sept. 2010.

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

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

[39] X. Zhang, N. Gupta, and R. Gupta. Pruning dynamic slices with confidence. In *PLDI*, pages 169–180, 2006.

[40] X. Zhang and R. Gupta. Matching Execution Histories of Program Versions. In *Proc. of joint Euro. Softw. Eng. Conf. and ACM Int'l Symp. on Found. of Softw. Eng.*, pages 197–206, Sept. 2005.