A Comprehensive Study of the Predictive Accuracy of Dynamic Change-Impact Analysis

Haipeng Cai and Raul Santelices

Department of Computer Science and Engineering
University of Notre Dame, Notre Dame, 46556, Indiana, USA
{*hcai, rsanteli*}*@nd.edu*

Abstract

The correctness of software is affected by its constant changes. For that reason, developers use change-impact analysis to identify early the potential consequences of changing their software. Dynamic impact analysis is a practical technique that identifies potential impacts of changes for representative executions. However, it is unknown how reliable its results are because their accuracy has not been studied.

This paper presents the first comprehensive study of the predictive accuracy of dynamic impact analysis in two complementary ways. First, we use massive numbers of random changes across numerous Java applications to cover all possible change locations. Then, we study more than 100 changes from software repositories, which are representative of developer practices. Our experimental approach uses sensitivity analysis and execution differencing to systematically measure the precision and recall of dynamic impact analysis with respect to the actual impacts observed for these changes.

Our results for both types of changes show that the most cost-effective dynamic impact analysis known is surprisingly inaccurate with an average precision of 38-50% and average recall of 50-56% in most cases. This comprehensive study offers insights on the effectiveness of existing dynamic impact analyses and motivates the future development of more accurate impact analyses.

Keywords: Impact analysis, dynamic analysis, impact prediction, accuracy study, software evolution

1. Introduction

Modern software is increasingly complex and changes constantly, which threatens its quality, reliability, and maintainability. Failing to identify and fix defects caused by software changes can have serious effects in economic and human terms. Therefore, it is crucial to provide developers with effective support to identify dependencies in code and deal with the impacts of changes that propagate via those dependencies. Specifically, developers must understand the risks of modifying a location in a software system *before* they can budget, design, and apply changes there. This activity, called *(predictive) change-impact analysis* [1, 2, 3], can be quite challenging and expensive because changes affect not only the modified parts of the software but also the parts where their effects propagate.

An existing importantapproach to assessing the effects of changes in a program is *dynamic* impact analysis [4, 5, 6, 7, 8, 9, 10, 11]. This approach uses runtime information such as profiles and traces to identify the entities that might be affected by changes under specific conditions—those created by the test suite for that program. The resulting *impact sets* (affected entities) of dynamic approaches that are safe for the execution sets utilized are smaller [6], and thus usually more manageable, than those obtained by safe static analyses as they focus on only a particular subset of all possible inputs (and executions accordingly) [9]. For scalability, most dynamic impact analyses operate on *methods* as the entities that can be changed and be impacted by changes [4, 5, 6, 7, 8, 9, 10, 11]. At the statement level, dynamic slicing [12, 13, 14], in its forward version, can be used for impact analysis in greater detail but at a greater computational cost [6, 7, 15].

Despite its attractiveness, however, dynamic impact analysis has not been evaluated for its ability to *correctly predict the actual impacts* that changes have on software. Techniques exist to *describe* the impacts of changes *after* changes have been made (e.g., [16, 17, 18]). However, for predictive purposes—*before* the changes are even known—the usefulness of dynamic impact sets remains a mystery. For instance, CHIANTI [8] and its applications [19, 20] evaluate their impact analysis results with respect to affected test cases or changes between pairs of program versions, but these approaches are *descriptive* [1] rather than *predictive*. The rest of the literature focuses only on comparing the sizes of dynamic impact sets (i.e., relative precision) and the relative efficiency of the techniques without considering how closely those impact sets approximate the real impacts of changes.

To address this problem, in this paper, we introduce a novel approach for assessing the *accuracy* (precision and recall) of dynamic impact analyses. The approach uses SENSA, a sensitivity-analysis technique we recently developed [21,

22]. We adapted SENSA for making large numbers of random changes efficiently across the software and running dynamic impact analysis on those change locations. While random changes do not necessarily represent all changes, the impacts they find (or not) can help identify *deficiencies in precision and recall* of dynamic impact analyses *across the entire software*. The benefit of this approach is that all methods in a program can be analyzed, in contrast with others based on code repositories which, if available, offer selections of changes that, although supposedly more representative of developer practice, are less comprehensive.

Nevertheless, it is important to also incorporate in a study of impact analysis the changes that developers typically make to complement the comprehensiveness of the new approach with the representativity of real changes. Thus, we designed our approach to support repository changes in addition to the random changes inserted by SENSA. Specifically, our approach takes changes committed by developers into SVN repositories and also changes (bug fixes) from the SIR repository [23] made by other researchers for their own studies.

To find the *ground truth*—the code actually impacted by changes—our approach uses *execution differencing* [17, 18, 24] on the program before and after each change is applied to determine which code is really affected (i.e., code that changes states or occurrences [25]). By design, we use the *same* test suite as the dynamic impact analysis to assess the accuracy of that analysis under the same runtime conditions. The similarities and differences between this ground truth and the impact sets indicate how accurate the evaluated impact analysis can be for predicting actual impacts.

Using this approach with both random and repository-based changes, we performed a comprehensive empirical study of the accuracy of dynamic impact analysis on multiple Java subjects. For dynamic impact analysis, we chose the best known and most cost-effective technique from the literature: PATHIMPACT [4] with *execute-after-sequences* (EAS) [9], which we call PI/EAS. (Another technique, INFLUENCEDYNAMIC [10], is only marginally more precise yet much more expensive, and also more complicated, than PI/EAS.) For different sets of changed methods in each subject, we obtained the impact set predicted by PI/EAS and computed its precision and recall with respect to the ground truth.

The results of our study are surprising. On average for all subjects, the *precision* of the impact sets ranged between 38% and 50% depending on the change type. In other words, at most one in two methods reported by PI/EAS was actually impacted by the studied changes. Moreover, the average *recall* of PI/EAS was about 50–56% except for SIR changes, for which the average recall was 87%. These results reveal that dynamic impact analysis can also miss many real impacts.

Interestingly, the accuracy of PI/EAS was lower for SVN changes, made by developers in practice, than for artificial changes (random and SIR). These results suggest that developers should not expect a great accuracy from existing dynamic impact analyses and that there is plenty of room for improving such techniques.

Our study also showed that, often, the precision was high and the recall was low or vice versa. We hypothesized and confirmed that, when the program execution is shorter *before* a change (when predictive impact analysis is performed) than after a change, runtime effects are missed (e.g., many methods execute only in the changed program). Interestingly, the precision in such cases is greater than usual, suggesting that methods in dynamic impact sets are more likely to be truly impacted if they execute relatively soon after the change.

In all, the main contributions of this paper are:

- An approach for evaluating the accuracy of dynamic change-impact analysis techniques with respect to the actual impacts of source-code changes
- An implementation of the approach that applies massive numbers of changes to support accuracy studies with both artificial and repository changes
- A comprehensive study—the first of its kind—on multiple Java subjects that estimates the accuracy of the most representative and cost-effective dynamic impact analysis known and shows the inadequacy of existing techniques for predicting the effects of changes

The rest of this paper is organized as follows. Section 2 details the problem addressed by, and the motivation of, this work. Section 3 provides the necessary background and a working example. Section 4 discusses the qualities of PI/EAS that affect its accuracy. Then, Section 5 presents our approach for assessing that accuracy with artificial and repository changes. Section 6 and Section 7 present our studies using this approach for both types of changes. Finally, Section 8 discusses related work and Section 9 concludes.

2. Problem and Motivation

The new paradigm of software engineering focuses on software evolution, which is characteristic of incremental changes [26, 27]. One of the two steps of designing incremental changes is impact analysis, a key activity during software development that assesses the full extent of the changes [26, 28]. In fact, several

industrial user studies have also shown that developers widely recognize the crucial role of impact analysis in their daily tasks [29, 30, 31], with views on impact analysis issues varying with different perspectives and organization levels [32].

However, developers face many challenges to impact analysis [29, 33, 31, 34], and one of the most critical issues is the *uncertain* results produced by existing analyses [30, 32]. In addition, an even more critical issue reported by developers is that available analyses are *incomplete* [32]. Taken together, these studies show that developers have already realized and encountered the *inaccuracy* of today's impact analysis in practice. And furthermore, such inaccuracy has been suggested as an issue with existing analysis techniques and tool supports that block their adoption in practice [32, 34].

On the other hand, despite of a large and growing body of research on impact analysis [3, 35], the empirically suggested inaccuracy has not yet been formally studied or systematically quantified [3]. Although a great number of automatic impact-analysis tools have been developed as well (e.g., [4, 5, 10]), the accuracy of most existing impact analyses was evaluated using relative measures only (e.g., the ratios of impact-set sizes of one technique over the other) with respect to the execution sets utilized by the analysis [3]. Particularly, when it comes to *predictive* impact analysis, empirical accuracy measurement with respect to actual impact sets (as ground truth) is still missing.

While predictive impact analysis plays a vital role in driving software evolution as it enables developers to assess potential risks and consequences of candidate changes during the planning phase for the changes, it is equally critical to gauge the accuracy of impact-prediction techniques through a comprehensive and in-depth study. For one thing, reporting false impacts (i.e., imprecise results) potentially causes wastes of time and other resources for developers inspecting the resulting impact sets; for another, failing to report true impacts (i.e., unsafe results) potentially threatens the quality and reliability of the evolving software. Additionally, results from a comprehensive study of this accuracy, and insights to the results, can motivate and guide future development of more advanced techniques for not only impact analysis itself but also its client analyses, such as fault localization [20], regress testing [5], and changeability assessment [36]. Without sufficient understanding of the accuracy of impact analysis, results from these client analyses tend to suffer from severe uncertainty as a consequence.

Specifically, in this work, we focus on the study of predictive dynamic impact analysis, which is usually much less conservative than its static alternatives, and reports potential impact sets with respect to concrete operational profiles of programs. In general, many applications of impact analysis are essentially enabled,

```
1 public class A {
2     static int M1(int f, int z) {
3         M2(f+z);
4         return new B().M3(f,1); }
5     void M2(int m) {
6         if (m > 0)
7             C.M5(); }}
```

```
8 public class B {
9     public static int t=0;
10    int M3(int a, int b) {
11        int n = b*b − a;
12        return n; }
13    static void M4() {
14        t = 10; }}
```

```
15 public class C {
16    public static boolean M5() {
17        return B.t > 10; }
18
19    public static void M0() {
20        if (A.M1(4,−3) > 0)
21            B.M4(); }}
```

PATHIMPACT trace: M0 M1 M2 M5 r r M3 r r r x

EAS first-last events: M0[0,8] M1[1,7] M2[2,4] M3[6,6] M4[-,-] M5[3,3]

Figure 1: The example program E, an example PATHIMPACT runtime trace, and the corresponding EAS *first* and *last* events.

or at least supported, by *dynamic* impact analysis (e.g., program debugging and regression testing) [3].

For our accuracy study, we utilize semantic dependence analysis [25] to obtain the ground truth, as the actual effects of code changes can be captured by *dynamic* semantic dependencies [37] with respect to those changes. Ideally, for a potential change location, we would need to exercise all possible changes at that location to get the full set of dynamic semantic dependencies, which however is impractical. Instead, we use limited numbers of changes to approximate that set, thus to *estimate* the predictive accuracy of dynamic impact analysis. Nevertheless, with a large variety and number of changes applied at each of the locations throughout a program, our study can still give reasonable and informative estimates.

3. Example and Background

Figure 1 shows an example program E used in this paper. In E, the entry method `M0` in class `C` calls methods `M1` and `M4` in classes A and B, respectively. `M1` receives two integers from `M0` and passes the sum to `M2`, which conditionally calls `M5`. Then, `M0` calls `M4`, which sets variable `t` that `M5` may read.

3.1. Dynamic Impact Analysis

Dynamic impact analysis uses execution information to compute change-impact sets (e.g., for a test suite) that estimate the impacts that any changes in a set of program locations might have on the entire program, at least for the executions considered. Developers typically apply the analysis on the unmodified program *before designing and applying any changes* to explore the potential impacts of changing parts of that program. This type of impact analysis is *predictive* [1].

Of existing method-level predictive dynamic impact analyses in the literature, PATHIMPACT [4] with *execute-after sequences* (EAS) [9], which together we call PI/EAS, has shown almost the best precision and about the best efficiency compared with the alternatives [9, 10, 4, 5, 7]. Only one other technique, INFLUENCEDYNAMIC [10], has shown a marginally better precision than PI/EAS but at a much greater cost. Although an efficient implementation of forward dynamic slicing could offer a statement-level option for dynamic impact analysis at a cost that may be acceptable by some developers, we could not include it in our present study due to the lack of known tool support. While some dynamic slicers are available indeed, they work backwardly and require great efforts to adapt for forward slicing. For example, adapting JavaSlicer [38] for such purpose would need redo more than half of its present implementation (as we confirmed with the authors). Nevertheless, we plan to include forward dynamic slicing for dynamic impact analysis in our future study when such tool support becomes available.

PI/EAS computes the impact set of a method m (or a set of methods m) based on the method-execution order of the program. The idea of PI/EAS is that any method called by or returned into its callers after m starts executing might be impacted by a program state modified at m. The original version of PI/EAS, PATHIMPACT, first collects the trace of the events of *entering* and *exiting* each method. Then, PATHIMPACT responds to a query for the impact set of a method m by finding in the trace the set of all methods entered or returned into after entering m. The set includes m itself.

To illustrate, Figure 1 on the bottom left shows an example trace for program E, where `r` indicates a method return, `x` is the program exit, and a method name (e.g., `M1`) represents the entry to that method. For a candidate change location `M2`, for example, PATHIMPACT first traverses the trace forward and identifies `M5` and `M3` as impacted because they are entered after `M2`. In this traversal, PATHIMPACT also counts two unmatched returns. Then, a backward traversal from `M2` finds the two matching methods `M1` and `M0` for those return events. The resulting impact set is {`M0`,`M1`,`M2`,`M3`,`M5`}. PATHIMPACT repeats this process for all occurrences of the candidate method in all traces considered and reports the union of the sets.

EAS improves PATHIMPACT to obtain the same results for a much lower cost. Instead of using traces, EAS only keeps track of the first and last time each method is entered or returned into. From this, we can infer the execution order of all methods and, thus, their dynamic impacts. To illustrate, Figure 1 on the bottom right shows the *first* and *last* values within square brackets for the methods of E. A "timer" starts at 0 and is incremented on each event. The first event for `M2` occurs at time 2 when it is entered. `M4` is not executed so its registers are

uninitialized. All other methods execute after time 2, so the impact set of `M2` is, again, {`M0,M1,M2,M3,M5`}. For another example, the impact set for `M3` is {`M0,M1,M3`} because only the last events for `M0` and `M1` occur after time 6—the time at which `M3` was first entered.

3.2. Execution Differencing

Differential execution analysis (DEA) [17, 18, 24] identifies *semantic dependencies* [25] at runtime of statements on changes—statements *truly affected* by those changes. Formally, a statement s is semantically dependent on a change c and a test suite T if, after change c is made, the *behavior* of s (i.e., the values computed by s or executions of s) changes when running T [18]. The approach requires the execution of the program before and after the change under the same conditions for all sources of non-determinism to ensure that a difference in the behavior of s is, in fact, caused by the change.

Although finding all semantic dependencies in a program is an undecidable problem, DEA detects at least a subset of those dependencies at runtime. With respect to a *concrete input (execution)* I *and change* c, such detected semantic dependence is precise and complete (sound). Thus, given an impact set M computed by any predictive dynamic impact analysis using I on a potential change location L, we can use DEA to obtain the *ground truth* (actual impacts) against M for I and c at L. To do that, DEA compares the execution histories of a program *before* and *after* a change is made. An *execution history* is the ordered sequence of statements executed and the values computed by them. The differences between histories mark the statements whose behaviors change and are thus semantically dependent on the change.

To illustrate, consider in Figure 1 the execution of E starting at `M0` and a change in line 6 to `if (m<0)`. DEA first executes E *before* the change to obtain the execution history

$$\langle 20(\textit{false}),\ 3(1),\ 6(\textit{true}),\ 7(),\ 17(\textit{false}),\ 4(\text{-}3),\ 11(\text{-}3),\ 12(\text{-}3)\rangle$$

Each element $s(v)$ in this sequence indicates that statement s executes and computes value v. DEA then runs the *changed* P and obtains the execution history

$$\langle 20(\textit{false}),\ 3(1),\ 6(\textit{false}),\ 4(\text{-}3),\ 11(\text{-}3),\ 12(\text{-}3)\rangle.$$

Finally, DEA compares the two histories and reports statements 6, 7, and 17 as truly affected at runtime (i.e., semantically dependent on the change) because 6 computes a different value and 7 and 17 execute only before the change is made.

To study method-level impact analyses, we adapted DEA to report all methods containing at least one affected statement. We call this variant MDEA. For our

example change, MDEA finds for `M2` the actually-impacted set {`M2`, `M5`} because those methods contain the affected statements 6, 7, and 17.

3.3. Sensitivity Analysis and SENSA

Sensitivity analysis [39] has been used to measure the influence of one part of a system on other parts and, particularly in software engineering, to relate requirements and components [40, 41]. Mutation analysis is also a form of sensitivity analysis, which is used to simulate common faults in programs so as to assess the ability of a test suite in error detection [42, 43]. Previously, we developed a technique and tool SENSA built on sensitivity analysis and execution differencing to predict and quantify potential impacts of candidate changes [22]. SENSA inputs a program P, a potential change location (statement) s and a test suite T, and outputs a quantified impact set of s.

The process of SENSA is essentially to find the set of statements in P, as the impact set, that are dynamically semantically dependent [37] on s relative to T. To that end, SENSA first instruments state-modification probes at s and then, at runtime, repeatedly changes the value computed at s to produce multiple modified executions. SENSA also runs P to produce the original execution. The last step of the SENSA process is to compute, via execution differencing, the impact set with respect to each modified execution against the original one. By calculating the frequency of each statement occurring in the impact sets from all the modified executions, SENSA obtains the quantified impact set of s.

Although both leveraging sensitivity analysis and modifying program states, SENSA differs from mutation analysis in two main aspects. First, SENSA focuses on particular program points (candidate change locations) to analyze their influences on all relevant statements while mutation analysis checking various points for their influences on program outputs. Second, unlike mutation analysis, SENSA modifications do not involve changing any operators in the program or using mutant operators [44]. By directly modifying the value computed at a specified location, SENSA guarantees producing and using different values in each different modified executions, thus is more efficient, for our study purposes, than mutation analysis, which employs mutations that may not effectively change program states from original ones. For instance, in the example of Figure 1, changing the relation operator at statement 6 will not change the value of the predicate if `m` equals 0.

4. Analytical Assessment

Intuitively, the conservative nature of PI/EAS (defined in Section 3.1) with respect to the analyzed executions can make the analysis imprecise. Moreover, predicting potential change-effects using dynamic analysis on a single program version can also affect the recall of the technique. This section provides an *analytical* examination of the implications of technical definition of PI/EAS of the predictive accuracy of this technique. This examination is a prelude to the comprehensive *empirical* analysis that follow in the next sections and it helps explain those results.

4.1. Precision

PI/EAS relies solely on runtime execution orders to identify, for a method m, its dynamic impact set. At a first glance, the technique seems safe as only methods that execute after m can be affected by m, while producing smaller impact sets than approaches based on runtime coverage [5] for almost the same cost [9]. However, not all methods executed after m are necessarily affected by m. Thus, PI/EAS can be quite imprecise.

For the example of Figure 1, PI/EAS predicts that the dynamic impact set of `M2` is I=`{M0,M1,M2,M3,M5}`. However, when applying the change to `if (m<0)` in line 6 (see Section 3.2), the set of truly-affected methods is M=`{M2,M5}`. Thus, the *predictive* precision of PI/EAS in this case is only $|I \cap M|/|I| = 2/5 = 40\%$. The imprecision is caused by the limited effects of the change, which prevents line 7 from executing and from calling `M5` but has no other consequences. Of course, for other changes in line 6, the precision can reach 100%. Therefore, we must empirically study the precision of PI/EAS for many changes to draw any conclusion.

In general, given a set of executions, PI/EAS can produce large and potentially imprecise impact sets for a method m in a program. This problem occurs when one or more executions continue for a long time after the first occurrence of m and a large number of methods are called or returned into during that process, but only a small portion of those methods are dynamically dependent on m. In some cases, the execution of the program at m goes deep into a call structure but, because of modularity, a change in m propagates only to some of those calls. Similarly, m might be called when the call stack is deep, making PI/EAS mark all methods in that stack as impacted even if many of them are completely unrelated to m in actuality.

4.2. Recall

Naturally, no method that can only execute and return before another method m is called for the first time can be affected by the behavior of m. Therefore, in a *descriptive* sense, PI/EAS has 100% recall. For example, for the program of Figure 1, PI/EAS reports all methods but `M4` as possibly impacted by `M2`, which has 100% recall because `M4` does not execute and, thus, cannot be dynamically impacted. For another example, the impact set for `M3`, which is {`M0,M1,M3`}, also has 100% recall in a descriptive sense because `M2` and `M5` are no longer executing when `M3` starts executing.

However, developers normally need to identify not only the effects of a method on a single version of the program but also the impacts that changing that method can have on the entire program, possibly *before* changes are designed and applied. This is the task of *predictive* impact analysis. A method m' that a dynamic impact analysis does not report as potentially impacted by a method m might be actually impacted by a change in m if that change that affects the control flow of the program such that m' executes after the changed m executes. As a consequence, the recall of a *predictive* dynamic impact analysis can be less than 100%.

To illustrate, consider again our example of Figure 1. If the expression `b*b-a` in line 11 changes to `a*a-b`, the value returned by `M3` and `M1` becomes 15 instead of -3. Thus, the expression at line 20 now evaluates to $true$ and line 21 executes, calling (and impacting) `M4`. However, the dynamic impact set for `M3` does not include `M4`. The change modifies the control flow of E so that `M4`, which did not execute before, now executes after `M3`. In other words, despite not executing before the change, `M3` is dynamically dependent on line 11 because, for the same input and a change in line 11, `M3` changes its execution behavior.[1] Thus, the predictive recall of PI/EAS for this example change is 75%.

4.3. Accuracy

To be useful in addition to efficient, a dynamic impact analysis must be accurate. Typically, neither a good precision nor a good recall alone is enough. Rather, a good balance is desired. On one hand, PI/EAS achieves 100% recall for a method m if *all* methods execute after m but only a few of them are truly impacted, which yields a low precision. On the other hand, if the program halts at method m, PI/EAS predicts an impact set $\{m\}$ for m with 100% precision but,

[1] More generally, a method that executes fewer or more times for the same input after a change, is dynamically impacted by that change [25].

after changes to m, many methods might execute after m so yielding a low recall. Therefore, in this paper, we also use an *F-measure* [45] to estimate the balance of PI/EAS. We use the first such measure:

$$F1 = 2 \times \frac{precision \times recall}{precision + recall}$$

To illustrate, consider again the change of line 6 in method `M2` to `if (m<0)` in our example. As we saw in Section 4.1, PI/EAS produces an impact set of {`M0,M1,M2,M3,M5`} for `M2`, whereas the actual impact set for this change is {`M2,M5`}. Thus, precision is 40% and recall is 100%, whereas the accuracy is $2\times(0.4\times1.0)/(0.4 + 1.0) = 57.1\%$. For another example, for the change in the expression in statement 11 in method `M3` from `b*b-a` to `a*a-b` (see Section 4.2), the precision is 100% and recall is 75%, for an accuracy of 85.7%.

4.4. Exception Handling

The PI/EAS approach, as published [9], can suffer from unpredictable results in the presence of unhandled exceptions that can make the runtime technique miss *return* or *returned-into* events. To process such events, PI/EAS assumes that an exception raised in a method m is caught by a *catch* or *finally* block in m before m exits or in the method that called the instance of m that raised the exception. However, this assumption does not hold for many software systems, including some of those studied in this paper.

If neither method m nor a sequence of (transitive) callers of m handle an exception thrown by m, the *returned-into* events for m and all methods in the call stack that do not handle the exception will be missed. As a result, those methods will not be added to the resulting impact set. To illustrate, in Figure 1, if an exception is raised in `M3`, it will not be handled. Thus, the *last* records for `M1` and `M0` will not be updated to reflect that they were returned into after `M3` exited abnormally, and the impact set for `M3` will miss `M1` and `M0`.

For our work, we decided to fix this problem by developing an improved version of PI/EAS that accounts for unhandled exceptions. Our design captures all return or returned-into events by wrapping all methods in special try-catch blocks. Those blocks catch unhandled exceptions, process the events that would otherwise be missed, and re-throw those exceptions. In the rest of this paper, whenever we mention PI/EAS, we refer to the version of this technique corrected by us.

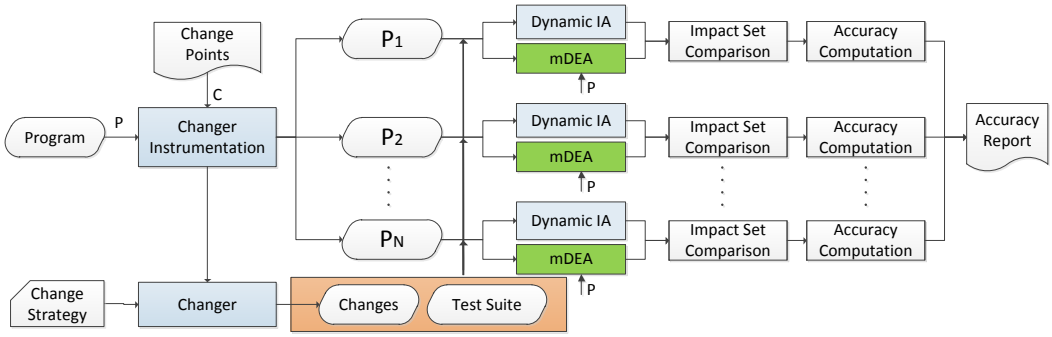

Figure 2: Process for estimating the accuracy of dynamic impact analyses through sensitivity analysis and execution differencing.

5. Experimental Approach

For a comprehensive study of the *predictive* accuracy of dynamic impact analysis, we consider both *artificial* and *repository* changes as these two types of changes complement each other. Repositories reflect how software evolves in practice but usually contain changes to only a fraction of all methods in the software. Artificial changes, in contrast, might not represent software evolution in practice but have two key benefits: they can be generated massively to cover all *analyzable* change locations (i.e., those executed at least once) and can reveal potential deficiencies in precision and recall.

In order to generate artificial changes for all analyzable methods, we used our sensitivity-analysis tool SENSA [21, 22] which makes random modifications to the code of each method. We call these changes SENSA *changes*. We also used as changes various bug fixes from the SIR repository [23] introduced by other researchers to study realistic faults. Finally, we retrieved numerous changes from popular SVN open-source repositories. As opposed to the SENSA changes, we refer to the changes from both SIR and SVN repositories as *repository changes*. (The SVN changes are, nevertheless, the repository changes we deem as most representative of developer practice.)

In this paper, we study the accuracy of PI/EAS with SENSA and repository changes separately. Next, we describe the experimental approach for each part of our study.[2] Also, to assess the validity of this approach for obtaining accuracy

[2]The entire toolkit that implemented our study framework is available to the public for download at `http://nd.edu/~hcai/deam`

results, we conducted two small case studies in which we manually determined the ground truth, for sample program inputs and impact-set queries, and checked it against the ground truth used by our experimental framework.

5.1. Approach with SENSA *changes*

To estimate the range of accuracy that developers can expect with the predictive dynamic impact analysis, we designed an approach that (1) systematically applies impact analysis to a large number of candidate change locations throughout the program, (2) changes those locations in groups of one to ten methods at a time, and (3) compares the predicted impact sets with the actual impacts found by MDEA as the ground truth.

5.1.1. Process

Figure 2 outlines our experimental approach. The process uses a *Changer* module that, for each change location (a statement in a method) in a location set C for program P, performs a number of changes in that location to produce one version of the program per change. For greater realism, each changed program version is treated as the *unchanged* (base) program for predictive impact analysis and the original P is treated as the "fixed", changed version. In other words, the changes can be seen as bug fixes.

The changer first instruments P at the locations C to produce a large number N of base (unchanged) versions of P called P_1 to P_N. Then, at runtime, the instrumentation in P invokes the changer for the points in C to produce the N base versions, one at a time, across which the C locations (statements) are distributed. A *change strategy* is provided for customization. By default, this strategy is *random*, which replaces the values or control-flow decisions computed at each change point with random values of the same type.

The replacement values generated for each execution of P are stored so that each base execution of P can be reproduced. Unlike similar tools, to speed up the process by avoiding disk-space blowup, our system uses only two versions of the program: the original P and the instrumented P controlled at runtime by the changer. At runtime, using the test suite provided with P, the approach applies dynamic impact analysis (dynamic IA) to each of the N base versions to obtain, for each method that contains at least one change location, its dynamic impact set. Then, MDEA is applied to that version and P with the same test suite to find the actual impacts (ground truth). We use the same test suite *on purpose* so we can compare predicted and actual impacts under the same runtime conditions.

In the last step, our approach compares the dynamic impact set of each changed method against the ground truth calculated by MDEA to determine the predictive precision, recall, and accuracy (F1 measure) of that impact set. The last step, shown on the right of Figure 2, computes statistics of these accuracy results for the final report for the subject program P.

5.1.2. Generation of Base Versions

At the core of our approach is the generation of N base versions from P. Every base version consists of P with one or more modifications, each made to one statement per method. These base program versions are the ones to which impact analysis is applied and P is the "fixed" (changed) version. To make the study comprehensive, the system selects the change set C to cover as much code and methods in P as possible. The approach compares the predicted impact set of each such method against the actual impacts of *all* changes in C located in that method to compute accuracy metrics.

To implement the changer, we adapted SENSA [21, 22], our sensitivity-analysis technique and tool for Java bytecode. SENSA can modify values of primitive types and strings in assignments and branching conditions. We call statements whose value can be modified by SENSA *modifiable* statements; other statements not supported by SENSA are normally affected directly by those supported by SENSA. Although most heap-object values are not directly supported now, any supported value within a heap object can be modified by SENSA at the location where that value is computed.[3] Therefore, the change set C is selected from all statements to which SENSA is *applicable*. We regard a method with no applicable statements as *non-applicable*.

The goal of our approach is to change every applicable statement in the program at least once. However, this can be impractical for large subjects. Therefore, we choose a well-distributed subset of those statements according to per-method limits L and L_{max} which default to 5 and 10, respectively. For each method m and applicable-statements set A_m in m, the change location set C_m for m is A_m if $|A_m| \leqslant L$. Otherwise, the size of C_m is limited to $min(|A_m|, L_{max})$ to ensure that at most L_{max} locations are used in m. In the latter case, to evenly cover m, the system splits the method into L_{max} segments of equal length (rounded). For each segment i of consecutive applicable statements in positions $\lceil max(0, i -$

[3]Modifications involving other unsupported values require applying changes not directly to that location but to the statement(s) that compute the supported parts of such values.

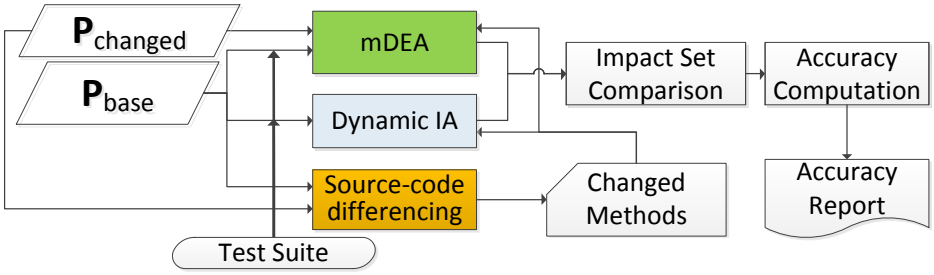

Figure 3: Experimental process for evaluating the accuracy of dynamic impact analyses with repository changes.

$1)\times|A_m|/L_{max}\rceil$ to $\lceil max(1,i)\times|A_m|/L_{max}\rceil$, the system randomly picks for C_m one statement in that segment. The union of all sets C_m is the set C.

5.2. Approach with Repository Changes

To evaluate the accuracy of dynamic impact analysis for the types of changes made in practice by developers (SVN repositories) and researchers (the SIR repository [23]), we created an experimental pipeline customized for each subject application of our study. This system (1) retrieves from the repository, configures, and compiles a series of software versions, (2) finds the set of all changed methods between each pair of consecutive versions, (3) computes the *predicted* and *actual* impacts of those changed methods using dynamic impact analysis (e.g., PI/EAS) on the first version and MDEA on both versions, respectively, and (4) calculates the predictive accuracy of the dynamic impact sets.

We describe first the experimental process for our study with repository changes and then our approach for finding the changed methods.

5.2.1. Process

The experiment process for studying repository changes is shown in Figure 3. For each pair of base version P_{base} and changed version $P_{changed}$ of a subject program P, the process first retrieves the source code from the repository and then uses a special differencing tool to find the set M of methods changed from P_{base} to $P_{changed}$. Next, it runs dynamic impact analysis (*Dynamic IA*) on the base version to produce the predictive impact set for M and MDEA to obtain the actual impacts, both using the test suite provided with the subject. The last step computes the accuracy of the impact set for these two versions of P. This process reuses some of the modules for dynamic IA, MDEA, and accuracy computation presented in Figure 2 for SENSA changes.

Since we focus on the study of *code* impact analysis, we ignored non-source changes such as updates in software documentation and changes in code comments. The process automatically checks against program versions that did not contain any source-code changes for a better performance than doing that during the retrieval of changed methods.

5.2.2. Retrieval of Changes

In contrast with the SENSA changes, which were seeded by our system to generate the base versions, repository changes are not immediately available as they have to be computed from existing sources like the Apache project's SVN servers and the SIR repository. The retrieval of repository changes is thus an essential step for this part of the approach. To that end, we developed a tool to help find changed methods between two given software versions. The tool finds the sets of methods added, deleted, and modified between two versions with the aid of a source-code differencer.

5.3. Execution Differencing as Ground Truth

As we calculate the predictive accuracy of dynamic impact analysis relative to actual impacts that are automatically produced by our framework using execution differencing as ground truth, it is crucial to check that such ground truth is acceptable with respect to the executions utilized for computing the impact sets under evaluation. To that end, we chose two of the seven subjects used in our empirical evaluation, Schedule1 and NanoXML, and performed one small case study for each one of them to analyze the obtained ground truths. Next, we summarize the study procedure and report the results.[4]

5.3.1. Procedure

Our case studies covered both single-method and multiple-method changes. For each study, we randomly selected two candidate change locations consisting of single methods and one change location consisting of multiple methods. To keep our manual analysis of change impacts manageable, we constrained these locations to those for which PI/EAS produced impact sets with at most 20 methods. In addition, for each change location, we randomly chose three test inputs from all the inputs that cover that change location.

Our manual identification of the *ground-truth* impact sets was based on our detailed understanding of the code structure, functionality, and purpose of the

[4]Full details are available at `http://nd.edu/~hcai/deam/manuinsp.html`

subjects. Furthermore, for each input and change, we first manually traced the execution of the subject starting from the change location, recording the outcome of control-flow decisions and collecting values computed at each statement. Next, we applied a random change at that location and repeated that process. Then, we determined the methods actually impacted by that change according to our understanding of the source code and the effects of the change on control flows and values. We utilized the Eclipse debugger to assist our effort.

5.3.2. Results

Our results show that, for all the changes and inputs we studied, the ground-truth impact sets manually determined constantly conformed to those computed by our framework using the MDEA technique. Although the order in which actually impacted methods were discovered during our manual examination was not always the same as that in which those methods were identified by MDEA from the execution-history differences, the eventual impact sets produced as ground truth from both approaches were consistent in all cases. Accordingly, the accuracy of PI/EAS relative to the manual ground truth for each corresponding impact set was always equal to that relative to the MDEA-based ground truth.

For Schedule1, for instance, the first single-method change was to modify a floating-point variable in a method at the core of this subject, which occurred deep in the call stack for executions on all the three inputs. As a result, 18 out of the total 24 methods in this program executed after the query, which were all identified as impacted by PI/EAS. However, only 8 of those 18 methods were actually impacted by the change according to our manual investigation, which were the same as all the ones found by MDEA; the other 10 were not really affected because they did not use that floating-point variable either directly or transitively.

Similarly, changing a loop variable in the second single-method case did not affect most of the methods executed after the change location, because the change, although escaped to three callees of the queried method, did not propagate further along or back to any callers. MDEA reported rightly just those three callees, which were exactly what we manually found as the ground truth in this case. In contrast, PI/EAS falsely included 13 additional methods in its impact set.

In the case of multiple-method changes, modifications to single statements in each of the two changed methods affected together many more methods than each change alone did. Yet, three methods were identified falsely as impacted by PI/EAS. Two of them were initializers that used constants only. The third false positive was the entry method that transitively called the two changed methods without using any of the changed values afterwards. These false-positive impacts

were included neither in our manual ground truth nor by MDEA in its actual impact set. On the other hand, the manual examination and MDEA both found the same set of true impacts.

Results from the cases with NanoXML were very similar to those with Schedule1 above, although most of the true impact sets were smaller. Constantly, MDEA found all actually impacted methods without including any false impacts in its results, according to our manual verification. While it happened to be more accurate with the first single-method change, PI/EAS suffered from low precision in the other two cases for the same reason as mentioned in the Schedule1 cases. Finally, as we expected, with both subjects, PI/EAS had slightly higher accuracy for multiple-method changes than for single-method ones in most cases. This is consistent with what we observed from the empirical study results (in Section 6.3).

In sum, for all the cases we studied, MDEA produced actual impact sets always the same as those we manually obtained. As we examined only a few cases for two relatively small subjects, results from these case studies may not be generalizable. Nevertheless, since we have carefully verified the conformance of MDEA results to manual ground truth, for both single-method and multiple-method changes with randomly chosen program inputs and queries, these results suggest that it is reasonably valid to evaluate the predictive accuracy of PI/EAS using MDEA-based ground truth.

6. Study with SENSA Changes

We first present our study of the predictive accuracy of PI/EAS using SENSA changes. Because changes can happen to a single method or multiple methods at once, we consider both single-method and multiple-method changes in this study.

6.1. Experimental Setup

For our SENSA-change studies, we chose eight Java subjects of a variety of sizes, complexities, and functionalities. Most of these subjects are widely-used, nontrivial open-source applications. We used the entire test suites provided with these subjects except for PDFBox, for which we considered 29 of its 32 test cases. The three remaining test cases cause our MDEA implementation to run out of memory (even on an 80GB RAM machine). Table 1 lists these subjects with brief descriptions and their statistics, including their sizes in non-comment non-blank lines of Java source code (*#LOC*), the total number of methods (*#Methods*), and the number of test cases (*#Tests*).

The first four subjects and JMeter came from the SIR repository for software testing studies. When applicable, the subject name includes the SIR version. Schedule1 is representative of small modules for specific tasks. NanoXML is a lean and efficient XML parser. Ant is a popular cross-platform build tool. XML-security is an encryption and signature component of the Apache project. JMeter is an Apache application for performance assessment of software. We obtained the other three subjects from their respective source-code repositories, some of which are still evolving. We chose a stable version of the subject from each repository. BCEL is the Apache library that analyzes and manipulates binary Java class files. PDFBox is a PDF document processing tool from the Apache project. The last subject, ArgoUML, is a UML modeling tool.

The subjects Ant, BCEL, JMeter, and ArgoUML exhibit some non-determinism—a few varying behaviors for the same test inputs—due to their use of the system time and random number generators. To ensure that MDEA does not report false-positive differences caused by non-determinism, we manually *determinized* these subjects by ensuring that, for each test case, the subjects used the same sequence of the system time and randomly generated values before and after each change. To assure that we did not accidentally break those subjects, at least for their test suites, we re-run those test suites on the determinized versions. We found no differences in outputs and assertion evaluations between the determinized and corresponding original versions.

We implemented our approach in Java to analyze the subjects in Java bytecode, as described in Section 5. We also implemented PI/EAS according to Section 3 with our exception-handling correction described in Section 4.4. We built this infrastructure on top of our Java-bytecode analysis and monitoring framework DUA-FORENSICS [46, 47], which is based on Soot [48], and our sensitivity-analysis tool SENSA [22].

We also found that some changes alter the length of the base executions considerably. To better understand the effects of those kinds of changes, our implementation classifies changes for which the number of PI/EAS events in the base version is 50% or less than in the changed version (the "fixed" program) as *shortening* (S) and the rest as *normal* (N).

6.2. Part I: Single-method Changes

For this study, we used our approach described in Section 5.1, where each base version is applied one change in one method at a time (i.e., the reverse of the SENSA change that "fixes" the program). Table 2 summarizes the results of this study. For each subject, the table reports the average precision, recall, and

Table 1: Experimental subjects and their characteristics

Subject	Description	#LOC	#Methods	#Tests
Schedule1	priority scheduler	290	24	2,650
NanoXML-v1	XML parser	3,521	282	214
Ant-v0	Java build tool	18,830	1,863	112
XML-security-v1	encryption library	22,361	1,928	92
BCEL 5.3	byte code analyzer	34,839	3,834	75
JMeter-v2	performance test tool	35,547	3,054	79
PDFBox 1.1	PDF processing tool	59,576	5,401	29
ArgoUML-r3121	UML modeling tool	102,400	8,856	211

accuracy for all changes in that subject. Since the data points were collected per change, methods that contain larger numbers of applicable change locations are better represented in those results. This is appropriate because those methods contain more locations that developers could change. The second column (*Scope*) indicates how much of the program is truly studied, which corresponds to the percentage of all statements belonging to the methods that are called at least once at runtime and contain at least one modifiable statement.

The row for each subject has three sub-rows, each named after the type of change (*C.T.*) in the third column: *All* (both normal and shortening), *N* (normal only), and *S* (shortening only). The extent of the changes made to each subject per category is indicated by the fourth column (*C.S.*) for the total number of executed and changed statements and the fifth column (*C.M.*) for the number of methods containing at least one changed statement. Note that the sums of numbers of methods for categories *N* and *S* can be greater than for *All* because some methods contain both *N* and *S* changes.

Next, the table shows the accuracy results per subject and change category, starting with the average number of impacted methods found by PI/EAS (*P.S.*) and the average number of actually-impacted methods identified by MDEA (*A.S.*) The next two columns show the average number of false positives (*#FP*) and false negatives (*#FN*) for PI/EAS with respect to the actual impacts. Finally, the last three columns show the average precision, recall, and accuracy (*F1*) of PI/EAS for the subject and change category. Each of those columns presents the mean and its 95% confidence interval (*conf. range*) obtained via the *non-parametric* Vysochanskij-Petunin inequality [49], which makes *no assumptions* about the normality of the data distribution.

To illustrate, consider the results for JMeter, for which 78% of its code was in methods that contained one or more changes, which were those analyzable by PI/EAS and MDEA. Of the 1439 statements on which changes were studied, distributed across 401 methods, 198 of them, distributed across 82 methods, con-

Table 2: Predictive accuracy of PI/EAS for single-method SENSA changes, for all changes and their two subsets: normal (N) and shortening (S).

Subject	Scope	C.T.	C.S.	C.M.	P.S.	A.S.	#FP	#FN	Precision		Recall		Accuracy (F1)	
									mean	conf. range	mean	conf. range	mean	conf. range
Schedule1	82%	**all**	**46**	**12**	**16.1**	**13.4**	**4.5**	**1.8**	**0.73**	**[0.59, 0.87]**	**0.90**	**[0.81, 0.99]**	**0.72**	**[0.59, 0.85]**
		N	12	6	16.7	5.8	11.0	0.2	0.33	[0.09, 0.56]	0.99	[0.95, 1.00]	0.43	[0.17, 0.69]
		S	34	12	15.9	16.0	2.2	2.4	0.87	[0.77, 0.97]	0.87	[0.75, 0.99]	0.83	[0.72, 0.93]
NanoXML -v1	85%	**all**	**379**	**129**	**74.6**	**54.2**	**39.3**	**18.8**	**0.46**	**[0.40, 0.52]**	**0.73**	**[0.68, 0.79]**	**0.40**	**[0.34, 0.46]**
		N	181	78	81.2	18.0	64.6	1.4	0.24	[0.16, 0.31]	0.95	[0.91, 0.98]	0.27	[0.20, 0.34]
		S	198	107	34.1	82.2	3.3	51.3	0.73	[0.66, 0.81]	0.31	[0.25, 0.36]	0.41	[0.35, 0.47]
Ant -v0	77%	**all**	**437**	**121**	**21.8**	**78.2**	**9.2**	**65.7**	**0.67**	**[0.62, 0.73]**	**0.53**	**[0.48, 0.58]**	**0.39**	**[0.35, 0.44]**
		N	381	102	17.6	27.6	8.8	18.8	0.65	[0.59, 0.71]	0.59	[0.54, 0.64]	0.43	[0.38, 0.48]
		S	56	22	49.9	422.4	12.4	384.8	0.81	[0.70, 0.91]	0.11	[0.05, 0.17]	0.14	[0.09, 0.20]
XML-security -v1	80%	**all**	**1405**	**297**	**149.1**	**208.1**	**53.8**	**112.8**	**0.70**	**[0.67, 0.73]**	**0.42**	**[0.39, 0.45]**	**0.40**	**[0.38, 0.43]**
		N	843	218	127.2	112.9	77.3	63.0	0.56	[0.52, 0.60]	0.38	[0.35, 0.42]	0.31	[0.28, 0.34]
		S	562	122	182.0	350.9	18.6	187.6	0.91	[0.89, 0.93]	0.47	[0.43, 0.52]	0.54	[0.50, 0.58]
BCEL 5.3	77%	**all**	**1523**	**436**	**257.8**	**88.5**	**232.3**	**63.0**	**0.29**	**[0.27, 0.32]**	**0.59**	**[0.56, 0.62]**	**0.17**	**[0.16, 0.18]**
		N	1345	399	266.3	35.9	250.6	20.2	0.25	[0.22, 0.27]	0.63	[0.60, 0.66]	0.16	[0.15, 0.18]
		S	178	86	193.4	485.7	94.3	386.5	0.65	[0.58, 0.73]	0.27	[0.19, 0.34]	0.23	[0.18, 0.29]
JMeter -v2	78%	**all**	**1439**	**401**	**81.6**	**51.6**	**54.5**	**24.4**	**0.42**	**[0.39, 0.44]**	**0.58**	**[0.56, 0.61]**	**0.38**	**[0.36, 0.40]**
		N	1241	357	78.7	32.5	57.1	10.8	0.38	[0.35, 0.41]	0.60	[0.58, 0.63]	0.37	[0.35, 0.40]
		S	198	82	99.9	171.3	38.4	109.7	0.66	[0.60, 0.72]	0.44	[0.37, 0.51]	0.42	[0.36, 0.48]
PDFBox 1.1	67%	**all**	**1092**	**268**	**131.7**	**144.3**	**75.0**	**87.6**	**0.58**	**[0.55, 0.61]**	**0.45**	**[0.42, 0.48]**	**0.35**	**[0.33, 0.38]**
		N	749	228	158.4	76.7	102.4	20.6	0.46	[0.43, 0.50]	0.55	[0.52, 0.59]	0.39	[0.37, 0.42]
		S	343	127	73.3	292.1	15.3	234.0	0.83	[0.80, 0.86]	0.22	[0.18, 0.26]	0.27	[0.23, 0.31]
ArgoUML -r3121	70%	**all**	**1239**	**421**	**81.6**	**51.4**	**56.9**	**26.6**	**0.39**	**[0.36, 0.41]**	**0.65**	**[0.63, 0.67]**	**0.37**	**[0.35, 0.39]**
		N	1043	371	77.3	33.3	58.8	14.8	0.34	[0.32, 0.37]	0.68	[0.65, 0.71]	0.35	[0.34, 0.37]
		S	196	70	104.9	147.6	46.9	89.7	0.62	[0.55, 0.70]	0.49	[0.43, 0.54]	0.44	[0.38, 0.49]
Overall (all subjects)	77%	**all**	**7560**	**2085**	**72.2**	**58.9**	**52.0**	**36.5**	**0.48**	**[0.47, 0.49]**	**0.55**	**[0.54, 0.56]**	**0.34**	**[0.33, 0.35]**
		N	5795	1759	68.5	30.6	58.4	13.4	0.39	[0.38, 0.41]	0.60	[0.58, 0.61]	0.31	[0.30, 0.32]
		S	1765	628	85.7	175.2	25.1	128.2	0.78	[0.77, 0.80]	0.38	[0.36, 0.40]	0.41	[0.39, 0.43]

tained changes that shortened the executions of the base program to less than half. On average for JMeter, the PI/EAS impact set had 81.6 methods, the actual impacts were 51.6, and the false positives and negatives of PI/EAS were 54.5 and 24.4 methods, respectively. The average precision of PI/EAS was 0.42 with 95% confidence that its real value is *not* outside the range [0.39, 0.44]. Recall and accuracy are presented similarly.

The last row presents the overall results for *all* changes in all subjects, so that every change has the same weight in those results. Thus, subjects with more changes (column *#Changed Statements*) have a greater influence in those results. Overall, the changed methods covered 77% of the code even though these methods were only a fraction of all methods in the subjects (see Table 1). This means that the methods that never executed or for which SENSA was not applicable were much smaller than the average. In total, the study spanned over 7500 changes. More than 3 in 4 of them were *normal*.

For all changes, on average, the precision of PI/EAS was 0.48, its recall was 0.55, and its accuracy was only 0.34. The non-parametric statistical analysis shows with 95% confidence that these values are no farther than 2 percentage points from the real value. (For individual subjects, which have fewer data points, the confidence ranges are wider). These numbers indicate that only a bit less than one in two methods reported by PI/EAS are actually impacted by those changes. Also, almost one in two methods truly impacted were missed by PI/EAS (low recall). Although, on average, the PI/EAS impact sets were close in size to the actual impact sets, the large numbers of false positives and false negatives led to a low accuracy. Thus, we conclude with high statistical confidence that, at least for these SENSA changes, the accuracy of PI/EAS is low. Hence, for many practical scenarios, dynamic impact analysis appears to need considerable improvements.

For a more detailed view of the accuracy of PI/EAS, Figures 4–6 present the distribution of the precision (*prec*), recall (*rec*), and F1 accuracy (*acc*) of all subjects for changes in *all*, *N*, and *S* categories, respectively. Each box plot shows the minimum (lower whisker), the 25% quartile (bottom of middle box), the 75% quartile (top of middle box), and the maximum (upper whisker) of the three metrics, respectively. The medians are marked by horizontal lines within the middle boxes. The vertical axis of each box plot represents the values of the metrics—precision, recall, and accuracy—for all changes in the corresponding subject.

For Schedule1, the simplest subject, the precision, recall, and accuracy were the highest of all subjects. This may be explained by the smaller number of methods that share many global variable accesses in Schedule1, which makes any change likely to truly impact the methods executed after it, and that control flows were not changed much thus most truly impacted methods executed after the change location. The box plots for Schedule1 also show the concentration on the top of the accuracy values for its 46 changes. NanoXML also had a high recall, possibly for similar reasons to Schedule1, but its precision was low—less than half the methods that execute after the change were truly impacted. This low precision suggests that NanoXML performs a larger number of independent tasks (so that changes to one task do not affect all other tasks).

For the largest six subjects, the average recall was much lower than what was observed with the two smallest ones, ranging from 0.42 to 0.65, suggesting that changes in them have greater effects on their control flow because missed methods (false negatives) not called after the execution of change locations in base versions are actually executed in changed versions after those locations. In other words, there seem to be many methods that execute under specific conditions satisfied only in changed program versions.

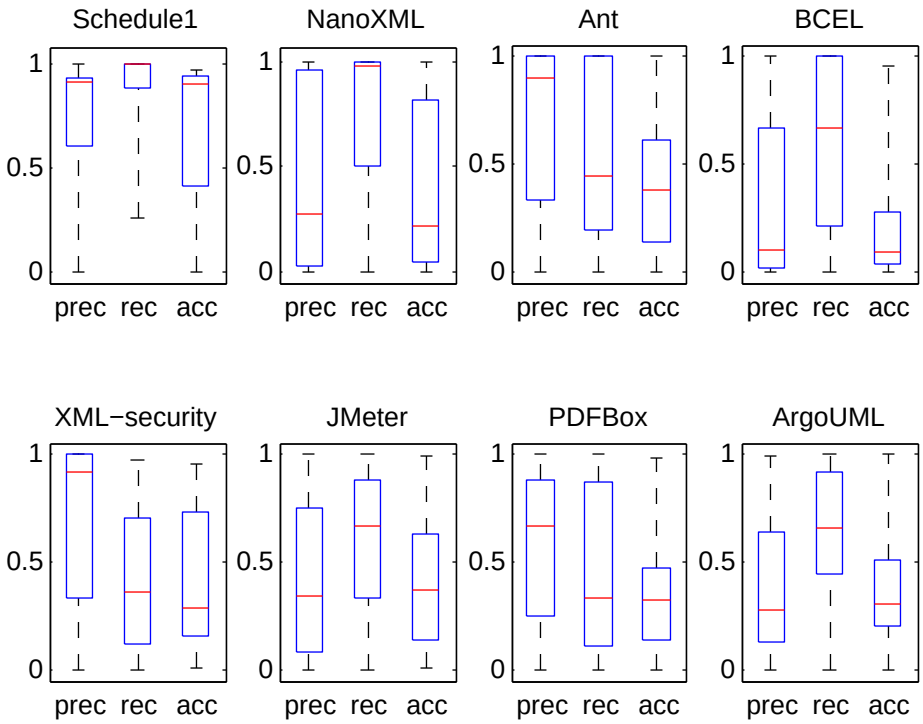

Figure 4: Distribution of accuracy of PI/EAS for *all* single-method SENSA changes.

As for precision, Ant and XML-security had a greater value than NanoXML and closer to Schedule1, suggesting that the degree of propagation of the effects of changes in those subjects was high, possibly by performing sequences of tasks that feed into each other. Among the four largest subjects, ArgoUML and BCEL had the lowest precision, suggesting that their internal tasks are less coupled. However, PDFBox exhibited a relatively high precision compared with other large subjects, likely due to the tight couplings among its internal components, which closely collaborate for its centralized style of PDF-document processing.

When considering the N and S categories separately, we can see that the changes in N usually have a higher recall than changes in S. This result was expected, as *normal* base versions of the subjects execute more methods and, therefore, have larger predictive impact sets found by PI/EAS. At the same time, the precision for S was greater than for N, which was also expected because the shorter executions analyzed by PI/EAS correspond to methods executed soon after each change. We conjecture that closer methods are more related to the changed method and, thus, actually impacted.

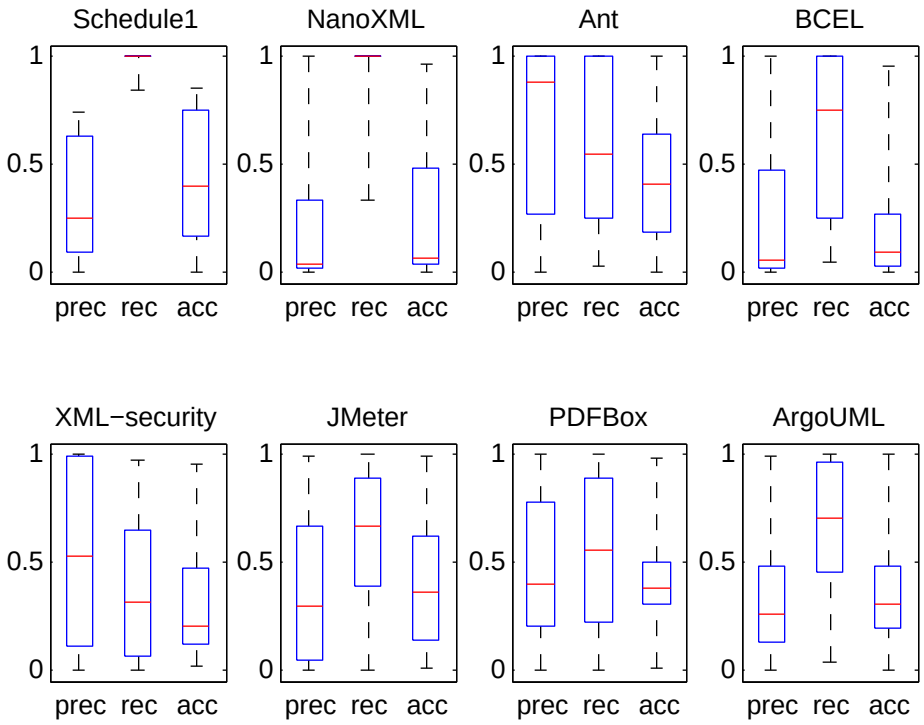

Figure 5: Distribution of accuracy of PI/EAS for *normal* (non-shortening) single-method SENSA changes.

The recall trend for N and S, however, does not apply to XML-security, where the recall is lower for N than for S. To understand this phenomenon, we manually examined the source code and executions of this subject by randomly picking five change points of type N with a recall below .0001. These changes, by definition of N, had traces of similar length before and after each change. However, the traces diverged for the most part after each change, making the actual impacts very different from the predicted impact sets. In contrast, for the changes of type S in this subject, the recall was greater, suggesting that in reality these changes did not affect the control flow of the program too dramatically.

6.3. Part II: Multiple-method Changes

Developers often change more than one method. To estimate the accuracy of PI/EAS with multiple-method changes, we used the same experimental approach as for the single-method study, but with each base version having multiple methods changed, where one statement was changed in each changed method. Given

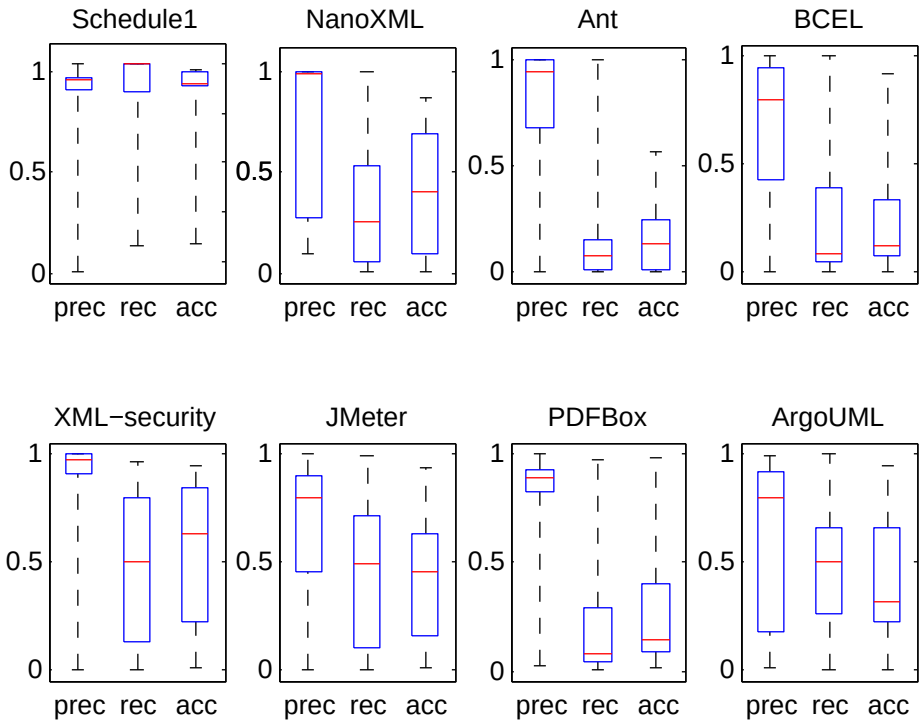

Figure 6: Distribution of accuracy of PI/EAS for *shortening* single-method SENSA changes.

the set C of all applicable change locations in the subject program P, our system *randomly* picks g locations from C, as *evenly distributed* across g different methods as possible (i.e., g locations were chosen such that each different location is within a different method, with one change per method). Then, we generate a base version by applying those changes to the subject. After picking g methods without replacement, we repeated the process by picking g new methods at a time until no more methods are left to change. To reduce the potential noise from grouping g methods randomly, we repeated the entire process *three times* and averaged the results. We performed our study for all values of g from 2 to 10.

Table 3: Predictive accuracy of PI/EAS for all multiple-method SENSA changes, for query (change-set) sizes from 1 to 10.

Subject	**#Methods changed**	**P.S.**	**A.S.**	**#FP**	**#FN**	**Precision**		**Recall**		**Accuracy (F1)**	
						mean	conf. range	mean	conf. range	mean	conf. range
Schedule1	1	16.1	13.4	4.5	1.8	0.73	[0.59, 0.87]	0.90	[0.81, 0.99]	0.72	[0.59, 0.85]
	2	16.3	17.0	2.1	2.7	0.87	[0.84, 0.91]	0.85	[0.77, 0.93]	0.83	[0.76, 0.89]
	3	15.4	17.6	1.8	4.0	0.88	[0.84, 0.91]	0.78	[0.68, 0.89]	0.79	[0.71, 0.87]
	4	15.3	17.9	1.7	4.2	0.88	[0.85, 0.91]	0.77	[0.65, 0.90]	0.79	[0.70, 0.88]

Table 3: *Continued*

Subject	#Methods changed	P.S.	A.S.	#FP	#FN	Precision		Recall		Accuracy (F1)	
						mean	conf. range	mean	conf. range	mean	conf. range
	5	14.6	17.9	1.7	5.0	0.87	[0.82, 0.91]	0.73	[0.58, 0.89]	0.75	[0.63, 0.87]
	6	13.2	18.1	1.5	6.4	0.85	[0.79, 0.91]	0.66	[0.47, 0.84]	0.69	[0.55, 0.84]
	7	13.9	18.2	1.5	5.8	0.88	[0.82, 0.93]	0.69	[0.51, 0.88]	0.73	[0.59, 0.87]
	8	13.0	18.3	1.6	6.8	0.84	[0.77, 0.92]	0.63	[0.42, 0.85]	0.68	[0.50, 0.85]
	9	11.5	18.3	1.4	8.2	0.82	[0.73, 0.91]	0.56	[0.30, 0.83]	0.61	[0.39, 0.83]
	10	11.7	18.4	1.4	8.1	0.86	[0.78, 0.93]	0.57	[0.33, 0.81]	0.64	[0.45, 0.82]
NanoXML	1	74.6	54.2	39.3	18.8	0.46	[0.40, 0.52]	0.73	[0.68, 0.79]	0.40	[0.34, 0.46]
	2	119.2	82.6	50.6	14.0	0.60	[0.56, 0.63]	0.86	[0.83, 0.88]	0.62	[0.59, 0.65]
	3	134.5	97.1	51.8	14.4	0.65	[0.60, 0.69]	0.89	[0.86, 0.91]	0.68	[0.64, 0.71]
	4	141.3	106.6	50.2	15.5	0.68	[0.63, 0.72]	0.89	[0.86, 0.92]	0.71	[0.67, 0.74]
	5	145.8	109.1	53.7	17.0	0.67	[0.62, 0.72]	0.89	[0.84, 0.93]	0.69	[0.65, 0.74]
	6	146.3	115.5	49.4	18.6	0.69	[0.65, 0.74]	0.88	[0.83, 0.92]	0.71	[0.66, 0.75]
	7	148.2	116.0	52.0	19.8	0.68	[0.63, 0.73]	0.88	[0.83, 0.92]	0.70	[0.65, 0.75]
	8	142.9	119.7	47.5	24.3	0.71	[0.65, 0.78]	0.84	[0.77, 0.91]	0.69	[0.62, 0.75]
	9	140.5	121.6	45.9	27.0	0.71	[0.65, 0.77]	0.82	[0.76, 0.89]	0.68	[0.61, 0.74]
	10	141.8	119.3	50.0	27.5	0.70	[0.64, 0.76]	0.83	[0.76, 0.90]	0.67	[0.60, 0.73]
Ant	1	21.8	78.2	9.2	65.7	0.67	[0.62, 0.73]	0.53	[0.48, 0.58]	0.39	[0.35, 0.44]
	2	35.7	113.8	17.2	95.3	0.59	[0.55, 0.63]	0.54	[0.49, 0.58]	0.36	[0.32, 0.40]
	3	37.9	120.0	18.1	100.2	0.59	[0.54, 0.63]	0.52	[0.48, 0.57]	0.35	[0.31, 0.39]
	4	39.6	126.3	18.8	105.6	0.59	[0.55, 0.64]	0.52	[0.48, 0.57]	0.36	[0.32, 0.40]
	5	40.3	129.4	18.8	107.8	0.59	[0.55, 0.64]	0.51	[0.46, 0.56]	0.36	[0.31, 0.40]
	6	43.4	141.7	19.9	118.2	0.58	[0.54, 0.63]	0.50	[0.45, 0.55]	0.35	[0.30, 0.39]
	7	45.3	148.6	20.4	123.7	0.60	[0.55, 0.65]	0.49	[0.43, 0.54]	0.36	[0.31, 0.40]
	8	48.8	157.6	22.3	131.1	0.58	[0.53, 0.63]	0.48	[0.43, 0.54]	0.34	[0.30, 0.39]
	9	50.4	166.5	22.8	138.8	0.59	[0.54, 0.64]	0.48	[0.42, 0.53]	0.35	[0.30, 0.39]
	10	52.1	170.2	23.5	141.7	0.58	[0.53, 0.63]	0.46	[0.41, 0.52]	0.33	[0.29, 0.38]
XML-security	1	149.1	208.1	53.8	112.8	0.70	[0.67, 0.73]	0.42	[0.39, 0.45]	0.40	[0.38, 0.43]
	2	164.7	190.4	62.3	88.0	0.64	[0.62, 0.66]	0.49	[0.46, 0.51]	0.44	[0.42, 0.47]
	3	183.9	218.5	63.5	98.1	0.67	[0.65, 0.69]	0.52	[0.49, 0.55]	0.48	[0.46, 0.51]
	4	206.4	241.5	68.5	103.6	0.68	[0.66, 0.71]	0.55	[0.53, 0.58]	0.52	[0.49, 0.54]
	5	220.6	267.6	65.7	112.7	0.71	[0.68, 0.73]	0.56	[0.53, 0.59]	0.54	[0.51, 0.57]
	6	230.1	283.3	68.7	121.9	0.72	[0.70, 0.75]	0.57	[0.54, 0.60]	0.55	[0.52, 0.58]
	7	241.8	303.9	69.4	131.5	0.74	[0.71, 0.76]	0.57	[0.54, 0.61]	0.56	[0.53, 0.59]
	8	251.4	314.5	71.7	134.8	0.74	[0.71, 0.76]	0.59	[0.55, 0.62]	0.57	[0.54, 0.60]
	9	265.4	337.1	69.4	141.1	0.76	[0.74, 0.79]	0.60	[0.56, 0.63]	0.59	[0.56, 0.62]
	10	270.3	344.3	79.1	153.1	0.74	[0.71, 0.77]	0.58	[0.54, 0.62]	0.57	[0.54, 0.60]
BCEL	1	257.8	88.5	232.3	63.0	0.29	[0.24, 0.34]	0.59	[0.55, 0.63]	0.17	[0.14, 0.20]
	2	271.8	156.3	191.5	76.0	0.39	[0.37, 0.42]	0.64	[0.61, 0.66]	0.30	[0.28, 0.32]
	3	284.7	163.9	201.9	81.0	0.39	[0.36, 0.41]	0.64	[0.61, 0.66]	0.30	[0.28, 0.32]
	4	296.4	182.7	200.7	86.9	0.40	[0.38, 0.43]	0.64	[0.61, 0.66]	0.32	[0.30, 0.34]
	5	311.3	178.7	219.7	87.1	0.38	[0.35, 0.40]	0.66	[0.63, 0.68]	0.31	[0.28, 0.33]
	6	315.6	190.3	218.8	93.4	0.39	[0.36, 0.42]	0.65	[0.62, 0.67]	0.32	[0.29, 0.34]
	7	325.2	205.3	219.5	99.5	0.40	[0.37, 0.43]	0.63	[0.61, 0.66]	0.33	[0.30, 0.35]
	8	340.9	218.2	226.2	103.5	0.40	[0.37, 0.43]	0.65	[0.62, 0.68]	0.34	[0.31, 0.36]
	9	349.9	229.7	227.4	107.3	0.40	[0.37, 0.43]	0.65	[0.62, 0.68]	0.34	[0.32, 0.37]
	10	354.4	237.3	227.6	110.5	0.41	[0.38, 0.44]	0.65	[0.62, 0.68]	0.36	[0.33, 0.38]
JMeter	1	81.6	51.6	54.5	24.4	0.42	[0.39, 0.44]	0.58	[0.56, 0.61]	0.38	[0.36, 0.40]
	2	90.0	62.0	57.8	29.8	0.42	[0.39, 0.45]	0.43	[0.41, 0.46]	0.35	[0.33, 0.37]
	3	98.9	67.7	62.8	31.7	0.42	[0.39, 0.45]	0.46	[0.43, 0.48]	0.36	[0.33, 0.38]
	4	108.3	73.0	68.1	32.9	0.42	[0.39, 0.45]	0.48	[0.46, 0.51]	0.37	[0.35, 0.40]
	5	118.0	78.6	73.0	33.7	0.42	[0.39, 0.46]	0.50	[0.48, 0.53]	0.39	[0.36, 0.42]
	6	123.5	84.0	76.1	36.6	0.43	[0.40, 0.46]	0.51	[0.48, 0.54]	0.39	[0.37, 0.42]
	7	134.9	90.7	81.3	37.1	0.43	[0.40, 0.46]	0.53	[0.50, 0.55]	0.40	[0.38, 0.43]
	8	146.6	98.8	87.1	39.3	0.43	[0.39, 0.46]	0.55	[0.52, 0.58]	0.42	[0.39, 0.45]

Table 3: *Continued*

Subject	#Methods changed	P.S.	A.S.	#FP	#FN	Precision		Recall		Accuracy (F1)	
						mean	conf. range	mean	conf. range	mean	conf. range
	9	152.5	103.7	89.6	40.9	0.44	[0.40, 0.47]	0.56	[0.53, 0.59]	0.43	[0.40, 0.46]
	10	161.7	109.2	94.0	41.5	0.44	[0.40, 0.47]	0.57	[0.54, 0.60]	0.44	[0.41, 0.47]
PDFBox	1	131.7	144.3	75.0	87.6	0.58	[0.54, 0.62]	0.45	[0.40, 0.50]	0.35	[0.32, 0.39]
	2	117.7	151.2	64.5	98.1	0.59	[0.57, 0.61]	0.41	[0.38, 0.44]	0.34	[0.32, 0.37]
	3	117.9	157.0	63.7	102.8	0.60	[0.57, 0.62]	0.40	[0.37, 0.43]	0.34	[0.32, 0.37]
	4	119.9	161.9	63.6	105.7	0.60	[0.58, 0.63]	0.41	[0.37, 0.44]	0.35	[0.32, 0.37]
	5	142.6	159.9	81.5	98.8	0.57	[0.55, 0.60]	0.45	[0.42, 0.48]	0.36	[0.33, 0.38]
	6	146.1	168.9	81.8	104.6	0.59	[0.56, 0.61]	0.46	[0.43, 0.49]	0.36	[0.34, 0.39]
	7	145.8	171.0	81.9	107.1	0.58	[0.56, 0.61]	0.45	[0.42, 0.49]	0.36	[0.33, 0.38]
	8	152.7	176.5	84.8	108.5	0.58	[0.55, 0.61]	0.46	[0.43, 0.49]	0.36	[0.33, 0.39]
	9	156.9	182.4	87.0	112.5	0.59	[0.56, 0.62]	0.47	[0.43, 0.50]	0.37	[0.34, 0.39]
	10	160.7	185.0	89.1	113.3	0.59	[0.56, 0.62]	0.47	[0.44, 0.51]	0.37	[0.34, 0.39]
ArgoUML	1	81.6	51.4	56.9	26.6	0.39	[0.36, 0.41]	0.65	[0.63, 0.67]	0.37	[0.35, 0.39]
	2	107.8	78.0	74.0	44.2	0.38	[0.36, 0.41]	0.53	[0.51, 0.55]	0.34	[0.32, 0.36]
	3	111.2	80.7	75.8	45.3	0.38	[0.36, 0.41]	0.53	[0.51, 0.55]	0.34	[0.32, 0.36]
	4	114.7	82.7	77.9	46.0	0.39	[0.36, 0.41]	0.54	[0.51, 0.56]	0.35	[0.33, 0.37]
	5	118.6	85.5	80.9	47.8	0.39	[0.36, 0.41]	0.53	[0.51, 0.56]	0.35	[0.33, 0.37]
	6	120.2	86.9	81.3	48.1	0.39	[0.36, 0.41]	0.54	[0.52, 0.56]	0.35	[0.33, 0.37]
	7	124.5	89.2	84.4	49.1	0.38	[0.36, 0.41]	0.54	[0.52, 0.57]	0.35	[0.33, 0.37]
	8	128.8	93.2	86.9	51.3	0.39	[0.36, 0.42]	0.54	[0.52, 0.56]	0.35	[0.33, 0.37]
	9	130.4	95.3	87.4	52.3	0.39	[0.37, 0.42]	0.55	[0.52, 0.57]	0.36	[0.33, 0.38]
	10	135.6	98.5	90.7	53.6	0.40	[0.37, 0.42]	0.55	[0.53, 0.57]	0.36	[0.34, 0.38]

Table 4: Predictive accuracy of PI/EAS for all multiple-method SENSA changes in all subjects

Subject	#Methods changed	P.S.	A.S.	#FP	#FN	Precision		Recall		Accuracy (F1)	
						mean	conf. range	mean	conf. range	mean	conf. range
Overall (all subjects)	1	72.2	58.9	52.0	36.5	0.48	[0.47, 0.49]	0.55	[0.54, 0.56]	0.34	[0.33, 0.35]
	2	147.4	124.7	88.7	66.0	0.49	[0.48, 0.50]	0.53	[0.52, 0.54]	0.37	[0.36, 0.38]
	3	155.0	133.0	92.7	70.8	0.49	[0.48, 0.50]	0.53	[0.52, 0.55]	0.37	[0.36, 0.38]
	4	161.8	141.8	94.3	74.3	0.50	[0.49, 0.51]	0.54	[0.53, 0.55]	0.38	[0.37, 0.39]
	5	172.0	144.6	102.7	75.4	0.49	[0.48, 0.50]	0.55	[0.54, 0.56]	0.38	[0.37, 0.39]
	6	174.5	150.8	103.0	79.3	0.49	[0.48, 0.51]	0.55	[0.54, 0.57]	0.38	[0.37, 0.40]
	7	179.0	157.0	104.5	82.5	0.49	[0.48, 0.51]	0.55	[0.54, 0.56]	0.39	[0.37, 0.40]
	8	186.3	163.7	108.2	85.6	0.49	[0.48, 0.50]	0.56	[0.54, 0.57]	0.39	[0.38, 0.40]
	9	190.3	170.0	108.6	88.2	0.50	[0.48, 0.51]	0.56	[0.55, 0.57]	0.39	[0.38, 0.41]
	10	192.8	172.8	110.4	90.4	0.50	[0.48, 0.51]	0.56	[0.55, 0.57]	0.39	[0.38, 0.41]

Table 3 shows the results of this study for all changes, in a format similar to Table 2 except that, for each subject, the results are listed per value of g (*#Methods changed*) for all changes instead of three categories. For comprehensibility, we do not include data per change category (N and S) although we comment on these later in this section. To facilitate comparisons, we do include the results for the single-method study (g=1) for *all* changes. All metrics were calculated in the same way as for single-method changes, including non-parametric 95%-confidence intervals for average precision, recall, and F1 accuracy.

To illustrate, consider the results for BCEL in Table 3. On average for all studied 10-method changes, for example, PI/EAS produced impact sets with 354.4 methods where 227.6 were false positives and 110.5 false negatives, leading to a precision of .41, recall of .65, and accuracy of .36.

For all subjects but Schedule1, both the impact sets reported by PI/EAS and the actual impacts kept growing as the number of changed methods increased. This correlation was expected because more changes often lead to larger aggregate impacts. For Schedule1, however, PI/EAS reported shrinking impact sets and decreasing recalls when more methods were changed, which can be explained by the small size of this subject, that a large fraction of its methods are already impacted by single changes, and that changes in Schedule1 *shortened* its executions while, as g grew, there were not many methods left to add to the aggregate (multiple-method) impact sets.

The numbers of false positives and false negatives tended to vary proportionally with the impact set sizes. As a result, the precision and recall numbers were relatively stable for most subjects for change sizes 2 to 10, with only small fluctuations. Interestingly, there was often a noticeable fluctuation between changes of size 1 and size greater than 1. Other than that, there were two noteworthy exceptions to the stability of the results. First, the recall for Schedule1 decreased almost steadily due to its decreasing execution lengths, and second, precision tended to increase considerably with the change size for NanoXML and, less dramatically, for XML-security and BCEL. Also, for each change size, the F1 accuracy was generally stable with tight confidence intervals.

In absolute terms, PI/EAS had the smallest precisions, below .5, for three of the four largest subjects: BCEL, JMeter, and ArgoUML. These low precisions suggest that these larger subjects execute many methods that are not necessarily related to (impacted by) each other, even when up to 10 methods are changed. Thus, these subjects seem to exercise different functionality at the same time that do not directly relate to each other, unlike PDFBox which appears to operate as a batch process where each step feeds into the next.

The observed comparisons and correlations suggest some subtle effects of the characteristics of programs on the accuracy of PI/EAS for those programs. Example such characteristics include the coupling among internal components and the sensitivity of test executions to random changes.

Overall, for multiple-method changes, PI/EAS still suffered from poor predictive accuracy, as it did for single-method ones. This technique often reported impact sets where half of its elements were false positives and missed about half of the actual impacts. Interestingly, although the sizes of impact sets given by

PI/EAS and those of the true impacts consistently grew as expected with the number of changed methods, the three accuracy metrics were quite stable with increasingly narrow confidence intervals. These results suggest that PI/EAS falls short for predicting the actual impacts of changes of different sizes.

6.4. Implications of the Results

The goal of this study was to assess, for as many change locations as possible, the predictive accuracy of the most cost-effective dynamic impact analysis technique in the literature. The quality of impact analysis is critical to tasks such as regression testing and maintenance. Despite a few differences between single-method and multiple-method changes, the results show that the predictive accuracy of PI/EAS can be surprisingly low. Although we cannot generalize the levels of inaccuracy observed with SENSA (random) changes, these numbers cast serious doubts on the effectiveness and practicality of PI/EAS when considering the possibility of changing almost any part of a program.

From these results, we first conclude that, at least for these subjects and these types of changes, the *precision* of PI/EAS can indeed suffer. A likely reason seems to be that this technique is quite conservative. It assumes that all methods executed during or after the execution of changed methods are *infected* [50, 51] by the change (i.e., carry on an affecting modification in the program state). In practice, however, methods can execute for different purposes and their order of execution does not always imply dependence.

Moreover, while PI/EAS appears at a first glance to be safe relative for the executions analyzed [9], our studies revealed that, for predictive purposes, PI/EAS also suffers from low recall. This drawback of dynamic impact analysis, which is orthogonal to the problem of not having a sufficient variety of executions, has not been emphasized in the past. Further, since the recall is measured with respect to the inputs utilized by PI/EAS in our study, adding more tests would not change the recall for those inputs, which come with the subjects we used in this work. On the other hand, the recall problem is particularly important for cases in which the unchanged program has relatively short executions, such as when there is a crashing bug that needs to be fixed. Therefore, developers who consider using PI/EAS may first want to determine whether a change would lengthen the execution of the program under analysis with respect to the inputs utilized.

6.5. Threats to Validity

The main *internal* threat to the validity of our conclusions is the possibility of implementation errors in our infrastructure, especially in the new mod-

ules PI/EAS and MDEA. However, both are built on top of our analysis and instrumentation framework DUA-FORENSICS, which has been in development for many years [46, 47] and has matured considerably, and SENSA, which has been in development for two years already [21, 22]. DUA-FORENSICS and SENSA have been carefully tested over time.

A second internal threat is the possibility for procedural errors in our use of the infrastructure, including our scripts for running experiments and analyzing data. To reduce this risk, we tested, inspected, debugged, and manually checked results from all phases of our process.

A third internal threat is the determinization process for Ant, BCEL, JMeter, and ArgoUML. It is possible that errors were introduced in these subjects. Therefore, we inspected and validated the determinized subjects by checking that their execution behavior and semantics of the programs were not affected—at least for the test suites we used. We compared the outcomes of all test cases for the determinized and original versions of these subjects, which are not supposed to depend on their non-determinism, and found no differences. To confirm that we did not miss other sources of non-determinism, we run the determinized programs multiple times and used MDEA to look for differences among them. We found no such differences. Many programs are almost impossible to determinize, but this is a limitation of the approach and not of our studies here. This limitation, however, could affect the application of our approach to other subjects in the future.

The main *external* threat to our studies is the representativeness of the changes that we used (fixes to random SENSA modifications). Yet, these changes directly implement the concept of right-hand-side function replacements to show semantic dependencies [25], which is ultimately what dynamic impact analysis looks for. Moreover, we studied a very large number of those changes distributed evenly across every subject.

Another *external* threat is the representativeness of our selection of subjects and test suites with respect to software in general. To limit this threat, we chose subjects of different sizes, coding styles, and functionality to maximize variety in our studies within our available resources. Most of our subjects are nontrivial, widely used, professionally developed, and have dozens if not hundreds of test cases covering most of their functionalities.

The main *construct* threat to the validity of our studies lies in the design of our approach and the ability of random modifications and their fixes to produce similar effects to other changes that can be made to software. While actual changes made by developers might be found in source-code repositories, our design ensured that *all* parts of the program were studied. (In a typical repository, only a fraction

of the system changes, even for long periods of time.) More importantly, the primary goal of this study was to find whether inaccuracy exists in dynamic impact analysis. Also, we made sure that the modifications had a real effect, even if just local, on the program at runtime. The changes studied represent at least a subset of all changes that developers can make.

Also, as discussed in Section 5.1.2, our changes were limited to the value types that SENSA currently can change, which are primitive types and strings. Nevertheless, almost all other values in a program can be directly affected by the values modifiable by SENSA. In addition, despite this limitation, the SENSA-based changes were evenly spread, covered a large portion of each subject, and allowed our study to include most non-trivial methods. Thus, we studied most parts on which a developer might run predictive impact analysis.

Another *construct* threat is the possibility of errors in the MDEA implementation we used to find the actual impacts of changes (the ground truth). To minimize this threat, we have tested and debugged this tool over four years [18] and we used subjects for which all or most test cases do not run for such a long time that the data produced cannot be analyzed by MDEA. We also applied MDEA to a subject with the same test suite used for dynamic impact analysis so that the same operational profile was considered.

Finally, a *conclusion* threat is the appropriateness of our statistical analysis and our data points. To minimize this threat, for the two parts of this study, we used a *non-parametric* analysis [49] that computes confidence ranges without making any assumptions about the normality of the data distribution. This is the safest way to statistically analyze any data set. In addition, we studied the precision and recall metrics as well as the F1 accuracy metric. We included the quartile distributions of all the data points for the single-method changes study. Also, for diversity of the data, we distributed the change points across each subject as evenly as SENSA permitted.

7. Study with Repository Changes

The study using SENSA changes enabled us to exhaustively estimate the predictive accuracy of dynamic impact analysis, as it considers all methods that can possibly be changed, with random changes applied to many locations in each method. Yet, such random changes might not necessarily represent those that developers make in practice. Thus, to complement that study for a truly comprehensive evaluation, it is necessary to investigate the accuracy with repository changes

as well, even though they are not available in the same numbers and coverage of the subjects as the SENSA changes.

This section presents our study with changes from software repositories. We focus on open-source repositories to measure the practical accuracy of PI/EAS with actual changes made by developers when developing and maintaining their software. We also use the SIR repository [23] to further assess the accuracy of PI/EAS with changes made by other researchers.

7.1. Experimental Setup

For our repository-change study, we chose the repositories for three Apache projects. We also picked four subjects from SIR. For the Apache projects, we checked out a contiguous series of subversion (SVN) revisions for each of the three projects. We started with the versions corresponding to the stable releases used for our study in Section 6 (Table 1). For the SIR-change study, we obtained seven single-method changes per subject directly from the seeded faults in the SIR repository. Similar to the SENSA-changes study, the SIR changes studied in this section correspond to the fixes of the faults inserted in those subjects.

To produce the method-level execution traces required by PI/EAS, for the SVN-change study, we adopted the test suite provided with the first version of the series we chose for each project and used that same test suite for all revisions of that project. For the SIR-change study, we used the test suite provided by SIR. We applied the same determinization process as in Section 6 to all revisions of Ant and JMeter to ensure that MDEA identifies actual impacts caused by only the changes committed to the repositories.

To obtain sufficient data points for our SVN study, for each project, we begun with a set of 30 consecutive revisions and kept adding more sets of consecutive revisions until we found at least a total of 30 revisions that contained source-code changes, excluding comments and declarations, that were covered by the test suite (and, thus, analyzable). For each pair of consecutive analyzable revisions, we treated the *set of all methods* changed between them as the location set to which we applied PI/EAS and the change for which we computed predictive accuracy using MDEA.

Table 5 gives the statistics of the three Apache projects used in our SVN-change study, including the revision range (*Revision range*), the number of revisions examined (*#Revisions examined*), the number of revisions covered by the test suite for the respective project (*#Revisions analyzed*), and the number of methods changed per covered revision (*#Methods changed*). For each pair of revisions, we considered methods that are deleted or modified as changed methods. We

omitted added methods as they do not exist in the base (unchanged) revision and, thus, cannot be queried by PI/EAS.

We implemented a tool for downloading SVN revisions and finding changed methods between those by comparing the abstract syntax trees generated from source code using the srcML [52] library. In our experiment, we obtained the differences for all revisions but we studied only those revisions with differences in *executable source code* (not comments or declarations) covered by *at least one test case* of the corresponding test suite. Then, for these analyzable revisions, we computed the same accuracy measures of the impact sets as in the SENSA-change study. For this SVN-change study, we do not report results for *shortening* changes because no change reduced any execution trace by a significant amount.

Table 5: Statistics of the subjects for the SVN-change study

Subject	**Revision range**	**#Revisions examined**	**#Revisions analyzed**	**#Methods changed**	
				mean	stdev
Ant	269450–269758	90	45	9.8	13.0
XML-security	350550–350859	60	32	25.8	87.1
PDFBox	924515–1038227	180	39	7.4	18.5
Total		**330**	**116**	**13.4**	**47.6**

7.2. Part I: SVN Changes

Table 6: Predictive accuracy of PI/EAS using SVN changes.

Subject	**Pairs of Revisions**	**C.M.**	**P.S.**	**A.S.**	**#FP**	**#FN**	**Precision**		**Recall**		**Accuracy (F1)**	
							mean	conf. range	mean	conf. range	mean	conf. range
Ant	44	429	432.3	224.3	323.0	115.1	0.24	[0.12, 0.35]	0.56	[0.41, 0.71]	0.22	[0.13, 0.31]
XML-security	31	799	293.5	394.5	101.1	202.1	0.60	[0.43, 0.76]	0.57	[0.44, 0.70]	0.48	[0.36, 0.60]
PDFBox	38	282	265.3	245.8	159.5	140.1	0.38	[0.28, 0.48]	0.38	[0.28, 0.49]	0.35	[0.26, 0.44]
Overall (all subjects)	**113**	**1510**	**338.0**	**278.2**	**207.1**	**147.4**	**0.38**	**[0.30, 0.46]**	**0.50**	**[0.42, 0.58]**	**0.34**	**[0.27, 0.40]**

In this study, for each pair of consecutive revisions per subject, we followed the experimental approach described in Section 5.2. We designated the older of the two revisions as the *base* version (P_{base}) and the newer one the *changed* version ($P_{changed}$). As already mentioned, we treated the set of all methods that were modified or deleted in the base version to obtain the changed version as the change locations for applying dynamic impact analysis (PI/EAS) and MDEA.

As Table 5 shows, we considered a total of more than 300 commits of source-code changes made by developers to three open-source projects which are still in

active development. Among those commits, we report the impact-analysis results of the changed methods for all *analyzable* revisions (i.e., revisions whose changes are executed by the respective test suite), for a total of over 100 revisions. In other words, about two thirds of all revisions that we examined only have changes in declarations, comments, or untested methods.

For each project, a few analyzable revisions had exactly one method changed with respect to their preceding analyzable revisions. Most revisions, however, contained multiple-method changes, with a few of them having *several hundred* of methods changed. Table 5 shows that, for all 116 analyzable revisions, on average, more than 13 methods changed. The large standard deviation of 47.6 suggests that the developers of these projects tended to vary considerably the scope, in methods, of the changes they made.

Table 6 lists the results of this study, where the second column (*Pairs of Revisions*) is the number of data points that contributed to the statistics in the columns to the right on that table for each subject. As mentioned earlier, each of these pairs consists of two consecutive *analyzable* revisions. The third column (*C.M.*) indicates the set of all methods changed between the first and last analyzable revisions, which may or may not include methods changed in non-analyzable revisions committed in between. The remaining columns show the average accuracy statistics (precision, recall, and F1) and non-parametric 95%-confidence intervals.

Overall, the predictive accuracy for PI/EAS on SVN changes was quite low. Precision was even lower than recall. The results in the last row for all changes in the three subjects show that PI/EAS predicted, on average, more impacts than the actual impacts caused by these changes but the majority of those impacts were false positives. An exception was the set of changes in XML-security, for which PI/EAS predicted fewer impacts than the actual impacts observed and had the highest average precision among the three subjects. Nevertheless, PI/EAS for this subject missed 43% of the actual impacts.

Individually, PI/EAS had the lowest precision for the smallest subject, Ant, and a recall for this subject as low as that for XML-security. This very-low precision caused the F1 accuracy to also be the lowest of the three subjects. PI/EAS for the largest subject in this study, PDFBox, had a more balanced average precision and recall. However, both measures were still quite low and recall in particular was much lower than that for the other two subjects. In all, PI/EAS only predicted about half of the actual impacts while reporting impact sets with more than 60% false positives. The confidence intervals in this study were wider than for the SENSA-change study because fewer data points were available. Another possible factor is the large variation in size of these SVN changes.

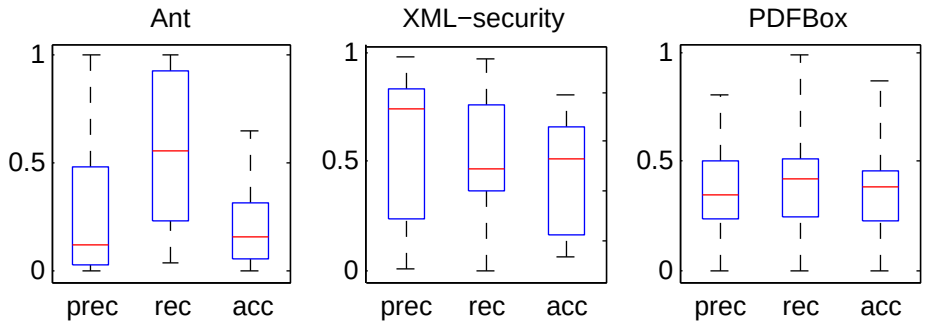

Figure 7: Distribution of accuracy of PI/EAS for *all* SVN changes.

To offer a more detailed view of the accuracy results for all revision pairs Figure 7 uses boxplots to indicate how the precision (*prec*), recall (*rec*) and accuracy (*acc*) distributed over all data points. The quartiles in these distributions confirm the overall tendencies shown in Table 6, where the predictive accuracy of PI/EAS was low for the three subjects studied. For instance, 75% of the revision pairs of Ant had a precision lower than 50 but more than half of all pairs had more than 50% recall, although the accuracy was still below 25% for most data points. The plots also show that, for PDFBox, its low average accuracy is explained by the majority of data points being below 50% for all three metrics, whereas, for XML-security, the middle 50% of the data points with respect to precision and recall explain the averages observed for that subject.

An important observation is that the recall of PI/EAS for SVN changes, similar to SENSA changes, was quite low even though the SVN changes *did not shorten* any execution. For PDFBox, in particular, PI/EAS had an even lower average recall for SVN changes than for SENSA changes (both single- and multiple-method ones). This observation confirms that the poor safety of PI/EAS found for random changes is not accidental and that it might not be attributed simply to the effects of changes on the length of executions.

Finally, for every subject of this study, PI/EAS exhibited an even worse average precision than for both types of SENSA changes, with the largest such gap observed for Ant. In all, we can conclude based on both studies that PI/EAS can suffer from very-low predictive accuracy.

7.3. Part II: SIR Changes

To complement and help compare our SENSA-change and SVN-change study results, we performed an additional study with changes corresponding to the fixes

Table 7: Predictive accuracy of PI/EAS using SIR changes.

Subject	#Chgs	C.M.	P.S.	A.S.	#FP	#FN	Precision		Recall		Accuracy (F1)	
							mean	conf. range	mean	conf. range	mean	conf. range
Schedule1	7	4	18.0	10.1	7.9	0.0	0.56	[0.38, 0.74]	1.00	[1.00, 1.00]	0.71	[0.57, 0.85]
NanoXML-v1	7	7	96.7	30.1	66.9	0.3	0.39	[0.01, 0.80]	0.99	[0.98, 1.00]	0.48	[0.09, 0.88]
XML-security-v1	7	6	189.7	159.4	119.3	89.0	0.53	[0.13, 0.92]	0.64	[0.24, 1.00]	0.40	[0.18, 0.61]
JMeter-v2	7	7	37.1	17.4	22.6	2.9	0.40	[0.05, 0.75]	0.84	[0.52, 1.00]	0.43	[0.12, 0.74]
Overall (all subjects)	**28**	**24**	**85.4**	**54.3**	**54.1**	**23.0**	**0.47**	**[0.30, 0.64]**	**0.87**	**[0.72, 1.00]**	**0.51**	**[0.36, 0.66]**

of faults introduced by other researchers for their own studies and shared at the SIR repository [23]. For this study, we chose from SIR four subjects listed in Table 1: Schedule1, NanoXML, XML-security, and JMeter. Our choice was guided by the availability of faults (real or artificial) and the availability of test cases covering those faults and, thus, covering the changes that fix them.

For the three largest subjects—NanoXML, XML-security, and JMeter—the maximum number of SIR changes usable for our approach was exactly seven. For Schedule1, more changes are available but we chose the first seven to prevent this subject from having a disproportionate weight in the overall results. Thus, for each subject, we used seven changes and run PI/EAS on the faulty versions to predict the impact set of the corresponding fault fixes. Each of these changes (fixes) usually involved one or a few more statements, all of them located in one method. Thus, all these changes are single-method changes. For all 28 changes, we computed predictive accuracy of PI/EAS using its predicted impact sets and the actual impacts found by MDEA after applying the changes.

Table 7, similar to Table 2 in format, shows the results of this study. The second column (*#Chgs*) indicates the number of changes we studied. We did not classify the SIR changes into the N and S categories because neither of those categories alone had enough data points to produce meaningful confidence intervals. The PI/EAS and actual impact set sizes were similar to those in Table 2 for all subjects except for JMeter, suggesting that the SIR changes for JMeter were less similar to the SENSA ones for this subject than for the other subjects, although the size ratio of predicted to actual impacts did not differ much.

For Schedule1, NanoXML, and JMeter, the average precision of PI/EAS was even lower for SIR changes than for single-method SENSA changes—by 2 to 17 percentage points. These numbers are actually closer to the category N (non-shortening) of SENSA changes than to the other category, suggesting that, for most SIR changes, the faulty subject runs for long enough for PI/EAS to identify as impacted a considerable number of unaffected methods. Interestingly, the con-

fidence ranges of precision resemble the average precisions for categories N and S in the SENSA study.

Recall, in contrast, was much higher for SIR changes on average per subject than the recalls for all single-method SENSA changes. Similar to the case of precision, though, the PI/EAS recalls for SIR changes were closer to those for the category N of single-method SENSA changes. However, the differences between recall and precision are still considerable, which suggests that, if not caused by the smaller number of data points, the recall of PI/EAS might not be as bad as for other types of changes (SENSA and SVN). Nevertheless, the recall for these SIR changes is still far from perfect. The overall 13% of false negatives might contain important impacts missed by PI/EAS and thus, possibly missed by users. This increase in recall, however, is unable to make up for the lower precision, as the F1 accuracy per subject is almost the same as for single-method SENSA changes.

7.4. Discussion of All Results

As we saw in Section 6, applying PI/EAS can be risky by missing many impacts in addition to falsely reporting many unaffected methods as impacted. Because those results were obtained for SENSA changes, which were randomly generated for each method and, for multiple-method SENSA changes, randomly grouped, we expected that those results would only indicate weaknesses of PI/EAS in general but not necessarily quantify those weaknesses as researchers and developers would normally experience them.

Unfortunately, our study on SIR changes, intended to represent fault fixes as one important type of change, show that PI/EAS can be even less precise for such changes (or at least those we studied). Although the F1 accuracy was higher for the SIR changes than for the SENSA changes due to a higher recall, a 51% average accuracy is probably still far from acceptable.

More importantly, the study on SVN changes made by developers in practice, even though they did not cover the entire subjects as the SENSA changes did, reveals that PI/EAS can perform even worse than estimated randomly via SENSA in terms of both precision and F1 accuracy for almost all subjects considered in both studies. Similarly, when taking all changes together for each study, recall for SVN changes was also lower than for artificial changes.

The 34% overall accuracy for SVN changes was noticeably lower than for all other types of changes studied in this work, indicating that, for usage scenarios that resemble the revision ranges we considered in these SVN repositories, PI/EAS (and similar dynamic impact analyses) has an even more disappointing

performance. These results also suggest strongly the need for developing and studying improved dynamic impact analyses that are more precise and safer.

A few interesting observations can be made in light of all these results. For instance, the single-method SENSA changes, based on random modifications, did not reflect well the SIR changes, although the latter were also single-method changes. There were large differences between the recalls of PI/EAS between the two types of changes. The SIR changes seem to represent fault-fixing scenarios where faults do not cause program executions to crash the program quickly, as about 20% of the SENSA changes did.

Another observation is that the studied SIR changes were not better at approximating "real" (SVN) changes than the SENSA changes for assessing the accuracy of PI/EAS. The average predictive accuracy of PI/EAS for SIR changes was, in fact, less similar to SVN changes than the accuracy for SENSA changes.

Finally, for XML-security, the subject for which we studied all three types of changes, the SIR changes were no different from the single-method SENSA changes in approximating the accuracy that "practical" (SVN) changes had. Multiple-method SENSA changes, however, had a much better accuracy than all other change types for this subject.

7.5. Threats to Validity

The validity of our study on repository changes is affected by the same *internal* and *conclusion* threats as the SENSA-change studies because we used the same tools, experimentation and data-processing scripts, approach to removing non-determinism, and statistical analysis. We also minimized these threats in the same ways. In addition, for the source-code differencing tool that identifies the changed methods between revisions, we manually checked the correctness of its outputs for sample revisions of each of the three studied repositories and carefully inspected the soundness of the results.

An *external* threat concerns the number and variety of subjects and changes (revisions) used for the repository-change studies. To address this limitation, for the SVN-change study, we chose evolving open-source projects of various sizes and functionality and examined a considerable number of revisions to identify over 100 analyzable changes. Also, the studied revision ranges have a fairly diverse set of changes in terms of change type and size. For the SIR-change study, we selected projects using similar guidelines to our other studies and we used as many fault fixes (seven per subject) as we could obtain from the SIR repository for the largest subjects. Those faults have been used in many studies and researchers often consider them as representative for testing and debugging studies.

Another *external* is our choice of test suite for SVN changes. We used the test suite provided with the first examined revision for all revisions studied. However, we saw no changes in those test suites for the range of revisions examined.

8. Related Work

Since PATHIMPACT was introduced by Law and Rothermel [4], dynamic impact analyses have been refined and studied but only for their relative precision in terms of the sizes of their impact sets and their relative efficiencies [7, 9, 53, 10, 11]. In previous work [54], we presented a preliminary study of the predictive accuracy of PI/EAS, but only for single-method SENSA changes. Our current work extends that study and increases its representativity by using, in addition, changes from source-code repositories, multiple-method SENSA changes, and more subjects. The results of our extended work provide developers and researchers a broader and deeper view of the effectiveness of the most cost-effective dynamic impact analysis in the literature.

Shortly after PATHIMPACT, which is based on compressed traces, another dynamic impact analysis called COVERAGEIMPACT was introduced by Orso and colleagues [5], which uses cheap information in the form of runtime coverage to obtain impact sets. The authors of both techniques later compared empirically the precision and efficiency of PATHIMPACT and COVERAGEIMPACT [7] and concluded that COVERAGEIMPACT is considerably less precise than PATHIMPACT, although it is cheaper.

Later, Apiwattanapong and colleagues developed the concept of *execute-after sequences* (EAS) to perform PATHIMPACT more efficiently without any loss of precision. Their approach requires only an execution-length-independent amount of runtime data that is almost as cheap to obtain as the data for COVERAGEIMPACT [9]. However, before our work, the precision of the resulting technique, which we call PI/EAS, was evaluated only in terms of its impact-set sizes against COVERAGEIMPACT. In this paper, in contrast, we evaluated the effectiveness of PI/EAS for a concrete, typical application: predicting *actual* impacts of changes.

To improve the precision of PATHIMPACT, Breech and colleagues conceived INFLUENCEDYNAMIC [10] which adds influence mechanisms to PATHIMPACT. However, the evaluation of INFLUENCEDYNAMIC showed very small improvements over PATHIMPACT while no clear variant of EAS exists for INFLUENCEDYNAMIC to reduce the cost of tracing (dependent on execution length) that INFLUENCEDYNAMIC incurs. That evaluation also focused only on the relative sizes of the impact sets rather than their accuracy.

Hattori and colleagues formally discussed the accuracy of impact analysis [55]. However, instead of *dynamic* impact analyses, they examined the accuracy of a class-level *static* impact analysis tool introduced in that same work, based on call-graph reachability with different depth values. Previously, at the statement level, we recentlystudied the accuracy bounds of dynamic forward slicing [37], which can be used for fine-grained (statement-level) dynamic impact analysis. In contrast, in our present work, we *comprehensively* study *dynamic* impact analysis and we do it at the method level, which can be more practical than static approaches and less expensive than statement-level analysis.

Sensitivity analysis [39] has been used for many software engineering tasks [56], such as those that connect components with requirements [40, 41]. Recently, we used sensitivity analysis and execution differencing to develop SENSA, a *statement-level* dynamic impact analysis technique [22] that finds and also quantifies impacts on statements using massive numbers of random changes. In our new work, we leveraged SENSA as an efficient tool to inject changes and analyze their effects across entire Java subjects to assess the precision and recall of PI/EAS as a representative method-level dynamic impact analysis.

Mutation analysis is a particular form of sensitivity analysis [42, 51] that has been extensively applied for software testing [57, 43, 58]. Indirectly, we benefited from mutation analysis because some of the faults in the SIR repository are the results of mutations. Moreover, SENSA can be seen as a tool that performs mutation, although for different purposes. However, the sensitivity analysis we used, specifically in the SENSA-change studies, is different from mutation analysis because SENSA is intended to simulate changes made to software, whereas mutations are intended to represent typical mistakes made by programmers.

Program differencing is the starting point for *descriptive* impact analysis, which we used to identify actual impacts. Many techniques have been proposed for differencing programs. Some use syntactic information [59, 60, 61, 62] and others perform semantic differencing [63, 16, 64], including execution differencing for dynamic analysis [65, 17, 18, 66]. In this work, we used execution differencing at method level in conjunction with an SVN library and the srcML tool [52] to automatically download source code and find changes and their effects.

Our MDEA technique and other execution-differencing approaches such as DEA [18] and Sieve [17] are *descriptive* impact analyses, which describe change-effects observed at runtime before and after those changes are applied. In contrast, we evaluated the most cost-effective *predictive* dynamic impact analysis known. This analysis can answer impact queries much earlier than descriptive ones, when only *potential* change locations (i.e., methods) have been identified.

Other related dynamic impact analyses exist, such as CHIANTI [8], which is also *descriptive*. CHIANTI compares two program versions and their dynamic call graphs to obtain the set of changes between versions and to map them to affected *test cases* and then affecting changes for each affected test. Our studies, in contrast, focus on *predicted* impacts on *code*. Another technique [67] uses dynamic impacts although as part of an iterative process that weighs statements to improve dynamic backward slicing for debugging. In all, before our work, there were no other accuracy studies of predictive dynamic impact analyses.

9. Conclusion and Future Work

In this paper, we presented a systematic approach for evaluating dynamic change-impact analyses which we applied to most representative and cost-effective technique in the literature. Using this approach, we performed the first study of the predictive accuracy of dynamic impact analysis. We estimated the accuracy, via precision and recall metrics, of PI/EAS for predicting the actual effects of thousands of injected changes in Java software using our sensitivity-analysis technique, 28 changes (fault fixes) made by other researchers, and more than a hundred changes from SVN repositories of open-source Java software systems. The results of our studies indicate that PI/EAS can suffer from low precision or low recall, or both and, thus, very low accuracy.

The accuracy levels observed for this dynamic impact analysis are probably much lower than expected, and probably too low for many tasks. Therefore, the knowledge gained in this work constitutes a note of caution to the community on the usefulness of dynamic impact analysis. At the same time, the results can also inform the understanding and use of dynamic impact sets. As our study did not show any strong or consistent correlation between the accuracy and the size, functionality or complexity of programs, for a variety of changes (e.g., simulated and repository ones), developers who want to use the current analyses should be aware of their low accuracy *in general*. However, developers may expect relatively higher accuracy for programs with tighter internal coupling while possibly better avoiding using the analyses for programs whose executions are greatly shortened by potential changes (or lengthened if the changes will be bug fixes).

The results also show the need for improvements or brand new techniques that provide developers with effective ways to take full advantage of the benefits of dynamic analysis for change-impact prediction, especially its ability to represent the behavior of software in practice and as a way to avoid the overly-conservative results of static analyses. While the low recall was mostly resulted from largely

altered control flows by potential changes, the low precision is very likely to be the result of insufficient use of program dependence information. Design of more accurate analyses, therefore, may explore from these perspectives accordingly.

Next, we will expand our experimental infrastructure to support the evaluation of other predictive impact analyses [68, 35, 69, 70, 71], including the automation of the determinization process to support execution differencing [72, 65, 66] and to enable our system to work efficiently with programs with long executions. The study will also be further expanded to include other interesting examinations, such as exploring the correlation between the predictive accuracy of dynamic impact analysis and certain metrics of the program under analysis (e.g., complexity measure), and evaluating the accuracy with even larger variety of changes (e.g., multiple changes in one method or multiple methods).

Driven by this accuracy study, we recently developed a new dynamic impact analysis that overcomes the precision problem by replacing execution-order relationships with method-level dependencies (data and control) computed via a one-time static analysis [73]. We are continuing along that line of research to study the effects of other forms of dynamic information (e.g., statement coverage) on the cost-effectiveness of dependence-based dynamic impact analysis. The experimental framework presented in this paper will be used to evaluate both the new technique and those effects.

Acknowledgment

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

References

[1] S. A. Bohner, R. S. Arnold, An introduction to software change impact analysis, In Software Change Impact Analysis, Bohner & Arnold, Eds. IEEE Computer Society Press, pp. 1–26, 1996.

[2] V. Rajlich, Software Engineering: The Current Practice, Chapman and Hall/CRC, 2011.

[3] B. Li, X. Sun, H. Leung, S. Zhang, A survey of code-based change impact analysis techniques, Software Testing, Verification and Reliability 23 (2013) 613–646.

[4] J. Law, G. Rothermel, Whole program Path-Based dynamic impact analysis, in: Proc. of IEEE/ACM Int'l Conf. on Software Engineering, 308–318, 2003.

[5] A. Orso, T. Apiwattanapong, M. J. Harrold, Leveraging Field Data for Impact Analysis and Regression Testing, in: Proc. of joint European Software Engineering Conference and ACM Int'l Symp. on the Foundations of Software Engineering, 128–137, 2003.

[6] J. Law, G. Rothermel, Incremental Dynamic Impact Analysis for Evolving Software Systems, in: Proceedings of the 14th International Symposium on Software Reliability Engineering, 430–441, 2003.

[7] A. Orso, T. Apiwattanapong, J. B. Law, G. Rothermel, M. J. Harrold, An Empirical Comparison of Dynamic Impact Analysis Algorithms, in: Proc. of Int'l Conf. on Softw. Eng., 491–500, ????

[8] X. Ren, F. Shah, F. Tip, B. G. Ryder, 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., 432–448, 2004.

[9] T. Apiwattanapong, A. Orso, M. J. Harrold, Efficient and Precise Dynamic Impact Analysis Using Execute-After Sequences, in: Proc. of Int'l Conf. on Softw. Eng., 432–441, 2005.

[10] B. Breech, M. Tegtmeyer, L. Pollock, Integrating Influence Mechanisms into Impact Analysis for Increased Precision, in: Proc. of IEEE Int'l Conf. on Software Maintenance, 55–65, 2006.

[11] B. Breech, M. Tegtmeyer, L. Pollock, A Comparison of Online and Dynamic Impact Analysis Algorithms, in: Proc. of European Conf. on Software Maintenance and Reingenering, 143–152, 2005.

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

[13] H. Agrawal, J. R. Horgan, Dynamic program slicing, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, 246–256, 1990.

[14] X. Zhang, R. Gupta, Y. Zhang, Precise Dynamic Slicing Algorithms, in: Proc. of International Conference on Software Engineering, 319–329, 2003.

[15] W. Masri, N. Nahas, A. Podgurski, Memoized forward computation of dynamic slices, in: Software Reliability Engineering, 2006. ISSRE'06. 17th International Symposium on, IEEE, 23–32, 2006.

[16] T. Apiwattanapong, A. Orso, M. J. Harrold, JDiff: A differencing technique and tool for object-oriented programs, Automated Software Engineering 14 (1) (2007) 3–36.

[17] M. K. Ramanathan, A. Grama, S. Jagannathan, Sieve: A Tool for Automatically Detecting Variations Across Program Versions, in: Proc. of IEEE/ACM Int'l Conf. on Automated Software Engineering, 241–252, 2006.

[18] R. Santelices, M. J. Harrold, A. Orso, Precisely Detecting Runtime Change Interactions for Evolving Software, in: Proc. of Int'l Conf. on Softw. Testing, Verif. and Valid., 429–438, 2010.

[19] M. Stoerzer, B. G. Ryder, X. Ren, F. Tip, Finding failure-inducing changes in java programs using change classification, in: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering, 57–68, 2006.

[20] X. Ren, O. C. Chesley, B. G. Ryder, Identifying failure causes in java programs: An application of change impact analysis, Software Engineering, IEEE Transactions on 32 (9) (2006) 718–732.

[21] R. Santelices, Y. Zhang, S. Jiang, H. Cai, Y. jie Zhang, Quantitative Program Slicing: Separating Statements by Relevance, in: Proc. of IEEE/ACM Int'l Conf. on Software Engineering – New Ideas and Emerging Results track, 1269–1272, 2013.

[22] H. Cai, S. Jiang, R. Santelices, Y. jie Zhang, Y. Zhang, SENSA: Sensitivity Analysis for Quantitative Change-impact Prediction, in: Proceedings of IEEE Working Conference on Source Code Analysis and Manipulation, 165–174, 2014.

[23] H. Do, S. Elbaum, G. Rothermel, Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact, Emp. Softw. Eng. 10 (4) (2005) 405–435.

[24] W. N. Sumner, X. Zhang, Comparative Causality: Explaining the Differences Between Executions, in: Proc. of IEEE/ACM Int'l Conf. on Software Engineering, 272–281, 2013.

[25] A. Podgurski, L. Clarke, A Formal Model of Program Dependences and its Implications for Software Testing, Debugging, and Maintenance, IEEE Transactions on Softw. Eng. 16 (9) (1990) 965–979.

[26] V. Rajlich, Changing the paradigm of software engineering, Communications of the ACM 49 (8) (2006) 67–70.

[27] V. Rajlich, P. Gosavi, Incremental change in object-oriented programming, Software, IEEE 21 (4) (2004) 62–69.

[28] V. Rajlich, Software Evolution and Maintenance, in: Proceedings of the Conference on the Future of Software Engineering, ISBN 978-1-4503-2865-4, 133–144, 2014.

[29] T. D. LaToza, G. Venolia, R. DeLine, Maintaining mental models: a study of developer work habits, in: Proceedings of the 28th international conference on Software engineering, ACM, 492–501, 2006.

[30] C. R. de Souza, D. F. Redmiles, An empirical study of software developers' management of dependencies and changes, in: Proceedings of the 30th international conference on Software engineering, ACM, 241–250, 2008.

[31] Y. Tao, Y. Dang, T. Xie, D. Zhang, S. Kim, How do software engineers understand code changes?: an exploratory study in industry, in: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, 51:1C–51:11, 2012.

[32] P. Rovegard, L. Angelis, C. Wohlin, An empirical study on views of importance of change impact analysis issues, Software Engineering, IEEE Transactions on 34 (4) (2008) 516–530.

[33] T. D. LaToza, B. A. Myers, Developers ask reachability questions, in: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1, ACM, 185–194, 2010.

[34] M. Acharya, B. Robinson, Practical Change Impact Analysis Based on Static Program Slicing for Industrial Software Systems, in: Proceedings

of IEEE/ACM International Conference on Software Engineering, Software Engineering in Practice Track, 746–765, 2011.

[35] S. Lehnert, A review of software change impact analysis, Tech. Rep., Ilmenau University of Technology, 2011.

[36] M. Ajrnal Chaumun, H. Kabaili, R. K. Keller, F. Lustman, A change impact model for changeability assessment in object-oriented software systems, in: Software Maintenance and Reengineering, 1999. Proceedings of the Third European Conference on, IEEE, 130–138, 1999.

[37] S. Jiang, R. Santelices, M. Grechanik, H. Cai, On the Accuracy of Forward Dynamic Slicing and its Effects on Software Maintenance, in: Proceedings of IEEE Working Conference on Source Code Analysis and Manipulation, 145–154, 2014.

[38] JavaSlicer, `https://www.st.cs.uni-saarland.de/javaslicer/`, [Accessed on 24-Nov-2014], 2014.

[39] A. Saltelli, K. Chan, E. M. Scott, Sensitivity Analysis, John Wiley & Sons, 2009.

[40] G. N. Rodrigues, D. S. Rosenblum, S. Uchitel, Sensitivity Analysis for A Scenario-based Reliability Prediction Model, in: Proceedings of Workshop on Architecting Dependable Systems, 1–5, 2005.

[41] M. Harman, J. Krinke, J. Ren, S. Yoo, Search Based Data Sensitivity Analysis Applied to Requirement Engineering, in: Proceedings of Genetic and Evolutionary Computation Conference, 1681–1688, 2009.

[42] R. A. DeMillo, R. J. Lipton, F. G. Sayward, Hints on Test Data Selection: Help for the Practicing Programmer, Computer 11 (4) (1978) 34–41.

[43] Y. Jia, M. Harman, An Analysis and Survey of the Development of Mutation Testing, IEEE Transactions on Software Engineering 37 (5) (2011) 649–678.

[44] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, C. Zapf, An experimental determination of sufficient mutant operators, ACM Transactions on Software Engineering and Methodology 5 (2) (1996) 99–118.

[45] R. E. Walpole, R. H. Myers, S. L. Myers, K. E. Ye, Probability and Statistics for Engineers and Scientists, Prentice Hall, ISBN 978–0321629111, 2011.

[46] R. Santelices, M. J. Harrold, Efficiently monitoring data-flow test coverage, in: Proc. of Int'l Conf. on Automated Softw. Eng., 343–352, 2007.

[47] R. Santelices, Y. Zhang, H. Cai, S. Jiang, DUA-Forensics: a fine-grained dependence analysis and instrumentation framework based on Soot, in: Proceedings of the 2nd ACM SIGPLAN International Workshop on State Of the Art in Java Program analysis, 13–18, 2013.

[48] P. Lam, E. Bodden, O. Lhoták, L. Hendren, Soot - a Java Bytecode Optimization Framework, in: Cetus Users and Compiler Infrastructure Workshop, 2011.

[49] D. Vysochanskij, Y. Petunin, Justification of the three-sigma rule for unimodal distributions, Theory of Probability and Mathematical Statistics 21 (1980) 25–36.

[50] R. Santelices, M. J. Harrold, Demand-driven propagation-based strategies for testing changes, Software Testing, Verification and Reliability 23 (6) (2013) 499–528.

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

[52] J. I. Maletic, M. L. Collard, Supporting source code difference analysis, in: 20th IEEE International Conference on Software Maintenance, IEEE, 210–219, 2004.

[53] B. Breech, A. Danalis, S. Shindo, L. Pollock, Online Impact Analysis via Dynamic Compilation Technology, in: Proc. of IEEE Int'l Conf. on Software Maintenance, 453–457, 2004.

[54] H. Cai, R. Santelices, T. Xu, Estimating the Accuracy of Dynamic Change-Impact Analysis using Sensitivity Analysis, in: IEEE 8th International Conference on Software Security and Reliability, 48–57, 2014.

[55] L. Hattori, D. Guerrero, J. Figueiredo, J. Brunet, J. Damasio, On the Precision and Accuracy of Impact Analysis Techniques, in: IEEE/ACIS Int'l Conf. on Computer and Information Science, 513–518, 2008.

[56] S. Wagner, Global Sensitivity Analysis of Predictor Models in Software Engineering, in: Proc. of IEEE Int'l Workshop on Predictor Models in Software Engineering, 3–10, 2007.

[57] Y.-S. Ma, J. Offutt, Y. R. Kwon, MuJava: An Automated Class Mutation System, Software Testing, Verifiifcation and Reliability 15 (2) (2005) 97–133.

[58] L. Zhang, M. Gligoric, D. Marinov, S. Khurshid, Operator-based and random mutant selection: Better together, in: 2013 IEEE/ACM 28th International Conference on Automated Software Engineering, IEEE, 92–102, 2013.

[59] W. Yang, Identifying syntactic differences between two programs, Software: Practice and Experience 21 (7) (1991) 739–755.

[60] B. Fluri, M. Wursch, M. PInzger, H. C. Gall, Change Distilling: Tree differencing for fine-grained source code change extraction, IEEE Transactions on Software Engineering 33 (11) (2007) 725–743.

[61] S. Raghavan, R. Rohana, D. Leon, A. Podgurski, V. Augustine, Dex: A semantic-graph differencing tool for studying changes in large code bases, in: 20th IEEE International Conference on Software Maintenance, IEEE, 188–197, 2004.

[62] Z. Xing, E. Stroulia, UMLDiff: An algorithm for object-oriented design differencing, in: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, ACM, 54–65, 2005.

[63] D. Jackson, D. A. Ladd, Semantic Diff: A tool for summarizing the effects of modifications, in: International Conference on Software Maintenance, IEEE, 243–252, 1994.

[64] S. Person, M. B. Dwyer, S. Elbaum, C. S. Păsăreanu, Differential symbolic execution, in: Proc. of Int'l Symp. on Foundations of Softw. Eng., 226–237, 2008.

[65] K. J. Hoffman, P. Eugster, S. Jagannathan, Semantics-aware Trace Analysis, in: Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation, 453–464, 2009.

[66] W. N. Sumner, T. Bao, X. Zhang, Selecting peers for execution comparison, in: Proceedings of the 2011 International Symposium on Software Testing and Analysis, ACM, 309–319, 2011.

[67] T. Goradia, Dynamic impact analysis: a cost-effective technique to enforce error-propagation, in: Proc. of ACM Int'l Symp. on Software Testing and Analysis, 171–181, 1993.

[68] M. C. O. Maia, R. A. Bittencourt, J. C. A. de Figueiredo, D. D. S. Guerrero, The hybrid technique for object-oriented software change impact analysis, in: 14th European Conference on Software Maintenance and Reengineering, IEEE, 252–255, 2010.

[69] P. Tonella, Using a concept lattice of decomposition slices for program understanding and impact analysis, IEEE Transactions on Software Engineering 29 (6) (2003) 495–509.

[70] D. Poshyvanyk, A. Marcus, R. Ferenc, T. Gyimóthy, Using information retrieval based coupling measures for impact analysis, Empirical Software Engineering 14 (1) (2009) 5–32.

[71] M. Gethers, B. Dit, H. Kagdi, D. Poshyvanyk, Integrated impact analysis for managing software changes, in: Proc. of IEEE/ACM Int'l Conf. on Software Engineering, 430–440, 2012.

[72] X. Zhang, R. Gupta, Matching execution histories of program versions, in: ACM SIGSOFT Symposium on the Foundations of Software Engineering, 197–206, 2005.

[73] H. Cai, R. Santelices, DIVER: Precise Dynamic Impact Analysis Using Dependence-based Trace Pruning, in: Proceedings of International Conference on Automated Software Engineering, 343–348, 2014.