COST-EFFECTIVE DEPENDENCY ANALYSIS FOR RELIABLE SOFTWARE EVOLUTION

A Dissertation

Submitted to the Graduate School
of the University of Notre Dame
in Partial Fulfillment of the Requirements
for the Degree of

Doctor of Philosophy

by

Haipeng Cai

Douglas Thain, Director

Graduate Program in Computer Science and Engineering
Notre Dame, Indiana

July 2015

COST-EFFECTIVE DEPENDENCY ANALYSIS FOR RELIABLE SOFTWARE EVOLUTION

Abstract

by

Haipeng Cai

Program *dependency analysis* is a fundamental approach to code-based impact prediction which underlies change planning and management, as well as a range of other dependency-based client analyses such as testing, debugging, and comprehension, all of which are crucial for reliable software evolution. However, existing such approaches suffer from difficult *cost-effectiveness* balancing: Fine-grained analyses offer detailed but large results at prohibitive costs, whereas coarser analyses achieve better efficiency at the cost of great imprecision. In addition, current techniques have little utility when applied to distributed systems consisting of decoupled components.

This dissertation addresses these challenges via three novel approaches to cost-effective dependency analysis, followed by a comprehensive study of prior alternatives. First, a quantitative semantic-dependency analysis is presented to separate program entities by relevance so as to prioritize impact prediction at statement level. The second approach exploits a diverse set of *hybrid* techniques that combine static and dynamic dependency analysis to offer not only improved but *variable* cost-effectiveness to fit varying budgets and task scenarios for impact prediction at method level. Finally, an efficient dynamic analysis is developed to discover method-level runtime dependencies both within and *across* multiple concurrent processes, which is shown instrumental in evolving distributed systems.

DEDICATION

To my dearly loving and ever-supporting parents

CONTENTS

FIGURES

TABLES

ACKNOWLEDGMENTS

I am grateful to my advisor Dr. Douglas Thain for his research guidance and other supports I needed to complete my study. I am also indebted to Dr. Raul Santelices for introducing me to program analysis and offering scrupulous advisement, without which this dissertation would not have been started.

I would like to thank Dr. Collin McMillan and Dr. Aaron Striegel for serving on my thesis committee and offering valuable comments, which have all helped improve this dissertation. Finally, I want to acknowledge the U.S. Office of Naval Research, for their generous grant, number N000141410037, which allowed me to pursue my doctoral research.

SYMBOLS

P	program under analysis
D	distributed system under analysis
P'	instrumented version of program P
T, I	test suite or input set
M	query set
q	a single query
L	list
A	a ranking
E	example programs

CHAPTER 1

INTRODUCTION

Constant code change drives the evolution of modern software systems, yet also poses risks to their quality and reliability, even more so as the systems become increasingly complex and subject to rapid iterations nowadays [148]. Although necessary changes also involve other forms of artifacts such as requirements and specifications, the most precise information about the behavior of a program is contained in its source code [79, 111]. This dissertation, therefore, specifically focuses on the particular category of changes that are made directly to the source code of a program.

For ensuring reliable software evolution, it is crucial to understand the consequences of *code changes* at potential program locations *before* actual changes are designed and made to those locations. Without sufficient knowledge about possible effects of candidate changes, developers can apply even a small modification in one piece of code of a system such that the entire system fails, because the impact of changes usually propagates beyond originating locations to the rest of the software [15, 97] due to *ripple effects* [27, 34, 45, 198]. In general, predicting the effects of proposed changes has become a critical step in the evolution-based software engineering paradigm today [146], whether the software be a centralized sequential program or a distributed concurrent system.

As the external behavior of a program is essentially the outcome of the internal interactions among its entities (e.g., statements or methods), the relationships among those entities have to be analyzed in order to address the effects of one entity or changes to that entity on others of the program. A fundamental approach to identifying and reasoning about such relationships is dependency analysis [114, 141, 181]. To determine the full extent of po-

tential changes to a program, impacts[1] of the changes need to be propagated from original change locations. This process relies on dependency analysis as well, as propagating the impacts is usually based on the navigation through program dependencies. In a broader sense, program dependency analysis is able to not only *predict* potential consequences of candidate changes, but also to support a variety of other software evolution tasks.

The main goal of this dissertation is to explore and study various ways of analyzing program dependencies to support *impact analysis* with respect to code changes [15, 109, 111], a vital software maintenance and evolution task that addresses the need for understanding potential change effects. In particular, this work aims to provide dependency-analysis techniques with various levels of tradeoffs between the *cost and effectiveness* of the techniques in the context of impact analysis for both centralized single-process and distributed multiprocess programs. More specifically, this work mainly targets *dynamic* program dependencies and corresponding applications—those concerning concrete operational profiles of programs that reflect their actual behaviors against specific inputs at runtime—through hybrid techniques which combine static and dynamic analysis.

1.1 Motivation and Challenges

Impact analysis is widely recognized as an integral and crucial step during software development [156, 180]. By far a large amount of research has been invested in impact analysis, mostly aiming at technical approaches such as novel algorithms (e.g., [12, 31, 32, 44, 68, 108, 110, 116, 134, 149, 167, 182]) with fewer targeting in-depth empirical studies such as comparative evaluations (e.g., [30, 111, 135, 177]). Regardless of the large volume and variety of work on impact analysis in the literature, however, existing techniques mostly adopt coarse analysis *only* [112, 175, 176] and, more critically, produce results that are generally quite rough (imprecise) [134, 135, 149]. For one thing, most analyses report

[1] When used as a noun, *impact* means *influence* as in the context of program slicing; when in the plural form, it is also short for *impacted entities*.

potentially impacted entities merely at the coarse granularity of methods, classes, or even packages (modules), without giving further details about the results (e.g., the exact code region in a reported method that is really relevant to the impacts). For another, extant approaches usually suffer from containing a large portion of false positives[2], missing truly impacted entities, or having both low precision and imperfect recall, in their resulting *impact sets* (set of entities identified as potentially impacted). In addition, built atop traditional dependency models which rely on explicit invocation among program components, today's solutions do not readily apply to distributed systems where loosely coupled or decoupled components communicate through implicit invocation (e.g., message passing).

The need for results of variable granularity. While impact sets are commonly requested for at relatively coarse levels— predominantly at method level [12, 31, 108, 116, 149] —with respect to fine-grained (statement-level) impacts, offering results of coarse granularity *only* is not enough because engineers do need to access more detailed information about the impact sets—specific statements responsible for the impacts [3]—for particular tasks that rely on fine results (e.g., fault cause-effect identification). For those tasks, the coarse granularity could incur excessive efforts for understanding the impacts, as developers would have to inspect the entire coarse entity in the impact set (e.g., a class) even if only a small portion of the reported set actually requires the inspection.

In reality, developers need different impact-analysis approaches that work at different levels of granularity, for different needs [52] due to, for instance, various levels of budget, change request, or variant perspectives of impact analysis [156]. For many usage cases, users do need fine-grained information about the impacts, while for some other cases, a coarser-level such as method-level analysis may suffice. Unfortunately, existing predictive analyses [15] often focus on the coarse results only—most of them work at the method level and fewer others at even coarser levels (e.g., file, package, or class levels). Addressing the

[2]Also referred to as false-positive impacts, meaning program entities that are actually not to be impacted but incorrectly reported as to be.

coarse granularity is not trivial as finer analyses, although more precise, imply a larger number of entities in the resulting impact sets, which may require an even higher cost of impact inspection. Statement-level impact analysis via forward program slicing could provide fine-grained (i.e., statement-level) impact sets. However, results from static slicing when applied to impact analysis would be too large and imprecise to be truly useful, while using dynamic slicing directly would be too expensive to be practically feasible.

The need for analyses of improved and variable cost-effectiveness. As commonly adopted, method-level impact analysis can provide a reasonable balance between fine granularity and scalability: It is generally much more precise than coarser-level analyses, as well as much less expensive than those dedicated to offering the finest-grained results. Yet, existing method-level analyses usually take fast approximation approaches, suffering from excessive imprecision as a result of over conservativeness. Such imprecision tends to waste developers' time and other resources for checking unnecessarily large impact sets and, more fatally, to risk introducing detrimental artifacts as a result of misguiding false impacts. While the imprecision is often suffered for the sake of efficiency, and rough rapid results are useful indeed, precise analysis at reasonable extra costs is at least equally, if not more, necessary as well [88], especially in light of the status-quo that most of the efficient techniques developed to this date are too imprecise to be realistically viable.

In practice, the diversity of impact-analysis techniques as mentioned above also consists in different levels of precision [88] and various cost-effectiveness tradeoffs [52, 156]. While intuitively greater precision would always be more desirable, developers have to balance between the cost and benefits of applying an analysis in accordance with their budgets because more precise analyses typically cost more too. For some tasks, such as test-suite selection, higher effectiveness may be more preferable even with compromised efficiency; whereas in other task scenarios, such as high-level code comprehension, quicker answers to impact-set queries would weigh over superior precision. However, prior approaches often provide merely a singular choice of precision and/or cost-effectiveness—they mostly

lie at two extremes, either efficient but too imprecise or more precise but overly expensive. Addressing the imprecision and low cost-effectiveness is also challenging, though, as removing false impacts usually necessitate more detailed analysis, which implies more computations hence higher total costs of performing the impact analysis.

In addition, the inaccuracy of today's impact analyses was *suggested* mainly through indirect evaluations based on relative metrics [12, 31, 108, 134]. Prior to this work, there has not yet been a systematic assessment of the accuracy (both precision and recall) of dependency-based impact analysis against true impacts (ground truth) expressed via semantic dependency [141]. Therefore, to enable practical applications of impact analyses, the effectiveness of existing approaches has to be thoroughly investigated first before the other issues are to be addressed.

The need for effectively evolving distributed systems. Yet another challenge to present impact-prediction techniques concerns their application scope. Although a great many approaches exist for analyzing centralized programs, very few is applicable to distributed systems [67, 92]. A major difficulty lies in the lack of explicit invocation in most distributed systems as a result of inter-component decoupling [72, 184], on which traditional, especially dependency-based, impact analysis relies. Although dependency analysis for particular types of distributed systems like the event-based ones [125] and those built on special language extensions [62] have received some attentions recently [67, 113, 142, 184], impact analysis for general[3] distributed systems that adopt neither well-defined inter-component interfaces nor message-typing specifications remains unaddressed as yet. On the other hand, the growing demands for, and increasing deployments of, the general-type distributed systems today entail effective impact analyses to support their evolution.

Techniques do exist for dynamic impact prediction in the presence of concurrency, by tracing method-execution events (i.e., control flows) and then inferring execution order

[3]Here being general means relying on no particular message typing, middleware framework constraints, or manual code editing or transformation such as developer annotations.

hence impact relations among methods (e.g., [12, 108]). As such, those techniques bypass the difficulty in analyzing explicit inter-component dependencies, yet they are capable of dealing with multi-threaded executions only. Accommodating them to multiprocess executions is seemingly straightforward but actually challenging, because the execution order of methods is not automatically observed in distributed systems. One primary reason underneath is that, under typical distributed system environments, the method-execution events are not synchronized with respect to the timing of their occurrences in multiple concurrent processes (at separated hosts). Accordingly, inferring the method execution order over all processes of the entire system, as is required for impact prediction, becomes infeasible.

1.2 Thesis Scope and Statement

The central objective of this work is to address the aforementioned challenges so as to facilitate evolving modern software systems reliably and cost-effectively. To reach that goal, this dissertation targets the design and evaluation of several advanced dependency analyses that enable various impact-prediction[4] techniques of variable cost-effectiveness tradeoffs in addition to significantly higher precision than existing peer alternatives without penalizing recall.[5] While the majority of the content is tilted to addressing traditional centralized software, the impact analysis of code changes for *general* distributed multi-process systems is also explored as the first of its kind. It is worth mentioning that while dependency-based impact analyses [15] are the predominating topic of this dissertation, they are the motivating instances and representative applications of the cost-effective code-based dependency analysis at the core. The advanced dependency analysis techniques presented are expected to readily apply to many other software engineering tasks as well, such as testing, debugging, refactoring, reverse engineering, and program comprehension.

[4]Without loss of clarity, the term *impact prediction* and *predictive impact analysis* are used interchangeably—*prediction* means that actual changes are *not* known.

[5]For all dynamic analyses in this work, a perfect recall, or the *safety* of analysis result, is claimed always *relative* to specific executions and the single version of the program under a *predictive* analysis.

The *thesis statement* is that advanced program dependency analyses, including dependency quantification and abstraction, can largely increase the cost-effectiveness of impact prediction with variable tradeoffs and thus greatly enhance the utility of the analysis to better support reliable software evolution, for both centralized and distributed systems.

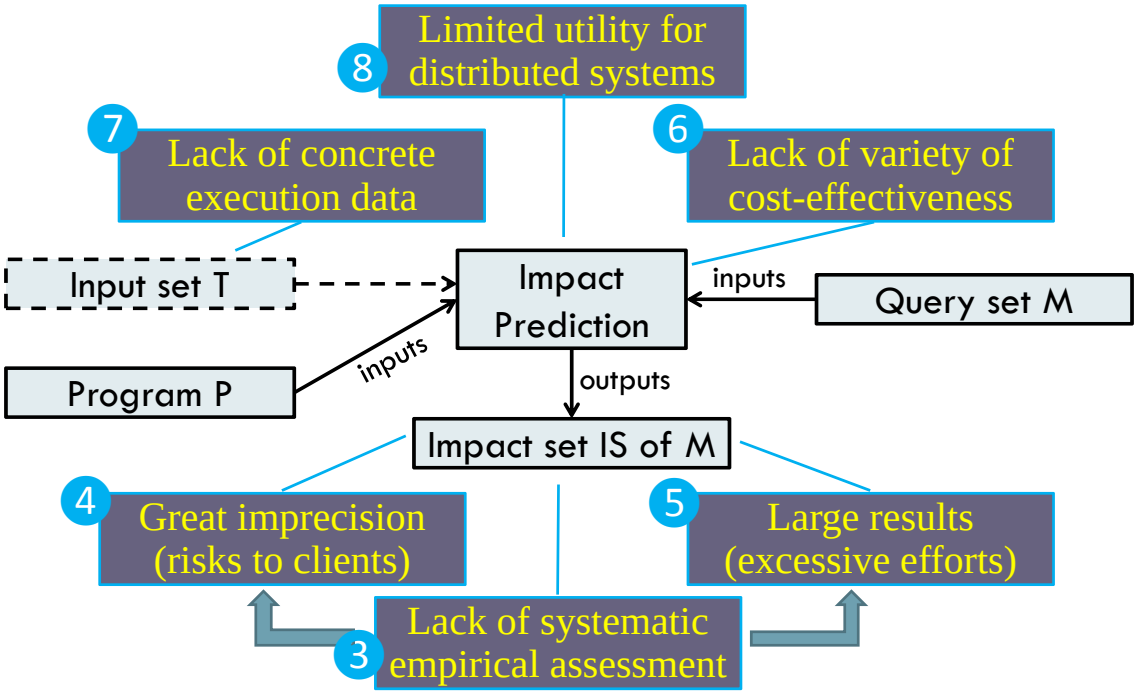

Figure 1.1: The common process flow of dependency-based impact prediction with the issues or challenges addressed in this dissertation annotated and each labeled by the number of the chapter where the issue is targeted.

1.3 Dissertation Overview

Specifically, to support the thesis statement, this dissertation presents (1) a comprehensive evaluation of existing dynamic impact prediction techniques, (2) a precise hybrid approach to dynamic impact prediction based on method-execution trace pruning, (3) a novel

semantic-dependency quantification approach to fine-grained (statement-level) impact prediction, (4) a unified framework of dynamic impact prediction that offers improved and variable cost-effectiveness, (5) a new program representation for method-level dependency-abstraction and its application to static impact prediction, and (6) a rapid and effective dynamic impact analysis for distributed programs.

Each of these approaches attempts to address one of the six issues or challenges faced by the current dependency-based impact prediction with different ones lying in different components of the common process flow of the prediction technique, as depicted in Figure 1.1. Note that these issues decompose the three major challenges categorized in Section 1.1, with the issue 5 corresponding to the need for variable impact-set granularity, issues 3, 4, 6, and 7 belonging to the need for improved precision and variable cost-effectiveness, and the issue 8 reflecting the need for impact analysis of distributed programs.

As is delineated, the impact prediction addressed in this work takes as inputs the program P under analysis, the set M of *queries* (entities of the program for which impacts are to be queried), and, if available, a specific input set T of P; next, using the given program information, the technique computes all entities in P that *may* be impacted by any member entity (or any changes to be made there) in M; as the output, the technique produces the impact set IS consisting of those potentially impacted entities. Concerning the optional input set T, when it is not available, the prediction is commonly realized through a purely static analysis—based on the code of P only—producing results that hold for any possible program inputs; when it is provided and utilized, however, either purely dynamic or hybrid analysis can be adopted to produce prediction results that are precise but hold for the executions derived from T only. Targeting the six problems faced by prior approaches as annotated, the rest of this dissertation is organized into the following chapters.

Chapter 2 provides common background concepts and techniques that are closely related to subsequent chapters to facilitate the understanding of this dissertation. It gives brief definition of different types of program dependencies, introduces the field of software

impact analysis, and presents representative approaches to dependency analysis and impact analysis, all illustrated through example programs with sample executions. It also summarizes the merits and weaknesses of the background techniques, of which those for impact analysis serve as baselines in the evaluation of the new analyses contributed by this work.

Chapter 3 presents a comprehensive study on the predictive accuracy of dynamic impact analysis to address the *lack of systematic empirical assessment* of existing such analyses. While previous approaches were expected to be imprecise due to their conservative nature [111, 135], the imprecision was identified mostly through empirical evaluations using relative precision measures only (e.g., [85, 134, 135]). Very few other analyses had their results assessed against ground truths, yet they either used only trivial experimental subjects [84, 116] or relied on the knowledge about actual changes *before* performing the analysis [149, 151, 162]. On the other hand, since developers need to apply impact analysis for understanding the effects of potential changes, it is clearly important to know how accurate the analysis results are expected to be. However, as yet an empirical assessment that systematically measures the accuracy (precision and recall) of change-impact prediction is still missing [111]. This chapter shows how to fill the gap by developing an experimental pipeline with which a large and diverse set of changes can be utilized to thoroughly gauge the predictive accuracy based on semantic-dependency analysis, and applied this pipeline to reveal the level of accuracy previous techniques offered.

Chapter 4 addresses the *great imprecision* of prior dynamic impact prediction by developing a significantly more precise analysis via dependency-based trace pruning. The study in Chapter 3 reveals that even the most cost-effective existing dynamic impact analyses report on average one false positive out of each two methods in their impact sets. For real-world software systems, especially those that are as large as containing thousands of methods, this level of imprecision could severely impede the adoption of such analyses since applying them implies a large amount of resources to be wasted. This chapter presents a novel approach that is much more precise while remaining sound and incurring

reasonable amount of extra costs. At the core of the new approach is a hybrid analysis that leverages static program dependencies to prune methods not directly or transitively dependent on the query at runtime that would otherwise be included in the impact set.

Chapter 5 aims to overcome the low cost-effectiveness of fine-grained (statement-level) impact prediction due to the *large results* it often produces by prioritizing impacts via semantic dependency quantification. Besides the great imprecision as attacked in the previous chapter, another problem developers have encountered with impact analysis is that the results from current techniques are too coarse or uncertain [156]. To understand the potential impacts of candidate changes, developers need to know which unchanged parts of a software may be affected. While knowing a coarser scope such as the classes or methods that contain the affected parts may help developers narrow down the search space as a useful starting step, missing precise locations (statements) that are exactly to be impacted could force developers to inspect the entire classes or methods. A fine-grained (statement-level) analysis, such as slicing, can overcome this problem, yet tends to produce even larger results—greater number of entities in the impact set. This chapter initiates a novel quantitative semantic-dependency analysis that allows developers to prioritize their inspection for more strongly impacted statements first, complementary to previous approaches that focus on reducing the sizes of impact sets.

Chapter 6 demonstrates a unified framework of dynamic impact analyses for improved and variable cost-effectiveness. To make impact analysis more practically adoptable, offering just better precision is not enough because different degrees of precision is required in different situations (e.g., applying the analysis to different program locations or for different data structures) [88]. In fact, several industrial studies (e.g., [52, 156]) also showed that different types of impact analyses are in demand for different budget plans and usage scenarios. To meet the variety of needs, not only is improved cost-effectiveness preferred, but also variable levels of tradeoffs between the cost and effectiveness of using impact analysis is requested. This chapter aims at an approach that offers flexible cost-effectiveness options

by leveraging not only both static and dynamic program information but also diverse forms of dynamic data, including method-event trace, statement coverage, and dynamic points-to sets. It unifies a sequence of existing and novel impact prediction techniques in one holistic framework, where variant degrees of effectiveness are accomplished at variant degrees of overheads when different amounts of program information are utilized in synergy, which addresses the *lack of variety of cost-effectiveness* with existing approaches.

Chapter 7 focuses on a program dependency abstraction at method level, and directly applies the abstraction to static impact prediction which accommodates situations with *lack of concrete execution data*. Fine-grained analyses, such as program slicing which works at statement level, offers more details in their results but often encountered scalability bottlenecks. Moreover, for tasks where coarser-grained results suffice, such as impact analysis, the extra cost incurred by fine analyses would end up with an even worse cost-benefit balance. To fit such tasks, previous research has explicitly attempted abstracting program dependencies to method level, yet existing abstraction models are either incomplete—implying unsoundness of the analysis—or too imprecise thus of little use to engineers. Some other approaches simply lift up fine-grained results to method level, suffering even greater inefficiency than fine analyses. This chapter describes a new representation of program dependencies, called the method dependency graph, which directly models both data and control dependencies at method level with intraprocedural dependency details abstracted away. It also shows how a novel static impact analysis leverages this new representation to attain significantly higher cost-effectiveness than peer alternatives.

Chapter 8 explores a cost-effective approach to predicting dynamic impacts in distributed systems. While the implicit invocations and references among components empowers the performance and scalability of modern distributed systems, they also bring unprecedented challenges to tradition program analysis such as dependency-based impact analysis, which relies on explicit invocations that are missing among components due to the decoupling paradigm. As a consequence, the rich collection of existing impact-analysis

techniques tend to be useless when applied to distributed systems. Although an emerging body of research started targeting the topic of distributed system impact analysis, existing approaches are mostly limited to specialized systems or languages, and all subject to particular constraints that greatly hinder their adoptability. This chapter presents a new dynamic analysis that identifies potential method-level impacts both within and across process boundaries in distributed systems consisting of physically separated components running in concurrent processes. Successfully applied to large-scale enterprise systems, the new technique overcomes the limitations of existing options thus surmounts their *low utility for distributed systems* by tracing local method-executions events while preserving the global partial ordering of all such events over concurrent processes.

Finally, Chapter 9 recapitulates the problems this dissertation addressed and approaches it presented to advance the state of the art in program analysis in general and dependency analysis with applications to software evolution in particular. It also summarizes the lessons learned from this work and envisions future directions of research.

1.4 Impact of This Work

The central theme of this work is initially driven by the need for effective impact analysis in the context of software evolution, which is suggested by the technical nature of existing analyses and demonstrated by previous industrial studies. The techniques and methodologies developed in this dissertation, however, are mainly motivated by a more general need for cost-effective program dependency analysis and based on the insights into several critical challenges faced by traditional algorithms underlying such analyses.

To systematically characterize these challenges before attempting to address them, this work starts with the first comprehensive study of dynamic impact analysis on the most cost-effective prior techniques. For engineers employing those techniques, results from this study inform about the level of precision and recall they can expect, giving them an important reference concerning whether and when the techniques should be adopted and

potential consequences of adoption. For relevant researchers, the results justify the development of more precise analyses and offer pointers toward promising avenues to that end; in addition, the experimental framework developed for this study can be directly utilized to gauge the accuracy of future techniques of kinds similar to dynamic impact prediction, and extended for evaluating other dynamic analyses, especially where a large-scale study against massive number of diverse types of changes is involved.

The static and dynamic impact prediction techniques presented in this work are expected to make impact analysis much more attractive to practitioners through the substantially enhanced precision and variable cost-effectiveness tradeoffs offered in one unified environment. With the safe but much smaller impact sets, the individual analyses will help engineers improve productivity by saving their efforts in inspecting potential effects of candidate changes when planning software upgrades or bug fixes, as well as assist them with overall development efficiency and quality by facilitating debugging and comprehension tasks. The unified framework is even more promising in practice as it provides more choices of techniques that all come with better cost-effectiveness than previous options, meeting various user needs in various task scenarios and application contexts. Additionally, the new method-level dependency abstraction not only enables more cost-effective (static) impact analysis in the absence of concrete operational profiles of programs, but also offers a new foundation for developing more efficient dynamic analysis.

This work will have positive impact on the development and evolution practice of distributed systems as well. First, there had been a lack of effective supports for analyzing potential impacts in this application domain before. With previous alternative techniques, engineers would have to either ignore impacts across the boundary of decoupled components (processes) or make the worst assumption that every method outside the local component (process) is potentially impacted thus has to be inspected. Given the success in applying it to several industry-strength distributed systems, the new approach presented is able to render practical usage of dynamic impact analysis for those engineers with im-

proved effectiveness at very-low costs. Second, this novel approach is the first attempt in developing realistically useful dynamic impact prediction for common-type distributed systems, thus it serves as an important starting point for more extensive future explorations of cost-effective techniques supporting program analysis of distributed systems.

A broader impact of this dissertation lies in its contributions to the area of program analysis and, in particular, the topic of code-based dependency analysis. Since the impact-prediction techniques presented essentially compute forward dependencies at either statement or method level, akin to traditional forward (static and dynamic) program slicing, they are immediately applicable to dependency-based applications where program slicing has long been underpinnings, such as testing and debugging. For example, applying the more precise dynamic impact prediction would give more effective regression test selection and prioritization, and using the impact analysis for distributed programs would facilitate debugging in multiple-process environments. More importantly, the two novel fundamental approaches, semantic-dependency quantification and syntactic-dependency abstraction, to which the various techniques probed in this work can be reduced, reveal new characteristics of program properties and semantics as well as new ways of modeling program dependencies, all of which can be exploited to address challenges facing existing solutions. The extensive evidences of successful exploitations as such in this dissertation, on the other hand, shed new light on understanding and balancing between the cost and effectiveness dimensions in the design space of dependency analysis in general.

CHAPTER 2

BACKGROUND

The new concepts, techniques, and methodologies presented in this dissertation stand on the basis of respective previous ones, especially those on program dependency, program slicing, execution differencing, impact analysis, and logical clocks in distributed computing environments. This chapter gives necessary background in these areas with illustrative code examples and sample supporting data such as method-execution event traces. Specifically, two major categories of program dependencies are differentiated with their relationship clarified, which lays the foundations of the general field of dependency analysis; both the static and dynamic approaches to impact analysis are introduced with their differences highlighted, which sets the backdrops of the new dynamic and static impact prediction techniques proposed in this work.

2.1 Program Dependency

Program dependencies model relationships among program statements that can be computed from the text of a program. This work concerns two major classes of program dependency, syntactic dependency and semantic dependency. These two classes of dependencies are closely interrelated and both able to characterize program behaviours, yet they are also different in nature focusing on disparate aspects of program behaviours.

2.1.1 Syntactic Dependency

Syntactic dependencies [141] of a program are derived directly from the syntax of the program. These dependencies are classified as control or data dependencies, which are the

```
1  float E1(int n, float s) {
2      int g = n;
3      if (g >= 1 && g <= 6) {
4          s = s + (7 − g) ∗ 5;
5          if (g == 6)
6              s = s ∗ 1.1;
7      }
8      else {
9          s = 0;
10         print g, " is invalid";
11     }
12     return s;
13 }
```

Figure 2.1: An example (monolithic) program $E1$ used for illustrating slicing.

building blocks of program slicing [83, 192]. Informally, a statement s_1 is *control dependent* [63] on a statement s_2 if a branching decision at s_2 determines whether s_1 necessarily executes. In Figure 2.1, for example, statement 4 is control dependent on statement 3 because the decision taken at 3 determines whether statement 4 executes. This dependency is *intraprocedural* because both statements are in the same procedure (function or method). Control dependencies can also be *interprocedural*—the two statements involved in the dependency are located in different procedures [169].

A statement s_1 is *data dependent* [7] on a statement s_2 if a variable v defined (written) at s_2 might be used (read) at s_1 and there is a *definition-clear path* from s_2 to s_1 in the program for v (i.e., a path that does not re-define v). For example, in Figure 2.1, statement 12 is data dependent on statement 4 because 4 defines `s`, 12 uses `s`, and there is a path $\langle$4,5,12$\rangle$ that does not re-define `s` after 4. Data dependencies can also be classified as intraprocedural (all definition-use paths are in the same procedure) or interprocedural (some definition-use paths cross procedure boundaries).

In addition, throughout this dissertation, formal parameters are treated as defined at the entry of each procedure, and the definition of a formal parameter is regarded as data depen-

dent on the corresponding actual parameter at each statement that may call that procedure. Specially, the parameters of entry methods, like the $E1$ function of the example program $E1$, are inputs and thus are not data dependent on any statement.

2.1.2 Semantic Dependency

Semantic dependencies of a program represent the actual behaviors that the program can exhibit, which its syntactic dependencies can only *overestimate*. Informally, a statement s is *semantically dependent* on a statement t if there is any change that can be made to t that affects the behavior of s. For example, in Figure 2.1, statement 6 is semantically dependent on statements 2, 3, 4, and 5 because they could be changed so that the execution of 6 changes (e.g., by not executing anymore after the change) or the state at 6 (i.e., the value of variable s) changes. In this case, the semantic dependencies of statement 6 coincide with its direct and transitive syntactic dependencies.

However, in this example, if statement 2 just declares that g is an alias of n (i.e., it is not an executable statement) and only values of n in [1..6] are valid inputs, the condition at 3 is always true and, thus, statement 6 is *not* semantically dependent on 3 despite being transitively syntactically dependent on that statement.

More formally, as defined by Podgurski and Clarke [141], a statement s_1 is semantically dependent on a statement s_2 in a program P if and only if:

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

Importantly, although directly computing semantic dependendices is well-known as undecidable, such dependencies can be overapproximated by syntactic dependencies. Generally, that a statement $s1$ is syntactically dependent on a statement $s2$ is the necessary, but not sufficient, condition of $s1$ being semantically dependent on $s2$.

2.2 Program Slicing

Program slicing [192] determines which statements in a program may affect or be affected by another statement. *Static* slicing [83, 192] identifies such statements for all possible executions whereas *dynamic* slicing [4, 103] does this for a particular execution. (Joining the dynamic results of multiple executions is called *union slicing* [23, 78].) A (static or dynamic) *forward slice* from statement s is the set containing s and all statements directly or transitively affected by s along (static or dynamic) control and data dependencies. Because slicing is based on the transitive closure of syntactic dependencies, it attempts to (over-)approximate the semantic dependencies in the program.

For example, the *static* forward slice from statement 4 in Figure 2.1 is the set $\{4,6,12\}$. Statement 4 is included in the slice as it affects itself. Statements 6 and 12, which use `s`, are in the slice because they are data dependent on the definition of `s` at 4. Another example is the *dynamic* forward slice from statement 2 in $E1$ for input $\langle n$=0, s=1$\rangle$, which is $\{2,3,9,10,12\}$. In this case, statement 3 uses `g` to decide that statements 9 and 10 execute next (i.e., 9 and 10 are control dependent on 3) and statement 12 is data dependent on 9.

The size of a static slice is generally large [26] but varies according to the precision of the underlying points-to analysis used. If a coarse points-to analysis is used in which a pointer can be any memory address, a forward slice from a statement s that defines a pointer p would include any statement that uses or dereferences any pointer (which may or may not point to the same address as p) if the statement is reachable from s. Yet, even when using a strong (and expensive) points-to analysis, static slices can still be quite large.

2.3 Execution Differencing

Differential execution analysis (DEA), or simply *execution differencing*, is designed to identify the runtime *semantic dependency* [141] of statements on changes. Although finding all semantic dependencies in a program is an undecidable problem, DEA tech-

niques [82, 149, 162, 172] can detect such dependencies on changes when they occur *at runtime* to underapproximate (find a subset of) the set of all semantic dependencies in the program. Although DEA cannot guarantee 100% recall of semantic dependencies, it does achieve 100% precision for the dependencies it finds. This is usually better than what dynamic slicing achieves [119, 162].

DEA executes a program before and after a change to collect and compare the execution histories of both executions [162]. The *execution history* of a program is the sequence of statements executed and, at each statement, the values computed or branching decisions taken at that statement. The differences between two execution histories reveal which statements had their *behavior* (i.e., occurrences and values) altered by a change—the conditions for semantic dependency [141].

To illustrate, consider an input $\langle n$=2, s=10$\rangle$ for $E1$ in Figure 2.1 and a change in statement 4 to `s=s`. DEA first executes $E1$ *before* the change for an execution history of $\langle$2(2), 3(*true*), 4(35), 5(*false*), 12(35)$\rangle$ where each element $e(V)$ indicates that statement e executed and computed the value set V. DEA then runs $E1$ *after* the change, obtaining the execution history $\langle$2(2), 3(*true*), 4(10), 5(*false*), 12(10)$\rangle$. Finally, DEA compares the two histories and reports 4 and 12, whose values changed, as the dynamic semantic dependencies on this change at statement 4 for that input.

2.4 Impact Analysis

Change impact analysis (CIA), or impact analysis, is a software development activity that identifies potential consequences of making a change to the software, or that estimates what needs to be modified to accomplish a change [15]. CIA is crucial to software maintenance tasks such as change understanding [180], or more broadly, change management [52]. It is also widely employed in many other software-engineering tasks, such as regression testing (e.g., test selection and prioritization) [76, 134], test-suite augmentation [161], and fault localization [152]. CIA can be classified in various ways from different

prospectives. Some recent taxonomy and survey of CIA in general can be found in [109] and those on code-based CIA in [111].

There are two usage scenarios of impact analysis: (1) *predictive* impact analysis [12, 108, 134] that is applied before proposed changes are made, thus *predicts* the effects of *potential* changes, and (2) *descriptive* impact analysis (e.g., [149, 151, 162]) that is applied after changes have been made, thus *describes* the effects of *actual* changes. Since predictive impact analysis does not use or assume any knowledge about actual changes, it allows for earlier assessment of possible consequences of candidate changes being proposed or designed, and enables developers to decide how to deal with change proposals and to estimate risks of making those proposed changes [8, 52]. As declared earlier, the CIA addressed in this dissertation is in the predictive setting.

According to the scope of software information utilized, impact analysis can be classified into two categories: traceability-based and dependency-based [15]. Traceability-based analysis [51] utilizes various forms of software artifacts, ranging from requirements and specifications to design documents and source code, to compute change impacts by defining and tracing the relationships among them. In contrast, dependency-based analysis (e.g., [66, 181]) attempts to identify effects of changes on semantic dependencies among program entities, and uses syntactic dependencies to approximate those effects as computing precise and complete semantic dependency is an undecidable problem. For impact analysis, this dissertation focuses on dependency-based approaches only.

On the other hand, according to the type of underlying analysis employed, impact analysis can be divided into two main classes, static and dynamic: static impact analysis (e.g., [3, 19, 81]) relies on static syntactic dependencies among program entities (e.g., statements, methods, or classes) to discover possible effects of some entities on the rest of the program for all possible executions, while dynamic impact analysis (e.g., [12, 40, 108]) mainly employs concrete execution data to find change-effects specific to the execution information utilized. There are also some hybrid approaches that combine static and dy-

```
1  public class A {
2      static int g; public int d;
3      String M1(int f, int z) {
4          int x = f + z, y = 2, h = 1;
5          if (x > y)
6              M2(x, y);
7          int r = new B().M3(h, g);
8          String s = "M3val: " + r;
9          return s;
10     }
11     void M2(int m, int n) {
12         int w = m − d;
13         if (w > 0)
14             n = g / w;
15         boolean b = C.M5(this);
16         System.out.print(b);
17     }
18 }
19 public class B {
20     static short t;
21     int M3(int a, int b) {
22         int j = 0;
23         t = −4;
24         if ( a < b )
25             j = b − a;
26         return j;
27     }
28     static double M4() {
29         int x = A.g, i = 4;
30         try {
31             A.g = x / (i + t);
32             new A().M1(i, t);
33         } catch(Exception e) {
34         }
35         return x;
36     }
37 }
38 public class C {
39     static boolean M5(A q) {
40         long y = q.d;
41         boolean b = B.t > y;
42         q.d = −2;
43         return b;
44     }
45     public static void
46       M0(String[] args) {
47         int a = 0, b = 3;
48         A o = new A();
49         String s = o.M1(a, b);
50         double d = B.M4();
51         String u = s + d;
52         System.out.print(u);
53     }
54 }
```

Figure 2.2: An example program $E2$ used for illustrating impact analysis.

namic analysis to identify change-effects (e.g., [68, 116]. The majority of impact-analysis approaches in this dissertation belongs to the dynamic (or hybrid) category, although a static approach is also explored based on program dependency abstraction (in Chapter 7).

2.4.1 Dynamic Impact Analysis

Dynamic impact analysis uses runtime data to find the impacts that entities such as methods or changes to those entities have on a program. In particular, this work focuses on *predictive* impact analysis [15, 108] which has no knowledge of any actual changes to the program. Such a dynamic impact analysis, also referred to as dynamic impact prediction,

PATHIMPACT: M0 M1 M2 M5 r r M3 r r M4 r r x
EAS first-last events: M0[0,11] M1[1,7] M2[2,4] M3[6,6] M4[9,10] M5[3,3]

Figure 2.3: Example method-execution event traces of the program $E2$ used by PATHIMPACT (top) and EAS (bottom)—an divide-by-zero exception occurred at line 31.

takes a program P, a test suite T, and a set of methods M and outputs an *impact set* containing the methods in P potentially impacted by M when running T.

One example dynamic impact-prediction technique is PATHIMPACT [108, 135], which collects runtime traces of executed methods. For each method m in M that is queried for its impacts, PATHIMPACT uses the method execution order found in the runtime traces of P for T. The analysis identifies as impacted m and all methods executed in any trace after m. Thus, the impact set includes all methods called directly or transitively from m, all methods below m in the call stack (those into which the program returns after m), and all methods called directly or transitively from those in the call stack below m.

Figure 2.3 shows on the top row an example trace, from running the example object-oriented (OO) program of Figure 2.2, for PATHIMPACT, where `r` is a method-return event and `x` the program-exit event. The remaining marks are the entry events of methods. For query `M2`, for example, PATHIMPACT first finds {`M5`, `M3`, `M4`} as impacted because these methods were entered after `M2` was entered and then finds {`M0`, `M1`} because these methods only return after `M2` was entered. Thus, the resulting impact set of `M2` is {`M0`, `M1`, `M2`, `M3`, `M4`, `M5`} for this trace. In cases where more than one trace exists, PATHIMPACT returns the union of the impact sets for all traces.

Method traces are much shorter than statement traces. Also, traces can be compressed. Nevertheless, the *execute-after sequences* (EAS) optimization [12] exists to reduce the computation and storage costs of PATHIMPACT without losing information. This approach exploits the observation that only the first and last occurrence of each method in a trace are needed. The resulting technique, referred to as PI/EAS, keeps track at runtime of those two events per method without collecting full traces.

To illustrate, Figure 2.3 shows on the bottom the *first* and *last* values (i.e., event-occurrence time) within square brackets for the methods of P. A "timer" starts at 0 and is incremented on each event. The first event for `M2` occurs at time 2 when it is entered. All other methods execute after time 2, so the impact set of `M2` is {`M0,M1,M2,M3,M4,M5`}, same as PATHIMPACT produced above. For another example, the impact set for `M3` is {`M0,M1,M3,M4`} because only the last events for `M0`, `M1`, and `M4` occur after time 6.

As a first step in combining dynamic information and dependency analysis for dynamic impact analysis, INFLUENCEDYNAMIC [31] attempted to improve analysis precision of EAS by considering method dependencies as investigated in this dissertation. It models interface-level data dependencies via parameter passing and returns between methods by an influence graph, on which impact sets of given queries are computed. However, the modeling is incomplete and its improvements in precision was not significant yet at the cost of noticeably penalizing efficiency. For an example, given the same input change set C={M2} for the same example program and execution trace as before, the resulting impact set is the whole program like PATHIMPACT and EAS gave above.

2.4.2 Static Impact Analysis

In contrast to dynamic impact analysis which produces results specific to the particular executions utilized by the analysis, static impact analysis computes impacts that hold for any possible runtime inputs. In the prediction setting, a dependency-based static impact analysis as addressed in this dissertation takes a program P under analysis and a query set M, and outputs the impact set of M; the prediction of impacts is mainly based on the compile-time analysis (e.g., traversal) of syntactic dependencies of P.

One latest technique for such a static analysis is built on top of a method-level program dependency abstraction, called static-execute after (SEA), which characterizes the execution order among methods of a program for all possible executions of the program. By definition, the SEA relation is the union of *call*, *return-into*, and *sequentially follow* rela-

tions, all considered transitively [24]. For SEA computation, the analysis first creates the interprocedural control flow graph (ICFG) of P and then keeps entry and call-site nodes with the rest removed, followed by shrinking strongly connected components of the remaining graph into single nodes. In essence, the SEA relations among all methods of P constitute an abstraction representation of P at the method level. Yet, this abstraction is built roughly on the basis of method-level control flows thus potentially suffers from great imprecision due to excessive conservativeness. Intuitively, since SEA abstraction models dependencies among methods, a static impact analysis based on SEA [165] works by taking all methods that have SEA relations with M as potential impacts.

To illustrate how the SEA-based impact analysis works and its imprecision, consider the example program $E2$ of Figure 2.2 and method `M0` as the query. First, the query itself is trivially included in the impact set. Then, since `M0` calls `M1` and `M4`, and also transitively `M2`, `M3` (both via `M1`), and `M5` (further via `M2`), the impact set of `M0` is {`M0`, `M1`, `M4`, `M2`, `M3`, `M5`}. Similarly, for every other possible query from $E2$, the impact set is constantly the entire program. However, these results are quite imprecise for this simple program: For example, none of `M1`, `M3`, and `M4` should be included in the impact set of `M5` because none of them are either data or control dependent on `M5`. It is anticipated that properly incorporating data and control dependencies in the underlying dependency model would largely overcome such imprecision with anticipated extra yet still acceptable overheads.

2.5 Logical Clocks in Distributed Computing

A distributed system is commonly composed of components running in spatially separated processes, leading to a possibly unpredictable order in which some events occur in the system. In distributed computing, different approaches have been proposed to address this issue [64, 106, 120], of which the one by Lamport [106] uses a logical clock for each process to partially order distributed events happening in all processes with a simple algorithm, which is referred to as the *Lamport timestamping algorithm* or LTS for short.

To manage the timing of distributed events, the LTS approach first defines a logical clock C_i for each process P_i, which is a function that assigns a number $C_i\langle a\rangle$ to an event a in P_i. Based on this definition, an event a happened before another event b if the number assigned to a is less than that assigned to b, or formally

$$a \longrightarrow b \implies C\langle a\rangle < C\langle b\rangle \tag{2.1}$$

which is called the *clock condition*. Then, to maintain this clock condition during system execution, the following two rules [106] should be observed by each process:

- Each process P_i increments C_i between any two successive events.
- If event a is that process P_i sends a message m, then the message contains a timestamp T_m=$C_i\langle a\rangle$.
- Whenever a process P_j receives a message m, the process sets C_j greater than or equal to its current value and greater than T_m.

For an EAS-based approach to impact analysis of distributed systems, as the one proposed in Chapter 8, this simple LTS algorithm can be utilized to preserve the partial ordering of distributed events across multiple processes running on separated machines. Furthermore, this ordering of events would enable inferring the causality between method execution events hence the computation of impacts of one method to be changed on other methods both within and across all components (processes) of the system.

CHAPTER 3

ASSESSING THE ACCURACY OF DYNAMIC IMPACT PREDICTION

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 an important, viable 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 chapter presents the first comprehensive study of the predictive accuracy of dynamic impact analysis in two complementary ways. First, it uses massive numbers of random changes across numerous Java applications to cover all possible change locations. Then, it studies 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.

Results for both types of changes show that the most cost-effective dynamic impact analysis known—prior to the new analyses this dissertation presents—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.

3.1 Problem Statement and Motivation

Modern software is increasingly complex and changes constantly, which threatens its quality, reliability, and maintainability. Failing to identify and fix defects caused by soft-

ware 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* [15, 111, 147], 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 approach for assessing the effects of changes in a program is *dynamic* impact analysis [12, 30, 31, 107, 108, 134, 135, 151]. 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 are smaller, and thus more usable, than those obtained by static analyses as they can better represent what occurs in practice. For scalability, most dynamic impact analyses operate on *methods* as the entities that can be changed and be impacted by changes [12, 30, 31, 107, 108, 134, 135, 151]. At the statement level, dynamic slicing [4, 103, 204], in its forward version, can be used for impact analysis in greater detail but at a greater computational cost.

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., [13, 149, 162]). However, for predictive purposes—*before* the changes are even known—the usefulness of dynamic impact sets remains a mystery. An exception is the study of the CHIANTI approach [151], but this study only evaluates which *test cases* might be affected by changes—which is the goal of CHIANTI—and not which *code* might be affected. 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, this chapter first introduces a novel approach, called DEAM, for assessing the *accuracy* (precision and recall) of dynamic impact analyses. The approach uses SENSA, a sensitivity-analysis technique recently developed [40, 164], for making large numbers of random changes efficiently across the software and running dynamic impact analysis on those change locations. Although random changes do not necessarily represent typical 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 any approach based on only code repositories which, if available at all, offer selections of changes that, although 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, the study is designed to support repository changes in addition to the random changes inserted by SENSA. Specifically, this approach takes changes committed by developers into SVN repositories and also changes (bug fixes) from the SIR repository [57] made by other researchers for their own studies.

To find the *ground truth*—the code actually impacted by changes—DEAM uses *execution differencing* [149, 162, 172] on the program before and after each change is applied to determine which code is really affected—code that changes the program states (i.e., values computed at the executed statements) or occurrences (i.e., whether statements are executed or not) [141]). By design, DEAM uses the *same* test suite as used by 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 impact analysis can be for predicting actual impacts.

Using this approach with both random and repository-based changes, a comprehensive empirical study of the accuracy of dynamic impact analysis was performed on multiple Java subjects. For dynamic impact analysis, the study chose the best known and most cost-

effective prior technique from the literature: PI/EAS (see Section 2.4.1). (Another technique, INFLUENCEDYNAMIC [31], improves the precision of PI/EAS but only marginally and at a greater expense.) For different sets of changed methods in each subject, the impact set predicted by PI/EAS are obtained and thus its precision and recall computed with respect to the ground truth.

3.2 Analytical Examination

3.2.1 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 [134] for almost the same cost [12]. However, not all methods executed after m are necessarily affected by m. Thus, PI/EAS can be quite imprecise.

For the example of Figure 2.2, PI/EAS predicts that the dynamic impact set of `M2` is I=$\{$`M0,M1,M2,M3,M4,M5`$\}$. However, when applying the change to `if (w<0)` in line 13, the set of truly-affected methods is M=$\{$`M2`$\}$ according to execution differencing (see Section 2.3). Thus, the *predictive* precision of PI/EAS in this case is only $|I \cap M|/|I| = 1/6 = 16.67\%$. The imprecision is caused by the limited effects of the change, which prevents line 14 from executing but has no other consequences. Of course, for other changes in line 13, the precision might reach 100%. Therefore, to draw any conclusion, the precision of PI/EAS for many changes must be empirically studied as well.

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 unrelated.

3.2.2 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 2.2, PI/EAS reports *all* methods as possibly impacted by `M2`, which trivially has 100% recall. For another example, the impact set for `M3`, which is {`M0,M1,M3,M4`}, 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 affects the control flow of the program such that m' executes after the changed m executes. Thus, the recall of a dynamic impact prediction can be less than 100%.

To illustrate, consider again the example of Figure 2.2. If the assignment `t=-4` in line 23 changes to `t=-1`, the original divide-by-zero exception at line 31 would not occur. Thus, the control would normally proceed, leading to the execution of `M2` and `M5` after `M3`. However, the dynamic impact set for `M3` does not include `M2` and `M5`. The change modifies the control flow of $E2$ so that these two methods, which did not execute before, now executes after `M3`. In other words, despite not executing before the change, they are dynamically dependent on line 23 because, for the same input and a change in that line, they change their execution behavior. (More generally, a method that executes fewer or more times for the same input after a change, is dynamically impacted by that change [141].) Thus, the predictive recall of PI/EAS for this example change is 4/6=66.67%.

3.2.3 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, after changes to m, many methods might execute after m so yielding a low recall. Therefore, in this chapter, an *F-measure* [191] is also used to estimate the balance of PI/EAS. Particularly, the first such measure is adopted:

$$F1 = 2 \times \frac{precision \times recall}{precision + recall} \tag{3.1}$$

To illustrate, consider again the change of line 13 in method `M2` to `if (w<0)` in the example program of Figure 2.2. As one example, PI/EAS produces an impact set of {`M0,M1,M2,M3,M4,M5`} for `M2`, as is seen in Section 3.2.1. However, the actual impact set for this change is {`M2`}. As a result, precision is 16.67% and recall is 100%, thus the accuracy is $2\times(0.1667\times1.0)/(0.1667 + 1.0) = 28.58\%$. For another example, for the change in statement 23 of method `M3` from `t=-4` to `t=-1`, as is seen in Section 3.2.2, the precision is 100% and recall is 66.67%, for an accuracy of 80%.

3.2.4 Exception Handling

The PI/EAS approach, as published [12], 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 chapter.

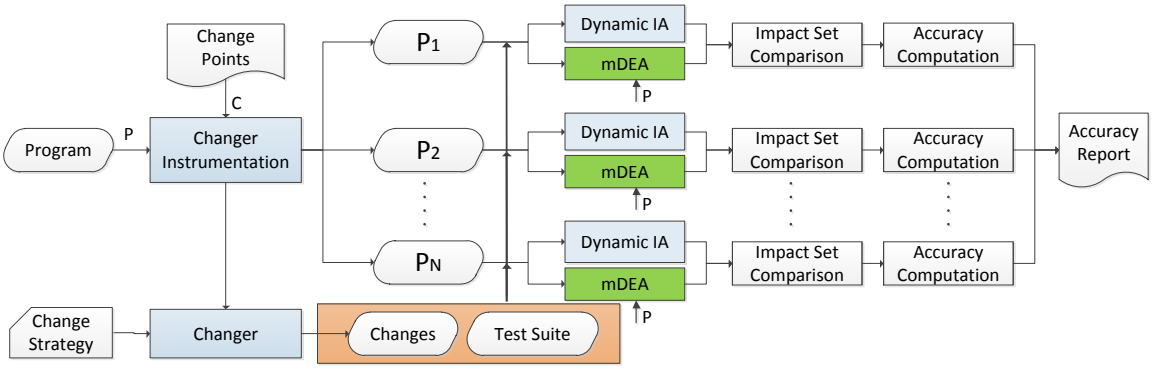

Figure 3.1: Process for estimating the accuracy of dynamic impact analyses through sensitivity analysis and execution differencing.

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 impact set. To illustrate, in Figure 2.2, 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 this work, the above problem is fixed 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 chapter, whenever mentioned, PI/EAS refers to the version of this technique with the correction applied.

3.3 Empirical Approach

For a comprehensive study of the *predictive* accuracy of dynamic impact analysis, both *artificial* and *repository* changes are considered 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.

To generate artificial changes for all analyzable methods, DEAM used the sensitivity-analysis tool SENSA [40, 164] which makes random modifications to the code of each method. These changes are thus called SENSA *changes*. Also used as changes are various bug fixes from the SIR repository introduced by other researchers to study realistic faults. Finally, numerous changes from popular SVN open-source repositories were retrieved. As opposed to the SENSA changes, the changes from both SIR and SVN repositories are referred to as *repository changes*. (The SVN changes are, nevertheless, the repository changes deemed as more representative of developer practice.) Next, we describe the experimental approach for our study with SENSA and repository changes separately.

3.3.1 Approach with SENSA Changes

To estimate the range of accuracy that developers can expect with the predictive dynamic impact analysis, a novel approach is designed, which (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.

3.3.1.1 Process

Figure 3.1 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). This study uses the same test suite *on purpose* so it can compare predicted and actual impacts under the same runtime conditions.

Lastly, DEAM 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. Finally, as shown on the right of Figure 3.1, DEAM computes statistics of these accuracy results for the final report for the subject program P.

3.3.1.2 Generation of Base Versions

At the core of DEAM 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 versions are the ones to which PI/EAS is applied and P is the "fixed" (changed) version. For comprehensiveness, DEAM selects the change set C to cover as much code and methods in P as possible, and compares the predicted impact set of each such method against the actual impacts of *all* changes in C located in that method.

To implement the changer, DEAM adapted SENSA [40, 164], a sensitivity-analysis technique and tool for Java bytecode. SENSA can modify values of primitive types and strings in assignments and branching conditions. (Other statements not supported by SENSA are normally affected directly by those supported by SENSA.) Therefore, the change set C is selected from all statements to which SENSA is *applicable*. A method with no applicable statements is called *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, our system chooses 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 $|A_m| \leq L$. 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 $\lceil max(0, i-1) \times |A_m|/L_{max} \rceil$ 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.

3.3.2 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 [57]), an experimental pipeline is created and customized for each subject application of this 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 process for this study and then how we retrieve the changed methods.

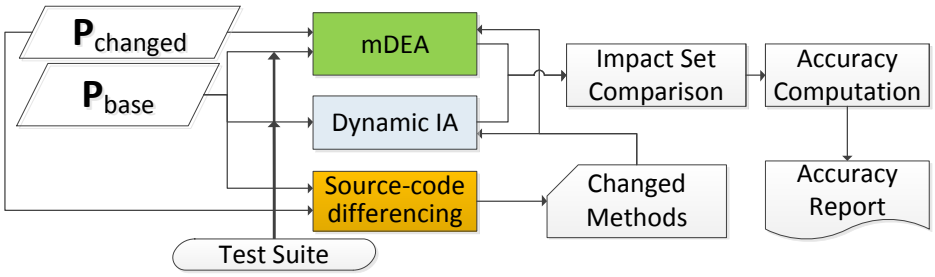

Figure 3.2: Experimental process for the accuracy study with repository changes.

3.3.2.1 Process

The experiment process for studying repository changes is shown in Figure 3.2. 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 3.1 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.

3.3.2.2 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.

TABLE 3.1

STATISTICS OF SUBJECTS FOR THE SENSA-CHANGE STUDY

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	1863	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

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.

3.4 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.

3.4.1 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 3.1 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 distinguishes the effects caused by changes from those caused by non-determinism, we *determinized* these subjects by ensuring that, for each test case, they used the same times and random values before and after each change. To check that we did not accidentally break those subjects, at least for their test suites, we re-run those test suites on the determinized versions and found no differences in outputs and assertion evaluations other than system times and random values.

We implemented our approach in Java to analyze the subjects in Java bytecode, as described in Section 3.3. We also implemented PI/EAS according to Section 2.4.1 in Chapter 2 with our exception-handling correction described in Section 3.2.4. We built this infrastructure on top of our Java-bytecode analysis and monitoring framework DUA-FORENSICS [159, 163], which is based on Soot [185], and our SENSA tool [40].

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).

3.4.2 Part I: Single-Method Changes

For this study, we used our approach described in Section 3.3.1, 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 3.2 summarizes the results of this study. For each subject, the table reports the average precision, recall, and 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 first column names the subject and the second column is *Scope*—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 columns *#C.S.*for the total number of executed and changed statements and *#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.*).

TABLE 3.2

ACCURACY OF PI/EAS FOR SINGLE-METHOD SENSA CHANGES

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	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	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	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	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]

TABLE 3.2

Continued

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
		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]
		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]**
JMeter	78%	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]
		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]**
PDFBox	67%	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]
		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]**
ArgoUML	70%	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]
		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]**
Overall (all subjects)	77%	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]

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 [189], 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, contained 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 3.1). 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 .48, its recall was .55, and its accuracy was only .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 be in need of considerable improvements.

For a more detailed view of the distribution of the accuracy of PI/EAS, Figures 3.3–3.5 present box plots for 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 can be explained by the shorter executions and smaller number of methods in Schedule1, which makes any change likely to truly impact the methods executed after it. 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 the same reasons as 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, ranging from .42 to .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. That is, many methods seem to execute under specific conditions satisfied only in changed program versions.

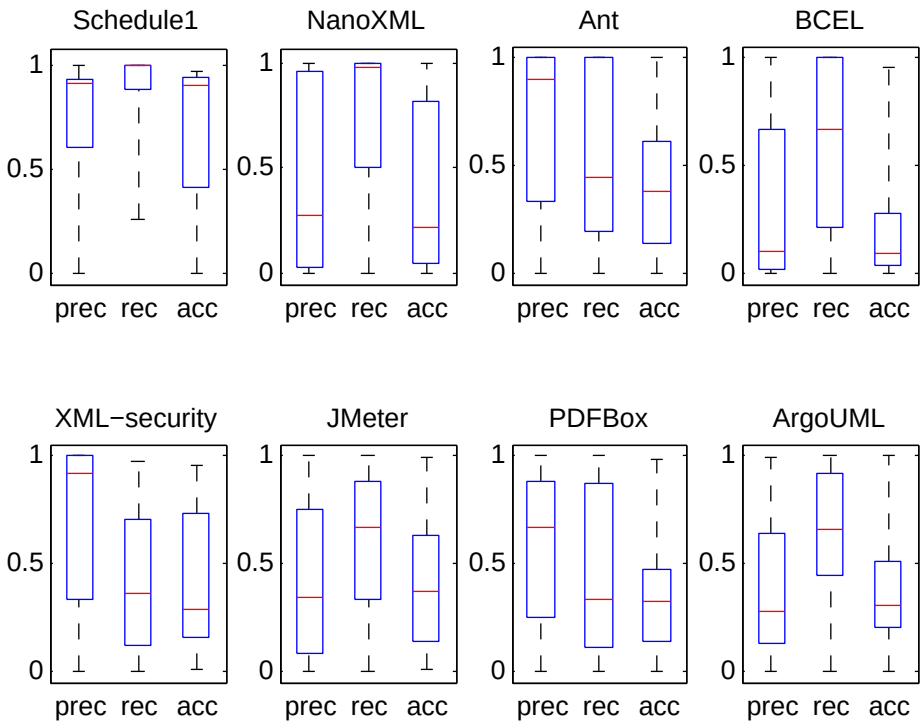

Figure 3.3: Distribution of accuracy of PI/EAS for *all* single-method SENSA changes.

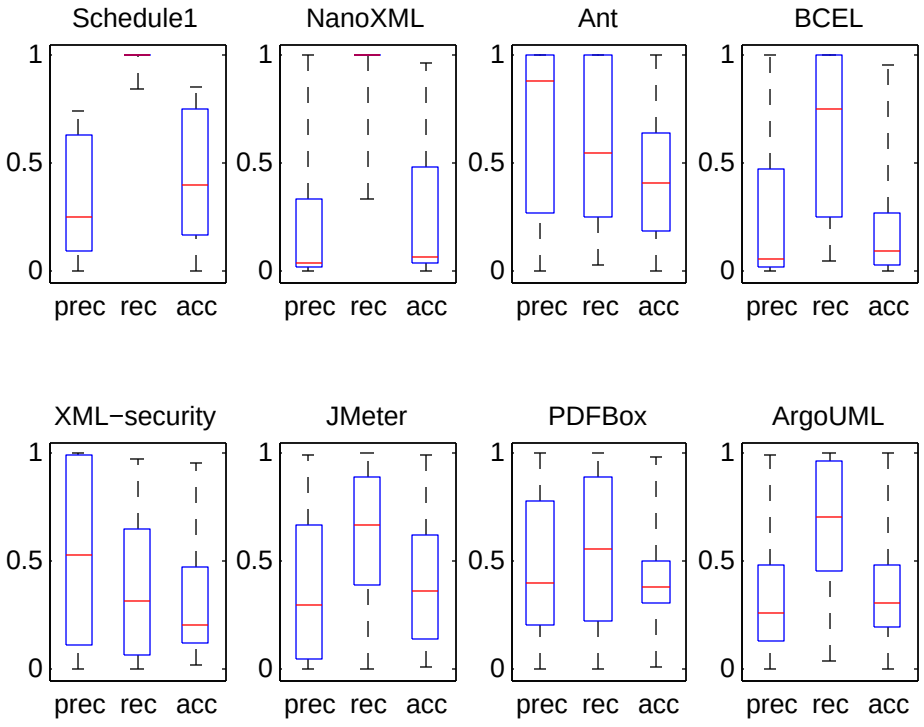

Figure 3.4: Distribution of accuracy of PI/EAS for *normal* (non-shortening) single-method SENSA changes.

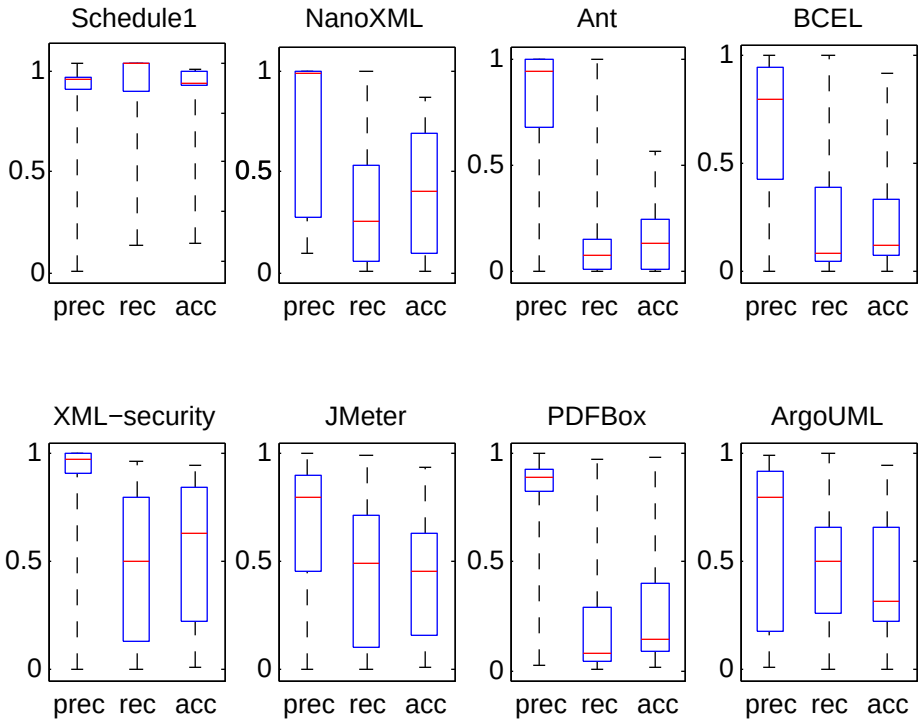

Figure 3.5: Distribution of accuracy of PI/EAS for *shortening* single-method SENSA changes.

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 relatively higher than in other subjects, 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 that closely collaborate on its centralized PDF-document processing.

When considering the N and S categories separately, 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.

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.

3.4.3 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 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., one change per method). Then, we generate a base version by applying those changes to the subject application. 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 between 2 and 10.

Table 3.3 shows the results of this study for all changes, in a format similar to Table 3.2 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 3.3. 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. The bottom ten rows recapitulate the average results over all individual data points from the eight subjects (rather than overall the per-subject summary statistics), separately for each of the ten query sizes. For example, the overall mean accuracy of all the nine-method queries is 0.39 within the confidence range of [0.38,0.41]).

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.

TABLE 3.3

ACCURACY OF PI/EAS FOR MULTIPLE-METHOD SENSA CHANGES

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]
	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]

TABLE 3.3

Continued

Subject	#Methods changed	P.S.	A.S.	#FP	#FN	Precision		Recall		Accuracy (F1)	
						mean	conf. range	mean	conf. range	mean	conf. range
	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]

TABLE 3.3

Continued

Subject	#Methods changed	P.S.	A.S.	#FP	#FN	Precision		Recall		Accuracy (F1)	
						mean	conf. range	mean	conf. range	mean	conf. range
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]

TABLE 3.3

Continued

Subject	#Methods changed	P.S.	A.S.	#FP	#FN	Precision		Recall		Accuracy (F1)	
						mean	conf. range	mean	conf. range	mean	conf. range
	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]
	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]

TABLE 3.3

Continued

Subject	#Methods changed	P.S.	A.S.	#FP	#FN	Precision		Recall		Accuracy (F1)	
						mean	conf. range	mean	conf. range	mean	conf. range
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]

TABLE 3.3

Continued

Subject	#Methods changed	P.S.	A.S.	#FP	#FN	Precision		Recall		Accuracy (F1)	
						mean	conf. range	mean	conf. range	mean	conf. range
	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]
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]

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.

3.4.4 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* [160, 188] 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 dependency.

Moreover, while PI/EAS appears at a first glance to be safe relative for the executions analyzed [12], 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. 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.

3.4.5 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 modules 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 [159, 163] and has matured considerably, and SENSA, which has been in development for two years already [40, 164]. DUA-FORENSICS and SENSA have been carefully tested over time.

Another internal threat lies in possible procedural errors in our use of the infrastructure, including the 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 in-

spected 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. Of course, many programs are almost impossible to determinize but this is a limitation of the approach and not of our studies on these subjects. This limitation, however, could affect the use of our approach for 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 [141], 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 concerns our selection of subjects and test suites with respect to software in general: These subjects are not necessarily representative of all applications to which our technique can be applied, and the test inputs may not exercise representative behaviour of respective subjects. 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 3.3.1.2, 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 [162] 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 [189] 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.

3.5 Study with Repository Changes

This section presents our study with changes from software repositories. In particular, 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 [57] to further assess the accuracy of PI/EAS with changes produced and used by other researchers.

3.5.1 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 3.4. For the SIR-change study, we obtained seven single-method changes per subject directly from the seeded faults in the SIR repository. As in 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 3.4 to all revisions of Ant and JMeter to ensure that MDEA identifies only actual impacts caused by the changes.

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 3.4 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

TABLE 3.4

STATISTICS OF 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

(*#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 [117] 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 significantly.

3.5.2 Part I: SVN Changes

In this study, for each pair of consecutive revisions per subject, we followed the experimental approach described in Section 3.3.2. We designated the older of the two revisions as

TABLE 3.5

ACCURACY OF PI/EAS USING SVN CHANGES

Subject	#C.M.	P.S.	A.S.	#FP	#FN	Precision		Recall		Accuracy (F1)	
						mean	conf. range	mean	conf. range	mean	conf. range
Ant	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	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	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)	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]

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 as the change locations for applying dynamic impact analysis (PI/EAS) and MDEA.

As Table 3.4 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 3.4 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 3.5 lists the results of this study, where the number of pairs of revisions (the third column of Table 3.4 minus 1) is the number of data points that contributed to the

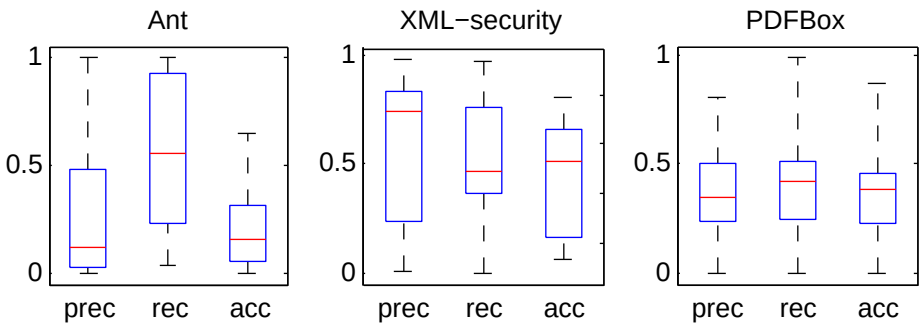

Figure 3.6: Distribution of accuracy of PI/EAS for *all* SVN changes.

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 second column (*#C.M.* for the number of *Changed Methods* as referred to before) 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.

To offer a more detailed view of the accuracy results for all revision pairs Figure 3.6 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 3.5, 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, both studies suggest that PI/EAS can suffer from very-low predictive accuracy.

TABLE 3.6

ACCURACY OF PI/EAS USING SIR CHANGES

Subject	#C.M.	P.S.	A.S.	#FP	#FN	Precision		Recall		Accuracy (F1)	
						mean	conf. range	mean	conf. range	mean	conf. range
Schedule1	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	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	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	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)	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]

3.5.3 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 of faults introduced by other researchers from the SIR repository [57]. For this study, we chose from SIR four subjects listed in Table 3.1: 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 3.6, similar to Table 3.2 in format, shows the results of this study. 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 3.2 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 impact sets to actual ones 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 confidence 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 (i.e., SENSA and SVN) changes. Nevertheless, the recall for these SIR changes is still far from perfect. The overall 13% false negatives might contain important impacts missed by PI/EAS and thus possibly by users too. Also, this increase in recall 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.

3.5.4 Discussion of All Results

As we saw in Section 3.4, 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, which are intended to represent fault fixes as one important type of code changes, show that PI/EAS can be even less precise for such changes (or at least with respect to 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, which were 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 changes 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 the artificial changes made through SENSA.

The 34% overall accuracy for SVN changes was noticeably lower than for all other types of changes studied in this work, indicating that, at least for usage scenarios that resemble the revision ranges we considered in these SVN repositories, PI/EAS (and similar dynamic impact analysis approaches) 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 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.

3.5.5 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, scripts, determinization process, 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 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* threat 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.

3.6 Related Work

Since PATHIMPACT was introduced by Law and Rothermel [108], 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 [12, 29–31, 135]. In previous work [41], 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 [134], 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 [135] 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 [12]. However, before our present work, the precision of the resulting technique (PI/EAS) was evaluated only in terms of its impact-set sizes against COVERAGEIMPACT. 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 [31] 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 [81]. 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 studied the accuracy bounds of dynamic forward slicing [94] *using sample slices*, 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 [158] has been used previously in software engineering [80, 153, 190]. Recently, we used it too with execution differencing to develop SENSA, a *statement-level* impact prediction technique [40] that finds and 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 [55, 188] that has been applied for software testing [93, 115, 202]. 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 in our study. Many techniques have been proposed for differencing programs. Some use syntactic information [65, 145, 195, 197] and others perform semantic differencing [13, 87], including execution differencing for dynamic analysis [82, 149, 162, 173]. In this work, we used execution differencing at the method level (MDEA) in conjunction with an SVN library and the srcML tool [117] to automatically download source code and find changes and their effects.

Our MDEA technique and execution differencing approaches such as DEA [162] and Sieve [149] are *descriptive* impact analyses because they describe the effects of changes as observed at runtime before and after those changes are applied. In contrast, in this chapter, we evaluated the most cost-effective *predictive* dynamic impact analysis known. This impact 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 [151], which is 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*. Our studies, in contrast, focus on *predicted* impacts on *code*. Another technique, by Goradia [71], 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 have been no other accuracy studies of predictive dynamic impact analyses.

CHAPTER 4

IMPROVING THE PRECISION OF DYNAMIC IMPACT ANALYSIS THROUGH DEPENDENCY-BASED TRACE PRUNING

Impact analysis determines the effects that the behavior of program entities, or changes to them, can have on the rest of the system. Dynamic impact analysis is one major approach to impact analysis that computes smaller *impact sets* than static alternatives for concrete sets of executions. However, existing dynamic approaches can still produce impact sets that are too large to be practically useful. To address this problem, this chapter presents a novel *dynamic* impact analysis called DIVER that exploits *static* dependencies to identify *runtime* impacts much more precisely without reducing safety and at acceptable costs. Our empirical evaluation shows that DIVER can significantly increase the precision of dynamic impact analysis at reasonable extra overheads thus offer a more cost-effective option relative to representative prior alternatives.

4.1 Problem Statement and Motivation

Different approaches to impact analysis provide different tradeoffs in accuracy, costs, and other qualities for computing *impact sets* (i.e., potentially affected entities). Static analysis can produce safe but overly-conservative impact sets [83]. Dynamic impact analysis, in contrast, uses runtime information such as coverage [134] or execution traces [108] to produce smaller and more focused impact sets than static analysis at the expense of some safety [31, 111, 135]. Yet, users looking for the *actual* behavior of the software, as represented by a set of executions, may afford unsafe results [108], making dynamic impact analysis an attractive option.

Different dynamic impact analyses also provide different cost-effectiveness tradeoffs. For example, COVERAGEIMPACT [134] is based on runtime coverage and ignores execution order, which makes it very efficient but also very imprecise [135]. Another technique, PATHIMPACT [108], is more precise by using execution order but is less efficient because it requires tracing [135]. For intermediate tradeoffs, optimizations of PATHIMPACT have been proposed [12, 30], including an incremental version [107]. Of all previous dynamic impact analyses, PI/EAS represents the most cost-effective technique, which preserves the precision of PATHIMPACT and is as efficient as *execute-after sequences* (EAS) [12].

Unfortunately, approaches such as PI/EAS can still have too many false positives. In previous work, we found that existing dynamic impact analyses such as PI/EAS can be too imprecise in practice [41]—on average, only about half of the methods reported as impacted are really impacted. In particular, for methods at the core of a software system, PI/EAS can include in the impact sets for such methods most or all methods in the system. For example, if querying the entry method of the Java application JMeter, the developer using PI/EAS will end up inspecting all 732 methods executed by the test suite, making impact analysis virtually impossible to use in practice. Therefore, because analyzing potential impacts is critical before applying changes, much smaller and precise impact sets are desirable for developers, as long as *safety* is preserved (i.e., no real dynamic impacts are missed).

An alternative is forward dynamic slicing [4], but it works at the statement level which makes it expensive and would have to be applied to all statements in a method. At the method level, hybrid techniques [116] combining static and dynamic analysis have been proposed to improve precision, such as INFLUENCEDYNAMIC [31]. However, these techniques improve precision over PI/EAS only marginally and at a much greater cost [31]. Another hybrid technique combines runtime coverage with text analysis [68] but it remains unclear how it performs with more precise dynamic data.

To address this imprecision, we developed a new dynamic impact analysis called DIVER that incurs the *same runtime costs* as PATHIMPACT but can be much more precise. PI/EAS

uses the execution order of methods to find all potential runtime impacts of a method m, which can be quite imprecise because, in general, not all methods executed after m are affected by m. DIVER, in contrast, uses a one-time static dependency analysis to *safely* reduce the impact set to only those methods that could have depended on m at runtime. Although applying that information increases the cost of querying for impact sets, the cost per query is still acceptable and multiple queries can be processed in parallel.

4.2 Approach

For our new impact analysis DIVER to be *safe* with respect to an execution set and also precise and practical, we need something much better than the *execute-after* relation of PI/-EAS, which is too conservative. The problem is that reaching a method m' after a method m at runtime is necessary for m to impact m', but not all such methods m' necessarily depend on m. To fix this problem, we propose to build first the (whole-program, static) dependency graph and then use it to find which of those methods really depend on m.

4.2.1 Overview

The process of computing the dependency graph and using it for dynamic impact analysis is shown in Figure 4.1. It works in three phases: static analysis, runtime, and post-processing. The inputs for the process are a program P, a test suite T, and impact-set queries M. To optimize the static-analysis phase, the process first runs a profiler which executes T on P to quickly find whether any exceptions are raised by a method but are not caught there or not caught at all. This lets the static analysis decide whether it can *safely skip* computing some control dependencies caused by unhandled exceptions.

After profiling, the static-analysis phase computes data dependencies (DD) and control dependencies (CD). For the CDs, the process first computes *regular* CDs caused by branches, polymorphic calls, and intraprocedural exception control-flows. For the remaining exception control-flows [169] (*ExInterCDs* in Figure 4.1), the process computes the

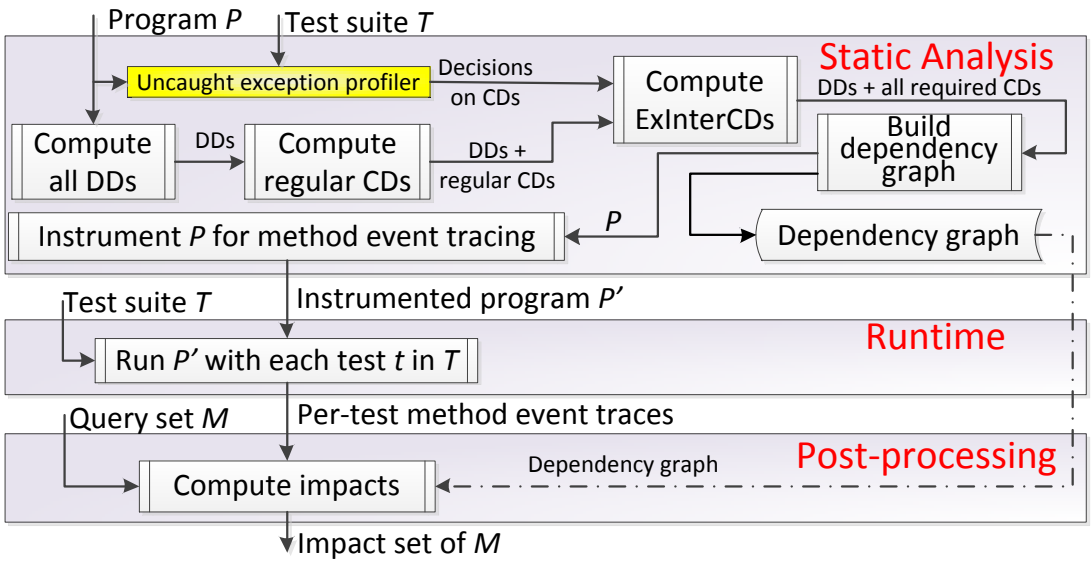

Figure 4.1: Process for dynamic impact analysis using DIVER.

CDs for the exception types that the profiler detected as not handled by the originating methods. After all dependencies are found, the dependency graph is built and passed to the *post-processing* phase (i.e., not needed at runtime).

For the runtime phase, the static analysis creates the instrumented version P' of P using only probes for monitoring *method-entry* and *returned-into* events (Section 2.4.1 in Chapter 2). The instrumented program P' is similar to that of PI/EAS except that, instead of tracking two values per method, it traces the *whole* sequence of method events—as PATHIMPACT does—because DIVER needs entire traces to determine *transitive* dependencies at post-processing. Nevertheless, DIVER compresses these traces on the fly at reasonable costs, as PATHIMPACT does, to make space costs acceptable. In all, the runtime phase executes P' with test suite T to produce one compressed method-level trace per test case.

The post-processing phase lets the user query the impact set of M. Any number of queries can be made at this point *without re-running the first two phases*. For each query, DIVER uses the dependency graph from the static phase and the traces from the runtime phase to identify all methods that depended directly or transitively on any method in M on any of those traces. M is included in the result because every method impacts itself.

$M0_e$ $M1_e$ $M2_e$ $M5_e$ $M2_i$ $M1_i$ $M3_e$ $M1_i$ $M0_i$ $M4_e$ $M4_i$ $M0_i$ x

Figure 4.2: Example execution trace of program $E2$ (see 2.2) used by DIVER.

To illustrate, consider the trace for input ⟨*a=0, b=-3*⟩ in Figure 4.2 where subscripts *e* and *i* denote the entry and returned-into events, respectively. For a query $M = \{\texttt{M2}\}$, DIVER traverses the trace to find which dependencies (from the dependency graph) are exercised due to methods executed after `M2` and, via those dependencies, which methods depended directly or transitively on any occurrence of `M2`. When DIVER finds `M2`, the impact set starts as $\{\texttt{M2}\}$. Then, the only outgoing dependency from `M2` in the graph is exercised because its target `M5` occurs next, so `M5` is impacted. Thus, DIVER finds the impact set $\{\texttt{M2}, \texttt{M5}\}$, in contrast to PATHIMPACT which reports $\{\texttt{M0}, \texttt{M1}, \texttt{M2}, \texttt{M3}, \texttt{M4}, \texttt{M5}\}$.

4.2.2 Dependency Graph and Propagation

The static dependency graph of the entire program is a key ingredient of our technique. Unlike the *system dependency graph* [83], however, the dependency graph built in the static phase of our technique does not include summary edges because DIVER is dynamic after all and, thus, does not require context-sensitive analysis. DIVER uses this graph only to prune runtime traces. *Interprocedural* (i.e., across methods) DDs in the dependency graph are classified into three types: *parameter* DDs from actual to formal parameters in method calls, *return* DDs from return statements to caller sites, and *heap* DDs from definitions to uses of *heap variables* (i.e., dynamically-allocated variables not passed or returned explicitly by methods). *Parameter* and *return* DDs are exercised at runtime only if the target method executes *immediately* after the source. Thus, the type of a DD lets DIVER at post-processing decide whether the dependency was exercised and the target method of that dependency was impacted by the source.

To facilitate our presentation of DIVER, we refer to the specific target statements of incoming interprocedural dependency edges to, and the source statements of outgoing in-

terprocedural edges from, a method as *incoming ports* (*IP*s) and *outgoing ports* (*OP*s) of that method, respectively. An impact propagating to a method via an incoming edge e will enter the method through the *IP* for e. If an impact propagates beyond this method through outgoing edges, it will exit through all *OP*s that are *reachable via intraprocedural (i.e., within the method) edges* from the *IP* for e. An impact that starts in a method will propagate through *all OP*s of that method.

4.2.3 Impact Computation

The post-processing phase of DIVER answers queries for impact sets using the dependency graph from the static phase and the traces from the runtime phase. Algorithm 1 formalizes this process.

The algorithm inputs a dependency graph G, an execution trace L and a queried method c, and outputs the impact set of c. *m(e)* gives the method associated with a method event e; *n(m)* is the set of dependency-graph nodes for all statements in method m; *m(z)* is the method to which port z belongs; *src(d)*, *tgt(d)*, and *type(d)* are the source node, target node, and type of edge d, respectively.

To maximize precision, an interprocedural edge d exercised for the i^{th} time in the trace propagates an impact to its target (IP port) only if the source (OP port) of d for that i^{th} occurrence has also been impacted. To that end, an impacted *OP* set per edge type, which starts empty at line 1, is maintained at lines 9, 17, and 25. These sets track impact propagations on ports to ensure that only the methods transitively reachable from c through impacted ports are reported as impacted. The impact set starts with the queried method c (line 2) and grows as the trace is traversed (lines 4–29).

Methods executed before the first occurrence of c cannot be impacted, so their events are skipped using a flag *start* (lines 3 and 5). Methods executed after c are checked to determine if they are impacted, for which two key decisions are made. First, the algorithm decides that the impact of c *propagates into* a method *m(e)* if there is an impacted port in

Algorithm 1 : COMPIS(dependency graph G, trace L, method c)

```
1:  ImpOPs := ∅   // map of edge type to set of impacted OPs
2:  ImpactSet := {c}  // impact set of c
3:  start := false, pre˙m := null  // preceding method occurrence
4:  for each method event e∈L do
5:      if ¬ start then {start := m(e) = c; if ¬start then continue}
6:      if e is a method-entry event then
7:          if m(e)=c then
8:              for each outgoing edge oe from n(m(e)) in G do
9:                  ImpOPs[type(oe)] ∪= {src(oe)}
10:             pre˙m := m(e) // method occurrence; continue
11:         for each incoming edge ie to n(m(e)) in G do
12:             if type(ie)=return∨src(ie)∉ImpOPs[type(ie)] then
13:                 continue
14:             ImpactSet ∪= {m(e)}
15:             for each outgoing edge oe from n(m(e)) in G do
16:                 if src(oe) is reachable from tgt(ie) in G then
17:                     ImpOPs[type(oe)] ∪= {src(oe)}
18:     else  // e is a method-returned-into event
19:         for each incoming edge ie to n(m(e)) in G do
20:             if type(ie)=parameter∨src(ie)∉ImpOPs[type(ie)] then
21:                 continue
22:             ImpactSet ∪= {m(e)}
23:             for each outgoing edge oe from n(m(e)) in G do
24:                 if src(oe) is reachable from tgt(ie) in G then
25:                     ImpOPs[type(oe)] ∪= {src(oe)}
26:     if pre˙m=m(e) then {continue}
27:     for each edge type t ∈ {parameter, return} do
28:         ImpOPs[t] \= {z | z ∈ ImpOPs[t] ∧ m(z) = pre˙m}
29:     pre˙m := m(e) // preceding method occurrence
30: return ImpactSet
```

ImpOPs that is the source of an interprocedural edge of the same type to *m(e)* (lines 11–14 for method-entry events and lines 19–22 for returned-into events).

The second key decision is to determine whether an impact *propagates out* of *m(e)* by finding the *OP* ports of *m(e)* that are reachable, via *intraprocedural* edges inside the method, from the *impacted IP* ports of that method (i.e., the target ports of impact-propagating edges according to the first decision). Those impacted *OP*s are added to *ImpOPs* to continue looking for impacts in the rest of the trace (lines 15–17 for method-entry events and

lines 23–25 for returned-into events). As for the queried method c, all of its *OP*s are added to *ImpOPs* when c executes (lines 7–10).

To determine impact propagations through interprocedural edges on the dependency graph, those edges can be classified into two categories, described next. All edges in each category share the same propagation rules.

Adjacent edge. DD edges of types *parameter* and *return* are classified as *adjacent* edges. An adjacent edge from method m to method m' models an interprocedural DD between the two methods. Through these edges, an impact can propagate from m to m' only if m' executes immediately after m. To realize this rule, an *OP* z that is the source of an adjacent edge is added to the impacted *OP* set *ImpOPs*, as other impacted *OP*s, when found to propagate the impact beyond *m(z)* in the trace. The port is then removed from that set (lines 26–28) after matching it to an *IP* in the immediate caller or callee because the corresponding parameter or return value should not be matched to *IP*s of methods that occur later in the trace. The method occurrence is tracked by *pre˙m*, initialized at line 3 and updated at lines 10 and 29.

Execute-anytime-after edge. All other interprocedural edges are *execute-anytime-after* edges. Such an edge from m to m' models an interprocedural CD or *heap* DD between these two methods such that an impact in m propagates to m' if and only if m' executes *anytime* after m in the trace. Such edges propagate impacts to their targets if their sources (*OP*s) are *impacted* when the targets are reached later. Thus, the sources of these edges are never removed from the impacted *OP* set *ImpOPs* once added to that set.

It is worth noting that the way in which propagations rules are applied depends on the type of method event being processed in the trace. For instance, no *return* edges are considered for impact propagation at method-entry events (lines 12–13) and no *parameter* edges are considered at returned-into events (lines 20–21) because of the semantics of those event types (see Section 2.4.1). Also, all *OP*s of the queried method c are marked as impacted at each entry event found for c. Thus, it is not necessary to do the same for

the returned-into events of c because the *OP*s of c are already marked as propagated at the entry of c in the trace.

In sum, according to the propagation rules of DIVER, for each event in the trace, the method associated with that event is added to the impact set if it is determined that at least one of its *IP*s is directly or transitively impacted by the queried method. After all events of the trace are processed in order, the algorithm returns as its output the resulting impact set for that trace (line 30). If multiple traces are available, one run of the algorithm per trace is required and the result is the union of the individual impact sets. Also, for the impact of multiple methods, the algorithm can be run once per method or can be easily adjusted to treat c as a set for efficiency.

4.3 Evaluation

This section presents our empirical evaluation of DIVER. Our goal was to assess the precision of this new technique and its practicality. Accordingly, we formulated the following two main research questions:

RQ1 How precise is DIVER relative to existing method-level dynamic impact analyses?

RQ2 How expensive in terms of time and space is DIVER?

4.3.1 Implementation

We first describe key aspects of our implementation of PI/EAS, our baseline for comparison, and of our new technique DIVER.

4.3.1.1 Exception Handling in PI/EAS

The original description of PI/EAS [12] deals with exceptions handled in the raising method or its caller. However, if neither method handles the exception at runtime, the *returned-into* events for all methods in the call stack that do not handle the exception will

not be logged and those methods can be mistakenly missing in the resulting impact set. To illustrate, consider the program of Figure 2.2 with `M2` as the query. If an exception is raised at `M5`, which is called by `M2` but never caught thereafter, the registers used by PI/EAS to record the *last* returned-into events for `M1` and `M0` will not be updated to reflect that these methods *executed after* `M2`. Thus, `M1` and `M0` will be missing from the impact set of `M2`.

To address this problem, we implemented a *corrected* version of PI/EAS, which we call PI/EAS$_C$. PI/EAS$_C$ captures all returned-into events by wrapping the entire body of each method in a try-catch block to identify uncaught exceptions. The added catch block, when reached by such an exception, creates the corresponding returned-into event (which would be missed otherwise) and then *re-throws* the exception to continue the execution as originally intended to preserve the semantics of the program.

4.3.1.2 DIVER

To build the dependency graph, we leveraged our dependency-analysis and instrumentation system DUA-FORENSICS [163]. For exceptional control dependencies, our implementation takes the *exceptional control flow graph* (ExCFG) provided by Soot [185] and applies both the classical algorithm for control-dependency computation [63] and the extended algorithm for interprocedural control dependencies [169].

In addition, we adapted multi-headed and multi-tailed ExCFGs by adding virtual *Start* and *End* nodes joining all head and tail nodes, respectively. Also, many types of programs, such as service daemons, contain infinite loops [150] which result in *tailless* ExCFGs. Thus, we treated all jumping statements for the outermost infinite loops as exit nodes. This treatment allowed us to directly apply the control-dependency analysis algorithms.

When computing interprocedural exception CDs, DIVER includes in the *throwable* set of each ExCFG node all exceptions, both *checked* (declared) and *unchecked* (undeclared) for that method, thrown by that node via a throw instruction or a method it calls that throws unhandled exceptions. DIVER also reuses the method-event monitoring we implemented

TABLE 4.1

EXPERIMENTAL SUBJECTS FOR DIVER EVALUATION

Subject	**#LOC**	**#Classes**	**#Methods**	**#Tests**
Schedule1	290	1	24	2,650
NanoXML	3,521	92	282	214
Ant	18,830	214	1,863	112
XML-security	22,361	222	1,928	92
JMeter	35,547	352	3,054	79
JABA	37,919	419	3,332	70
ArgoUML	102,400	1,202	8,856	211

for PI/EAS$_C$. DIVER, however, uses this feature to capture the *whole* traces it needs (just like PATHIMPACT) instead of the two registers per method needed by PI/EAS$_C$.

4.3.2 Experiment Setup

4.3.2.1 Subjects

We chose seven Java programs of various types and sizes, as summarized in Table 4.1, for our experimental study. The size of each subject is measured as the number of non-comment non-blank lines of code (LOC) in Java. Except for JABA [73], which is Java-bytecode analyzer provided directly by its authors, the other subjects, and the test suites coming with them, are the same as corresponding ones used in the accuracy study of dynamic impact analyses in Chapter 3 (see Section 3.4.1).

4.3.2.2 Methodology

For our experiments, we applied PI/EAS$_C$ and DIVER separately to each subject on a Linux workstation with a Quad-core Intel Core i5-2400 3.10GHz processor and 8GB DDR2

RAM. To obtain the method traces, we used the entire test suites provided with the subjects except for JABA, for which we used only the 70 tests for which our current implementation would not run out of memory. (We expect in the future to extend our experiment using an optimized tool and more RAM.) For each of the two techniques, we separately computed the impact sets for method in each subject. As expected, some methods are never executed and thus have empty dynamic impact sets, so we excluded them from our results.

To compare the analysis precision between the two techniques, we calculated for each query the impact set size for DIVER and PI/EAS$_C$ and the *size ratio* of the first to the second. We computed the means, medians, and standard deviations of those metrics for each subject and all queries per subject. Because both techniques over-approximate the concept of *semantic* (true) dependency [141], their results are *dynamically safe* (i.e., safe with respect to the test suites). Thus, the ratios reflect the precision improvements that DIVER attains over PI/EAS$_C$ for these test suites.

To measure and compare the efficiency of the techniques, we computed for each subject the time and space costs of their respective static-analysis and runtime phases. This was needed only once per subject and technique because all queries at post-processing reuse the results of the first two phases. For the post-processing phase, we collected the time and space costs *per query* (per method) and report the means, medians, and standard deviations of these costs for all queries per subject and overall for all subjects.

4.3.3 Results and Analysis

This section presents the results of our study. We report and discuss the relative precisions of DIVER and PI/EAS$_C$ and the costs that both techniques incur.

4.3.3.1 RQ1: Precision of Impact Analysis

Figures 4.3—4.9 present the details of the precision results for DIVER and PI/EAS$_C$. For each subject, the chart depicts the impact set sizes given by the two techniques for all

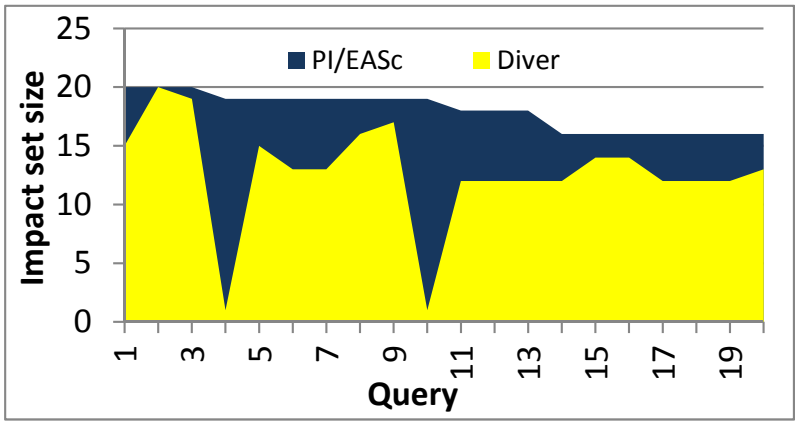

Figure 4.3: Impact set sizes of DIVER vs. PI/EAS$_C$: Schedule1.

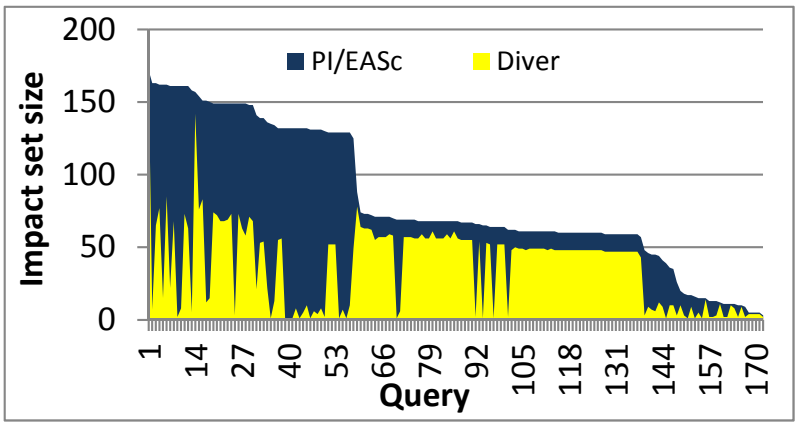

Figure 4.4: Impact set sizes of DIVER vs. PI/EAS$_C$: NanoXML.

queries for that subject. In each chart, the X axis represents the queries and the Y axis the resulting impact-set sizes. To facilitate visual comparisons, results are sorted by decreasing PI/EAS$_C$ impact-set size. For example, for the first query in Schedule1, PI/EAS$_C$ reported 20 possibly-impacted methods for Schedule1's test suite whereas DIVER reported (safely) that only 15 of them could have been impacted.

The graphs confirm that, for all queries, DIVER never produced a larger impact set than PI/EAS$_C$. We also verified that, for every query, the DIVER result was a subset of the PI/-EAS$_C$ result. Moreover, for a majority of queries, DIVER reported much smaller impact sets and, therefore, DIVER achieved a much greater precision overall. The contrast between

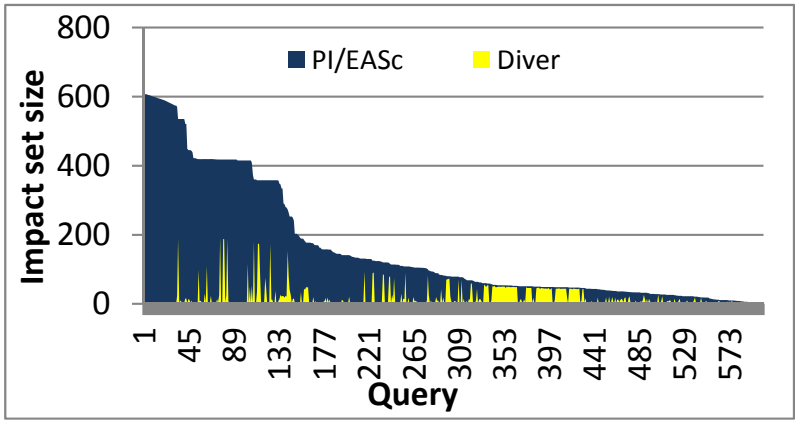

Figure 4.5: Impact set sizes of DIVER vs. PI/EAS$_C$: Ant.

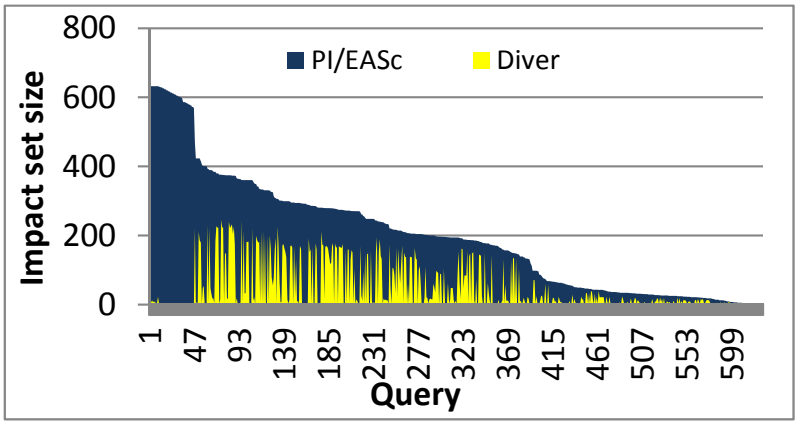

Figure 4.6: Impact set sizes of DIVER vs. PI/EAS$_C$: XML-security.

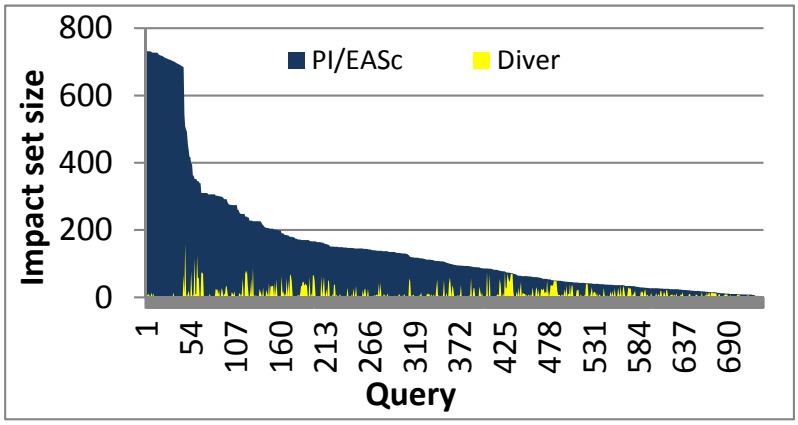

Figure 4.7: Impact set sizes of DIVER vs. PI/EAS$_C$: JMeter.

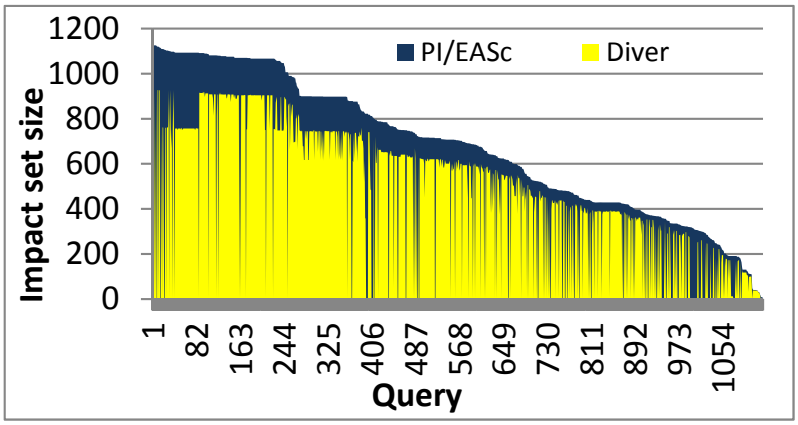

Figure 4.8: Impact set sizes of DIVER vs. PI/EAS$_C$: JABA.

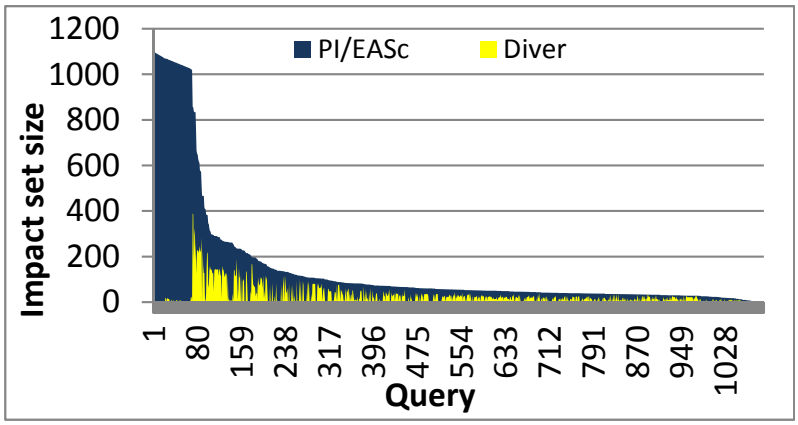

Figure 4.9: Impact set sizes of DIVER vs. PI/EAS$_C$: ArgoUML.

DIVER and PI/EAS$_C$ is considerable and it is particularly sharp for four of the five largest subjects for which, on average, DIVER computed impact sets with less than 50%, and as little as 19%, of the impacts reported by PI/EAS$_C$.

Table 4.2 provides three statistics per subject and over all queries (at the bottom row) for the corresponding data points: the mean, standard deviation (*stdev*), and median (*med*) of the impact set (*I.S.*) sizes of the baseline technique PI/EAS$_C$ and DIVER. The *#Query* column lists the number of single-method queries per subject, which is equal to the respective method-level *test coverage*. To the right, the table presents the same three statistics but for the impact-set size ratio of DIVER to PI/EAS$_C$.

TABLE 4.2

PRECISION OF DIVER RELATIVE TO PI/EAS$_C$

Subject	#Query	PI/EAS$_C$ I.S. size			DIVER I.S. size			I.S. size ratio			Wilcoxon
		mean	stdev	med	mean	stdev	med	mean	stdev	med	p-value
Schedule1	20	18.0	1.6	18.5	12.8	4.7	13.0	71.3%	24.5%	75.0%	6.65E-05
NanoXML	172	82.6	48.1	68.0	37.1	28.9	48.0	51.7%	33.1%	60.8%	2.40E-30
Ant	607	159.5	173.4	79.0	17.9	34.3	4.0	25.7%	33.6%	6.9%	2.94E-100
XML-security	632	199.8	168.4	194.0	45.1	68.1	7.0	28.8%	30.3%	12.4%	4.79E-102
JMeter	732	149.6	172.6	96.0	12.3	18.5	4.0	18.8%	25.1%	6.6%	8.58E-122
JABA	1,129	677.0	301.2	705.0	471.9	308.2	550.0	66.9%	33.7%	84.0%	1.51E-186
ArgoUML	1,098	151.0	261.2	54.0	27.6	44.9	13.0	31.5%	26.0%	33.3%	1.66E-181
Overall average		291.4	325.4	140.0	141.4	253.2	13.0	38.3%	35.1%	31.7%	3.03E-07

The results in the table confirm what can be gleaned from Figures 4.3–4.9. Large numbers of false positives for PI/EAS$_C$ were identified as such and discarded by DIVER. For example, PI/EAS$_C$ identified 151 methods on average in its impact sets for ArgoUML, whereas DIVER reported only 28 with a mean ratio of 31.5%. (These values are means of ratios—not ratios of means.) Also, the large standard deviations indicate that the impact-set sizes fluctuate greatly across queries for every subject except Schedule1. The comparatively-small median sizes in many cases imply that large deviations are caused by a few queries with large impact sets.

For Ant, XML-security, JMeter and ArgoUML, the medians are especially small compared to the means, which indicates that DIVER has an especially-greater precision than PI/EAS$_C$ for many queries. The results also suggest that DIVER is even stronger with respect to PI/EAS$_C$ for larger subjects, which are more representative of modern software. For JABA, however, DIVER has larger impact sets and ratios. We examined JABA and

found tight inter-module couplings. Thus, PI/EAS$_C$, which reports impacts simply on execution order, tends to guess correctly more impacts than for other subjects. Finally, for the smaller subjects Schedule1 and NanoXML, DIVER is less effective than average possibly due to the proximity and interdependency of their few methods.

We applied two statistical analyses to these results. The first one is the Wilcoxon signed-rank one-tailed test [191] for all queries in each subject and also for the set of all queries in all subjects. This is a *non-parametric* test that makes no assumptions on the distribution of the data. The last column in Table 4.2 shows the resulting p-values. For $\alpha = .05$, the null hypothesis is that DIVER is not more precise than PI/EAS$_C$. These p-values show strongly that the null hypothesis is rejected and, thus, the superiority of DIVER is statistically significant for these subjects and test suites.

The second analysis is Cliff's delta [48] for the *effect size* of the precision differences, which is also non-parametric. We used a 95% confidence interval per subject using the sizes of the impact sets for DIVER and PI/EAS$_C$ as the experimental and control groups, respectively. For all subjects but Schedule1, the effect sizes are .98 to 1.0 with confidence intervals within [.95, 1.0]. For Schedule1, the effect size is .95 with interval [.78, .98]. These values mean that, if the results *were normally distributed* (just for the discussion), about 84% of the DIVER impact sets would be smaller than for PI/EAS$_C$ [49]. For the other 16%, the impact sets would be the same—PI/EAS$_C$ cannot outperform DIVER as for any query the impact set of the latter is always a subset of that of the former.

In all, for seven Java subjects of different types and sizes, DIVER can *safely* prune 38% of the impact sets computed by PI/EAS$_C$. This amounts to an increase in precision by a factor of 2.61 (i.e., by 161%) over the (almost) best existing method-level technique.

4.3.3.2 RQ2: Efficiency

Table 4.3 reports the time costs of each phase of each technique, including the time costs of uncaught exception profiling (column *Prof.*), static analysis, test-suite execution

TABLE 4.3

TIME COSTS IN SECONDS OF DIVER AND PI/EAS$_C$, INCLUDING THE OVERHEADS OF PROFILING UNCAUGHT EXCEPTIONS FOR DIVER

Subject	**Prof.**	**Static analysis**		**Runtime**			**Post-processing**					
							PI/EAS$_C$			DIVER		
		PI/EAS$_C$	DIVER	Normal	PI/EAS$_C$	DIVER	mean	stdev	med	mean	stdev	med
Schedule1	12.7	4.8	5.6	4.0	10.1	15.7	0.7	0.1	0.8	14.6	6.0	14.2
NanoXML	12.1	11.3	14.4	0.4	1.0	5.4	0.1	0.1	0.0	6.2	8.8	0.5
Ant	29.2	27.3	142.4	1.2	1.5	2.0	0.1	0.1	0.0	3.2	7.6	0.1
XML-security	37.1	33.4	157.7	4.3	4.8	14.8	0.0	0.0	0.0	7.4	9.6	0.4
JMeter	50.5	38.3	371.9	12.1	12.6	14.8	0.0	0.0	0.0	2.3	7.8	0.2
JABA	62.1	55.3	289.2	11.0	11.9	14.0	0.3	0.2	0.3	78.3	82.5	59.2
ArgoUML	190.1	172.3	7465.2	8.2	9.7	11.2	0.0	0.1	0.0	15.9	58.2	0.1
Overall average	81.8	72.8	2046.5	7.7	8.6	11.6	0.1	0.2	0.0	26.4	60.0	0.9

for the non-instrumented program (*Normal*) and for both techniques, and post-processing. All costs except for post-processing were incurred only once per subject. For the last phase, we show per subject and overall the means, standard deviations, and medians of query costs. The last row shows averages weighted by the number of queries per subject.

The profiling numbers suggest that automatically finding the static-analysis settings is cheap—a minute or less for most subjects and three minutes for ArgoUML. As expected, for static analysis, DIVER incurred higher costs than PI/EAS$_C$. For both techniques, these costs increase with the size of the program, with DIVER growing faster. However, other than for ArgoUML, the DIVER static analysis finished within 6.2 minutes, which seems reasonable because this is done only once per program to support all possible queries. For a larger subject such as ArgoUML, a two-hour static analysis seems acceptable, which can be done during nightly builds.

TABLE 4.4

SPACE (STORAGE) COSTS OF DIVER AND PI/EAS$_C$

Subject	**Runtime data (MB)**		**Dependency graph size (MB)**
	PI/EAS$_C$	DIVER	
Schedule1	1.0	8.2	.1
NanoXML	.4	2.4	.9
Ant	1.0	2.0	5.5
XML-security	.5	3.8	5.4
JMeter	.5	.8	9.0
JABA	2.5	15.0	20.4
ArgoUML	1.4	7.3	40.8
Overall average	1.3	6.8	18.5

For the runtime phase, both techniques had small overheads. For the post-processing phase, due to the traversal of longer traces, DIVER needed more time than PI/EAS$_C$. However, the average cost of 26.4 seconds per query seems practical, albeit a little less for JABA. Yet, 78.3 seconds per queried method still seems acceptable, and multiple queries can be processed in parallel since the impact computation of any single query is independent of that computation for other queries.

Table 4.4 shows the space costs of the two techniques for the relevant phases. As expected, the DIVER traces (compressed on the fly with GZIP) use more space than the PI/EAS$_C$ registers. In addition, DIVER incurs the cost of storing the dependency graph (DG) during static analysis for use later in queries. One expected correlation can be seen between space use—especially trace sizes—and post-processing times (Table 4.3), which is that longer traces lead to greater post-processing costs. However, these space costs of DIVER at a few dozen MB or less are quite low for today's storage availability.

In all, DIVER achieved significantly greater precisions for these subjects at acceptable time and space costs. Only the static-analysis cost for ArgoUML suggests that more optimizations in our implementation or the dependency graph construction algorithm, or both, might be needed for larger subjects with longer executions.

4.3.4 Threats to Validity

The main *internal* threat to the validity of our results is the possibility of implementation errors in DIVER. However, DIVER is based on Soot and DUA-FORENSICS, both of which have matured over the years. In addition, we manually checked individual results of every phase of DIVER for our subjects. Another *internal* threat is the possibility of errors in our experimental and data-analysis scripts. To reduce this risk, we tested and debugged those scripts and checked their operation for each phase.

Yet another *internal* threat is the risk of missing static dependencies due to Java language features such as reflection, multi-threading, and native methods. However, we confirmed that, for our study subjects running on their test suites, there was no use of such features except for native methods in Java libraries, which are modeled by DUA-FORENSICS [163]. For JMeter and Ant, the two subjects that used reflection, we refactored their code to obtain reflection-free, *semantically-equivalent* versions (at least according to their test suites). Thus, one more internal threat is the possibility of internal errors in these refactored subjects not observed in the test-case outcomes. Also, for Schedule1, we used a version translated from C to Java. To avoid relevant threats, we verified that the outcomes of all 2,650 test cases remained the same.

The main *external* threat to the validity of out study is our selection of Java subjects and their test suites. This set of seven subjects does not necessarily represent all types of programs from a dynamic impact-analysis perspective. To minimize this threat, we picked our subjects such that they were as diverse as possible in source-code size, application domain, coding style, and functional complexity.

Another *external* threat is inherent to dynamic analysis: the test suites we used cannot exercise all behaviors of the respective subjects. Many methods were not covered by the test suites, so we could not apply the techniques to those methods. Thus, our results must be interpreted in light of the extent of the ability of those test suites to exercise their subjects. To address this issue, we chose subjects for which reasonably large and complete test suites were provided—at least from a functional point of view. Also, most of these subjects and test suites have been used by other researchers thus can be seen as good benchmarks.

The main *construct* threat is in our experimental design. Without additional knowledge, we gave the same weight to the impact sets of every method. Yet, developers may find some methods more important than others in practice, and thus the reported average precision improvements of DIVER might not represent the actual precisions that developers will experience. Also, we used impact set sizes as inverse indicators of precision assuming that recall is not affected. This is the safe for program understanding, but for predicting the impacts that *actual* changes will have, recall might be less than perfect if those changes modify the control flow of the program to execute methods not reported by DIVER.

Finally, a *conclusion* threat to validity is the appropriateness of our statistical analyses. To reduce this threat, we used a non-parametric hypothesis test and a non-parametric effect-size measure which make no assumptions about the distribution of the data (e.g., normality). Another *conclusion* threat concerns the data points analyzed: we applied the statistical analyses only to methods for which impact sets could be queried (i.e., methods executed at least once).

4.4 Related Work

Law and Rothermel introduced PATHIMPACT [108] to compute dynamic impacts based on the execution order of methods and extended their technique for incremental impact analysis of evolving software [107]. Apiwattanapong and colleagues proposed EAS [12] to safely reduce the size of PATHIMPACT traces, while Breech and colleagues proposed an

online version [29] and other performance optimizations [30] for PATHIMPACT. This technique improves the efficiency of PATHIMPACT but not its precision. DIVER also uses the whole method-level execution traces, as PATHIMPACT does, but it does so while *improving* its precision significantly.

Many researchers have used structural information for impact analysis. Orso and colleagues presented COVERAGEIMPACT [134] which uses method-level test coverage and static forward slicing. And their technique was found to achieve greater efficiency but lower precision than PI/EAS [135]. Similar to EAS, the static execute-after (SEA) approach [90] also utilizes execution orders to find impacted methods but based on the static call graph [19] instead of runtime traces. Other static impact analyses exist [15, 109, 111], including some based on static slicing [3], coupling measures [143], change-type classification [174], and concept lattices [176, 182]. Static impact analyses are more conservative than dynamic ones, which makes them less precise but more general. DIVER also performs a static analysis—for dependency graph construction—but, instead of producing a conservative impact set directly from that graph, it uses the dependency graph to guide the analysis of runtime traces for precise dynamic-impact analysis.

Impact-analysis techniques based on dependency analyses other than static slicing have been explored as well. Sun and colleagues proposed OOCMDG [174] and LoCMD [176] to model dependencies among classes, methods, and class fields, which are directly derived from class-method memberships, class inheritance, method-call relations, and use relations between classes and methods. Their techniques were also extended for impact analysis with hierarchical slicing [175] and for multiple levels of granularity [140]. These models, however, include only structural dependencies (e.g., call edges) based on object-oriented features whereas our dependency graph models all interprocedural data and control dependencies for all types of software. CALLIMPACT [74] computes change impacts using variable definitions and uses but, like SIEVE [149] and CHIANTI [151], it is a *descriptive* impact analysis—describing the effects of concrete changes between two given versions.

INFLUENCEDYNAMIC [31] combines dependency analysis and dynamic information for impact analysis. However, it considers only a subset of the method dependencies that DIVER models and its precision improvements over PATHIMPACT are only marginal. Huang and Song [84] extended INFLUENCEDYNAMIC for objected-oriented programs by adding dependencies between fields, as the RWSETS tool does [60], and improve it further by including dependencies due to runtime class inheritances [85]. Another hybrid technique, SD-IMPALA [116], extends its pure static-analysis based version IMPALA [81] to improve the recall of impact analyses by utilizing the runtime traces of method events and field accesses. However, these approaches model partial data dependencies only and none of them achieve a noticeably-better precision than PI/EAS as DIVER remarkably does.

Dynamic slicing [4], specifically in its forward version [98], seems to be an option for dynamic impact analysis at first glance. While possibly viable for relatively simple, small programs, this fine-grained technique would be too expensive for impact analysis [12, 107]. Although dynamic dependencies among methods, hence the set of methods potentially impacted by the method (i.e., impact-set query) that contains the slicing criterion, can be easily obtained by first running the slicing algorithm and then uplifting statement-level dynamic slices to method level, the total computation would incur at the least the time cost of dynamic slicing. Yet, dynamic slicing itself is known to have scalability issues , especially with large and long-running software applications [204–206]. In essence, DIVER computes dynamic dependencies at method level too, but it does so much more efficiently through (method-level) dependency approximation [114] using lightweight dynamic information (i.e., method-execution event traces) in collaboration with static dependencies.

CHAPTER 5

PRIORITIZING FINE-GRAINED IMPACT ANALYSIS VIA SEMANTIC-DEPENDENCY QUANTIFICATION

Software is constantly changing. To ensure the quality of this process, when preparing to change a program, developers must first identify the main consequences and risks of modifying the program locations they intend to change. This activity is called (predictive) change-impact analysis. However, existing impact analyses suffer from two major problems: lack of fine granularity (e.g., working at method level or even coarser levels) and large sizes of their resulting impact sets. Finer-grained analyses (e.g., statement-level) such as slicing give more detailed impact sets which, however, are also even larger in size. While different impact-set reduction approaches have been proposed at different levels of granularity, the challenge persists as very-large impact sets are still produced, impeding the adoption of impact analysis due to the great costs of inspecting those impact sets.

To address these challenges, this chapter presents a novel dynamic-analysis technique called SENSA that combines sensitivity analysis and execution differencing. SENSA not only provides fine-grained (statement-level) impact sets but also *prioritizes* potential impacts via *semantic-dependency quantification* for program slices. We evaluated the benefits of impact prioritization using SENSA with respect to static and dynamic forward slicing via an extensive empirical study of open-source Java applications and three case studies. Our results show that SENSA can offer much better cost-effectiveness than slicing in assisting developers with impact inspection and bug cause-effects understanding.

5.1 Problem Statement and Motivation

Existing impact-analyses techniques usually work at coarse levels of granularity only, such as methods and classes (e.g., [12, 33, 68, 108, 140, 151]). While these techniques provide a first approximation of potential change impacts, they do not distinguish which *statements* in particular in the impacted methods or classes are actually responsible for the impacts. Without such further details that are indeed needed for some tasks, such as fault cause-effect understanding, users dealing with those tasks would end up with wasting time examining irrelevant parts in the entire coarse entities. Moreover, the coarse techniques can miss code-level dependencies that are not captured by higher-level structural relationships among classes or methods [186], which leads to false negatives. Therefore, coarse analyses can inaccurately report change impacts, which, according to the study in [156], is a critical issue that developers encountered in practice when using impact analysis.

Another issue with today's impact analyses is that they often produce very-large impact sets [3, 31, 108, 134, 135, 165]. To understand the effects of potential changes, developers need to inspect the impact sets of those potential changes. One strategy is to check the entire impact set, akin to the "retest-all" strategy in regression testing [154], but this strategy incurs too much effort. At the method level, existing dynamic impact analyses can report over 60% false positives in their impact sets [41], whereas static impact analyses are even more imprecise [134]. Such imprecision tends to result in very large impact sets. For example, impact sets of a static impact analysis for WebKit can be as large as 41,000 methods [165]. Fully inspecting such enormous impact sets is impractical.

At the statement level, the forward version of *program slicing* [83, 192] could be an attractive option for fine-grained impact analysis (e.g., [3]), yet it can also be inaccurate. For instance, *static* slicing can report too many potentially-affected statements that are not truly affected [26] whereas *dynamic* slicing [4, 25, 103] gives smaller results but can still suffer from imprecision [94, 119, 162] in addition to low recall. In an industrial study by Acharya and Robinson [3], the large impact-set sizes were one of the most critical problems of us-

ing impact analysis—the impact sets, as large as 343,758 lines of code, cannot possibly be fully explored by developers. Indeed, producing large impact sets, hence incurring excessive inspection costs, is one of the main challenges to today's impact analysis, even more so as modern software is increasingly complex and software evolution is usually subject to tight schedules due to the volatility of requirements, technology and knowledge [148]. In fact, a problem reported as even more critical than the coarse granularity is that developers' time for for impact analysis is quite restricted [156].

A possible solution to this challenge is impact-set reduction, akin to the test selection approach in regression testing [154]. Unfortunately, researchers have tried various such approaches but the resulting impact sets can be still quite large. For instance, a large amount of research has aimed to reduce the impact-set sizes by improving the precision of impact analysis, both at the statement level (e.g., [6, 171, 205]) and coarser levels (e.g., [31, 35, 108]). However, the size problem persists as large impact sets are still produced frequently by these techniques. Moreover, impact-set reduction techniques can have additional drawbacks such as sacrificing scalability and risking the loss of result safety.

However, we believe that *not all statements in the large impact sets are equally strongly impacted*, thus not all of them have always the same *priority* for inspection. For tasks such as change understanding and program comprehension, developers need to isolate the *key* relationships between components of a program that explain its behavior. In particular, when preparing to change a program, developers must first identify the *main* consequences and risks of applying the changes before properly designing and propagating those changes.

With these motivations, we propose to distinguish the *degree of influence* of statements in a program and separate affected statements by *likelihood* (or *strength*) of being truly affected so that users can decide which code to inspect first. This impact prioritization approach, akin to test case prioritization [155], has not yet been exploited for impact analysis.

Our approach is different from, yet complementary to, existing impact-set reduction techniques. Instead of reducing the impact-set sizes, we *prioritize* entities (statements)

in the impact sets so that impacts of higher priority can be inspected earlier than others, with the priority of a statement measured by the *strength of impact* of the change on that statement. Our approach effectively provides an (under-)approximation of the *semantic* dependencies [141], an undecidable problem, by finding a subset of all statements *definitely* affected by another statement and the *frequency* at which they are affected.

Our new technique and tool, called SENSA [40], combines *sensitivity analysis* [158] and *execution differencing* [82, 149, 162, 173, 203] to estimate those frequencies as impact strengths. Unlike a different approach by Masri and Podgurski [119], which observes information-flow strengths for existing executions, SENSA quantifies semantic dependencies between statements for a much larger set of executions, which are derived from existing executions to explore the semantic dimensions of such dependencies. By systematically modifying program states in a way similar to *value fuzzing* [178], SENSA can better determine the effects of statements and find dependencies that could otherwise be missed.

As a dynamic analysis, SENSA can be applied only to statements executed at least once. However, the number of executions considered by SENSA is large because of the many modified executions it created and analyzed. Also, dynamic analyses have proven quite useful when executions exist (e.g., [12, 25, 61, 103, 108, 160, 173]) or can be generated (e.g., [17, 28, 75, 95, 136, 139, 179]). The goal of SENSA is to address the existence and frequency of semantic dependencies for a *specific set* of runtime behaviors—those caused by the provided executions and alternative executions derived automatically from them.

To evaluate SENSA, we empirically compared the effectiveness of SENSA and of static and dynamic forward slicing for *predictive impact analysis* on various candidate change locations across six Java subjects. For each location, we ranked statements by impact frequency (likelihood) according to SENSA and by breadth-first search as proposed by Weiser for slicing [192]. Then, we estimated the effort a developer would spend inspecting the rankings to find all statements impacted by actual changes (both bug-fixing and bug-introducing changes) in those locations for the same test suite. Our results indicate

with statistical confidence that SENSA outperforms both forms of slicing at *predicting the actual impacts* of such changes by producing more accurate rankings describing the real dependencies on those locations.

To further demonstrate the benefits of prioritizing impacts with SENSA, we also performed three case studies to investigate how well SENSA and program slicing isolate the *cause-effect chains* that explain how bugs propagate to *failing points* (crash locations or outputs) and, thus, how those bugs can be fixed. To achieve this goal, first, we manually identified the statements that are impacted by each bug and that propagate the erroneous state to a failing point. Then, we computed how well SENSA and slicing isolate those event chains. While unable to draw general conclusions, we found again that SENSA was better than slicing for isolating the specific ways in which buggy statements make a program fail.

5.2 Approach

The goal of SENSA is, for a program P and an input set (e.g., test suite) T, to *detect* and *quantify* the effects on P of the *runtime behavior* of a statement C or any *changes* in C. To this end, SENSA combines sensitivity analysis [158] and execution differencing [162].

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

5.2.1 Overview and Example

Every statement s has a role in a program. This role is needed, for example, when s is being considered for changes and *predictive* impact analysis must be performed for s. Ideally, to find the role of s, one should identify all statements that *semantically* depend on s [141]. Semantic dependency considers *all possible changes* to the computations performed at s for all possible inputs to represent the *effects* of the behavior of s.

Unfortunately, computing semantic dependency is an undecidable problem, although for *individual* changes and executions the semantic dependencies can be determined. For (descriptive) impact analysis, DEA can tell which statements are dynamically impacted by a change. However, *before* developers can design and apply a change to a statement, they first need to know the *possible effects* of changing that statement.

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

By design, the modifications made by SENSA are constrained to *primitive values* and strings computed by statements. To determine the sensitivity of the program on a computation for any other object, such as a method call c to a data structure, the user must identify the statement(s) of interest in that operation that compute the supported values that make up the data structure and apply SENSA to that (those) statement(s) instead of c.[1]

We use the example program $E1$ in Figure 2.1 again to illustrate how SENSA works for inputs (2,10) and (4,20). Suppose that a developer asks for the effects of line 1 on the rest of $E1$. SENSA instruments line 1 to invoke a state modifier and also instruments the rest of the program to collect the execution histories that DEA needs. The developer also configures SENSA to modify variable g in line 1 with values in the "valid" range [1,6].

For each input I, SENSA first executes $E1$ without changes to provide the *baseline* execution history for DEA. Then, SENSA re-executes $E1$ on I five times—once for each other value of g in range [1,6]. We list the execution histories for this example and test input

[1]Naturally, SENSA can be extended in the future to modify all non-primitive values—a more generic modification. How to make those changes useful and valid remains to be investigated.

TABLE 5.1

EXECUTION HISTORIES OF PROGRAM $E1$ FOR AN EXAMPLE TEST INPUT (2,10)

Run	**Execution history**
baseline	⟨2(2), 3(true), 4(35.0), 5(false), 12(35.0)⟩
modified #1	⟨2(1), 3(true), 4(40.0), 5(false), 12(40.0)⟩
modified #2	⟨2(3), 3(true), 4(30.0), 5(false), 12(30.0)⟩
modified #3	⟨2(4), 3(true), 4(25.0), 5(false), 12(25.0)⟩
modified #4	⟨2(5), 3(true), 4(20.0), 5(false), 12(20.0)⟩
modified #5	⟨2(6), 3(true), 4(15.0), 5(true), 6(16.5), 12(16.5)⟩

(2,10) on Table 5.1. The execution histories for the test input (4,20) are similar to these. Finally SENSA applies DEA to the execution histories of the baseline and modified runs of each test case and computes the *frequency* (sensitivity) (i.e., the fraction of all executions) at which each line was *impacted* by having its state or occurrences changed.

A developer can use these frequencies directly or through a ranking of affected statements by frequency. In our example, the resulting ranking is $\langle\{2, 4, 12\}, \{5, 6\}, \{3, 9, 10\}\rangle$ where lines 2, 4, and 12 are tied at the top because their states (the values of g and/or s) change in all modified runs and, thus, their sensitivity is 1. Lines 5 and 6 come next with sensitivity 0.2 as line 5's state changes in one modified run and 6 executes for one modification on each input (when g changes to 6)—line 6 is not reached in the baseline execution. Lines 3, 9, and 10 rank at the bottom because 3 never changes its state and 9 and 10 never execute. In contrast, static forward slicing ranks statements by dependency distance from line 1. These distances are found by a breadth-first search (BFS) of the program-dependency graph—the inspection order suggested originally by Weiser [192]. Thus, the result for static slicing is the ranking $\langle\{2\}, \{3, 4, 5, 10\}, \{6, 9, 12\}\rangle$. For dynamic slicing,

a BFS of the dynamic dependency graph [4], which is the same for both inputs, yields the ranking $\langle\{2\}, \{3, 4, 5\}, \{12\}\rangle$.

To illustrate the usefulness of these rankings in this example, consider their application to *predictive* change-impact analysis. Suppose that the developer eventually decides to change line 2 to `g = n + 2`. The *actual* set of impacted statements for this change and input is $\{2, 4, 5, 6, 12\}$. This is exactly the set of statements placed at the top two levels of the SENSA ranking. In contrast, static slicing predicts statement 3, from among the four predicted statements, as the second most-likely impacted statement, but that statement is not really impacted. Static slicing also predicts statement 6 as one of the least impacted, even though this statement is actually impacted after making this concrete change.

Dynamic slicing, perhaps against intuition, performs even worse than static slicing in this example. The ranking for dynamic slicing misses the actually-impacted statement 6 and predicts statement 3, which is not really impacted, as the second most-impacted. Note that, in the context of this chapter, a *forward* version of relevant slicing [6] would not perform better either, although in general it may achieve a higher recall, than dynamic slicing. In this example, the forward relevant slice is identical to the dynamic slice.

Naturally, the usefulness of SENSA depends on the executions and modifications chosen as well as application-specific aspects such as the actual change made to the statement analyzed by SENSA. If, for example, the range [0,8] is used instead of [1,6] to modify g in line 2, the sensitivity of statements 5 and 6 will be higher because they will not execute for some of the modifications. (The sensitivity of statement 4 does not change as it is always affected either by state changes or by not executing when g is not in [1,6].) Also, the sensitivity for 3, 9, and 10 will be non-zero because g can now be outside [1,6]. In this particular case, the SENSA ranking does not change but the frequencies change, and the developer's assessment of the possible impacts of line 2 might rely on those quantities.

Program $E1$ is a simple example that contains only eight statements, thus it does not require much effort to identify, quantify, and rank potential impacts, regardless of the ap-

proach used. Yet, in more realistic cases, the differences in prediction accuracy between SENSA and both forms of slicing can be substantial, as shown in Sections 5.3 and 5.4.

5.2.2 Applications

SENSA has, potentially, a myriad of applications in software engineering. Virtually any task that uses program dependencies and slicing can benefit from semantic-dependency discovery and quantification. In this chapter, we focus on the two applications that originally motivated our development of SENSA.

The first application is *predictive* change-impact analysis [15, 108]. Before designing and applying any changes, a developer identifies the location(s) that might need to be changed, and the implications of changing that location(s) must be assessed. Most impact analyses point the developer to the potentially-affected parts of the program which might need to be inspected, possible changed too, and later tested against the changes made. Because the affected parts can be large, SENSA quantifies those impacts to help the developers focus first on the most-likely impacted parts. We study this application in Section 5.3.

The second application is *failure comprehension*. After localizing a fault, to fix it properly, a developer might need to understand how that fault leads to a *failure point* somewhere else in the program—a failed assertion, a bad output, or the location of a crash. SENSA can isolate the most-likely ways in which that fault propagates to the failure point by giving high scores to the statements that participate in that propagation. Program slicing can report many ways (effects) in which a fault propagates to a failure point, but not all such effects are necessarily erroneous. SENSA, as demonstrated in Section 5.4, can focus the developer better on the effects that matter.

5.2.3 Scope

As a dynamic analysis, SENSA requires the existence of at least one test case that covers the analyzed statement. Frequently, however, test suites do not achieve 100% coverage of

their programs. Therefore, SENSA is only applicable to covered statements or when new covering executions can be created. As with any dynamic analysis, the results of SENSA are also subject to the quality and representativity of the test cases and, in particular, those covering the analyzed statement C. The more behaviors are exercised for C, the more dependencies SENSA can find and quantify, and the more representative those behaviors are of the real usage of C, the more accurate the quantification will be.

The quality of SENSA's predictions is also a function of the accuracy modification strategies for modeling the effects of C's behavior. Intuitively, the more modifications are made to C, the more effects of C are reflected in the results. Therefore, the user will make these strategies modify C and re-execute the program as many times as it is practical according to that user's budget. It is worth noting that, in a change-impact analysis scenario, the test suite for the subject will be used as the input set for SENSA. This test suite, perhaps with a few updates, will be used again after the changes. Thus, the change effects that the developer will experience will be subject to the same or similar runtime conditions as the one exploited by SENSA for predictive change-impact analysis.

5.2.4 Formal Presentation

SENSA is a technique that, for a statement C (e.g., a candidate change location) in a program P with test suite T (or, more generally, an input set) computes for each statement s in P a relevance value between 0 and 1. These values are estimates of the frequencies or sizes of the influences of C on each statement of P (or, more precisely, the static forward slice of C). Next, we present SENSA's process and algorithm.

5.2.4.1 Process

Figure 5.1 shows the diagram of the process that SENSA follows to quantify influences in programs. The process logically flows from top to bottom in the diagram and is divided into three stages: (1) *Pre-processing*, (2) *Runtime*, and (3) *Post-processing*.

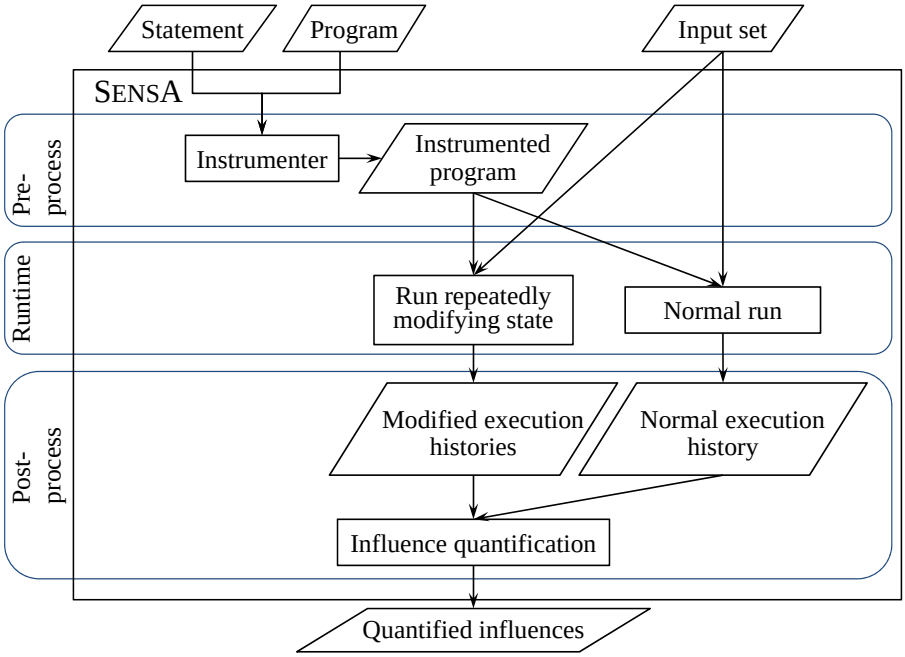

Figure 5.1: The SENSA process for dependency quantification.

For the first stage, at the top, the diagram shows that SENSA inputs a *Program* and a *Statement*, which we denote P and C, respectively. In this stage, an *Instrumenter* inserts at C in program P a call to a *Runtime* module (which executes in the second stage) and also instruments P to collect the execution histories of the program for DEA, including the values written to memory [162]. (Branching decisions are implicit in the sequences of statements in execution histories.) The result in the diagram is the *Instrumented program*.

In the second stage, SENSA inputs an *Input set* (test suite) T and runs the program repeatedly modifying the value produced by C (*Run repeatedly modifying state*), for each test case t in T. SENSA also runs the instrumented P without modifications (*Normal run*) as a baseline for comparison. For the modified executions, the runtime module uses a user-specified *strategy* (see Section 5.2.5) and other parameters, such as a value range, to decide which value to use as a replacement at C each time C is reached. Also, for all executions, the DEA instrumentation collects the *Modified execution histories* and the *Normal execution history* per test case.

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

5.2.4.2 Algorithm

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

For the second stage, the loop at lines 7–16 determines, for all test cases t, how many times each statement is impacted by the SENSA modifications. At line 8, SENSA executes t without modifications to obtain the baseline execution history. Then, line 10 executes this same test for the number of times indicated by the parameter given by SENSA-MODS(). Each time, SENSA modifies the value or branching decision at C with a different value.[2]

If not specified, a default value of 20 from SENSA-MODS() is used, so that as many as 20 distinct modifications will be made at C, while constrained by the value range that

[2]As discussed in the overview, modifications are only made to primitive values computed at C. Any modifications involving non-primitive values requires applying SENSA not to C but to the statement(s) that compute the primitive parts of such values.

Algorithm 2 : SENSA(program P, statement C, test suite T)

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

the user provided and the data type of the value computed at C. The maximum number of distinct values can be less if there are only a few distinct values (e.g., two for boolean types). This default value was not arbitrarily determined but an empirical threshold identified in our initial experiments, through which we found that, at least for the subjects and changes we studied, more than 20 modifications do not increase the cost-effectiveness of SENSA.

For the third stage, line 11 asks DEA for the differences between each modified run and the baseline run. In lines 12–14, the differences found are used to increment the *frequency* counter for each statement affected by the modification. Then, the loop at lines 17–19 divides the influence counter for each statement by the total modified runs, which normalizes this counter to obtain its influence frequency in range [0,1]. This influence frequency is used by SENSA as the measure of impact likelihood of a change on a statement affected by that change, with higher (lower) value indicating stronger (weaker) impact.

5.2.5 Modification Strategies

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

Users can specify parameters such as the modification strategy to use to pick each new value for a statement. The choice of values affects the quality of the results of SENSA, so we designed two different strategies while making it straightforward to add other strategies in the future. The built-in modification strategies are:

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

Moreover, SENSA *skips* modifications that make a program run for a very long time or forever— when the running time of the modified program exceeds 10 times the time taken by the original program; modifications that cause early terminations do not need special treatment, though, since SENSA can work the same way as with normal executions.

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

As an example of other potential strategies to add, consider one based on values observed in the past at C that replace the values currently computed at C. First, the strategy would collect all values observed at C for the test suite. Then, the strategy picks iteratively

from this pool each new value to replace at C. This strategy would seek values that are more meaningful to the program P because P has computed them at some point.

The way in which SENSA modifies program executions, especially when the *Random* strategy is adopted, is similar in spirit to value fuzzing [178], a security testing technique that inputs to the program anomalous data including invalid, unexpected, and random values. However, the values that SENSA generates for the modifications are not necessarily invalid or unexpected. In fact, SENSA tries to produce valid values by allowing users to specify the range of value modification and the increment, for the *Random* and *Incremental* modification strategy, respectively.

It is worth emphasizing that, despite of the modification strategy used, the changes SENSA made to a statement are strictly constrained to *the value* computed at that statement—SENSA does not change the statement itself. It is also noteworthy that SENSA modifications are different from the mutations adopted in mutation testing [93], although both are intended to change program states. First, SENSA neither changes operators in the program nor uses any mutant operators [131], as mutation analysis does. Second, SENSA guarantees producing and using different values, both from the original ones and from those used before during the same execution, for different modifications. In contrast, applying a mutation operation may not change the program state from the original one (e.g., changing a logical operator at a predicate might not change the value computed there).

5.3 Study: Impact Prediction and Prioritization

To evaluate SENSA, we first studied its ability to *predict* the impacts of changes of a particular kind—*fault fixes*—in the context of the test suites of the programs containing those changes. We compared these predictions with those of *static* and *dynamic* slicing using Weiser's traversal of slices [192].[3] The rationale is that the more closely a technique

[3]Relevant slicing [6, 77] is an option in between for comparison, but a forward version must be developed first. We expect to do this in future work.

TABLE 5.2

EXPERIMENTAL SUBJECTS FOR SENSA EVALUATION

Subject	Short description	#LOC	#Tests	#Changes
Schedule1	priority scheduler	301	2,650	7
NanoXML-v1	XML parser	3,521	214	7
XML-security-v1	encryption library	22,361	92	7
JMeter-v2	performance tester	35,547	79	7
Ant-v2	project builder	44,862	205	7
PDFBox 1.1	tool for pdf files	59,576	32	7

approximates the actual impacts (including the ordering of impacts) that changes will have, the more effectively developers will focus their efforts to maintain and evolve their software. To this end, we formulated three research questions—two about effectiveness and the third about efficiency:

RQ1: How accurately does SENSA predict the real impacts of changes versus slicing?

RQ2: How accurately does SENSA predict these impacts in comparison to slicing under budget constraints?

RQ3: How expensive is it to use SENSA?

The first two questions address the comparative *effectiveness* of SENSA overall and per ranking-inspection effort—when users can only inspect a portion of the predicted ranking in practice—respectively. The third question targets the *practicality* of the technique.

5.3.1 Experimental Setup

We implemented SENSA in Java as an extension of our dependency analysis and instrumentation system DUA-FORENSICS [159, 163], which also provides static and dynamic slicing and execution differencing for computing the actual impacts of changes.

DUA-FORENSICS computes forward static slices by traversing data and control dependencies starting from the slicing criterion (change locations), with data dependencies and control flow are determined using context- and flow-insensitive points-to analysis. DUA-FORENSICS also performs forward dynamic slicing based on dynamic dependency monitoring. More details about DUA-FORENSICS and its dependency analyses for slicing can be found in [163]. To run our experiments, we used a Linux workstation with an 8-core 3.40GHz Intel i7-4770 CPU and 32GB DDR2 RAM.

We studied six Java subjects of different types and sizes and seven changes (fault fixes) per subject, for a total of 42 changes. (Seven is the maximum number of available changes for most of the subjects.) Table 5.2 describes the subjects. Column *#LOC* shows the size of each subject in non-comment non-blank lines of code. Column *#Tests* shows the number of tests for each subject. Column *#Changes* shows the number of changes studied per subject. More details on these subjects These subjects are a subset of those used in the accuracy study in Chapter 3 (see Section 3.4.1), although here we used the v2 (instead of v0) version of Ant. For subjects that exhibit some non-determinism (e.g., JMeter and Ant used here), we adopted the same manual determinization approach as utilized in that study as well.

Table 5.3 lists the ids of the faults whose fixes we used as the changes for our experiment. The faults for all subjects except PDFBox were introduced by other researchers and contributed to the SIR repository [57]. For PDFBox, we obtained version 1.1.0 from its SVN repository [11].and we used a random number generator to select seven statements. For each statement, we randomly applied a mutation operator (and value when appropriate) from among the *sufficient* mutant operators identified by Offutt and colleagues [131]. Our SENSA homepage [40] gives details on these PDFBox changes.

For each fault, the "fixed" (changed) program is the program *as is*—without the fault. Each fault fix modifies, adds, or deletes one to three lines of code. For XML-security and JMeter, only seven changes are covered by at least one test case (coverage is a minimum requirement for dynamic analysis). Thus, for consistency, we chose the first seven changes

TABLE 5.3

FAULT IDs WHOSE FIXES USED AS CHANGES IN OUR STUDY

Subject	**Source**	**Fault ids**
Schedule1	SIR	v1, v2, v3, v4, v5, v6, v8
NanoXML	SIR	v1s1, v1s2, v1s3, v1s4, v1s5, v1s6, v1s7
XML-security	SIR	v1s2, v1s3, v1s5, v1s14, v1s16, v1s17, v1s20
JMeter	SIR	v2s1, v2s2, v2s5, v2s6, v2s11, v2s13, v2s19
Ant	SIR	v2s3, v2s4, v2s5, v2s6, v2s7, v2s8, v2s15
PDFBox	SVN	Version 1.1.0 – s1, s2, s3, s4, s5, s6, s7

provided with the other SIR subjects. For Schedule1, v7 involves two methods so we chose v8 instead. (Future plans include studying SENSA on multiple changes.)

5.3.2 Methodology

Figure 5.2 shows our *experimental* process for SENSA. The inputs of SENSA are a program, a statement (e.g., a candidate change location), and the test suite of the program. SENSA quantifies the runtime influence of this statement on every other statement of the program. The output of SENSA that we use is the set of program statements ranked by decreasing influence. For tied statements in the ranking, the rank assigned to all of them is the average position in the ranking of those tied statements. To be compared against static slicing, the SENSA ranking includes at the bottom, tied with influence zero, those statements in the static forward slice *not found* by SENSA to be affected.

Similarly, because forward dynamic slicing usually does not find all statements truly affected by potential change locations, our empirical approach also assigns to the statements in the static forward slice not found by dynamic slicing an influence of zero and adds them to the bottom of the dynamic slicing ranking to enable comparisons with SENSA and static

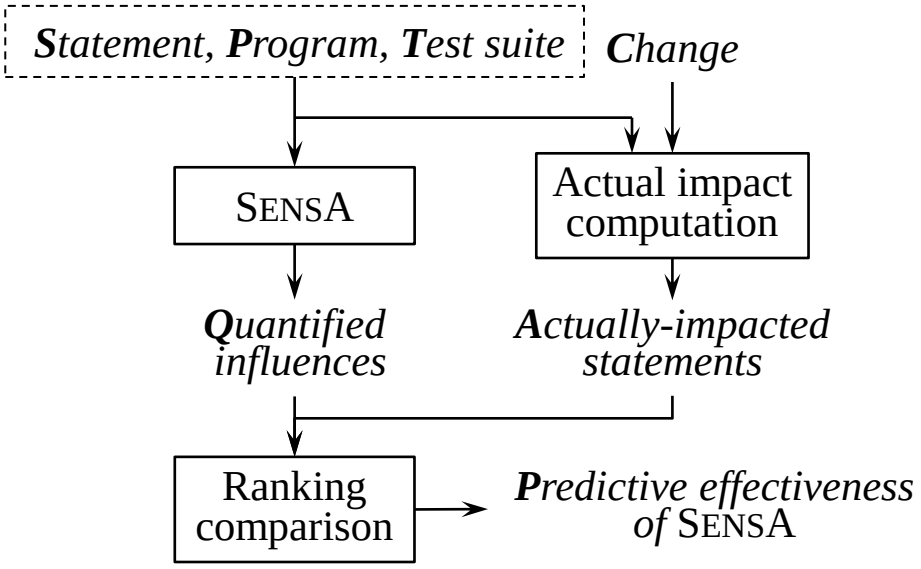

Figure 5.2: The SENSA process for impact prediction, where the location is known but not the actual change, and ground truth (actual impacts) to evaluate those predictions.

slicing. Figure 5.3 illustrates the relationships among the three types of rankings for the example of Section 5.2.1.

To the right of the experimental process diagram (Figure 5.2), an *Actual impact computation* takes the same three inputs and a change for the selected statement. This procedure uses our execution-differencing technique DEA [162] to determine the *exact* set of statements whose behavior changes when running the test suite on the program before and after this change. It is crucial to note that this step of computing actual impacts with DEA on both program versions (before and after the change is made) is used only for comparative evaluation purposes—SENSA, as a *predictive* impact-analysis technique, does not assume or use any knowledge about the actual changes.

The procedure *Ranking comparison* at the bottom of the diagram measures the *predictive effectiveness* of the SENSA ranking for the selected statement—when the actual change is not even known—by comparing this ranking with the ranking of the set of actually-impacted statements after the change is designed and applied to that statement. This *actual-impact* ranking is computed the same way as is the SENSA ranking—by run-

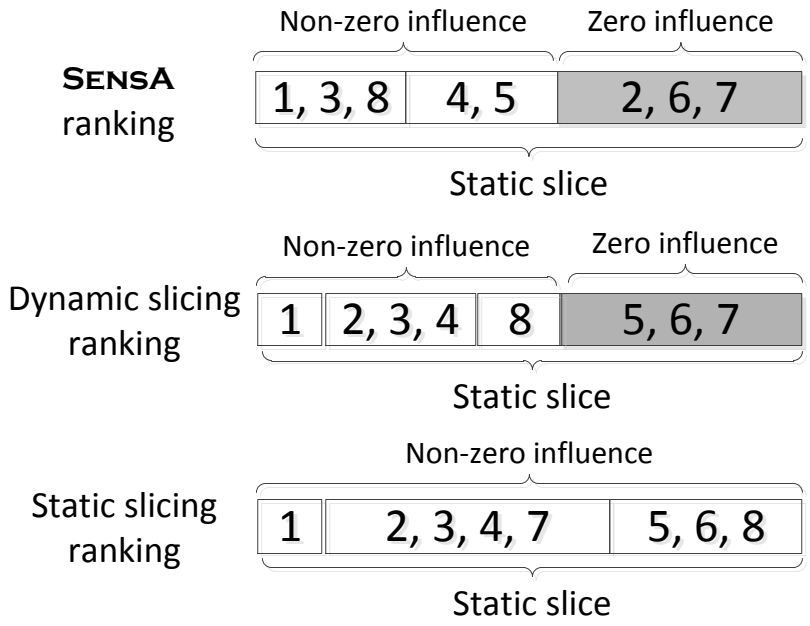

Figure 5.3: An illustration of the three impact-ranking strategies and the relationships among them, using the example program and rankings of Section 5.2.1. Statements sharing the same rank are placed in one cell, and the ranks decrease from left to right. In each ranking, statements with *non-zero influence* are those found to be affected, while statements in the static slice but not found by the strategy are all assigned *zero influence* and appended to the bottom of the ranking. As such, each of these rankings includes the entire static slice.

ning DEA on the versions before and after the change—except that for the actual-impact ranking the modified program is the version in which the *actual* change has been applied.

The experimental process makes a similar comparison, not shown in the diagram, for the rankings obtained from *breadth-first searches* (BFS) of the dependency graphs for static and dynamic slicing. BFS is the traversal order of slices proposed by Weiser [192]. We should note that, while the semantics of slicing do not guarantee a stronger impact of the slicing criterion on statements closer to the top of the slice, as an impact-prioritization strategy, the ordering of statements in a slice ranking is the result of a BFS traversal of the dependencies of those statements on the criterion. Therefore, for developers who inspect the impacts in a forward direction, it is natural to compare the BFS ordering of slices to the impact-strength-based ranking of SENSA. For dynamic slicing, we join the dynamic slices for all executions that cover the selected statement. This is known as a *union slice* [23].

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

Ideal case. As a basis for comparison against the studied techniques, we computed the inspection cost for the *ideal* scenario for each change. This ideal case corresponds to a ranking in which all statements impacted by the change are placed at the top of that ranking. No other ranking can make a better prediction. The ideal ranking is the actual-impact ranking which includes at the bottom all statements in the static forward slice that are not in the actual impact set. Those statements have zero influence and are tied at the bottom of the ranking, as we do for SENSA and dynamic slicing for meaningful comparisons.

For RQ1, we computed, for all changes of each subject, the average inspection costs for the entire rankings for SENSA, static slicing, and dynamic slicing. For each of these three impact-prioritization ranking strategies, we calculated this inspection cost for each change by dividing the sum of the ranks produced by the strategy for that change by the sum of the worst-case ranks (i.e., the length of the ranking given by the strategy), of all *actually-impacted* statements with respect to that change. Suppose L and A are the ranking given by a strategy and the actual-impact ranking for the change, respectively, the inspection cost for L is computed as

$$\frac{\sum_{\text{statement } s \in A} \text{the rank of } s \text{ in } L}{|A| \times |L|} \tag{5.1}$$

Then, we report such ratios as percentages for each change and compute the average over all seven changes per subject.

Example. Consider again the example discussed in Section 5.2.1 which has 8 lines that are all in the static slice. For SENSA, lines 2, 4, and 12 are tied at the top of the ranking with

average rank (1+2+3)/3 = 2 and lines 5 and 6 are tied at average rank 4.5. SENSA does not detect differences in lines 3, 9, and 10, which get tied at the bottom with rank 7. If a change is actually made that impacts lines 2–5 and 12, the *average inspection cost (effort)* of using the predicted SENSA ranking (before making the change) is the average rank of those lines divided by the static slice size, or (2+2+2+4.5+7)/(5×8) = 43.75%. For static and dynamic slicing, the average efforts are computed similarly with respect to the actual impacts.

For RQ2, we computed, for each ranking, the percentage of actually-impacted statements found in each fraction from the top of the ranking—for fractions $\frac{1}{N}$ to $\frac{N}{N}$, where N is the size of the ranking. The result for each change in each subject is a set of two-dimensional points, with each point (x, y) representing that $y\%$ of the actually-impacted statements can be found by inspecting the top $x\%$ of the statements in the ranking. Let L be the ranking for that change, to obtain those points, the experiment process traverses L to calculate y for each x when incrementing x by $\frac{1}{N}$.

To *merge such points across all changes* per subject for comparison purposes, we first linearly interpolated the data points for each change with an interval of 0.1% (i.e., 1,000 points for each change in the subject) and then computed the average per point for all seven changes. We report, visually using a plot of those points, the average cost-effectiveness curve per subject. Note that the plots are *usually not smooth curves*, even for the ideal case, because the size of L can be much larger than that of the corresponding actual impact set after all zero-influence static-slice statements are added to L.

For RQ3, we measured the execution time costs of the subjects for their provided test suites and the time each phase of SENSA takes on that program and test suite.

Test suite choice. By design, the test suites we use to compute the SENSA and dynamic-slicing predictions *before* making changes are the same we used to find the actual impacts (the ground truth) *after* making the changes. To the casual reader, using the same test suite to obtain both the prediction and ground truth might seem biased. However, we chose to do so because developers will normally use the entire test suite of the program for SENSA

before they decide their changes and then the same (occasionally updated) test suite to observe the actual impacts *after* defining and applying those changes. Therefore, using the same test suite before and after is not only appropriate, but necessary for evaluation.

5.3.3 Results and Analysis

5.3.3.1 RQ1: Overall Effectiveness

Data. Table 5.4 presents the average inspection efforts per subject (seven changes each), using Equation 5.1 to obtain the cost per change, and the average effort and standard deviation for all 42 changes. As explained before, the units are percentages. For the *Ideal* case, the effort is an absolute value representing the minimum possible effort—the best possible prediction. For each of *Static slicing*, *Dynamic slicing*, and SENSA with *Random* (SENSA-*Rand*) and *Incremental* (SENSA-*Inc*) strategies, the table shows the average effort required to find *all* actual impacts in their respective rankings.

For example, for XML-security, the best position on average in the ranking for all statements impacted by the changes is 5.0% of the static forward slice. On top of this, static and dynamic slicing add 26.94% and 40.37% average inspection effort, respectively, for a total of 31.94% and 45.37%. These extra efforts can be seen as the *imprecision* of the techniques. Meanwhile, SENSA-*Rand* and SENSA-*Inc* impose 8.15% and 16.49% extra effort, respectively, over the ideal case—considerably less than slicing.

Analysis. The *Ideal case* results indicate that the number of statements impacted in practice by the changes in our study, as a percentage of the total slice size, decreased with the size of the subject—from 47.90% in Schedule1 down to 5% or less in the four largest subjects. This phenomenon can be explained by two factors. First, the larger subjects consist of a variety of loosely coupled modules and the changes, which are thus more scattered, can only affect smaller fractions of the program. The second factor is that static slicing will find connections among those modules that are rarely, if ever, exercised at runtime—aliasing in larger object-oriented subjects is a major reason for the imprecision

TABLE 5.4

AVERAGE INSPECTION EFFORTS FOR BUG-FIXING CHANGES

Subject	**Ideal (best) case**	**Average effort**			
		Static slicing	Dynamic slicing	SENSA Rand	SENSA Inc
Schedule1	47.90	50.14	48.34	48.01	48.01
NanoXML	8.84	22.71	27.09	20.27	22.37
XML-security	5.00	31.94	45.37	13.15	21.49
JMeter	0.15	9.26	24.65	7.50	7.51
Ant	3.21	39.16	41.55	29.84	23.76
PDFBox	2.72	39.77	47.51	34.73	34.73
mean	11.30	32.16	39.08	25.59	26.31
standard deviation	17.26	16.52	14.47	20.92	21.86
		p-value w.r.t. static slicing:		0.0037	0.0121

of slicing. For Schedule1, however, there is little use of pointers and most of the program executes and is impacted on every test case and by every change. In fact, an average of 97.8% of the statements in the slices for Schedule1 were impacted. These factors explain the much greater inspection efforts needed by all techniques in this subject.

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

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

For NanoXML, the ideal average effort is much lower at 8.84%. Thus, impact prediction can be much more effective in this case. For this subject, SENSA-*Rand* was the best variant of SENSA, requiring 2.44% less effort on average than static slicing. SENSA-*Inc*, however, was only slightly better than static slicing.

For XML-security, in contrast, both variants of SENSA—especially SENSA-*Rand*—were considerably more effective than static slicing, which required more than six times the ideal effort to isolate all impacts, whereas SENSA-*Rand* required less than three times the least possible effort. For JMeter, however, SENSA-*Inc* was almost as good as SENSA-*Rand*. Both outperformed upon static slicing. Although the difference was not large, the levels of effort below 10% make that difference important.

For Ant, SENSA-*Inc* was the best variant with a considerable decrease of 15.4% in effort compared to static slicing. Finally, for PDFBox, both variants of SENSA reduce the average effort with respect to static slicing by about 5%.

Importantly, on average for all 42 changes, both versions of SENSA outperformed static slicing, with SENSA-*Rand* requiring 6.57% less effort than static slicing and 13.49% less than dynamic slicing to capture actual impacts. The standard deviation for both variants of SENSA, however, is greater than for both forms of slicing, which suggests that they are less predictable than slicing.

Statistical significance. To assess how conclusive is the advantage of both variants of SENSA over static slicing—the best slicing type—we applied to our 42 data points a Wilcoxon signed-rank one-tailed test [191] which makes no assumptions about the distribution of the data. Both p-values, listed in the last row of Table 5.4, are less than .02,

TABLE 5.5

AVERAGE INSPECTION EFFORTS FOR REVERSE ANALYSIS

Subject	Ideal (best) case	Average effort			
		Static slicing	Dynamic slicing	SENSA Rand	**SENSA** Inc
Schedule1	47.90	50.15	48.30	48.05	48.03
NanoXML	8.84	22.64	20.66	11.23	12.24
XML-security	5.00	31.91	45.16	9.02	9.33
JMeter	0.15	9.26	20.46	8.63	9.34
Ant	3.21	37.27	33.45	15.94	15.32
PDFBox	2.72	39.77	42.60	28.60	28.58
mean	11.30	31.83	35.11	20.24	20.47
standard deviation	17.26	16.40	17.11	20.35	20.10
		p-value w.r.t. static slicing:		7.16E-06	1.48E-05

indicating that the superiority of both types of SENSA—especially SENSA-*Rand*—is statistically significant.

Reverse analysis. To understand the factors behind these results, we looked closely at how the techniques operated on these changes. We found that, because these are fault fixes and the predictive techniques were applied to the *faulty* versions of the programs, for many of these faults the executions covering them stopped shortly after, typically with an unhandled exception. Static slicing does not use these executions, but SENSA and dynamic slicing do. As a consequence, SENSA often had less data available than usual. In many cases, SENSA could not "guess" the modifications that would fix the fault and therefore missed the impacted statements that execute only when the fault is fixed—after the real change is applied.

This problem does not occur if SENSA analyzes executions that proceed normally after the candidate change location, which is the case for most other change-impact prediction scenarios. Therefore, we also studied the *reverse* versions of our changes (i.e., bug-introducing changes) where the fixed program is used instead. Although the reverse versions of these fault fixes might not be as common and representative of software changes, they nevertheless correspond to modifications that can occur in software. More importantly, studying these reverse changes provides insights on how SENSA would compare to slicing when executions continue normally after the analyzed location.

Table 5.5 shows the results per subject and overall for the reverse changes. For these changes, the predictions of SENSA—especially SENSA-*Inc*—improve considerably. Dynamic slicing also improves but remains ineffective. Static slicing changes little as it does not depend on runtime data. The *Ideal* case is the same because execution differencing is symmetric. In all, SENSA outperforms static slicing with even stronger statistical significance for these changes.

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

5.3.3.2 RQ2: Effectiveness Distribution

Developers might not always be able to examine the entire rankings produced by SENSA. In such cases, they can focus only on a fraction of each ranking—normally those statements at the top. Thus, developers can prioritize their inspection efforts by analyzing as many highly-ranked impacts as possible within their budget. To understand the effects of such

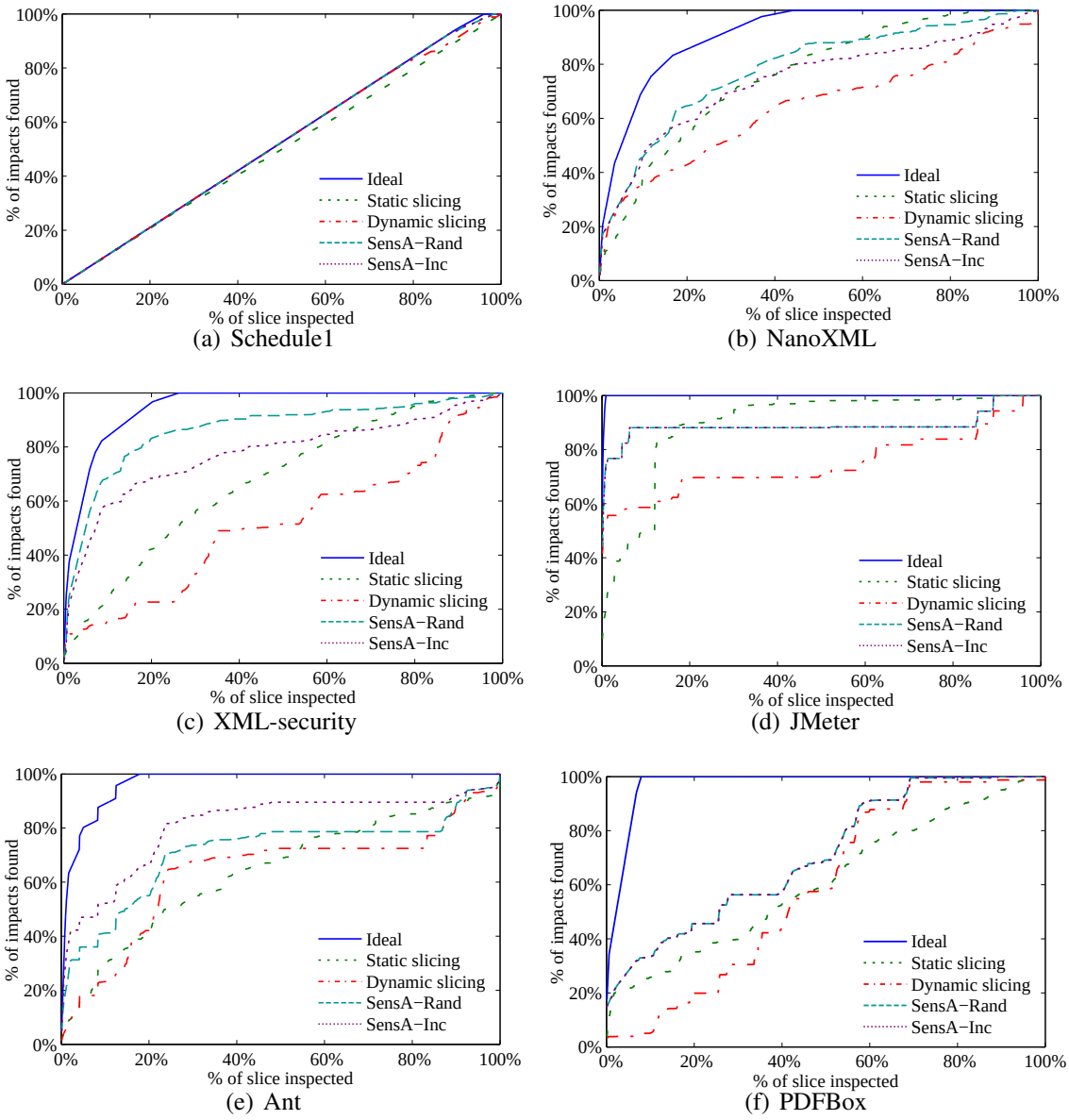

Figure 5.4: Impacted code found per inspection effort for the six Java programs we studied, using different impact ranking (prioritization) strategies: the ideal case, static slicing, dynamic slicing, SENSA-*Rand* (SENSA with the *Random* strategy), and SENSA-*Inc* (SENSA with the *Incremental* strategy). In each diagram, the x axis shows the percentages of statements in a ranking that need be inspected, in the forward direction, in order to reach the percentages of statements, shown on the y axis, for the associated actual-impact ranking. For each subject and ranking strategy, the curve displayed is the average of the curves for all changes for that subject, which are merged after being normalized to 1,000 data points each using linear interpolation. Because the sizes of all the rankings are the same per change as those of the corresponding static slices, which are usually much larger than the sizes of the corresponding actual-impact sets, the curves may not be smooth—even for the ideal cases.

prioritizations, we investigated the effectiveness of each portion of the SENSA and slicing rankings starting from the top.

Data. Figure 5.4 shows, for each subject and all fault fixes in that subject, the average cost-effectiveness curves of five examination orders: the ideal (best possible) ranking, SENSA with *Random* and *Incremental* strategies, and static and dynamic slicing. The horizontal axis indicates the fraction of the ranking examined from the top of the ranking that predicts impacts. The vertical axis indicates the percentage of actually-impacted statements found within that fraction of the ranking, on average for all changes in the subject. Note that the results in Table 5.4 are the average of the Y values for the corresponding rankings.

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

At the beginning, the curves for SENSA—especially SENSA-*Rand* for NanoXML and XML-security and SENSA-*Inc* for Ant—grow faster than those for slicing. For Schedule1, because of the high baseline (ideal) costs, all curves are very close. For that subject, the SENSA curves overlap with the ideal curve for about 90% of the ranking, whereas the dynamic-slicing curves breaks from them at about 75%. In other words, SENSA correctly predicts virtually all impacts for inspection budgets of 90% or less, whereas dynamic slicing has the same benefit for budgets up to 75%. Static slicing, however, always stays slightly below the ideal.

The results for the other five subjects, which are more representative of modern software, indicate important cost-effectiveness benefits of SENSA with respect to slicing up to about 60% of the ranking for NanoXML and 75% for PDFBox. For XML-security, SENSA-*Rand* outperforms slicing at 80% and SENSA-*Inc* does so by 65%. For Ant,

SENSA-*Inc* is more cost-effective than static slicing up to about 85% of the ranking and SENSA-*Rand* is more cost-effective up to 60%.

JMeter is, to some extent, an exception. Both variants of SENSA outperform static slicing in JMeter only up to 18% of the ranking. Beyond those points, static slicing becomes more competitive than SENSA up to 90% of the ranking. Yet, the superiority of SENSA for JMeter is substantial before this 18% point, which explains the overall advantage of SENSA reported in Table 5.4. More importantly, SENSA outperforms slicing considerably where we believe it matters the most, which is at the top of the ranking.

Reverse changes. For the reverse versions of the 42 changes, we omit the graphs as this is only a supplementary exercise. (The full details can be found in our SENSA home page [40].)For these changes, we found that the cost-effectiveness for the top portions of the SENSA rankings was even greater than for the original changes. This was expected given the greater overall superiority of SENSA for the reverse changes revealed in Table 5.5. For NanoXML, XML-security, Ant, and PDFBox, SENSA was always better than static slicing from the top of the ranking to the point where static slicing reaches 100% effectiveness. For JMeter, SENSA was more cost-effective than static slicing up to about 24% of the respective rankings and by a greater amount than for the original changes.

Conclusion. For these subjects, changes, and test suites, SENSA is not only more effective than slicing overall (i.e., at 100% of the ranking), as Tables 5.4 and 5.5 show, but also this superiority is concentrated at the top of the resulting rankings, which are the areas likely to be inspected first. Therefore, SENSA is especially more cost-effective than slicing when users are constrained by a budget which forces them to prioritize their inspection by some metric (e.g., influence).

5.3.3.3 RQ3: Computational Costs

Provided that executions exist that cover the statements to analyze, the other major factor that affects the practicality of SENSA is its computational cost.

Data. To study the cost factor, we collected the time it took SENSA on the 42 changes in our experimental environment (see Section 5.3.1) using the respective test suites. Table 5.6 first shows the average time in seconds it takes to run the *entire* test suite for each subject without instrumentation (column *Normal run*). The next three columns report the average time in seconds taken by each of the three stages of SENSA per subject.

Analysis. First, the pre-processing stage (column *Static analysis*) performs static slicing, which is needed by our experiment and is also necessary to instrument the program for SENSA, dynamic slicing, and DEA for actual impacts. As expected, this time grows with the size and complexity of the subject, where the three largest subjects (35-65K LOC) dominate. The average costs per subject were less than 16 minutes, which we consider acceptable for an unoptimized prototype tool.

The runtime stage (column *Instrumented run*) of SENSA repeatedly executes 20 times (the default) those test cases that cover the candidate change location. In contrast with the first stage, the cost of the runtime stage is proportional to the number of test cases that cover those locations. The number of test cases available for our subjects is inversely proportional to the subject size (Table 5.2), which explains the cost distribution seen in the table.

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

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

The results for the reverse changes (not shown), which have longer executions, are not very different. We found that, for these changes, SENSA is no more than 1.3 times costlier than for the original changes. Thus, longer executions do not seem to cause an explosion

TABLE 5.6

AVERAGE TIME COSTS OF SENSA IN SECONDS

Subject name	Normal run	Static analysis	Instrumented run	Influence ranking
Schedule1	186.0	6.1	4,756.8	1054.1
NanoXML	15.4	16.5	773.1	10.0
XML-security	65.2	179.3	343.9	20.8
JMeter	40.5	604.6	2,967.8	0.3
Ant	75.2	942.9	439.0	7.0
PDFBox	5.4	504.0	1,094.0	7.7

in the computational costs of SENSA. Also, our data (not reported in the table) shows that dynamic slicing was constantly much more expensive than SENSA by incurring total costs of several hours on average.

Conclusion. The observed costs are encouraging for the practicality of SENSA for three main reasons. First, we think that developers using our non-optimized prototype can in many cases accept to get the impact predictions of SENSA if they are provided within the time frames that all subjects but JMeter exhibit. (For Schedule1, a smaller subset of its large test suite can be used.) Second, the cost-benefit ratio of prioritizing the inspection efforts with SENSA can be even smaller for developers inspecting larger impact sets, and the overhead can be more acceptable when the impact sets are too large to be fully inspected. Third, our prototype implementation can be significantly optimized by parallelizing the large number of runs made by SENSA, to the extent that even JMeter might be analyzable at reasonably low costs by SENSA.

5.3.4 Threats to Validity

The main *internal* threat to the validity of our study is the potential presence of implementation errors in SENSA for sensitivity analysis and DUA-FORENSICS [163] underneath for execution differencing and slicing. Although SENSA is a research prototype developed for this work, we have tested and used it for more than one year already. Meanwhile, DUA-FORENSICS has been in development for many years [159] and has matured considerably. Another internal threat is the possibility of procedural errors in our use of SENSA, DUA-FORENSICS, and related scripts in our experimental process. To reduce this risk, we tested, manually checked, and debugged the results of each phase of this process.

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

Importantly, we *do not claim* that our results generalize to *all kinds* of changes in software. We only studied changes that represent *bug fixes* and small corrections as a first demonstration of SENSA. These changes are commonly available for experimentation. Also, SENSA is currently applicable to a *few* statements at a time but small changes are hard to find in other sources, such as code repositories. Larger changes in those repositories could be broken down into smaller pieces, or SENSA could be adapted for larger changes. We intend to explore this more broadly in future work on impact analysis. In all, this study on bug fixes highlights one of the many potential applications of SENSA. We present and study a second application in Section 5.4.

For *construct* validity, one threat can be our choice of ground truth—the *actual* impacts of changes—and the method to compute it. We used execution differencing (DEA) to find,

for a test suite, which statements behave differently in response to changes in another statement. DEA, like SENSA, works at the *statement* level, unlike repository-mining methods which are coarser and possibly noisier. Also, the actual impacts found via DEA are a subset of *all impacts* a change can have, so we chose subjects for which reasonable test suites exist. Moreover, we used the same test suites for SENSA so that its predictions apply to the same runtime profile.

Another construct threat is the *metric* we used to compare the effectiveness of SENSA and slicing. Intuitively, this metric correlates with the benefits that developers experience when using a predictive technique that places actual impacts higher in a ranking. However, the fidelity of this metric and the usefulness of SENSA for this task can only be, ultimately, studied on software developers in real production environments.

Finally, a *conclusion* threat is the appropriateness of our statistical analysis and our data points. To minimize this threat, we used a well-known test that makes *no assumptions* about the distribution of the data [14, 191]. Another issue could be the use of an equal number of changes per subject. For each of the larger SIR subjects, however, seven was the largest number of faults we could find and we deemed inadequate to study less than seven changes in smaller subjects. Moreover, SENSA outperformed slicing by greater margins for larger subjects. Thus, we expect that the statistical significance will increase if we drop changes from small subjects.

5.4 Case Studies: Failure Analysis

To further investigate the effectiveness of SENSA, we studied how well it predicts the *cause-effect chains* that link faulty code to a particular failure. Unlike impact analysis, whose goal is to identify *all impacted code* for maintenance, this task looks for the specific *subset of all effects* of a fault that make a program fail.

To that end, we conducted three short case studies of fault-effects isolation. After a (possibly) faulty statement is identified during *fault localization* [96, 171], a developer must

decide *how* to fix it. This decision requires understanding how the fault really affects the *failing point* (a failed assertion, a bad output, or crash location). However, not all effects of a fault are necessarily erroneous or are responsible for the failure. The interesting behavior is the chain of events that propagates this fault to the failing point. Unlike Zeller's approach that *finds* cause-effect chains based on differencing program states between passing and failing runs [201], SENSA inputs a change (location) and *prioritizes*, in addition to finding, the effects of that change to assist their inspection and understanding.

We performed our case studies on the first fault provided at SIR [57] for each of the three subjects: NanoXML, XML-security, and JMeter. For each fault, we identified the *first* failing point (one statement in each case) where the fault is manifested. Given the faulty statement, we *manually* identified the sequence of all statements that propagate the fault to the failing point. We discarded affected statements that did not participate in this propagation to the failing point. All statements are Java bytecode instructions in a readable representation—the Jimple intermediate representation (IR) of Soot [185].

Given a chain of events—the set of propagating statements—and the bug fix provided with the subject, we computed how close to the top the chain is in the rankings computed by SENSA and by forward static and dynamic slicing from the fault. While forward slicing is not typically used for debugging, these small case studies are intended for investigating the effectiveness of SENSA against both slicing techniques and in assisting with fault understanding, which may help with debugging, rather than performing debugging or fault-localization tasks directly. Specifically, we calculated the average rank of the statements in this chain in each ranking to estimate how well those rankings highlight the effects of the fault that actually cause the failure.

Although three case studies are insufficient for statistically-significant conclusions (the manual effort for these studies is large), these cases shed light on the workings of the three techniques that we studied for this application. Next, we present our results and analysis for these brief case studies.

5.4.1 NanoXML

Fault v1s1 in NanoXML is located in a condition for a *while* loop that processes the characters of the DTD (Document Type Definition) of the input XML document. The execution of this fault by some test cases triggers a failure by failing to recognize the end of the DTD, which then causes an unhandled exception to be thrown when parsing the next section of the document by using the method that parses DTD. The bug and its propagation mechanism are not easy to understand because the exception is not thrown immediately after the execution of the faulty statement. After exhaustive inspection, we manually identified the 21 Java Jimple [185] statements that constitute the entire cause-effect chain from fault to failure.

All 21 statements that cause the failure—which will help design the bug fix—are placed by SENSA-*Rand* in the top 18.54% of the ranking and by SENSA-*Inc* in the top 23.14%, whereas static slicing places them in the top 33.53% and dynamic slicing puts them in the top 26.28%. Thus, both SENSA strategies isolate the entire cause-effects chain better than both forms of slicing.

The average inspection effort for finding these statements, computed with the method of Section 5.3.2, is 7.08% for SENSA-*Rand*, 12.49% for SENSA-*Inc*, 8.86% for static slicing, and 7.44% for dynamic slicing. Therefore, for this particular fault, dynamic slicing was, comparatively, much more effective than for most impact-analysis cases studied in Section 5.3. Nevertheless, SENSA-*Rand* was still slightly better than dynamic slicing, which approaches the favorable trend observed for SENSA-*Rand* on faulty programs.

5.4.2 XML-security

Fault v1s2 in XML-security is revealed by only one of the 92 test cases available. This test fails because of an assertion failure due to an unexpected value. Manually tracing the execution backwards from that assertion to the fault location reveals that the fault caused an incorrect signature on the input file via a complex combination of control and data

flow. The complete sequence of events for the failure trace contains more than 200 Jimple statements. Many of those statements, however, are in helper functions that, for practical purposes, work as atomic operations. Therefore, we skipped those functions to identify a more manageable and focused cause-effect chain that can be more easily understood.

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

The entire forward static slice consists of 18,926 Jimple statements. Thus, users would find the entire chain within the first 1,255 statements in the SENSA ranking. Using static and dynamic slicing, instead, users would need 6,274 and 3,388 statements, respectively. Thus, for this fault, a developer can find the fault effects that make the assertion fail much faster than using slicing.

5.4.3 JMeter

For this case study, we chose again the first fault provided with JMeter—v2s1—and we picked, from among all 79 test cases, the one test that makes the program fail for v2s1. The failing point is an assertion by the end of that test. Despite the much larger size of both the subject and the forward slice from this fault, the fault-propagation sequence consists of only 4 statements.

Static slicing ranks two of those statements in the top 1% and the other two statements farther away. SENSA, in contrast, places the entire sequence on its top 1%, making it easier to distinguish the failure-related statements for this fault from the other, non-failing effects of it. The inspection of the entire failure sequence using static slicing would require a developer to go through 2.6% of the forward slice, or 848 statements. For SENSA, this cost would be only 0.1%, or 32 statements.

As we observed for dynamic slicing in Section 5.3, considering the effects of fixing this fault, dynamic slicing would cost much more than SENSA and static slicing to identify the fault-propagation sequence. To find all four statements in the sequence, users would have to traverse 12.6% of the slice, which corresponds to 4,094 statements. Once again, this case gives SENSA an advantage over static slicing and especially over dynamic slicing in isolating the failure sequence.

5.5 Related Work

In preliminary work [164], we outlined an early version of SENSA and we showed initial, promising results for it when compared with the predictions from breadth-first traversals of static slices [171, 192]. In this chapter, we expanded our presentation of SENSA, its process, algorithm, and modification strategies. Moreover, we extended our experiments from two to six Java subjects, included dynamic slicing in our comparisons, and added three case studies of cause-effects isolation using SENSA.

A few other techniques discriminate among statements within slices. Two of them [71, 205] work on dynamic backward slices to estimate influences on outputs, but do not consider impact influences on the entire program. These techniques could be compared with SENSA if a backward variant of SENSA is developed in the future. Also for backward analysis, thin slicing [171] distinguishes statements in slices by pruning control dependencies and pointer-based data dependencies incrementally as requested by the user. Our technique, instead, can be used to automatically estimate the influence of statements in a static slice in a safe way, *without dropping* any of them, to help users prioritize their inspections.

In [118, 119], Masri and Podgurski followed an information-theoretic approach to measure the strength of dynamic information flow between variables through dynamic data and control dependencies. Their concept of dynamic flow strength is similar to the impact strength we used in SENSA for semantic-dependency quantification. However, although their approach can also be employed for quantifying dynamic dependency, we designed

a more comprehensive approach to measuring dependency strengths by using fuzzed program executions and execution differencing, in contrast to using existing executions only as in [118, 119]. On the one hand, fuzzing helps SENSA alleviate the drawbacks of using a limited set of executions, which may not represent well the usage pattern of the analyzed parts of the program. On the other hand, we could enhance in future work the approach of Masri and Podgurski via fuzzing.

Program slicing was introduced as a backward analysis for program comprehension and debugging [192]. Static forward slicing [83] was then proposed for identifying the statements affected by other statements, which can be used for change-impact analysis [15]. Unfortunately, static slices are often too big to be useful. Our work alleviates this problem by recognizing that not all statements are equally relevant in a slice and that a dynamic analysis can estimate their relevance to improve the effectiveness of the forward slice. Other forms of slicing have been proposed, such as dynamic slicing [103], union slicing [23, 78], relevant slicing [6, 77], deletion-based slicing approaches [47, 56, 199], and the already mentioned thin slicing [171], all of which produce smaller backward slices but can miss important statements for many applications. Our technique, in contrast, is designed for forward analysis and does not trim statements from slices but scores them instead.

Dynamic impact analysis techniques [12, 108, 151], which collect execution information to assess the impact of changes, have also been investigated. These techniques, however, work at a coarse granularity level (e.g., methods) and their results are subject strictly to the available executions. Our technique, in contrast, works at the statement level and analyzes the available executions and, in addition, multiple variants of those executions to predict the impacts of changes. Also, our technique is *predictive*, unlike others that are only descriptive [151, 162] (using knowledge of the changes made).

Mutation testing is a specific form of sensitivity analysis that simulates common programming errors across the entire program [5, 55, 93, 115]. Its purpose is to assess the ability of a test suite to detect errors by producing different outputs. This approach is re-

lated to testability-analysis approaches, such as PIE [188], which determine the proneness of code to propagate any errors to the output so they can be detected. Similar to these approaches, SENSA modifies program points to affect executions but it focuses on points of interest to the user (e.g., candidate change locations) and analyzes not only the influences on outputs but also the influences on all statements.

Many fault-localization approaches [2, 18, 53, 96, 171], although not directly related to SENSA, share a common aspect with our work: they assess their effectiveness in terms of the inspection effort for finding certain targets in the program. And the inspection effort is also often measured as the percentage of the program or a slice that must be inspected to reach those targets, similar to what we do in the evaluation of SENSA. Yet, for fault localization, those targets are faults, whereas in our work they are the influences of a statement.

Fuzzing is a well-known technique that has been widely used in software security testing and quality assurance [178], especially for detecting software vulnerability and reliability issues [21]. However, to the best of our knowledge, fuzzing has not been exploited for dependency quantification or impact analysis before. Yet, while SENSA shares the spirit of fuzzing, it does not necessarily use invalid or unexpected values.

Finally, symbolic execution has been utilized in characterizing the impacts of changes that have been applied to a program on its behaviour [157], where the impacts are expressed via path conditions that reflect differences in program semantics cause by those changes . While working in the prediction setting without knowledge about concrete changes, SENSA could benefit too from symbolic execution or similar other techniques that can explore the space of program states more efficiently than the random value generation we current adopt. For example, considering only the values that exercise distinct paths (e.g., via different branches) by solving related path conditions would allow SENSA to examine either the same number of unique runtime modifications faster (hence higher efficiency) or more modifications (hence higher effectiveness) with the same time budget.

CHAPTER 6

UNIFYING HYBRID IMPACT PREDICTION FOR IMPROVED AND VARIABLE COST-EFFECTIVENESS

Dynamic impact analysis can greatly assist developers with managing software changes by focusing their attention on the effects of potential changes relative to concrete program executions. While dependency-based dynamic impact analysis (DDIA) provides finer-grained results than traceability-based approaches, traditional DDIA techniques often produce imprecise results, incurring excessive costs thus hindering their adoption in many practical situations. This chapter presents the design and evaluation of a DDIA framework and its three new instances that offer not only much more precise impact sets but also flexible cost-effectiveness options to meet diverse application needs such as different budgets and levels of detail of results. By exploiting both static dependencies and various dynamic information including method-execution traces, statement coverage, and dynamic points-to sets, the new techniques achieve that goal at reasonable costs according to our experiment results. Our study also suggests that statement coverage has generally stronger effects on the precision and cost-effectiveness of DDIA than dynamic points-to data.

6.1 Problem Statement and Motivation

Analyzing the impacts of constant changes is crucial for successful software evolution [148], even more so as modern software is increasingly complex. In comparison to traceability-based approaches, dependency-based analysis produces finer-grained *impact sets* (potentially affected entities), which are generally more useful for understanding

code-level changes [180], while for such tasks traceability-based analyses are usually insufficient [156]. Furthermore, among the dependency-based techniques, static approaches compute impact sets for all possible executions that are often highly imprecise due to their conservative nature [83, 112]. In contrast, dynamic impact analysis produces more focused results by using runtime information [31, 111, 135] that represents specific subsets of all executions. For developers looking for concrete program behavior or understanding the effects of potential code changes relative to those specific subsets, impact sets given by dynamic impact analysis are preferable. Therefore, this chapter targets dependency-based dynamic impact analysis (DDIA).

For clients of impact analysis such as regression testing [134], program comprehension [34], and change understanding [180], imprecise impact sets are clearly undesirable because developers using such results could waste time and other resources on examining entities that are falsely reported as impacted. Moreover, the imprecision can adversely affect the effectiveness of application tasks. For example, when used in program understanding, false-positive impacts may suggest spurious complications that make the program harder to understand than it should be. For another example, applying changes based on imprecise impact sets could lead to severe consequences such as system failure.

Unfortunately, developing a DDIA technique of practical cost and effectiveness, even on the method level, remains a challenging problem. For instance, COVERAGEIMPACT [134] is highly efficient but also very imprecise [135]. In contrast, PATHIMPACT [108] is more precise yet less efficient than COVERAGEIMPACT [135]. Since these two techniques, much research focused on performance optimizations (e.g.,[12]) while only a few invested on improving the precision (of PATHIMPACT), yet their improvements were marginal only and at much greater costs (e.g., [31]). To the best of our knowledge, the most cost-effective method-level DDIA prior to our work remains to be PATHIMPACT combined with its optimization *execute-after sequences* (EAS) [12], which we call PI/EAS. The slightly more precise technique is INFLUENCEDYNAMIC [31] but it is much less efficient than PI/EAS.

Nonetheless, existing techniques can still suffer from great imprecision. Our recent study [41] revealed that the mean precision of PI/EAS was about 50% only. Although some other techniques may provide better precision (e.g., [151]), they are *descriptive* impact analyses [15] applicable only *after* actual changes are made. However, for many software evolution tasks, impact analysis needs be performed *before* making those actual changes [148]. In software industry, it has been reported that delayed impact analysis is among the highest-priority issues that both organization and developers encountered [156]. Therefore, we focus on *predictive* impact analysis [15], which predicts *potential* impacts of candidate changes without any assumptions about actual changes to be made.

This chapter presents three cost-effective DDIA techniques within a unified framework that are much more precise than PI/EAS, called *Trace Only* (*TO*), *Trace plus statement Coverage* (*TC*), and *Full Combination* (*FC*). While all use the dependency graph of the target program [35], they differ in the amount of dynamic data used for the analysis. By leveraging different combinations of static and dynamic program information, including statement coverage and dynamic alias data, these techniques also offer multiple levels of cost-effectiveness tradeoffs of DDIA, and thus provide developers with flexible technical options, which are in growing demand in practice nowadays [52, 156, 180].

We have applied this framework to seven Java subjects of up to 220K lines of code in size. Our results show that these new analyses are generally all cost-effective, with continuous and significant precision gains over the baseline approach PI/EAS at reasonable costs. In addition, among the three instances, *TC* attained the best cost-effectiveness in most cases as regards the overall analysis time overhead, which implies that statement coverage can play a strong role in general DDIA design. The study also suggests that more precise points-to data may not translate to significant gains in the precision or cost-effectiveness of DDIA, akin to previous such findings in the context of program slicing [122]. In comparison, statement coverage mostly leads to greater precision improvements at costs that are better paid off than do dynamic points-to data.

Beyond these three instances, more others can be instantiated from our framework as well. To differentiate those from the three we study here, and also for brevity, we hereafter refer to as DIAPRO any of *TO*, *TC*, and *FC*, or them together. Note that DIAPRO only prunes dependencies (and consequent impacts) that it ascertains are not exercised by the employed dynamic data, with *TO*, *TC*, and *FC* incrementally doing so in a conservative manner. Thus, we regard DIAPRO as safe with respect to the execution set it utilizes. Under this assumption about recall, we express the precision as the impact-set size ratio of DIAPRO to the baseline, which is appropriate at least for evaluating relative precision gains as we focus on in this work as previously adopted as well [12, 135].

6.2 Approach

To prune false-positive impacts given by PI/EAS or INFLUENCEDYNAMIC, we need more precise information than both the *execute-after* sequences of PI/EAS and the incomplete dependency analysis of INFLUENCEDYNAMIC. To that end, we propose to utilize the (whole-program) static dependencies, together with the *execute-after* relation between methods, and, optionally, statement coverage and dynamic alias information, to develop DDIA techniques that offer better precision with various cost-benefit tradeoffs.

6.2.1 Overview

We first introduce the design of a unified framework for DDIA, from which both the two existing and three proposed DDIA approaches can be instantiated. This framework, as shown in Figure 6.1, helps illuminate the rationale underneath various levels of DDIA precision and cost, as well as the relationship among them.

The framework works in three phases: static analysis, runtime, and post-processing, as shown at the top of the diagram which sketches the process flow of a typical DDIA. The inputs for the entire process are a program P, an input set (e.g., test suite) T of P, and a set M of queries (i.e., the set of methods for which impact sets are to be queried). In

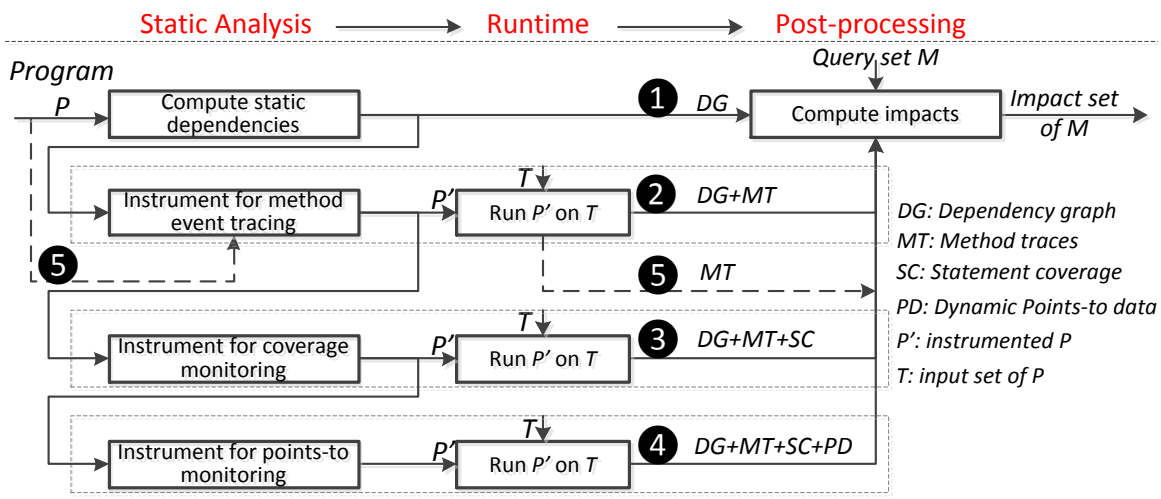

Figure 6.1: A unified framework for DDIA that incorporates static dependency information and various forms of dynamic data to achieve multiple-level cost-effectiveness: The marked paths (circled 1 to 5) illustrate five of its instances we studied.

most cases, a *mandatory* step is to create the dependency graph of P used by all the DDIA techniques we proposed. There are five different workflow paths (numbered with 1 through 5) that correspond to five alternative approaches to computing the impact set of M. The output of the process (framework) is consistently the resulting impact set of M.

The first path directly leads to the impact-computation step without using any execution information, constituting a static impact analysis. Although we do not include it in the study here, it still provides a viable option to developers that we plan to explore in the future. Path 5 effectively leads to PI/EAS, which we use as the baseline approach for aforementioned reasons. The other three paths correspond to the three proposed DDIA techniques: path 2 to *TO* using method traces only, path 3 to *TC* with statement coverage added to *TO*, and path 4 to *FC* further including dynamic points-to data on top of *TC*.

When building the dependency graph, the framework first runs a profiler which executes T on P to make decisions on the inclusion of exceptional control dependencies. Since such decisions can greatly affect the graph size hence overall performance of the DDIA, this step is an integral part of the framework. The details about the exception profiler and

how the dependency graph is constructed according to the profiling results are described in Section 4.2. Next, runtime monitors for collecting constituent dynamic information (method trace, statement coverage, and dynamic points-to data) are inserted through bytecode instrumentation, configured based on the needs of an instance. The outputs of this phase are the dependency graph and instrumentation version P' of P.

During the runtime, the framework executes T on P' to produce the dynamic information configured during static analysis. For example, all the three forms of dynamic data will be produced in *FC*, while in *TO* only the method traces will be generated and collected. To reduce the storage at small time overheads well paid off by overall cost savings, dynamically generated data are all compressed on the fly. For programs of relatively long-lasting runs, such extra data processing can be particularly instrumental and often necessary.

The last phase, impact computation is essentially the post-processing of all the static and dynamic information collected during the previous phases. The general idea is to prune executed methods that have no dependencies on the query exercised by the dynamic data utilized. Intuitively, with more dynamic data employed for such pruning, more precise impact set will be obtained and, at the same time, larger costs of the entire analysis will be incurred. When multiple methods are queried, the process computes the impact set for one method at a time and then takes the union of all. Note that for any number of queries with respect to the same execution set, the previous phases are performed once only, with their outputs shared by the impact computation for all queries.

6.2.2 Trace Only (*TO*)

The method execution trace is the single type of dynamic data used in *TO* (path 2 in Figure 6.1). Corresponding to DIVER [35], *TO* finds methods from the trace directly or transitively dependent on the query using the dependency graph. Details about this technique are presented in [35], and here we recap only the key ideas as follows.

TO consists of two major steps. The first step conceptually corresponds to the entire PI/EAS analysis: using the method trace, filtering methods never executed after the query thus cannot be impacted in that trace. However, an impact set obtained as such is only a rough approximation of methods dynamically dependent on the query. The execution order in the trace implicitly exercised runtime *control flows* but not *control dependencies* of the program, with data dependencies entirely ignored. Thus, *TO* takes the second step to further prune methods that are neither data nor control dependent on the query using the static dependencies, which is crucial for its achieving a better precision than PI/EAS.

When tracking *impact propagation* along the method trace, *TO* differentiates three types of data dependencies, *parameter*, *return* and *heap*, so as to apply different propagation rules for precise false-impacts pruning. To be exercised, the first two types need an *immediate posterior* relation, whereas the last type just a *posterior* one, between the events of a method already affected and the method being considered. Further, interprocedural dependencies through which impacts may propagate into and out of a method are referred to as *incomong* and *outgoing dependencies* of that method, respectively.

6.2.3 Trace plus Coverage (*TC*)

On top of the hybrid approach *TO*, the *TC* technique goes further to add statement coverage to the DDIA (path 3 in Figure 6.1). While the method execution trace informs the analysis with method coverage and execution order, the statement coverage offers a form of finer-grained execution data to enable a finer-grained pruning for the DDIA. With *TO*, it is implicitly assumed that statements bridging incoming to outgoing dependencies within a method are always executed. In essence, impact propagation through inside each method is based on *static*, rather than *dynamic*, dependencies.

However, this conservative assumption can lead to falsely identified impact propagation hence false impacts. *TC* thus exploits statement coverage to prune those false impacts. There are two alternative ways of such pruning: (1) *pre-pruning*, which prunes edges on

which at least one statement is not covered, from the dependency graph before it is applied to post-processing, and (2) *post-pruning*, which incorporates the statement coverage data into the impact computation algorithm [35] along with the dependency graph, without pre-processing the graph itself. We expect that both strategies contribute equivalently to the effectiveness (precision), but differently to the overhead, of DDIA.

6.2.4 Full Combination (*FC*)

FC (path 4 in Figure 6.1) combines all the three types of execution data performing the finest-grained analysis among the five DDIA instances we studied. By employing statement coverage on top of method execution traces, *TC* supposedly addresses false positives due to spurious transitive dependencies, but only does that partially. Another source of imprecision can be resulted from spurious data dependencies due to imprecise static pointer analysis, which is a well-known problem as true points-to sets are often conservatively approximated only during static analysis. Fortunately, however, precise points-to sets can be obtained at runtime by monitoring the actual memory addresses each pointer refers to.

FC thus attempts to further improve the precision of DDIA by exploiting dynamic points-to data. This technique collects the full set of allocation sites that each pointer points to during program execution, by monitoring memory addresses of pointer targets. Then, during the post-processing phase, spurious aliasing-induced data dependencies can be identified by checking the intersection of relevant points-to sets. Given a data dependency $s1 \rightarrow s2$ in the dependency graph, for example, *FC* will regard it as a spurious dynamic dependency if there does not exist a variable $v1$ defined on $s1$ and a variable $v2$ used on $s2$ such that the points-to set of $v1$ and that of $v2$ have a non-empty intersection.

In collaboration with the method trace, which contains all method execution instances, dynamic points-to sets can be collected and employed at two granularity levels: method level and method-instance level. The method level strategy maintains a single points-to set of each heap variable (exercised by the runtime input) in a method m that contains

allocation sites pointed to by that variable for all instances of m. In contrast, the method-instance level data includes such points-to sets for each instance of m separately.

Once identified, spurious dependencies are used for pruning false impacts from the method execution trace. The method level data can be applied either before or during impact computation, similar to the pre-pruning and post-pruning strategies described for *TC* above. However, only post-pruning can be adopted with the method-instance level data as the dependency graph is static and does not incorporate information about method execution instances.

In comparison, the method-level data tends to be conservative thus only *approximates* precise points-to data that the method-instance level captures, yet with potentially lower time and space overheads than the other. In this chapter, we study two variants of *FC*, FC_{ml} and FC_{mil}, which use the method and method-instance level points-to data, respectively.

6.2.5 Other Instantiations

Beyond the above three new analyses, more others can be instantiated from this DDIA framework as well. For example, as marked by path 5, PATHIMPACT and EAS are both the instance that does not use the static dependency model (dependency graph) but purely relies on the method-level execution trace to predict impacts of the input queries. INFLUENCEDYNAMIC can be instantiated from this framework too, which would be a variant of *TO* that ignores intraprocedural data dependencies and all control dependencies (although it does utilize control flows), and uses a different impact computation algorithm based on the method trace and a partial dependency graph called *influence graph* [31].

The framework could also be instantiated such that a DDIA technique would use the dynamic points-to sets as the only type of dynamic data, or even without using the dependency graph (i.e., adding dynamic alias analysis to PI/EAS). While even more instances potentially exist, we leave them for future exploration for space reasons.

```
1  public class A {
2      static int g; public int d;
3      String M1(int f, int z) {
4          int x = f + z, y = 2, h = 1;
5          if (x > y)
6              M2(x, y);
7          int r = new B().M3(h, g);
8          String s = "M3val: " + r;
9          return s;}
10     void M2(int m, int n) {
11         A a2 = _trans(this);
12         C.M5(a2);
13         int w = m − d;
14         if (w < 0)
15             g = m / w;}}
16 public class B {
17     static short t;
18     int M3(int a, int b) {
19         int j = 0;
20         t = −4;
21         if ( a < b )
22             j = b − a;
23         return j;}
24     static double M4() {
25         int x = A.g, i = 5;
26         try {
27             i = x / (i + t);
28             new A().M1(i, t);
29         } catch(Exception e) { }
30         return x;}}
31 public class C {
32     static boolean M5(A q) {
33         long y = q.d;
34         boolean b = B.t > y;
35         q.d = −2;
36         return b;}
37     public static void
38       M0(String[] args) {
39         int a = 0, b = 3;
40         A o = new A();
41         String s = o.M1(a, b);
42         double d = B.M4();
43         String u = s + d;
44         System.out.print(u);}
45     }
```

PATHIMPACT: M0 M1 M2 M5 r r M3 r r M4 r r x

DIAPRO: $M0_e$ $M1_e$ $M2_e$ $M5_e$ $M2_i$ $M1_i$ $M3_e$ $M1_i$ $M0_i$ $M4_e$ $M4_i$ $M0_i$ x

Figure 6.2: The example program $E3$ (top) and its execution traces (bottom) used by PATHIMPACT and DIAPRO for illustrating the DDIA framework.

6.2.6 Illustration

We illustrate DIAPRO using the example program and traces of Figure 6.2. In the example trace used by DIAPRO (bottom right of the figure), *e* denotes method-entry events while *i* denotes method-returned-into events. Suppose the query set is {`M5`}, and function `_trans` is a library call that clones the input object and returns the clone after transforming it. Then, the ground-truth impact set is {`M5`} in this example case.

PI/EAS again finds the entire program (all methods) of E impacted as every method executes after the first entry event of `M5`. INFLUENCEDYNAMIC does not prune any method from that imprecise impact set because the influence [31] of `M1` propagates to each of

other methods according to its influence graph for this case. The impact set of DIAPRO starts with $\{$M5$\}$ upon the occurrence of event $M5_e$, growing as more methods are found dependent on M5 during the traversal of the method trace with reference to, if available, statement coverage and/or dynamic points-to sets.

Next, control returns into M2. With *TO*, which assumes that line 15 is covered and object q at line 35 points to the same allocation site (of line 40) as the base object of field d at line 13, the (*heap*) DD of M2 on M5 (via instance field d) is exercised, so M2 is regarded as impacted. Later in the trace, when M4 is entered, another *heap* DD, of M4 on M2 (via class field g), is exercised and M4 is thus added to the impact set: The impact reached to (line 13 of) M2 from (line 35 of) M5 continues to propagate (via line 15) to (line 25 of) M4. The first and second *heap* DD here are the only outgoing dependency of M5 and M2, respectively. As a result, the impact set computed by *TO* is $\{$M2, M4, M5$\}$.

However, with *TC*, which checks statement coverage and finds that statement 15 is not covered (w is 5 at line 14), the second *heap* DD above is not exercised. As a result, *TC* reports a more precise impact set $\{$M2, M5$\}$. Finally, on top of *TC*, *FC* further checks dynamic points-to data and thus finds that the two base objects, of field d at line 35 and that at line 13, are not aliased at runtime, so the first *heap* DD is skipped too. Consequently, *FC* reports M5 as the only impacted method, thus gives the most precise (also accurate) impact set. (Coincidentally, in this example, *FC* would find the accurate impact set even without using the statement coverage, because by checking dynamic aliasing data the impact would not even propagate into M2 hence M4 would also be pruned.)

6.3 Evaluation

This section presents our empirical evaluation of the DDIA framework we proposed, by studying the DIAPRO techniques as its representative instances: *TO*, *TC*, and *FC* (both variants). Our main goal with this empirical study was twofold. First, we wanted to assess the precision of the new techniques and its practicality in terms of time and space costs

against the most cost-effective prior DDIA, PI/EAS, as the baseline approach.Second, we intended to analyze the effects of different types of dynamic data on the cost-effectiveness of DDIA in order to inform future design of better DDIA techniques.

6.3.1 Experiment Setup

The DDIA framework was implemented in Java based on the Soot byte-code analysis framework [185] and our dependency-analysis toolkit DUA-FORENSICS [163]. For the dependency graph construction including the exceptional control dependency analysis, and PI/EAS facilities including the method execution trace instrumentation and monitoring, we reused relevant modules of DIVER [35]. In the implementation of DIAPRO and PI/-EAS, we included the exception-handling enhancement [41] to ensure the safety of analysis results relative to the execution data utilized.

For statement coverage monitoring capable of exceptional control flows, we computed whole-program reverse dominance frontiers and branch-induced control dependencies [7], whereby statement coverage was monitored indirectly through branch coverage monitoring. For dynamic alias analysis, we piggybacked the collection of object memory addresses on the method-event monitors. Monitoring method and method-instance level points-to sets shares the same instrumentation with differences in runtime monitors.

For impact computation, *TO* implements the same as DIVER [35]; *TC*, for which we adopted the post-pruning strategy for a uniform implementation for all DIAPRO techniques (recall that FC_{mil} can not adopt the pre-pruning algorithm), skips any edges of the dependency graph having at least one node whose underlying statement is not covered during the method trace traversal; *FC*, for both variants, prunes the method trace similarly to *TC* but checks *heap* edges only.

We chose seven Java programs of various types and sizes, as summarized in Table 6.1, for our experimental study. The size of each subject is measured as the number of non-comment non-blank lines of code (*#LOC*) in Java. For each subject, the last two columns

TABLE 6.1

EXPERIMENTAL SUBJECTS FOR DIAPRO EVALUATION

Subject	**#LOC**	**#Tests**	**#Methods**	**#Queries**
Schedule1	290	2,650	24	20
NanoXML	3,521	214	282	172
Ant	18,830	112	1,863	607
XML-security	22,361	92	1,928	632
JMeter	35,547	79	3,054	732
JABA	37,919	70	3,332	1,129
ArgoUML	102,400	211	8,856	1,098

give the number of total methods (*#Methods*) and that of the subset (*#Queries*) covered by at least one of the tests (*#Tests*) used by our dynamic analyses. The same set of subjects is previously used for evaluating DIVER (see Chapter 4) and a more detailed description on these subject programs can be found in Section 3.4.1.

6.3.2 Experimental Methodology

For our experiments, we applied *TO*, *TC*, FC_{ml}, FC_{mil}, and PI/EAS separately to each subject on a Linux workstation of an Intel i5-2400 3.10GHz processor and 8GB DDR2 RAM. To obtain the method traces and other dynamic data, we used the entire test suite provided with each subject except for JABA, for which we use the shortest-running 70 tests only on which the current implementation did not run out of memory. Per technique and subject, we computed the impact sets for all methods, each as an individual query. Expectedly, some methods were not executed by any test and thus had empty impact sets. We excluded them from our results. To assess the cost-effectiveness of DIAPRO, we computed three main metrics against the baseline technique PI/EAS as follows.

The first measure is precision. Without actual changes available hence the lack of ground-truth impact sets, this precision is measured relatively through the impact-set size ratios of DIAPRO to PI/EAS. We report the mean and five quartiles of the precision for all queries per subject. It is crucial to note that DIAPRO attempts to improve the precision yet without penalizing the recall of PI/EAS, because the additional dependency analyses it employs for pruning false impacts of PI/EAS are still all conservative in nature.

The second metric is the time and space costs. For each technique and subject, we collected the time cost (per-thread CPU time) separately per DDIA phase; for static-analysis and runtime phases that are needed once for all queries, we did one measurement of time; for post-processing costs, we calculated the mean and standard deviation of query time for all queries; for space costs, we gauged the sizes of disk data used.

The third metric is the average cost-effectiveness of DIAPRO. To see how many gains in effectiveness (precision) different forms of dynamic data contribute with respect to the extra costs incurred by using them, we compared the ratio of precision gain to cost increase among the three DIAPRO instances relative to the baseline, for the one-time cost of the first two DDIA phases and per-query post-processing cost separately. This data analysis helps reveal the effects of different dynamic data on the cost-effectiveness of DDIA.

For a further analysis of those effects, we intended to assess the statistical significance of the differences in precision among the DIAPRO instances. To that end, we performed a set of paired Wilcoxon signed-rank tests [191] where the two groups were the impact-set sizes given by each pair of techniques being contrasted. We adopted this non-parametric test to lift the assumption about the normality of underlying data distribution. In addition to p-values computed separately for each subject at the 0.95 confidence level, we also report combined p-values for each pair of techniques using the Fisher method [124].

6.3.3 Results and Analysis

6.3.3.1 Precision

The main precision results are shown in Figure 6.3, where the per-subject data points are summarized by separate plots. In each plot, the impact-set size ratios (y axis) to the baseline are grouped by the DIAPRO instances we studied, with each group characterized by a single boxplot that consists of the maximum (upper whisker), 75% quartile (top of middle box), 25% quartile (bottom of middle box) and the minimum (lower whisker). The central dot within each middle box indicates the median, surrounded by a pair of triangular marks that represent the comparison interval of that median. These comparison intervals, together within each plot, give a quick indicator of the statistical significance of the differences in medians among the four groups in that plot: For any two groups, their medians are significantly different at the 5% significance level if their intervals do not overlap.

Overall, *TO* greatly reduced the sizes of baseline impact sets, with *TC*, FC_{ml}, and FC_{mil} continuously improving the relative precision, albeit slightly. Also, except for Schedule1, there always existed cases in which DIAPRO cut some off the entire impact set reported by the baseline analysis—DIAPRO reports empty impact sets for queries with empty method body—leading to the minimal size ratios of 0%. And except for Ant and XML-security in addition to Schedule1, DIAPRO was always able to prune some false impacts—The maximal ratios are constantly below 100% for other subjects.

The largest gains in precision were seen with Ant, XML-security, and JMeter, for which DIAPRO drastically prunes the baseline impacts by over 80%. ArgoUML also got a substantial impact-set reduction from DIAPRO, which reported no more than 35% of the methods produced by the baseline. With Schedule1 and JABA, DIAPRO improved relatively less, probably because these programs have dense dependencies among their entities as a result of tight inter-function couplings. However, in general neither statement coverage nor dynamic points-to data contributed to the precision gain of DIAPRO over the baseline so much as the static dependency information did. On the other hand, the comparison intervals

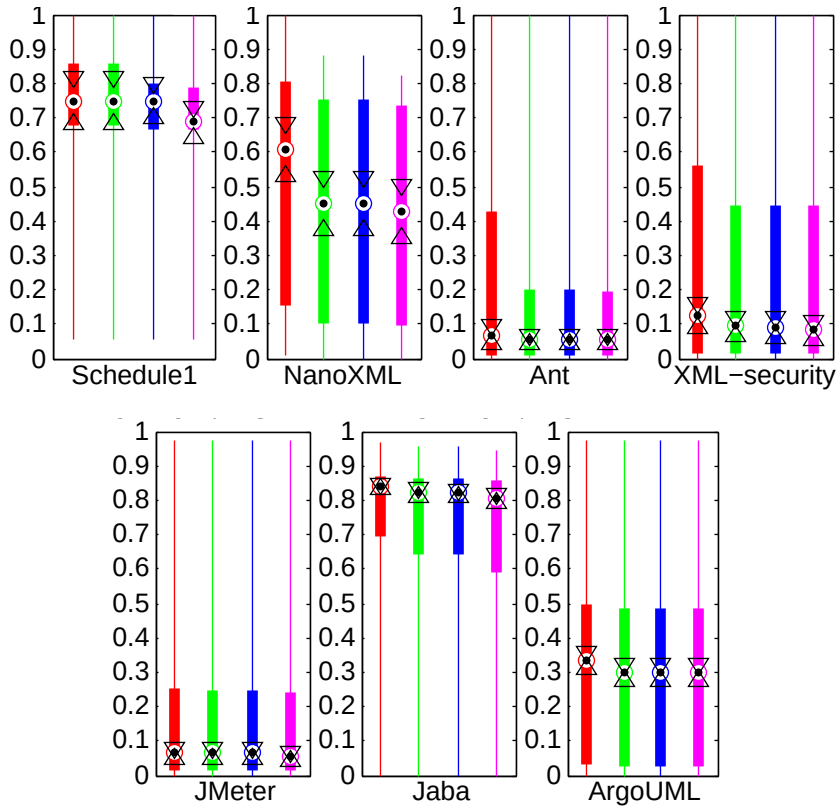

Figure 6.3: Precision of the DIAPRO techniques expressed as impact-set size ratios against PI/EAS (the lower the ratio, the better).

show no significant differences in the medians of precision among all DIAPRO instances, except for those between *TO* and the other three in the only case of NanoXML.

The complementary results on precision shown in the left five columns of Table 6.2 further demonstrated the effects of the additional dynamic data beyond method traces, where the means of impact-set size ratios reveal that both *TC* and *FC* gained only little in precision on top of *TO*. Between the two forms of dynamic information, however, statement coverage appears to be more effective than dynamic points-to data. Consistent with the boxplots, the overall mean ratios, as listed in the bottom row of the table, show that relative advantage of statement coverage. Note that, as in other tables too, these *overall* numbers are not simple averages over the per-subject means listed in the table but weighted averages by the number of queries per subject as shown in the last column of Table 6.1.

TABLE 6.2

AVERAGE PRECISION AND QUERYING COST OF THE DIAPRO TECHNIQUES VERSUS PI/EAS

Subject	Mean I.S. size ratio to PI/EAS (%)					Query time in seconds: mean (stdev)			
	TO	*TC*	FC_{ml}	FC_{mil}	PI/EAS	*TO*	*TC*	FC_{ml}	FC_{mil}
Schedule1	71.3	71.3	66.6	65.1	0.7 (0.3)	14.6 (6.0)	15.7 (5.9)	19.2 (5.8)	44.3 (6.7)
NanoXML	51.7	45.6	45.5	43.5	0.1 (0.1)	6.2 (8.8)	6.4 (8.9)	5.6 (7.7)	7.9 (10.4)
Ant	25.7	17.2	17.2	16.9	0.1 (0.1)	3.2 (7.6)	3.4 (7.9)	3.3 (7.2)	5.2 (9.7)
XML-security	28.8	24.8	24.6	24.1	0.1 (0.1)	7.4 (9.6)	8.0 (10.4)	8.2 (10.5)	16.9 (20.9)
JMeter	18.8	18.2	18.1	17.6	0.1 (0.1)	2.3 (7.8)	2.3 (7.9)	1.8 (5.6)	2.2 (6.2)
JABA	66.9	63.8	63.3	61.5	0.3 (0.2)	78.3 (82.5)	99.7 (102.5)	82.6 (83.0)	105.2 (99.7)
ArgoUML	31.5	29.4	29.2	29.2	0.1 (0.1)	15.9 (58.2)	15.9 (57.8)	12.6 (42.8)	15.8 (49.9)
Overall	38.3	34.8	34.6	33.8	0.1 (0.2)	26.4 (60.0)	32.0 (72.1)	26.7 (57.9)	35.1 (70.8)

Finally, dynamic alias data led to slightly larger precision gain at method-instance level than method-level, which is most noticeable with Schedule1, NanoXML, and JABA, possibly due to relatively heavier use of pointers in these subjects. Overall, DIAPRO reduces the baseline impact sets by 61%–66%, implying an increase in precision of 160%–200%.

6.3.3.2 Efficiency

Table 6.2 summarizes the means and standard deviations (in the parentheses) of query costs incurred by the baseline and all DIAPRO instances, in the right five columns. As expected, DIAPRO incurred higher cost during post-processing than the baseline because of the larger execution data they traversed: full method execution trace by DIAPRO versus two integers per method by PI/EAS (*P.E.*). Among the DIAPRO instances, query cost generally grows as more dynamic data is used. An exception is FC_{ml}, though, for which adding method-level dynamic aliasing data even reduced the cost with respect to *TC* for five out

TABLE 6.3

SPACE AND OTHER TIME COSTS OF DIAPRO VERSUS PI/EAS

Subject	Prof.	Static analysis cost (s)				Runtime cost (s)						Execution data size (MB)				
		P.E.	*TO*	*TC*	*FC*	*Norm.*	*P.E.*	*TO*	*TC*	FC_{ml}	FC_{mil}	*P.E.*	*TO*	*TC*	FC_{ml}	FC_{mil}
Schedule1	13	5	6	11	17	4	10	16	19	36	91	1.0	8.2	10.2	14.8	20.2
NanoXML	12	11	14	25	39	1	1	5	7	14	26	0.4	2.4	3.0	3.5	4.9
Ant	29	27	142	170	311	1	2	2	4	9	25	1.0	2.0	3.8	5.1	7.2
XML-security	37	33	158	190	280	4	5	15	20	30	70	0.5	3.8	4.2	5.9	11.6
JMeter	51	38	372	408	764	12	13	15	28	42	61	0.5	0.8	1.1	1.6	2.4
JABA	62	55	289	326	600	11	12	14	25	51	115	2.5	15.0	16.8	23.0	24.8
ArgoUML	190	172	7,465	7,542	11,998	8	10	11	23	38	87	1.4	7.3	8.1	11.4	17.8
Overall	82	73	2,047	2,115	3,392	8	9	12	21	35	76	1.3	6.8	7.9	10.9	14.3

of the seven subjects. A possible reason is that spurious alias-induced data dependencies account for a relatively large portion of all dependencies in these subjects. In consequence, pruning those spurious DDs using dynamic points-to sets sped up searches on the reduced dependency graph during the process of impact computation.

Naturally, higher costs are associated with subjects of denser dependencies (e.g., JABA and ArgoUML), longer traces (e.g., JABA), and/or larger test input (e.g., Schedule1). The lowest efficiency of DIAPRO was seen with JABA, which has a much larger dependency graph and longer traces than other subjects. In addition, the generally large standard deviations suggest the post-processing time fluctuates greatly across different queries in most subjects. The *overall* numbers show that the query cost of DIAPRO is about half a minute per query. In fact, the average cost for the other six subjects than JABA would be 10–20s. In all, the query cost of DIAPRO looks reasonable. Further, multiple queries can be easily parallelized since their computation is independent of each other.

Comparing across the DIAPRO instances reveals that the method-instance level alias data is much more expensive than the method level data and statement coverage, which mostly brought small overheads only with respect to *TO*. This is not surprising because the instance level data can be substantially larger than the other two. The right five columns of Table 6.3 compare the sizes of such execution data used by PI/EAS and DIAPRO. As can be seen, the largest jump among the DIAPRO instances occurred when the method-instance level dynamic points-to data was added to the analysis. On the other hand, as expected also, the space cost constantly grows from *TO* to FC_{mil} because of the continuous addition of dynamic data. Yet, such costs are all quite small—the largest by FC_{mil} with JABA is about 30M only. Another source of space cost comes from the dependency graph (not shown in the table), which is no more than 41M (the largest, with ArgoUML), though.

The rest of Table 6.3 shows the time costs of the first two phases of DDIA techniques studied here, including the uncaught-exception profiling costs incurred by all DIAPRO techniques (*Prof.*) and the execution time of inputs on the original (uninstrumented) programs (*Nr.*). Mostly, both the profiling cost and static-analysis time tend to increase with the program size, with the peak number seen by the largest subject ArgoUML. Similar to previous observations, the method-instance level dynamic alias data again led to the greatest cost growth, in both the static-analysis and runtime phases. Nonetheless, the profiling is generally cheap, as is the generation of addition dynamic data at runtime. Static analysis is mostly efficient too, except for the largest subject ArgoUML. However, as the runtime phase, the static analysis incurs one-time costs for all queries for the same program version, and it can be incorporated in nightly builds in practice.

In sum, the three DIAPRO instances come with time and space costs higher than the baseline which, however, are still reasonable. Note that such extra time and spaces are, by design, natural additional costs for the sake of better effectiveness. Also, utilizing more dynamic information causes increases in both costs as expected, with the method-instance level dynamic alias analysis incurring the largest overhead among the DIAPRO instances.

TABLE 6.4

WILCOXON P-VALUES FOR NULL HYPOTHESES THAT THERE ARE NO DIFFERENCES IN MEANS WITH RESPECT TO VARIOUS PAIRED CONTRASTS AMONG PI/EAS AND DIAPRO TECHNIQUES

Subject	**PI/EAS:*TO***	***TO*:*TC***	***TO*:FC_{ml}**	***TO*:FC_{mil}**	***TC*:FC_{ml}**	***TC*:FC_{mil}**	FC_{ml}:FC_{mil}
Schedule1	1.73E-05	1	0.4854	0.2261	0.4854	0.2261	0.5523
NanoXML	9.41E-24	0.0053	0.0052	0.0001	0.9922	0.1298	0.1325
Ant	8.06E-130	0.0033	0.0025	0.0014	0.9126	0.7766	0.8583
XML-security	1.22E-104	0.1029	0.0507	0.0196	0.733	0.4659	0.6965
JMeter	2.71E-181	0.6369	0.5411	0.1745	0.8892	0.3777	0.4571
JABA	6.43E-42	0.0384	0.0129	0.0001	0.5083	0.0184	0.0423
ArgoUML	1.77E-176	0.1575	0.0995	0.0906	0.8086	0.7703	0.9599
Fisher Overall	0	0.0006	4.99E-05	2.79E-09	0.9935	0.1449	0.4326

6.3.3.3 Effects of Dynamic Data

Our statistical testing results for all the studied DDIA techniques are shown in Table 6.4. For each subject, the table presents the p-values from seven paired two-sided Wilcoxon tests each comparing one pair of DDIA instances, for which the null hypothesis was constantly the means of the impact-set sizes for the two compared techniques being equal. The hypothesis testing covered all combinations of the DIAPRO instances, but only considered *TO* when compared to the baseline. We stopped continuing comparing other DIAPRO instances to the baseline as the significance was found constantly quite strong with *TO*, and we saw that those others kept improving in precision over *TO*.

The numbers indicate that DIAPRO is strongly significantly more precise than PI/EAS, and in the majority of cases statement coverage contributes significantly to the precision of DDIA while dynamic points-to data generally does not. And nevertheless the dynamic

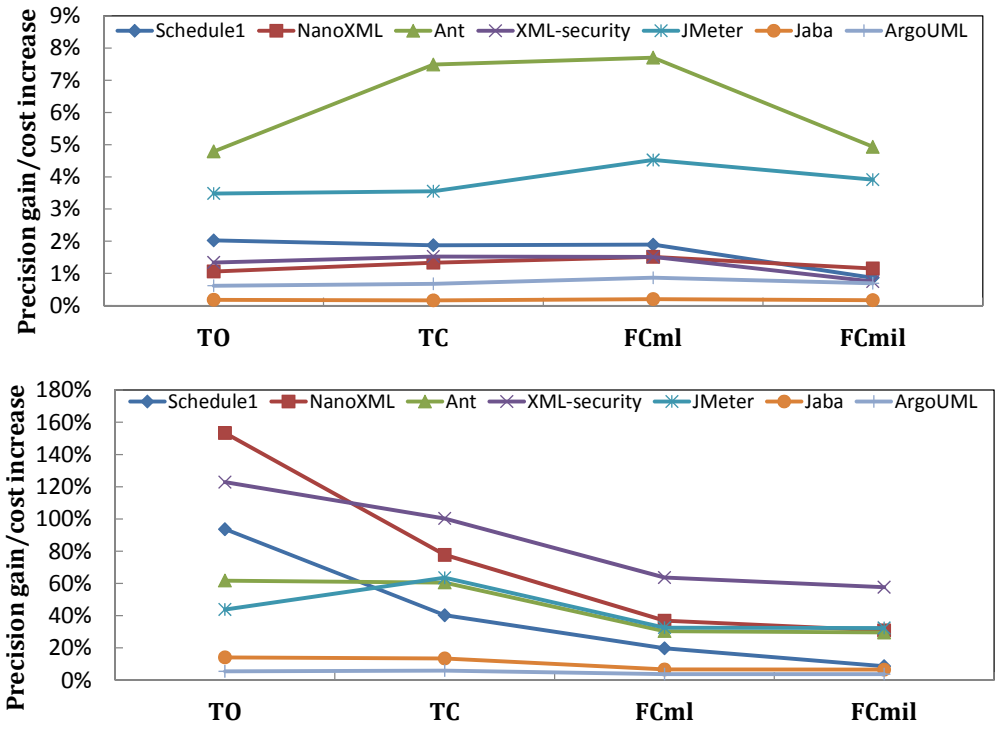

Figure 6.4: Cost-effectiveness of the DIAPRO techniques expressed as the ratios of their precision gain to the increase in the average query cost (top) and total cost of the first two phases (bottom), both against PI/EAS.

alias analysis has stronger effect on the precision when applied at method-instance level than done at method level. The bottom row lists the combined p-values for all subjects per test, which confirmed the same contrasts. In all, these observations resound with those from the precision results expressed by impact-set size ratios discussed earlier.

Finally, Figure 6.4 puts the precision and costs together showing the cost-effectiveness of the DIAPRO instances. In both plots, the y-axis indicates the precision gains (for a size ratio of r, the precision gain is $(1 - r)/r$) divided by the factor of cost increases, of each of these instances, shown on the x-axis, relative to the baseline results. The per-query post-processing cost and one-time cost of the first two phases combined are separately considered, shown in the left and right plot, respectively. We do not consider the space costs in this regard as those costs are all marginal, almost negligible relative to today's storage resources, and they are all one-time costs.

When considering the query cost only, FC_{ml} appears to be the most cost-effective DIAPRO instance for any subject. When considering the static-analysis and runtime costs, however, *TO* has the best cost-effectiveness for all subjects but Ant and JMeter, for which *TC* is the best. *TC* is always more cost-effective than the two variants of *FC*, though. Taken together, these results suggest that the most cost-effective option may vary as certain parts of the overall costs are weighed more than others, which implies that DIAPRO allows users to choose different best options for varying needs. For example, if developers readily afford the one-time cost but are less tolerant for possibly long query time, they would not bother applying additional dynamic data such as statement coverage and dynamic points-to sets here. On the other hand, developers concerned about large static-analysis cost with very large subjects but willing to wait for impact computation may find that the effectiveness gain given by statement coverage and method-level dynamic alias analysis pays off the extra cost utilizing those additional dynamic data will incur.

Nevertheless, it is also possible that developers would choose a DDIA technique preferably based on its precision gain even if it may not be the most cost-effective option: They opt for better precision anyway no matter whether the added costs are best paid off. For example, they may choose the most expensive technique FC_{mil} for its highest precision among the five instances studied, if the relatively largest cost is still affordable to them. For those developers, DIAPRO does provide more technical options than existing alternatives.

As we mentioned earlier, the impact sets give by DIAPRO are all safe relative to the execution set utilized by the analysis. Further, since *TO*, *TC*, and *FC* prune false-positive impacts continuously with increasing amount of dynamic data, the impact set of a query produced by *TC* is a subset of that of the same query by *TO*, and similar inclusion relations hold for *FC* compared to *TO* and *FC* to *TC*. At the same time, such incremental precision gains come with growing overheads in general. Accordingly, developers are suggested to adopt the DIAPRO instance that best fits their effectiveness need, time and storage budget, and availability of program information.

6.3.4 Threats to Validity

One *internal* threat to the validity of our results is the possibility of implementation errors in the five instances of our framework and our experimentation scripts. However, all the DDIA instances were based on Soot and DUA-FORENSICS that have both matured over many years, and we verified the scripts manually for each experimental step. We verified cautiously the modules of our framework used for monitoring and applying the additional dynamic data. For threats common to DIVER, we adopted similar solutions.

The main *external* threat is that our study subjects may not be representative of all real-world software. And an additional such threat lies in the limited coverage of the inputs available for the chosen subjects that we used in our dynamic analyses. To reduce such limitations, we purposely chose as many and much diverse subjects as possible that come with reasonably large and complete set of inputs, which were also often used by other researchers before.

The main *construct* threat concerns our use of relative impact-set size comparisons for precision contrasts, and the assumption about the safety of impact sets given by our techniques. With respect to actual impact sets for concrete changes, the recall may not be perfect especially when those changes modify control flows of programs at runtime. Nevertheless, our analyses are safe relative to the execution data utilized for the single program version available to them.

Finally, a *conclusion* threat is the appropriateness of our statistical analyses. To reduce this threat, we used a non-parametric hypothesis test which makes no assumptions about the distribution of the underlying data (e.g., normality). Another *conclusion* threat concerns the data points analyzed: We applied the statistical analyses only to methods for which impact sets could be queried (i.e., methods executed at least once). To minimize this threat, we adopted this strategy consistently for all experiments and compared the techniques of interest with respect to those methods only.

6.4 Related Work

In contrast to Chapter 4 which presents DIVER, this chapter focuses on a unified DDIA framework to provide multiple levels of cost-effectiveness tradeoffs for impact prediction, and studies the effects of two additional types of dynamic data on those tradeoffs.

The baseline PI/EAS comes from the method-level DDIA introduced by Law and Rothermel [108] and its performance optimization EAS [12] by Apiwattanapong and colleagues. Our DIAPRO techniques all utilize the method execution events as by PI/EAS as the common form of dynamic data to prune interprocedural dependencies that are not exercised by runtime inputs. Unlike PI/EAS that relied solely on the method execution order, however, DIAPRO leverages static program dependencies in addition to that data and other types of dynamic information to significantly improve precision against PI/EAS. Impact analysis based on *static-execute-after* relations [90] also exploits method execution order but that for a static approach (considering all possible program inputs).

DDIA approaches employing both static and dynamic information (i.e., hybrid) have also been explored before, such as INFLUENCEDYNAMIC [31] and its extension in [84]. While these techniques improve PI/EAS in terms of analysis precision, they all model partial dependencies yet exploit a single type of dynamic data (the method execution trace) only. Previous studies have shown that none of them achieved significant precision gain over PI/EAS [31, 84]. SD-IMPALA [116] also explored hybrid DDIA yet it focuses on improving the recall of DDIA and does that at the cost of penalizing precision. It uses call graphs as the static dependency model and is shown even less precise than its predecessor Impala [81] and PI/EAS. In contrast, DIAPRO is built on a complete (albeit conservative) dependency model to accomplish significantly better precision not only than PI/EAS but even further beyond DIVER we recently developed by using diverse dynamic information. It is difficult to include INFLUENCEDYNAMIC in our study as its design is constrained to procedural languages such as C while our current DIAPRO implementation targets object-oriented software, and also its environment is different from that of DIAPRO.

Statement coverage has been used in regression testing [59], test generation [101], and in general as a test metric [193], but not yet directly for predictive DDIA or even impact analysis generally, to the best of our knowledge. For DDIA, Orso and colleagues used coverage data but at method level to guide impact computation [134], which is much less precise than PI/EAS [135], though. We used statement coverage in *TC* and *FC* to improve the precision of predictive DDIA and showed that using statement coverage can greatly contribute to the precision and cost-effectiveness of DDIA.

Mock and colleagues used dynamic points-to data to improve the precision of program slicing and intensively studied the effects of that data on slice sizes [122]. Their study examined flow-sensitive and flow-insensitive dynamic points-to sets for both variables and function pointers, and found that dynamic pointer analysis does not significantly lead to better slicing precision in general. With the two *FC* instances of DIAPRO, we exploited dynamic points-to data too but for DDIA. We examined two types of such data also but differentiated them by method instance instead of by pointer dereference site, based on our different application contexts and needs. While our finding that dynamic points-to data may not translate to significantly higher precision for DDIA is akin to theirs, our study suggests higher overhead of using dynamic points-to data relative to the total cost of DDIA, at the method-instance level in particular, than what they found in the context of program slicing.

Forward dynamic slicing [98] could work as the finest-grained DDIA, yet it would be too expensive for a method-level impact analysis [12, 107] since it would have to be applied to most, if not all, statements inside the queried method. On the other hand, dynamic slicing can be an extended instance of our DDIA framework as it implicitly exploited statement coverage and (statement-instance level) dynamic points-to data too (and more). While we assume that dynamic slicing would incur much higher cost than DIAPRO, it may still be worth comparing them empirically, especially on their cost-effectiveness, for DDIA. In that regard, variants of dynamic slicing such as relevant slicing [6] and quasi slicing [187] will be of interest as well as they have different levels of efficiency, precision, and/or recall.

CHAPTER 7

ABSTRACTING PROGRAM DEPENDENCIES USING THE METHOD DEPENDENCY GRAPH

While empowering a wide range of software engineering tasks, the traditional fine-grained software dependency (TSD) model can face great scalability challenges that hinder its applications. Many dependency abstraction approaches have been proposed, yet most of them either target very specific clients or model partial dependencies only, while others have not been fully evaluated for their accuracy with respect to the TSD model, especially in approximating forward dependencies on object-oriented programs.

To fill this gap, this chapter presents a new dependency abstraction called the method dependency graph (MDG) that approximates the TSD model at method level, and compares it against a recent TSD abstraction, called the Static-Exectue-After (SEA), concerning forward-dependency approximation. We also evaluate the cost-effectiveness of both approaches in the application context of impact analysis. Our results show that the MDG can approximate TSD safely, for method-level forward dependency at least, with little loss of precision yet huge gain in efficiency; and for the same purpose, while both are safe, the MDG can achieve significantly higher precision than SEA at practical costs.

7.1 Problem Statement and Motivation

Analyzing dependencies among program entities underlies a wide range of software analysis and testing techniques [141]. While traditional approaches to dependency analysis offer fine-grained results [83], they can face severe scalability challenges, especially with modern software of growing sizes and/or increasing complexity [3, 90]. On the

other hand, for many applications where results of coarser granularity suffice, computing the finest-grained dependencies tends to be excessive, ending up with low overall cost-effectiveness. One example is impact analysis [15], where results are commonly given at method level [35, 37, 89], whereas statement-level results can be too large to fully utilize [3]. In other contexts such as program understanding, method-level results are also more practical to explore than are those of the finest granularity.

Driven by varying needs, different approaches have been explored to abstract program dependencies to coarser levels [31, 43, 200]. While these abstraction models have been shown to be useful for their particular client analyses, they either capture only partial dependencies among methods [31] or dependencies among components at the levels of granularity (e.g., class or even file level) which can be overly coarse for most tasks. More critically, none of such approaches were designed or fully evaluated as a general program dependency abstraction regarding the accuracy relative to their underlying full models (e.g., fine-grained statement-level ones) as ground truth.

In [90], a method-level dependency abstraction, called the *static-execute-after/before* (SEA/SEB), was proposed to replace traditional software dependencies (TSD) based on the system dependency graph (SDG) [83, 90]. This approach approximates dependencies among methods via control flows using the interprocedural control flow graph (ICFG) and was shown to as almost precise as static slicing based on the TSD model while coming at low costs and perfect (100%) recall. Later, the SEA was also applied to static impact analysis proven more accurate than peer techniques [183] while improving regression test selection and prioritization [165] as well.

Unfortunately, previous studies of the accuracy of SEA/SEB either targeted procedural programs [90] or focused on backward dependencies based on the SEB only [89]. The remaining relevant studies addressed the accuracy of SEA-based forward dependencies, with some indeed on object-oriented programs, yet the accuracy of such dependencies was assessed either not at the method level, but at class level only [24]; or relative to ground

truth not based on TSD, but on repository changes [91] or programmer opinions [183], and in the specific application context of impact analysis. While forward dependency analysis is required by many dependency-based applications, including impact analysis that SEA/SEB has been mainly applied to, the accuracy of this abstraction with respect to TSD, for object-oriented programs in particular, remains unknown.

However, to inform developers about the reliability of results given by SEA-based impact analysis techniques, it is important to access SEA's accuracy in approximating forward dependencies on which the impact analysis is based. In addition, according to the definition of SEA, such impact analysis identifies dependencies among methods based on their connections via control flows only. Although data and control dependencies are realized through control flows at statement level, thus the approach is expected to be safe (of 100% recall), ignoring the analysis of them can naturally lead to false-positive dependencies. And understanding the extent of such imprecision is still an unanswered but critical question.

Program dependency analysis has been underlying impact analysis, a crucial step during software evolution [15, 148] for which a typical approach is to find the *impact set* (the set of potentially impacted program entities) of points of interest, such as those for change proposals, by analyzing program dependencies with respect to those points. Despite of a large body of research invested [111], today's impact-analysis techniques still face many challenges, most of which can be reduced to the struggle between the cost and effectiveness of the techniques or their results [3, 12, 36, 41].

In this context, we lately developed DIVER [35, 37], a dynamic impact analysis that was shown to be much more precise than its previous alternatives with reasonable overheads. Given a program and its test inputs, this technique first builds a detailed statement-level dependency graph of the program, and then guides, using static dependency information in that graph, the impact computation based on method execution traces generated from the test inputs. However, during its post-processing phase, intraprocedural dependencies carry excessive overheads as they cannot be pruned by the execution traces of method-level

granularity (in essence, they are conservatively assumed to be all exercised due to the lack of statement-level dynamic data [37]). Therefore, for hybrid analysis using method-level execution data only, intraprocedural dependencies can be abstracted away.

A few approaches devoted to abstracting program dependencies to method level exist [114, 200] which, however, are as heavyweight as or even more than the TSD model [83] that either do not scale to large programs or come with excessive costs. DIVER derives method-level dependencies from statement-level ones based on the TSD model, thus it also suffered from certain costs that could be avoided. Therefore, the overheads of DIVER would be reduced without losing precision, implying increase in overall cost-effectiveness, if it *directly* models method-level dependencies to capture only necessary information used by the dynamic analysis.

The applicability of dynamic impact analysis is constrained by the availability, and quality, of program inputs, though. When such inputs are not available, impact analysis would be performed using static approaches. In the current literature, the most cost-effective method-level static impact analysis we are aware of is based on the SEA relations among methods [165]. Such analyses input the program under analysis and a query and add all methods that possibly statically execute after the query into its impact set as the output. Yet, intuitively this approach can be very imprecise because of its highly conservative nature—it roughly approximates syntactic dependencies with control flows only.

To fill the above gap, this chapter presents an alternative method-level dependency abstraction called the method dependency graph (MDG). For one thing, the MDG directly models dependencies among all methods of a program, with detailed dependencies within methods abstracted away, and does so in a context-insensitive manner, thus it is more efficient than TSD [83, 200]. On the other hand, this abstraction captures whole-program control and data dependencies, including optionally those due to exception-driven control flows [168], thus it is more informative than coarser models like call graphs or ICFG. With the MDG, we attempt to not only address the above questions concerning the latest peer

approach SEA/SEB, but also to attain a more cost-effectiveness dependency abstraction over existing alternative options.

We implemented the MDG and applied it to impact analysis for Java, which are both evaluated on five non-trivial subject programs. We computed the accuracy of the MDG for approximating forward dependencies in general and the cost-effectiveness in supporting impact analysis; we also compared the accuracy and efficiency of the MDG relative to TSD-based forward slicing against SEA. Our results show that the MDG can approximate TSD for perfect recall (100% recall) and high precision (85-90% mostly) with great efficiency for method-level forward dependencies. Our study also demonstrated that the MDG abstraction can significantly enhance the cost-effectiveness of impact analysis over the SEA approach. While not a program representation like the MDG, SEA was originally proposed as a substitute for TSD. Thus, for brevity, we refer as SEA to both the abstraction approach based on SEA relations and the SEA analysis itself when comparing them to the MDG and MDG-based analysis.

7.2 Approach

We first give a high-level description of the dependency abstraction using the MDG and then a definition of the MDG as a program representation model. Finally, algorithms for constructing the MDG on an input program are presented. We use both graph and code examples to illustrate our approach.

7.2.1 Dependency Abstraction

To create the method-level abstraction of program dependencies, we take some basic concepts from IC (integrated circuit) design for a visual analogy: the entire software is conceptualized as a hardware module (e.g., a mother board or IDE controller) consisting of multiple chipsets; a class is akin to a single chipset that is composed of a group of chips; each method is a single chip with a certain number of input and output ports; inter-

procedural dependencies are wires connecting among individual chips and intraprocedural dependencies are sub-scale wires in the circuits inside each chip. Now the purpose of the dependency abstraction via MDG is to zoom out the (statement-level) dependency graph into the hardware module such that each method is viewed as a chip where only the I/O ports are externalized while the internal circuits are hidden as in a black box, although those circuits must exist such that each input port is connected, via some path inside, to at least one output port—in a program, there might exist input ports that do not reach out any output ports, corresponding to input parameters of a method that do not escape the method; there could also exist output ports that are not reachable from any input ports, corresponding to some output parameters or returns that do not depend on any input parameters.

Based on such analogy and conceptualization, for defining the MDG, we reuse the the terms *port* and, accordingly, *incoming ports* (*IP*s) and *outgoing ports* (*OP*s), incoming dependency and outgoing dependency, introduced in the DIVER design [35]. However, instead of simply naming such terms from the vertices on interprocedural dependency edges after a dependency graph is constructed using a whole-program statement-level data and control flow analysis, we define the MDG based on these terms before constructing it and use them to guide the construction of the MDG. Concretely, we will explicitly find the ports without first computing interprocedural and intraprocedural dependencies, but instead first locating the input and output ports and then connecting them *among* methods; we will compute intraprocedural dependencies for connecting input to output ports *inside* methods and discard them afterward, and will not keep them in the MDG. To facilitate the description of MDG and the design of application algorithms based on MDG (e.g., dynamic impact analysis using MDG), we also reuse the interprocedural DD classification in DIVER: parameter, return and heap (see Section 4.2.2).

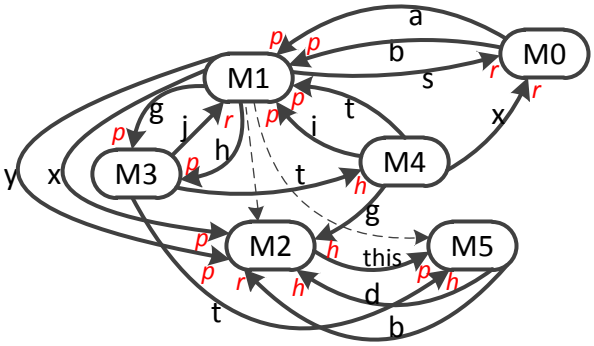

Figure 7.1: The method dependency graph (MDG) of program $E2$.

7.2.2 MDG Definition

An MDG is a directed graph that consists of method-level data and control dependencies, with summary dependency edges abstracting intraprocedural data and control dependencies inside each method. A method m' is *data dependent* on a method m if m defines a variable that m' might use, whereas a method m' is *control dependent* on a method m if a decision in m determines whether m' (or part of it) executes. We continue to handle control dependencies for exceptional control flows as we did in the dependency graph used in both DIVER and our DDIA framework. Summary dependencies of a method m connect the IPs to OPs of m each representing that an IP is reachable to an OP via at least one intraprocedural dependency chain inside m.

Each MDG node n_m represents a method m, represented by a tuple that incorporates three elements: method identifier for m (e.g., a method index), list of IPs and list of OPs. An MDG edge from a method m to a method m' connects a particular OP of m to a particular IP of m', expressing either a method-level data or control dependency of m' on m. Therefore, the MDG of a program has the same number of nodes as that of the methods in the program, but has the number of edges corresponding to the number of interprocedural edges in its (statement-level) dependency graph.

Figure 7.1 shows the MDG of our example program $E2$ of Figure 2.2 where each node represents a method and each edge a dependency of the target node on the source. CDs are

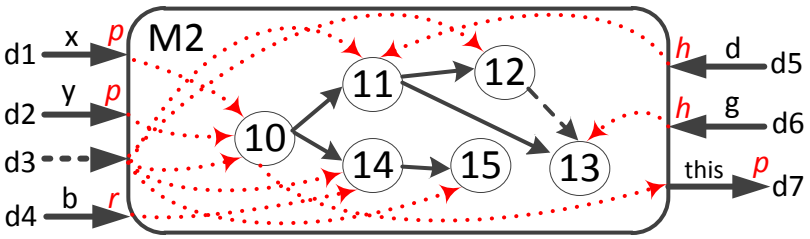

Figure 7.2: Statement dependencies in `M2` used to find *summary* dependencies of outgoing on incoming dependencies in the MDG. Statement dependencies are *discarded* after analyzing the method.

dashed edges and DDs are solid edges. Each DD edge is labeled with the corresponding variable and its arrow is labeled with the DD type (*p* for parameter, *r* for return, and *h* for heap). In the example, `M2` has six incoming dependencies (five DDs and one CD), such as the DD labeled `g` with arrow *h* caused by the use of heap variable `g` defined in `M4`. Method `M2` has also an outgoing DD edge to `M5` because it calls `M5` with parameter `this`, which is an implicit parameter of `M2`.

Figure 7.2 shows the statement-level dependencies *within* `M2` (i.e., its PDG [63]) and the incoming and outgoing dependencies of that method, named *d1–d7* for convenience. Dotted (not dashed) edges indicate the connections between method- and statement-level dependencies. For M2 and every other method, a reachability analysis on these connections identifies the summary dependencies of outgoing dependencies on incoming dependencies. For `M2`, only *d1*, *d2*, and *d3* reach the outgoing dependency *d7*. Thus, the *summary dependencies* for `M2` are $\langle d1, d7\rangle$, $\langle d2, d7\rangle$ and $\langle d3, d7\rangle$.

7.2.3 Construction of the MDG

Finding the dependencies in the MDG requires an *intraprocedural* analysis of the *statements* of each method. However, unlike statement-level graphs [83], only summaries of reaching definitions, reachable uses, and control dependencies for *call sites* and *exit nodes* are kept in memory after each intraprocedural analysis. For the CDs, we first compute *reg-*

Algorithm 3 : BUILDMDG(program P, exception set *unhandled*)

```
1: G := empty graph // start with empty MDG of P
2: IP := OP := ∅   // maps of methods to incoming/outgoing ports
   // Step 1: find ports
3: for each method m of P do
4:     FINDDDPORTS(m, IP, OP)
5:     FINDCDPORTS(m, IP, OP)
   // Step 2: connect ports
6: for each method m of P do
7:     for each DD port z ∈ OP[m] do
8:         add {⟨z, z'⟩ | ∃m' s.t. z'∈IP[m'] ∧ data·dep(z,z')} to G
9:     COMPUTEINTERCDS(G, unhandled, m, IP, OP)
10:    pdg := GETPDG(m)
11:    for each port z ∈ IP[m] do
12:        add {⟨z, z'⟩ | z'∈OP[m] ∧ reaches(z,z',pdg)} to node G_m
13: return G
```

ular CDs caused by branches, polymorphic calls, and intraprocedural exception control-flows. For the remaining exception control-flows [169] (*ExInterCDs*), we compute the CDs for the exception types not handled by the originating methods, detected using the same exception profiler as used in DIVER.

The MDG construction algorithm uses *intraprocedural* statement-level information for each method. This information is *discarded* after analyzing each method, once ports and summary dependencies are identified. Hence, the algorithm does not incur the time and space overheads of statement-level interprocedural analysis. Only the method-level information required by the MDG is kept.

For a node (method) m, the *IP*s of m for DDs are the *uses* of variables v in m reachable from definitions of v in other methods or recursively in m. The *OP*s of m for DDs are the *definitions* in m that reach uses in other methods or recursively in m. The following are the cases in which ports for DD can be identified:

- For heap variables, including exceptions, whose definitions are *OP*s and uses are *IP*s
- For method calls, where actual parameters (at call sites) are *OP*s and corresponding formal parameters (at method entries) are *IP*s
- For method returns, where returned values at callees are *OP*s and returned values at corresponding caller sites are *IP*s

The CD *IP*s of a method m are denoted by special locations within m whose executions depend on external (or recursive) decisions such as calling m or returning to m with an unhandled exception. Those control-flow decisions are the *OP*s. Concretely, CD ports are identified for these cases:

- The entries of all methods that can be invoked are *IP*s. For a non-polymorphic call site (which can only call one method), every branch and CD *IP* that guards it is an *OP*. For a polymorphic call site (which has multiple target methods), the call site itself is an *OP* because it decides which method is called
- For an unhandled exception x thrown by a method m, the *entry points* of all blocks that can handle x (e.g., *catch* statements) at callers of m are *IP*s. The conditions that cause the exception to be thrown (i.e., the branches and CD *IP*s that guard its throwing or the instruction that conditionally throws it) are *OP*s

The cases listed for DD and CD ports set the rules for matching ports to determine the (interprocedural) DD and CD edges of the MDG—an *OP* can connect only to an *IP* for the same case. Thus, an MDG edge e from m to m' links an *OP* of m (the source of e) to a compatible *IP* of m' (the target of e) according to these cases.

An MDG node represents a method, its *IP*s, its *OP*s, and the *summary dependencies* that map *IP*s to *OP*s in that method. An *OP* p_o is summary-dependent on an *IP* p_i if there is a path from p_i to p_o in the (intraprocedural) statement dependency graph of the method [63]. With this information and the MDG edges, a client analysis such as DDIA can find which methods are impacted by a method m by traversing the MDG from m and all *OP*s of m conditioned to edges whose source *OP*s are summary-dependent on the *IP*s that are targets of traversed (impacted) edges.

Algorithm 3 describes the process for building an MDG. We use the following helper notations to facilitate the presentation: a caller (call) site *crs* (*cs*) is a tuple $\langle m, s \rangle$ where m is the caller (set of callees) and s the calling statement; *actual˙params*(*cs*) is the actual parameter list of a call site *cs* and *formal˙params*(m) the formal parameter list of m; *return˙sites*(m) is the set of return statements in m and *return˙type*(m) the return type of m; *D(rs)* is the definition of the return value in a return statement *rs*; *U(crs.s,rs)* is the use at

Algorithm 4 : FINDDDPORTS(m, *IP*, *OP*)

```
1:  for each call site cs in m do
2:      for each callee m' of cs do
3:          add {D(a, cs) | a ∈ actual_params(cs)} to OP[m]
4:          add {U(f, m') | f ∈ formal_params(m')} to IP[m']
5:  if return_type(m)≠void then
6:      add {D(rs) | rs ∈ return_sites(m)} to OP[m]
7:      for each caller site crs of m do
8:          add {U(crs.s,rs) | rs ∈ return_sites(m)} to IP[crs.m]
9:  for each heap variable definition hd in m do add hd to OP[m]
10: for each heap variable use hu in m do add hu to IP[m]
```

a caller site *crs* of the value returned by a return statement *rs* in a method called by *crs*. We also denote a formal parameter f at the entry of m as the use *U(f,m)* and an actual parameter a in a call site *cs* as definition *D(a,cs)*.

The algorithm inputs the program P under analysis and a set of unhandled exceptions and outputs the MDG of P. The exception set contains, by default, all possible exceptions for safety. As mentioned earlier, the exception profiler identifies a potentially-smaller set for efficiency while staying safe. First in the algorithm, the DD and CD ports are identified for all methods of P via FINDDDPORTS and FINDCDPORTS, respectively. Next, the algorithm computes all DD edges, CD edges, and summary dependencies by connecting the ports that match (e.g., actual and formal parameters).

DD edges between methods are created in lines 7 and 8 by matching each DD *OP* z to each DD *IP* port z' that is data dependent on z according to any of the three cases described earlier. CD edges are created via COMPUTEINTERCDS in line 9, which matches CD ports according to the rules for CDs listed earlier. CDs due to exceptions are included only for exceptions in the set *unhandled*. Finally, the algorithm computes the *summary* dependencies within each method (lines 10–12). For each method m, given its PDG [63] (line 10) which contains all intraprocedural dependencies, the algorithm matches each *IP* with every *OP* that the *IP* can reach in that PDG. For each match, a summary edge $\langle z, z' \rangle$ is added to the node G_m for m in MDG G (line 12).

Algorithm 5 : FINDCDPORTS(m, *IP*, *OP*)

```
 1: add entry of m to IP[m]  // entry represents all CD targets for callers
 2: for each edge ⟨h, t⟩ in GETCDG(m) do
 3:     if t is a single-target call site then {add h to OP[m]}
 4:     if t unconditionally throws unhandled exception in m then
 5:         add h to OP[m]
 6: for each multi-target call site cs in m do {add cs.s to OP[m]}
 7: for each statement s in m do
 8:     if s catches interprocedural exception then {add s to IP[m]}
 9:     if s conditionally throws exception unhandled in m then
10:         add s to OP[m]
```

The helper Algorithm 4 shows the details for FINDDDPORTS. For the input method m, the algorithm first traverses all call sites to find, for each call site and callee, the definition and use of actual and formal parameters, respectively (lines 1–4). Then, for methods that return values (line 5), the returned values are added, as pseudo-definitions, to the *OP*s of m (line 6) and the use of that value at each caller site are added to the *IP*s of the caller methods (lines 7–8). Finally, the algorithm finds and adds all definitions and uses of heap variables in m to the corresponding *OP* and *IP* sets.

The helper Algorithm 5, FINDCDPORTS, first identifies as *IP* the entry point of m. This point represents the decision to enter the method, which in a PDG is the *true* outcome of the *Start* node [63]. Then, using the control dependency graph (CDG), lines 2–5 mark as *OP*s the decisions that guard single-target calls and unconditional throwers of unhandled exceptions. Those decisions can be branches, the entry of m (target of caller dependencies), and targets of callee dependencies (interprocedural exception catchers and calls to methods that might return abnormally [169]). Then, all multi-target call sites in m are added to the *OP*s of m (line 6). Lines 7–10 find the *IP*s that catch interprocedural exceptions and *OP*s that throw exceptions conditionally (e.g., null dereferences).

7.3 Empirical Evaluation

We evaluated our technique as a dependency abstraction in general and its application to impact analysis in particular. For that purpose, we computed the precision and recall of forward dependency sets derived from the MDG against forward static slices, both at method level, and compared the same measures and efficiency against SEA. Accordingly, we seek to answer the following three research questions:

RQ1 How accurately can the MDG and SEA abstract the full TSD model in terms of approximating forward dependencies?

RQ2 Can the dependency abstractions (the MDG and SEA) archive significantly better efficiency than the TSD model for forward dependency analysis?

RQ3 Are the MDG and the static impact analysis directly based on MDG more cost-effective than the SEA and SEA-based static impact analysis?

7.3.1 Experiment Setup

We briefly discuss key implementation issues and describe the subject programs used for obtaining the following empirical results. All our studies were performed on a Linux workstation with a Quad-core Intel Core i5-2400 3.10GHz CPU and 8GB DDR2 RAM.

7.3.1.1 Implementation

We implemented the MDG, SEA, impact analysis tools based on the two abstractions, and the method-level TSD-based forward static slicer all on top of our dependency-analysis and instrumentation system DUA-FORENSICS [163], which is built on the Soot byte-code analysis framework [185]. To compute control dependencies, including those due to exception-driven control flows, we used the *exceptional control flow graph* (ExCFG) provided by Soot as recently did in [35, 39] as well. The ExCFG was also employed to create the interprocedural component control flow graph (ICCFG) [24, 90], on which SEA

TABLE 7.1

EXPERIMENTAL SUBJECTS FOR MDG EVALUATION

Subject	**Description**	**#LOC**	**#Methods**
Schedule1	priority scheduler	290	24
NanoXML	XML parser	3,521	282
Ant	Java project builder	18,830	1,863
XML-security	encryption library	22,361	1,928
JABA	bytecode analyzer	37,919	3,332

was implemented using the *on-demand* algorithm presented in [89]. For static slicing, we directly used the context-sensitive forward static slicer as part of DUA-FORENSICS with results lifted up to method level. Both the slicer and SEA implementations utilized the same call graph facilities given by Soot with the rapid type analysis applied.

A static impact analysis based on the MDG was also implemented, which simply gives as the impact set the transitive closure on the MDG starting from the input query (more precisely, starting from each *OP* of the query, and then taking the union of all such closures). The SEA-based impact analysis produces as the impact set of a given query the set of all methods that are in the SEA relation with that query.

7.3.1.2 Subject Programs

We selected five Java programs of diverse application domains and sizes for our evaluation. Table 7.1 lists the basic characteristics of these subject programs, including the number of non-blank non-comment lines of Java code (*#LOC*) and number of methods (*#Methods*) defined in each subject. The first four subjects are all obtained from SIR [57], for which we picked the first versions available. JABA was received from its authors.

7.3.2 Experimental Methodology

This main goal of our study is to addresses the accuracy of the MDG against both the SEA approach and the TSD model. Since impact sets computed by the static impact analysis based on the MDG and SEA are also the method-level forward dependency sets used by the accuracy study, we *simultaneously* evaluate the accuracy of these two abstraction models and the static impact analysis techniques based on them. We also study the efficiency of all these approaches. For this study, we applied the MDG- and SEA-based static impact analysis tools, and the method-level TSD-based forward static slicer, to each of the five subjects. We collected the forward dependency set (i.e., the impact set or method-level forward slice) of every single method defined in each subject as a query by running each of the three tools on that query separately.

To obtain the method-level forward slice of a query from the slicer, we computed the statement-level forward slice of every applicable slicing criterion, and then took the union of the enclosing methods of statements in those slices. We also collected the CPU time elapse as the querying cost per such query. Next, we calculated the following metrics.

First, we calculated the precision and recall of forward dependency set produced by the MDG and SEA for each query using the corresponding forward slice given by the static slicer as the ground truth: The precision metric measures the percentage of dependencies produced by the abstraction approaches that are true positives (i.e., included in the forward slice), while the recall measures the percentage of dependencies in the forward slice that are included in the dependency set produced by the abstraction approaches. We report the distribution of the entire set of data points for these two metrics per subject.

Second, we computed the forward-dependency querying time costs of the MDG, the SEA, and the forward static slicing. We also report the time costs of building the program representations (i.e., ICCFG and the MDG) used by the abstraction approaches. These two types of costs are calculated separately to give more detailed efficiency results that users may need for better planning their budgets: The times for abstracting program dependen-

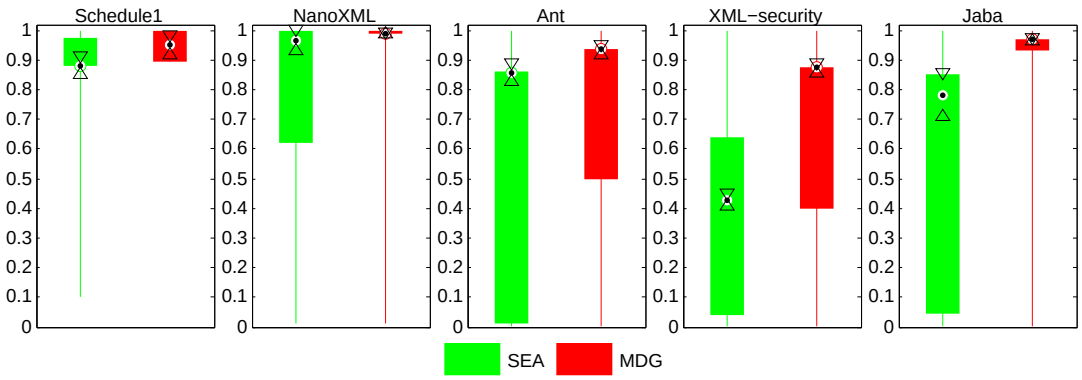

Figure 7.3: Precision of the MDG and SEA relative to TSD-based forward slices as the ground truth (with constantly 100% recall).

cies are one-time costs in the sense that for any queries (criteria) the abstract dependency models can be reused in the query processing phase; while the querying time is incurred for each individual query.

Finally, we applied a non-parametric statistical test, the Wilcoxon signed rank test [191], to assess the statistical significance of mean difference in each of the above two metrics between the two abstraction approaches against the method-level static forward slicing. For the statistical test, we adopted a confidence level of 95% with the null hypothesis for no difference in the means. We also report the significance over all subjects when applicable by combing the per-subject p-values using the Fisher method [124].

7.3.3 Results and Analysis

7.3.3.1 Accuracy of Dependency Abstractions (the MDG and SEA versus TSD)

Figure 7.3 shows the precision results of the two abstraction approaches, as listed on the x axis, in approximating forward dependencies relative to the TSD model, where the y axis represents the precision. For each subject, a separate plot characterizes all the data points we analyzed, which consists of two boxplots each for the data from one of the two approaches with that subject.

Each of the boxplots includes five components: the maximum (upper whisker), 75% quartile (top of middle box), 25% quartile (bottom of middle box), the minimum (lower whisker), and the central dot within each middle bar indicating the median. Surrounding each such dot is a pair of triangular marks that represent the comparison interval of that median. The comparison intervals within each plot together express the statistical significance of the differences in medians among the two groups in that plot: their medians are significantly different at the 5% significance level if their intervals do not overlap.

The results indicate that the MDG can approximate the TSD-based forward dependencies with generally very high precision in most cases: for the majority of queries in all subjects, the precision was around 90%, according to the medians, and even low-end 25% of the queries had a precision between 45% (the lowest, with XML-security) to 98% (the highest, with NanoXML). For NanoXML and JABA, the precision was over 95% for 75% of queries, and for Schedule1 it was also as high as 90%. In these cases, we found many queries that share the same dependency sets, possibly due to the existence of dependency clusters as previously investigated [165]. The worst overall precision was seen by XML-security, for which the MDG gave a precision of no more than 85% for 75% of its 1,928 queries. Another subject that received mostly lower precisions than other subjects, except for the worst-case subject XML-security, was Ant, where 25% of queries had forward dependency sets less than 55% precise.

While the MDG did not seem to have lower precisions for larger subjects—in fact, it performed almost as well with the two largest subjects as with the two smallest ones—SEA did see such trend, although not constantly. Except for the same worst case as has been seen by the MDG (with XML-security), both the other two largest subjects had generally much lower precision from SEA than what the MDG gave. Besides Schedule1 and NanoXML, for which it was as almost precise as the MDG, SEA produced results of less than 5% precise for 25% of queries in other three larger subjects. For XML-security, in particular, the SEA precision did not even reach 50% for 50% of queries.

However, with both the MDG and SEA, there were cases in which the dependency abstraction missed almost all true-positive forward dependencies (precision close to zero), though much fewer seen by the MDG. We inspected certain samples of such cases and found that in most of the resulting dependency sets only one was true positive: the query itself. While the most possible common reason was the conservativeness of both abstraction approaches, the fact that the MDG has a lot less such bad cases than SEA was more probably due to their different underlying technique in nature: the MDG models both data and control dependencies, which tends to be less conservative than SEA which considers roughly control flows only.

Supplementary to the bloxplots, the left three columns of Table 7.2 gave another statistics, the means, of the precision results. Mainly due to the existence of bad cases discussed above, the means were dropped down considerably relative to the numbers seen in the distribution of all data points, for both approaches. Comparison between the MDG and SEA reveals that both had an overall similar trend in the fluctuation of means across the five subjects: where the MDG had relatively lower mean precisions, did the SEA too. Nonetheless, the p-values (the fourth column) show statistical significant differences quite strongly in the means between the two approaches, further confirming the advantage of the MDG over SEA on top of the significance in medians shown by the comparison intervals on the boxplots of Figure 7.3.

Over the total data points of all subjects, the MDG had a precision of 72% on average, significantly (with p close to 0) higher than the 49% precision archived by SEA. Note that such means were largely skewed as discussed above and, thus, in most cases, users should anticipate much higher precision from the MDG than from SEA. Additionally, for the SEA, in contrast to previous accuracy studies that reported very high precision from it, at class level [24] or at method level but with procedural programs (in C) considered only [90], our results suggest that object-oriented programs may contain much more method-level data and control dependencies that can not be accurately captured by control flows only,

TABLE 7.2

PRECISION MEANS OF THE MDG VERSUS THE SEA AND TIME COSTS OF BOTH RELATIVE TO FORWARD SLICING

Subject	**Mean precision**			**Abs. time (s)**		**Dependency querying time (ms): mean (stdev)**			
	SEA	MDG	p-value	SEA	MDG	SEA	MDG	p-value	Slicing
Schedule1	0.81	0.94	2.4E-02	3	4	6 (3)	4 (2)	0.4E-02	124 (194)
NanoXML	0.77	0.88	2.0E-09	4	9	9 (12)	3 (4)	7.9E-07	1,267 (3,095)
Ant	0.55	0.72	7.8E-79	17	130	64 (62)	45 (43)	1.1E-08	34,896 (74,210)
XML-security	0.40	0.67	4.6E-83	22	77	50 (67)	43 (34)	4.5E-16	24,092 (46,601)
JABA	0.58	0.84	1.9E-33	28	302	213 (201)	121 (221)	3.6E-10	444,188 (801,631)
Overall	0.49	0.72	6.8E-195	14.8	104.4	131.4 (294.1)	53.3 (66.7)	1.9E-35	55,737.9 (241,084.9)

when compared to other cases (e.g., method-level forward dependencies on object-oriented programs) studied before.

Recall (MDG and SEA versus TSD) Given the conservative nature of both abstraction approaches and the consistency that all the studied techniques were implemented on the same analysis infrastructures (call graph, points-to analysis, and data and control flow analysis facilities, etc.), we expected that both MDG and SEA are safe. Our results confirmed this hypothesis: for all the data points we collected, the *recall was constantly 100%* for both approximation models. Consequently, the precision numbers reported here can be readily translated to accuracy values (e.g., for a precision p, the F1 measure of accuracy is $2p/(1+p)$). Thus, we only show the precision here for the evaluation on accuracy.

Answer to RQ1: *Both the MDG and SEA can approximate the TSD-based forward dependencies safely; the MDG can also give high precisions in most cases, while SEA is significantly less precise in general.*

7.3.3.2 Efficiency of Dependency Abstractions (MDG and SEA versus TSD)

The rest of Table 7.2 focuses on the efficiency of the two abstraction approaches versus the TSD model, including the time in seconds (*Abs. time*) consumed by building the underlying graphs (*ICCFG* used by the SEA and the MDG), the time in *milliseconds* taken by the three techniques for querying forward dependency sets based on respective dependency models (*dependency querying time*). For the latter, the table lists the means and standard deviations (*stdev*) of all data points for each subject. The p-values shown in the ninth column report the statistical significance of the Wilcoxon test on the mean differences in querying costs between the two abstraction models.

In terms of the graph construction time, since the MDG as a program representation contains much finer-grained information than the ICCFG, the MDG approach costs always more than the SEA, as we expected. Yet, in all cases, the costs were at most five minutes, which should be quite affordable. Note that the costs did not increase with subject sizes, implying that users would not necessarily expect increasingly higher costs for subjects of growing sizes. Intuitively, the internal logic complexity of programs is an additional factor. Moreover, this phase is required only once for all possible impact-set queries, for the single program version the client analysis works with at least. Finally, in case of larger overheads, this step could be included as part of nightly build in practice.

Also, the querying-time results to the right of the table show that the higher one-time costs of the MDG approach were easily paid off: compared to SEA, the MDG costed constantly less in terms of the mean querying time and with smaller variations. In addition, the p-values tell that, for all individual queries, the MDG was more efficient than SEA with strong significance in all cases. In all, over all subjects, the mean querying cost on the MDG is 53ms, less than half of that incurred by SEA.

Compared to the cost of the full TSD model (the last four columns), however, both abstraction models appeared to be much cheaper. For the two smallest subjects, querying the method-level forward dependencies was 100x faster on the two abstraction models than

on the full TSD model; for other subjects, the speedup over TSD was almost 1000x. As an example, the huge variations of the TSD querying costs suggest that, in some cases, a single query can take as long as a few hours, as we experienced in experimentation. Between the two abstraction approaches, on average over all queries, SEA costs 1.47% of the per-query time incurred by the TSD-based forward slicing, in contrast to 0.44% by the MDG.

Answer to RQ2: *Both the MDG and SEA are reasonably efficient, and much (100–1000x) cheaper than the TSD model for querying method-level forward dependencies.*

7.3.3.3 Cost-Effectiveness for Static Impact Analysis (MDG versus SEA)

From the foregoing comparisons, it has been seen that the MDG as a TSD approximation approach is generally much more precise than the SEA alternative, for approximating forward dependencies at least. The numbers in Table 7.2 also suggest that the MDG provides significantly faster dependency-set querying than SEA too. In addition, for the cases we studied, both approaches were confirmed to be safe, always giving perfect recall relative to the fine-grained TSD-based forward static slicing.

Note that the forward-dependency sets studied above can also be regarded as impact sets from the perspective of static impact analysis. Therefore, taken together, the advantages in the precision and querying efficiency of the MDG over the SEA implies that, besides the graph building overhead which is a one-time cost and mostly quite practical, the MDG-based static impact analysis is potentially more cost-effective than the SEA-based analysis.

Answer to RQ3: *The MDG incurs expectedly larger yet still affordable one-time cost, and tends to be significantly more cost-effective for static impact analysis, than the SEA.*

7.3.4 Threats to Validity

The main *internal* threat to the validity of our results is the possibility of implementation errors in the tools used for our study (the analyses based on the MDG and SEA, and the forward static slicer). However, the underlying infrastructures, Soot and DUA-

FORENSICS, have both been tested, improved, and matured for many years. To reduce errors in the code written on top of these frameworks, we manually checked and verified the correctness of our implementations using both the example program and two subjects Schedule1 and NanoXML for randomly sampled queries. An additional *internal* threat is the possibility of errors in our experimental and data-analysis scripts. To minimize this risk, we tested and debugged those scripts and checked their functionalities against the requirements of our experimental methodology.

Another *internal* threat is the risk of misguiding the experiments with inaccurate ground truth obtained from the static slicer. However, this threat has been mitigated in several ways. First, the slicer, as part of DUA-FORENSICS, has been used and stabilized along with the entire framework over seven years. Second, most of the core facilities that could affect the accuracy of this slicer are adopted or extended from the Soot framework, which is a static-analysis platform widely used by many researchers. Finally, since those core facilities are shared by our implementation of the slicer, the MDG, and SEA, possible biases, if any, in the results derived from comparing among these tools have been greatly reduced.

The main *external* threat to the validity of out study is our selection of study subjects. This set of five Java subjects does not necessarily represent all types of programs used in real-world scenarios. To address this threat, we picked our subjects such that they were as diverse as possible in size, application domain, coding style, and complexity. Also, the subjects we used in our study have been extensively used by many researchers before. Another *external* threat is that the forward slicing algorithm we used in our study may not be the optimal one in terms of precision and efficiency, thus using a more sophisticated slicer would possibly lead to different study results. Similarly, while the SEA algorithm we adopted is the latest one we are aware of that processes one query at a time, which is justified for comparing per-query processing time costs between the two abstraction approaches, different efficiency contrasts may be obtained if comparing the total time of processing all possible queries of a program at once (using the batch SEA-computation algorithm in [89]).

The main *construct* threat lies in our experimental design. Without any additional knowledge, we gave the same weight to the forward dependency sets (impact sets) of every method (query). However, in practice, developers may find some methods more important than others and, thus, the reported precision results might not exactly represent the actual results that developers would experience. To address this potential concern, we adopted the same experimentation procedure when obtaining the dependency sets from the two approaches (the MDG and SEA) we compared.

Finally, a *conclusion* threat to validity is the appropriateness of our statistical analyses. To reduce this threat, we used a non-parametric hypothesis testing which makes no assumptions about the distribution of underlying data points (e.g., normality). Additionally, we collected and analyzed data points for all methods as queries in each subject in order to avoid relevant biases.

7.4 Related Work

Program Dependency Abstraction. Most existing dependency-abstraction approaches were designed for specific applications, and also mostly did not directly model or subsume complete method-level (data and control) dependencies. For instance, the *program summary graph* [43] was originally developed to speed up interprocedural data-flow analysis, which is similar to the data dependency abstraction part of the MDG but is built based on program call structure only and, thus, can be not only imprecise but also unsafe from the perspective of approximating a full program dependency model.

Particularly targeting impact analysis, several abstract dependency models were proposed, such as the *lattice of class and method dependency* [176] and the *influence graph* [31]. These abstractions consider only partial dependencies: The former only captures structural dependencies among classes, methods, and class fields that are directly derived from objected-oriented features, such as class-method memberships and class inheritance, and method-call relations; the latter models only data dependencies among methods in an

overly conservative manner while ignoring analysis of intraprocedural dependencies, which has been shown to be highly imprecise (precision close to a transitive closure on the ICFG) [31]. The RWSets tool in [60] abstracts data dependencies via field reads and writes but ignores control and other data dependencies.

Other approaches explicitly attempted to model method-level dependencies. One example is the *abstract system dependency graph* (ASDG) [200], which is built by first creating the entire SDG and then simplifying statement-level edges thereof. In [114], an extended TSD model is described to generalize the definitions of interprocedural dependencies, directly at both statement and procedure levels. Directly derived from or developed atop the SDG [83], these models are at least as heavyweight as the underlying TSD model itself.

The SEA/SEB abstraction [90] to which we compared the MDG is developed on the simplified ICFG (i.e., ICCFG) to capture method execution orders statically, which is motivated by the dynamic version of such orders proposed in [12]. Developed also for interprocedural data-flow algorithms, the *program super graph* [127] connects per-procedure control-flow graphs with call and return edges, similar to the ICFG but enclosing calling contexts as well. The context-sensitive CFG in [129] is proposed to visualize CFGs for program comprehension, which also simplifies each intraprocedural CFG as by the SEA/SEB.

In contrast, the MDG we proposed directly models method-level dependencies that explicitly include both data and control dependencies. However, compared to the TSD model, the MDG dismisses expensive interprocedural data-flow analysis with context-sensitivity ignored as well, which makes it conservative yet enables it to be relatively lightweight. The summary edges in the MDG are also different from those of the same name used in the SDG and ASDG: Those edges were used to help represent calling contexts in the SDG and transitive data-flow across procedures in the ASDG; we use such edges to abstract reachability from incoming to outgoing dependencies within each method.

On the other hand, it is worth noting that this work is constrained to examining dependency abstractions with respect to the TSD model, while, as previous work revealed

and studied [24, 200], hidden dependencies that cannot be captured by TSD also exist, especially in object-oriented software. And other researchers have shown that the SEA approach can help discover those dependencies [24]. Therefore, it would be also of interest to investigate in that regard using the MDG and in comparison to the SEA/SEB.

Impact Analysis. Static impact analysis provides impact sets for all possible program inputs. At method level, a main approach to such analysis is to find methods that are directly or transitively dependent on the given query. In comparison to the SEA-based impact analysis that requires a reachability algorithm [165, 183], a static impact analysis based on the MDG simply computes the transitive closure from the query. The MDG-based impact analysis also gives more information regarding *how*, in addition to whether, impacts propagate across methods (through the ports and edges between them), thus it tends to better support impact inspection and understanding, than the SEA-based approach.

Static program slicing has been directly used for static impact analysis, but it was shown to have challenges from overly large results and/or prohibitive costs [3]. Many other static approaches exist, which utilize various types of program information, such as code structure and version repository [111, 183]. Our static impact analysis based on the MDG utilizes method-level dependencies to offer an efficient approach with a precision comparable to fine-grained static slicing. Note that a large body of other impact-analysis techniques has been developed but is *descriptive* [15, 111], such as SIEVE [149], CHIANTI [151], and the impact analysis based on static slicing and symbolic execution [157]. These approaches require prior knowledge about actual changes made between two program versions. In contrast, the impact analysis we focused on in this chapter is *predictive*, which inputs a single program version without knowing the actual changes, thus it gives prediction of possible impacts based on information from the single version of the program rather than describing the impacts of those concrete changes that have already been applied.

CHAPTER 8

EFFICIENT DYNAMIC IMPACT ANALYSIS FOR DISTRIBUTED SYSTEMS

With increasing number of distributed systems being deployed to meet the high performance and scalability needs of today's computing infrastructure, it is crucial to provide fundamental techniques for evolving these systems. Dynamic impact analysis is one of such techniques, which helps developers assess potential change effects for concrete operational profiles of the systems. Unfortunately, while techniques for this analysis are available, most of them were applicable to centralized programs only, with few others aiming at distributed systems yet limited to specific programming paradigms and/or language features. This chapter presents DistEA, an efficient dynamic impact analysis that works for general distributed programs where components communicate via message passing. By partially ordering distributed method-execution events and leveraging the executes-after relations among methods, DistEA provides a highly efficient means for developers to understand the impacts of one entity on others across multiple components and processes of the system.

8.1 Problem Statement and Motivation

Constant code changes drive the evolution of software during its life cycle but can also pose threats to software reliability [146]. Thus, it is crucial to understand potential consequences of those changes *before* applying them to candidate program locations. To accomplish this task, developers often need to perform impact analysis [15, 156, 180] with respect to those locations, an integral step of modern software development process [148]. In particular, for developers working with specific operational profiles of programs, dy-

namic impact analysis [15, 111] tends to be a more attractive option as it narrows down the search space of such impacts to the concrete context of those profiles.

During the past two decades, research on dynamic impact analysis has been extensively invested [111], resulting in a rich and diverse set of relevant techniques and tools (e.g., [12, 35, 37, 108]). However, most of existing such approaches mainly address sequential programs only, with much less targeting concurrent yet centralized software [69, 72, 104, 138, 194] while very few applicable to distributed systems where components running in multiple processes concurrently over separate machines. On the other hand, to accommodate the increasingly demanding performance and scalability needs of today's computing infrastructure, more distributed systems than centralized software are being deployed, raising an urgent call for technical supports, including impact analysis, for effective maintenance and evolution of distributed multiple-process programs [22, 67, 92, 138].

Code analysis techniques for distributed systems were actually explored early on [58, 99, 102] and ratcheted up recently [20, 123, 130, 137], largely focusing on fine-grained analysis of dependencies among program entities (e.g., statements). However, the majority of these approaches was developed for procedural programs [20]. For distributed object-oriented programs, backward dynamic slicing algorithms have been proposed [20, 123], yet they were either not implemented and evaluated [137] or applied to small experimental programs only [20, 123]. These algorithms might be adapted to their corresponding forward version such that impact analysis could be built on them, yet it is still unknown whether these slicing algorithms can work with and scale up to real-world large distributed systems. Also, the statement-level dynamic slicing would be too heavyweight for impact analysis which is commonly adopted at method level.

Yet, it remains challenging to develop an effective and efficient impact analysis for distributed systems. One major difficulty lies in the lack of explicit invocations or references among the components in most distributed programs as a result of inter-component decoupling [72, 92, 184], while traditional impact-analysis approaches usually rely on those

explicit information for computing dependencies among program entities. Lately, various dependency-analysis techniques other than slicing have been proposed [67, 113, 142, 184]. While efficient for impact analysis, these approaches either are constrained to specific application domains such as distributed *event-based* systems (DEBS) [125] or rely on specialized language extensions such as EventJava [62]. Other approaches are potentially applicable in a wider scope, yet they depend on additional information such as particular formats of execution logs [113] or suffer from overly-coarse granularity (class-level) [67, 113, 142] and/or incompleteness [126], in addition to imprecision, of their analysis results.

This chapter presents DistEA, a technique that offers automatic supports for dynamic impact analysis for commonly deployed distributed systems where components have no explicit invocations or references to each other but communicate through network I/Os. By exploiting the partial ordering of, hence the happens-before relations [106] among, distributed method-execution events, DistEA computes potential impacts of one method in a distributed system on other methods of the system both within and across its components and concurrent processes. Utilizing the execution-after-sequence (EAS) [12], DistEA offers results that is safe relative to the set of executions utilized, with high efficiency while relying on neither well-defined inter-component interfaces nor message-typing specifications as needed by existing options for DEBS. By design, DistEA works with centralized programs running in single processes as well, either concurrently or sequentially.

DistEA has been applied successfully to four distributed Java programs, including two enterprise systems, which has shown that the tool is capable of working with large distributed systems of blocking, non-blocking, or mixed, network communications. We also evaluated our technique on these subjects with respect of its effectiveness and performance. Our empirical study demonstrated that DistEA, as a dynamic impact analysis technique, can greatly reduce developers' efforts in evolving distributed systems with highly promising scalability: overall, DistEA reports only x% of impacts that would have to be all inspected with existing options, and the average analysis time of DistEA is only a few seconds.

We also report more detailed results that shed light on the general runtime behaviour and inter-component logic coupling in distributed systems. These results suggest that DistEA can have various other applications beyond impact analysis. For instance, the partial ordering of distributed events realized by DistEA can potentially be utilized to support software-engineering tasks such as code comprehension and behaviour modeling of distributed systems. Also, while our current implementation of DistEA addresses Java programs, the technical approach itself can be readily adopted for other programming languages.

8.2 Approach

To realize efficient dynamic impact analysis for distributed programs at method level, we build DistEA based on the order of method executions. To that end, DistEA leverages lightweight instrumentations to capture those method-execution events and partially order them across all processes of the program, and infers execute-after relations among methods according to the partially-ordered events to compute potential impacts of any query.

This section first presents the essential concepts underlying our approach, including the definition of method events that DistEA deals with and the rationale of impact prediction from sequences of those events. Next, it gives an overview and illustration on how DistEA works as a whole. Finally, it describes details of the analysis algorithms of DistEA.

8.2.1 Essential Concepts

8.2.1.1 Method-Execution Events

In the general context of distributed systems, an event is defined as any happening of interest observable from within a computer [106]. In DEBS [125], a particular type of distributed systems, however, events are usually expressed as messages, defined by their constituent attributes, that are transferred among components of the system (e.g., [67]). While working for a broad scope of multiprocess programs with respect to both definitions

of events, DistEA neither makes any assumption nor reasons about the structure or content of messages flowing among distributed systems components.

Concretely, for dynamic impact analysis, DistEA monitors and utilizes the following two major classes of events: communication events and internal events, as informally defined as follows. As we focus on analyzing multiprocess programs, we distinguish components in a distributed system as such that each runs in a separate process.

- *Communication Event.* A communication event E_C represents an action of data exchange between two components $c1$ and $c2$, denoted as $E_C(c1,c2)$ if $c1$ initiates E_C, which attempts to reach $c2$. We further distinguish only two main categories of such events: *sending a message* and *receiving a message*, according to the underlying message flow direction.
- *Internal Event.* An internal event E_I represents an action in a component c, denoted as $E_I(c)$ if E_I happens in c. Further, we differentiate three categories of internal events at method level: *entering a method*, *returning from a method*, and *returning into a method*, denoted as m_e, m_x, and m_i, respectively, for the method m.

Note that, for internal events, we capture both the return (m_x) and returned-into (m_i) events for each method m. Yet, we distinguish them during static analysis only, and treat them equally during runtime monitoring. The reason is to deal with interleaving method-execution sequences in the case of multithreaded executions, as discussed in detail in [12].

8.2.1.2 Impact Inference

As mentioned earlier, one major challenge to developing DistEA is to infer the execute-after relation in the presence of asynchronous events in multiprocess concurrent executions. Fortunately, in the absence of synchronized physical timing for all processes, maintaining the logic notion of time to discover the causality relations among method events suffices for the inference of execution-after order of associated methods that DistEA needs. In essence, the happens-before relations among method-execution events semantically correspond to the execute-after relations among those methods. And such happens-before relations can be revealed from the partial ordering of the distributed method events [106].

Specifically, given two executed methods $m1$ and $m2$, we map execute-after (EA) relation between them to the happens-before relation between their internal events, as follows:

$$EA(m1, m2) \Longleftrightarrow m1_e \longrightarrow m2_x \bigvee m1_e \longrightarrow m2_i \tag{8.1}$$

where the notion "$\longrightarrow$" denotes the happens-before relation that signals causal relationship. Without loss of generality, $m1_e \longrightarrow m2_x$ (or $m1_e \prec m2_x$), $m1_e \longrightarrow m2_i$ (or $m1_e \prec m2_i$), and EA($m1$,$m2$) all equally imply that "$m2$ executes after $m1$, thus $m2$ may be affected by $m1$ or any change to occur in $m1$.", hence the equivalence mapping between the execute-after relation and (internal) method-event partial ordering above.

Based on this equivalence mapping, for a given query c, finding the set of impacted methods $IS(c)$ is reduced to finding methods from multiprocess method-execution traces that satisfy the partial ordering with c with respect to their relevant internal events:

$$IS(q) \ = \ \{m \mid q_e \prec m_i \vee q_e \prec m_x\} \tag{8.2}$$

Note that only internal events are directly used for impact-relation inference, while communication events are utilized to maintain the partial ordering among all those internal events during multiprocess executions of the distributed system under analysis.

8.2.2 DistEA Overview

8.2.2.1 Process

The overall process flow of DistEA is depicted in Figure 8.1. DistEA expects three major inputs: the system D under analysis, a set I of program inputs for D, and the query set M. Optionally, a message-passing API list L can also be specified to help DistEA identify program locations where probes for communication events should be instrumented (see Section 8.3.1 for more details). The output of DistEA is a set of potential impacts of M, computed from the given inputs with the following steps (as numbered in this figure).

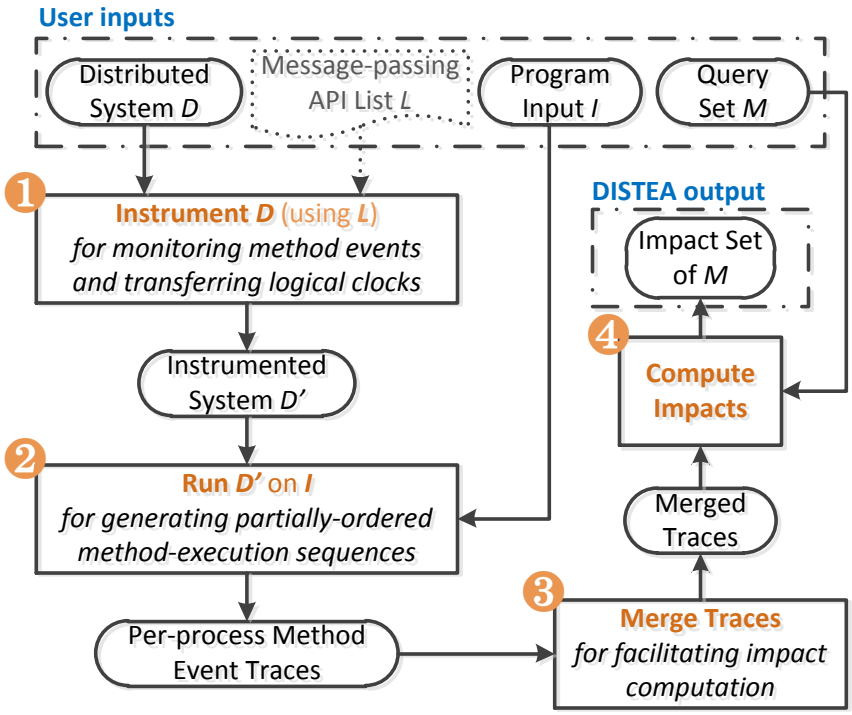

Figure 8.1: Overall process flow of DistEA, where the numbered steps are described in Section 8.2.2.1.

The *first step* shows the static analysis in DistEA, where the input program D is instrumented for both monitoring method-execution events and exchanging logic clocks among concurrent processes. This step results in the instrumented code D' of D, which executes on the given input set I in the *second step*. The resulting method-execution events are time-stamped with respect to the logic clocks exchanged among all processes during the runtime. The timestamps attached to the method events implicitly preserve the partial ordering of those events. Next, in the *third step*, traces produced from different processes are merged in order to facilitate impact computation. As the processes are supposed to run on separated machines, DistEA collects these traces to one machine before merging them. Finally, in the *fourth step*, DistEA takes the query set M and computes the impact set of M from the merged traces.

```
1   public class C {
2       Socket sock = null;
3       public C(String host, int port) {
4           sock = new Socket(host, port);}
5       void shuffle(String s) {...}
6       char compute(String s) {
7           shuffle(s);
8           sock.writeChars(s);
9           return sock.readChar(); }
10      static int main(String[]){
11          C c = new C('localhost',2345);
12          char r = c.compute(a);
13          System.out.println(r);
14          return 0;
15  }}
```

```
16  public class S {
17      Socket sock = null;
18      public S(int port) {
19          sock = new Socket(port);
20          sock.accept(); }
21      char getMax(String s) {...}
22      void serve() {
23          String s = sock.readLine();
24          char r = getMax(s);
25          sock.writeChar(r); }
26      static int main(String[]) {
27          S s = new S(2345);
28          s.serve();
29          System.out.println('Done.');
30          return 0;
31  }}
```

Figure 8.2: A distributed program $E4$ including two components: C (client) and S (server).

8.2.2.2 Illustration

To illustrate the above process flow, consider the example program $E4$ of Figure 8.2. DistEA first instruments E and produces instrumented code for both components. Next, suppose the instrumented server and client components S' and C' are deployed on two distributed hosts A and B, respectively, and S' starts first before an user launches C'. While running concurrently, S' and C' generates two method-event sequences in two separate processes, as shown *in full* in the first and last two columns in Table 8.1, respectively. As is shown, logic clocks are updated upon communication events. For instance, the logical clock of the server process is first updated upon the event E_c(C,S) to 10, which is greater by 1 than the current logical clock of the client process.

Now suppose the query set M={`S::getMax`}. DistEA merges event traces produced by the two processes and, by inferring impact relations from the timestamped events, it gives {`S::getMax`, `S::serve`, `S::main`, `C::compute`, `C::main`} as the impact set of M. As is demonstrated, DistEA can predict impacts across distributed components (processes) of the input system. For instance, when the developer makes a code change

TABLE 8.1

EXAMPLE METHOD-EXECUTION EVENT TRACES OF PROGRAM $E4$

Server process		**Client process**	
Method Event	Timestamp	Method Event	Timestamp
`S::main`$_e$	0	`C::main`$_e$	0
`S::init`$_e$	1	`C::init`$_e$	1
`S::init`$_i$	2	`C::init`$_i$	2
`S::init`$_x$	3	`C::init`$_x$	3
`S::main`$_i$	4	`C::main`$_i$	4
`S::serve`$_e$	5	`C::compute`$_e$	5
E_c(C,S)	-	`C::shuffle`$_e$	6
`S::getMax`$_e$	10	`C::shuffle`$_i$	7
`S::getMax`$_i$	11	`C::shuffle`$_x$	8
`S::getMax`$_x$	12	`C::compute`$_i$	9
`S::serve`$_i$	13	E_c(C,S)	-
E_c(S,C)	-	E_c(S,C)	-
`S::serve`$_x$	14	`C::compute`$_x$	14
`S::main`$_i$	15	`C::main`$_i$	15
`S::main`$_x$	16	`C::main`$_x$	16

in the method `getMax` in the server, the methods `compute` and `main` in the client are potentially affected thus need inspections by the developer before applying that change.

8.2.3 Analysis Algorithms

8.2.3.1 Partial Ordering of Internal Events

At the core of DistEA is the maintenance of causality among method events across distributed system components, and that causality analysis is based on the order in which

those method events occur based on a system of logical clock. It is important to note that the order is only a partial ordering in distributed systems [106]. Maintaining this order is realized through monitoring communication events in DistEA.

Two main options exist for maintaining the partial ordering on top of a logic notion of time: the Lamport timestamps [106] (LTS) and vector clocks [64, 120]. In comparison, LTS is lighter-weight as it just maintains a single counter as the logical clock for each process, while vector clock contains more information, such as process context which can be used for deriving concurrency relationships among communicating processes. Since DistEA does not use those extra information for impact inference (in its current design at least), we adopt LTS in this work.

Based on the original LTS algorithm (see Section 2.5), to observe the partial ordering DistEA needs to maintain a logical clock for each process and exchange the logic clocks among processes when communicating data (messages) among them. Algorithm 6 summarizes in pseudo code the DistEA algorithm for partially ordering internal events. The logical clock of the current process C is initialized to 1 upon process start, as is the global variable `remaining`, although initialized to 0, which tracks the length of data most recently sent by the peer *sender* process. The rest of this algorithm consists of two parts, which are trigged upon the occurrence of communication events during system executions.

The first part corresponds to function SENDMESSAGE, a runtime monitor trigged upon each message-sending event. The monitor piggybacks (prepends) two extra data item to the original message: the total length of the data to send sz, and the logical clock C, with the present value, of this *sender* process (lines 2–3); then, it sends out the packed data (line 4). The second part corresponds to the other monitor RECVMESSAGE which is trigged upon each message-receiving event. After reading the incoming message into a local buffer d (line 6), the monitor decides whether to simply update the size of remaining data to read and return (lines 7–9), or to extract two more items of data first: the new total data length to read, and the logical clock of the peer *sender* process (lines 10–16). In the latter case, the

Algorithm 6 Monitoring communication events

let C be the logical clock of the current process
remaining = 0 // remained length of data to read

```
 1: function SENDMESSAGE(msg) // on sending a message msg
 2:     sz = length of sz + length of C + length of msg
 3:     pack sz, C, and msg, in order, to d
 4:     write d
 5: function RECVMESSAGE(msg) // on receiving a message msg
 6:     read data of length l into d from msg
 7:     if remaining > 0 then
 8:         remaining -= l
 9:         return d
10:     retrieve and remove data length k from d
11:     retrieve and remove logical clock ts from d
12:     remaining = k - length of k - length of ts - l
13:     if ts > C then
14:         C = ts
15:     increment C by 1
16:     return d
```

two items are retrieved, and removed also, from the incoming message (lines 10–11), and then the remaining data length is reduced by the length of data already read in this event. Next, the local logical clock is compared to the received one and updated to the greater (lines 13–15). Lastly, the monitor returns the original message as sent (with the prepended data taken out).

For this algorithm to work without interfering the message-passing semantics of original systems, it is necessary to bookkeep the length of remaining data (via the variable `remaining`). In real-world distributed systems (e.g., Zookeeper [10] as we studied), it is common that between two communicating processes the *receiver* process may receive, through several reads, the data sent in a single write by the *sender* process. For example, a first read just retrieves data length so that appropriate size of memory can be allocated to take the actual data content in a second read. Not only is it unnecessary to attempt retrieving the prepended data items (data length and logical clock) in the second read since the first one should have already extracted them, but also such attempts can interfere the original network I/O operations.

8.2.3.2 Monitoring Internal Events

Impact inference in DistEA relies on the order of internal events of all executed methods, which can be easily retrieved based on the timestamps attached to those events. As proved in [12], for each method, tracking only its *first-entrance* and *last returned-into* events is sufficient to capture the essential information for execute-after relation inference and the dynamic impact analysis in EAS. Thus, instead of recording all those timestamps (as shown in Table 8.1), it suffices to bookkeep just two key timestamps for each method m in DistEA: the one at which the first m_e event happened, and the one at which the last m_i or m_x event happened, whichever occurred later, just as done by EAS when considering multithreaded executions [12].

In the online algorithm for monitoring internal events, we utilize two counters per method as EAS did to record the two key timestamps *for each process*. However, we use the per-process logical clock, instead of a global integer counter used by EAS, to update the per-method counters during runtime. On the other hand, the logical clock itself in each process is maintained as follows:

- It is initialized to 0 upon the start of current process.
- It is increased by 1 upon each internal event.
- It is updated upon each communication event in the way as shown in Algorithm 6.

8.2.3.3 Trace Processing and Impact Computation

DistEA computes impacts based on execute-after relations between the queried method and each candidate method possibly impacted. These relations are identified from traces of internal events, which are produced by all components of the system in concurrent processes at runtime, and generally on spatially distributed machines. However, impact computation needs to refer to the traces from all the separated processes. One solution would be to transfer *in runtime* (by additional instrumentation) the per-process traces to a central

process in which the impact-computation algorithm will be executed, yet it could incur too much extra communication traffics which can affect the original system performance [22].

In DistEA, we adopt a simpler approach with which each component (process) produces its own execution traces locally, and then the traces are transported *offline* to a central machine where per-process traces are merged to *global traces* (traces of the entire system execution). From the global traces, impacts can be readily computed with respect to any given query by searching methods that have the execute-after relation with that query according to Equation 8.1 and then computing the final impact set according to Equation 8.2.

8.3 Evaluation

To evaluate DistEA, we utilized its implementation as discussed above, and attempted to seek answers to the following three research questions:

RQ1 How effective is DistEA in predicting impacts relative to existing alternative options?

RQ2 How does impacts within processes compare with impacts across process boundaries?

RQ3 How efficient is DistEA in terms of all costs it incurs?

Our main goal with this empirical study was to investigate the effectiveness (RQ1) and efficiency (RQ3) of DistEA. We also intended to examine the distribution of impacts given by DistEA with respect to how they propagate within and across multiple components (processes) in distributed systems (RQ2).

8.3.1 Implementation

The implementation of DistEA consists of three main parts: the static analyzer, runtime monitors, and post-processor. As our tool is currently implemented for Java programs, we limit the following discussion to the context of the Java language. The entire DistEA tool, including its library dependencies, is available online for download.

8.3.1.1 Static Analyzer

The primary role of the static analyzer is to identify proper locations in the input program and instrument there such that appropriate runtime monitors are to be triggered upon relevant method-execution events. Finding an accurate set of those locations for the instrumentation is crucial to the soundness and precision of our technique. In this static analysis, we utilized the Soot byte-code analysis framework [185] for both looking for instrumentation points and actually achieving the byte-code-level instrumentation.

The instrumentation consists of two major steps. First, DistEA inserts in each method probes for the three types of internal events as EAS did [12]. In addition, we continue to adopt the improvement for capturing return and returned-into events the original EAS algorithm would miss in the case of unhandled exceptions, as detailed in Chapter 3. Moreover, we considered calls to `System.exit` as an additional case of program return. The second step is to insert probes for communication events. Among other user inputs, the message-passing API list L is expected to give the full prototype of each unique API used in the input system for network-based I/O operations. This list facilitates the identification of instrumentation locations based on string matching, but is optional. when not provided by users, a list including basic Java network I/O APIs will be used by default, which currently covers two common cases (blocking and non-blocking network I/Os), as well as our four experimental subject programs: Java socket I/O [133] and Java NIO [132].

8.3.1.2 Runtime Monitors

Corresponding to the probes inserted in the two steps of static analysis, two classes of runtime monitors are implemented. The first class is those for monitoring and timestamping internal events, which implement the algorithm described in Section 8.2.3.2. In addition, since some programs (e.g., service daemons) terminate only upon external kills, we hook in the monitor for program termination to ensure the output of method event sequences. The second class implements Algorithm 6 for monitoring communication events and exchang-

ing logic clocks among processes. Specifically, for the second class, instead of transferring logic clocks via separate calls to the monitors in addition to the original network I/O API calls, the probes actually *replace* those original calls, and the corresponding monitors take over the transfer of original messages so that they can manipulate directly on the message (or data buffers) to piggyback the extra data items (data length and logical clock).

In short, the extra data items are carried on by the original messages. Our experience suggested that the piggybacking strategy is more viable than inserting additional calls to network I/O APIs, especially in dealing with selector-based, non-blocking communications [16]. We applied the ShiVector tool [1], which uses additional I/O calls for vector-clock transferring, to our subjects, and with two of them (NioEcho and ZooKeeper) it did not work properly because of the interference caused by those additional calls. One reason as we verified with those subjects is that, for a pair of an original and the corresponding additional call, the two messages may not be read in the same order by the *receiver* process as in which they are sent by the *sender* process. As a result, an original message-receiving call may encounter extraneous data in the message hence the interference.

8.3.1.3 Post-processor

The post-processer actually answers impact-set queries. To that end, it collects distributed traces through a helper script which passes the traces to the impact-computation algorithm. The algorithm itself is implemented in Java, as are all other parts of DistEA. Currently, event traces are serialized after compression upon the process termination. Thus, the post-processor decompresses per-process traces before merging them, and then computes the impact set.

8.3.2 Experiment Setup

We evaluated DistEA on four distributed Java programs, as summarized in Table 8.2. The size of each subject is measured by the number of non-comment non-blank source

TABLE 8.2

EXPERIMENTAL SUBJECTS FOR DistEA EVALUATION

Subject	**#SLOC**	**#Methods**	**Inputs (size and type)**	**#Queries**
MultiChat	470	37	1 integration test	25
NioEcho	412	27	1 integration test	26
ZooKeeper	62,450	4,813	1 integration test	749
			1 system test	817
			1 load test	798
			195 unit tests	2,780
Voldemort	163,601	17,843	1 integration test	2,048
			1 system test	1,056
			1 load test	1,323
			9 unit tests	3,421

lines of code (*#SLOC*), and that of methods defined in the subject, both in Java code we actually analyzed. The last two columns list the input sets we used in our dynamic analysis, including the type and size of each set, and the number of methods (*#Queries*) covered in that set that we all used as impact-set queries.

MultiChat [70] is a Java chat application where multiple clients exchange messages through a central server broadcasting a message sent by one client to all others, using Socket I/Os (blocking, stream-oriented) only. NioEcho [170] is a network program using Java NIO only (non-blocking, buffer-oriented I/Os), where the server provides a simple echo service by which a client just gets the same message back that it sent to the server. ZooKeeper [10, 86] is a high-performance coordination service for distributed systems to easily archive consistency and synchronization. Voldemort [9] is a distributed key-value storage system supporting data replication and partitioning. It is an open-source clone of

Amazon's Dynamo system [54] and now used at companies such as LinkedIn. For all these programs, we checked out from their official online repositories for the latest version by the time of this chapter.

We chose these subjects and their input sets such that a variety of program sizes, types (application domains), and uses of both blocking and non-blocking network I/Os, are all considered; we chose input sets to cover different types of inputs when possible, including system test, integration test, load test, and unit test. Except for the integration tests, other types of inputs come with the subjects as integral parts. For unit tests, we chose only those leading to multiprocess executions from original full sets.

For each subject, we created the single integration test in which we started one server process and one client process on two separated machines and manually performed client operations that cover basic server functionalities. The four subjects can all work in a client/server mode, which enabled us to created these integration tests as such. Specifically, for MultiChat and NioEcho, the client requests were sending random text messages; for ZooKeeper, the client operations were, in order: creating a node, looking up for it, checking its attributes, changing its data association, and deleting it; for Voldemort, the client operations were, also in order: adding a key-value pair, querying the key for its value, deleting the key, and retrieving the pair again.

8.3.3 Experimental Methodology

To the best of our knowledge, there is no other dynamic impact analysis techniques for distributed systems addressed by DistEA. We could not compare to slicing techniques either as we are not aware of the existence of any such slicers readily available that are fully designed and implemented for distributed programs[1] , while designing and implementing one would require considerable effort.

[1]The latest such techniques [20, 123] are limited to a subset of (Java) language constructs and not evaluated against large real-world distributed systems.

Therefore, we consider as existing alternatives to DistEA two options: ignoring impacts outside the process in which the query first executes (referred to as *local process*, versus which all others are *remote processes*); taking all methods executed outside local process as impacted. In most cases, the latter (method-level coverage, referred to as *MCov*) is a safer and more viable option. For brevity, we refer to the set of potentially impacted methods executed in local process as *local impact set*, versus those in remote process as *remote impact set*; correspondingly, the sets of all methods in local process and remote processes are referred to as *local covered set* and *remote covered set*, respectively. We regard these covered sets as impact sets given by *MCov* as a baseline technique.

We measure the effectiveness of DistEA by comparing impacts it predicted to those given by *MCov* in terms of overall impact-set size ratios. Further, we also examine the composition of impact sets concerning its three subsets: *local impact set*, *remote impact set*, *common impact set* (intersection of the previous two sets). Accordingly, we also measure effectiveness of DistEA with respect to these subsets relative to correspondingly *MCov* results. When computing the impact set for a query, for each type of inputs, we take the union of per-input impact sets of all inputs of that type (e.g., impact sets of all unit tests are unionized for ZooKeeper and Voldemort) as each type of inputs attempts to represent a different operational profile of the system. Finally, beside the impact-set querying time, we report the static-analysis and runtime costs of DistEA, and storage costs of event traces, together as efficiency metrics. All machines used in our experiments are Linux workstations with an Intel i5-2400 3.10GHz processor and 8GB DDR2 RAM.

8.3.4 Results and Analysis

8.3.4.1 RQ1: Effectiveness

Figure 8.3 summarizes the major effectiveness results, where one single plot depicts the distribution of all data points for each subject and input type shown as the plot title. Each plot includes three box plots showing that distribution for the three data categories listed on

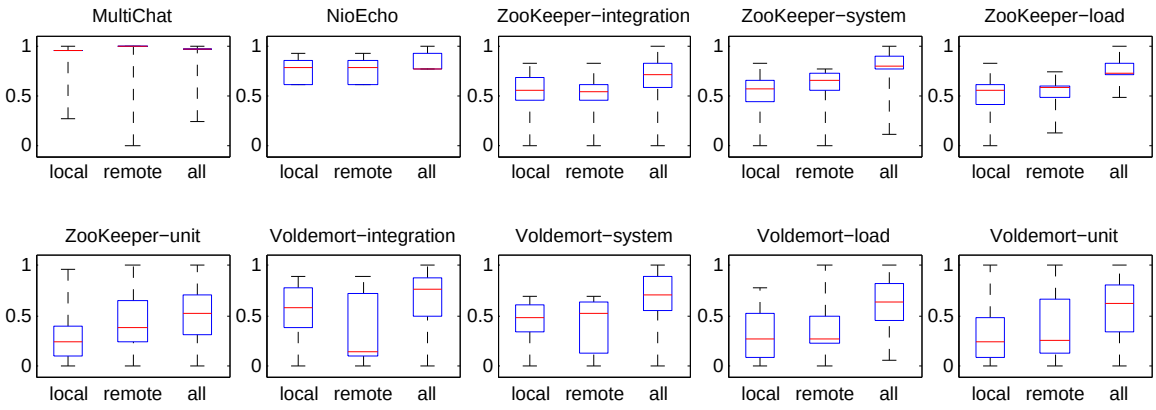

Figure 8.3: Effectiveness of DistEA expressed as the ratios (y axes) of its per-query impact-set sizes, including those of the local and remote subsets (x axes), versus *MCov* as the baseline, for each subject and input-set type (atop each plot as the title).

the x axis: size ratios of *local*, *remote*, and holistic (*all*) impact sets to corresponding covered sets, with each data point expressing the effectiveness metric, as the y axis indicates, for a single query. Thus, the number of data points of each plot is that of the queries (in Table 8.2) exercised on the corresponding subject and input set for obtaining those data.

The results show that DistEA was not quite more effective than *MCov* for the two small subjects, MultiChat in particular, as it predicted most (over 85%) of the covered methods to be impacted for the majority of queries. However, for the two large subjects, the impact sets produced by DistEA were much smaller than those given by the baseline approach, with a reduction by over 30% in impact-set sizes in most cases. One possible reason for the difference between these two groups is that the few methods all devoted to a simple task (given by the input) in the small subjects tend to have more methods executed after any other methods. Yet, for all subjects, the effectiveness of DistEA appeared to be higher with respect to holistic impact sets than to the two subsets. This was expected because these two subsets, from both DistEA and *MCov*, have intersections of considerable sizes (as discussed later) while those of DistEA were generally smaller than corresponding *MCov* sets.

In all, despite of the almost constant existence of bad cases, where the maximal ratios were close to 1, DistEA greatly reduced the result sizes of *MCov*, to about 65%, in most

cases. Also, it is noteworthy that, besides reporting mostly below 60% of covered methods as impacts in local processes, which is essentially the effectiveness of EAS, DistEA also predicted impacts in remote processes with a close overall average effectiveness, of which EAS and other existing peer techniques would be incapable. Finally, supplementary to the statistics (medians and three quartiles) represented in the box plots, the mean effectiveness is recapped in Table 8.3, which reaffirmed the foregoing observations.

8.3.4.2 RQ2: Impact-Set Composition

To highlight the needs for analyzing remote impacts, Figure 8.4 plots the impact-set composition of each individual query numbered on the y axis, where the x axis indicates the percentage of the three complementary subsets (local, remote, and common) in the holistic impact set, for each subject and input set. Thus, different from those in Figure 8.3 and other references in this chapter, where the local and remote impact sets may overlap, here the two subsets do not since the common impacts between them were taken away and put into the common sets in order to better clarify the composition.

Since its two components share no methods as we verified, MultiChat has not seen any common impacts. Similarly in NioEcho, only a couple of methods were invoked in its two processes, resulting in relatively small common impact sets in contrast to the two large subjects. One prominent observation is that for all other eight cases, remote impacts always dominated corresponding holistic impact sets. For one thing, this could partially explain the higher size ratios of remote impact sets to remote covered sets than those of local ones as seen in Figure 8.3. On the other hand, the contrast of remote to local impact sets suggests that impacts do not just propagate to methods within local processes but to far more beyond (to remote components).

In additional, in almost all cases (queries), there were methods executed after the query both in local and remote processes. This finding implies that in large distributed systems, components often share quite a few methods for common functionalities. In this sense, the

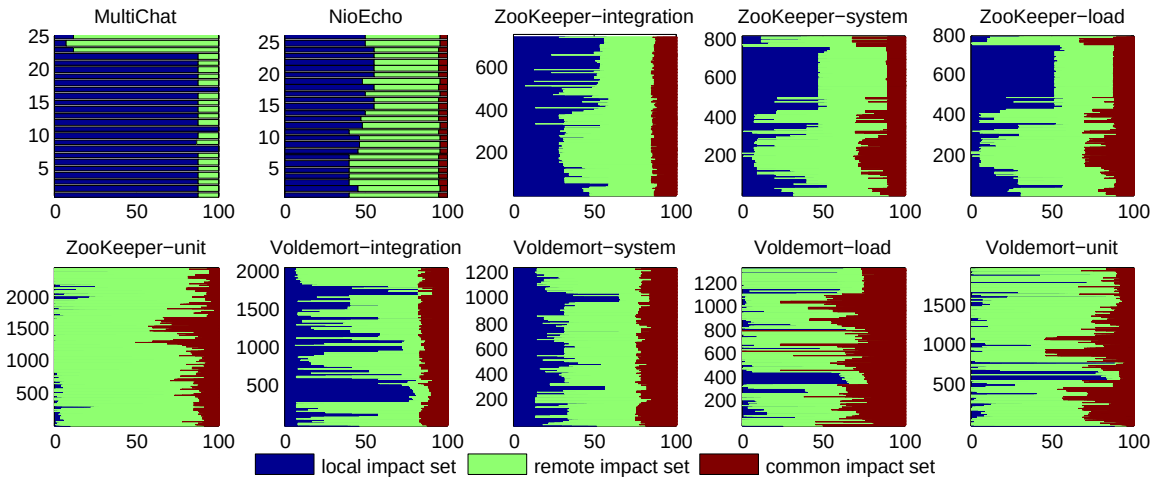

Figure 8.4: Composition of impact sets given by DistEA for all queries (y axes) expressed as the percentages (x axes) of local, remote, and common impact sets in the whole impact set per query, for each subject and input-set type (atop each plot).

sizes of common impact sets can actually serve as an indicator of functional overlapping, message coupling (coupling via message passing) in particular, among components of distributed systems. For instance, according to Figure 8.4, Voldemort seems to have stronger message coupling than ZooKeeper, for the studied input sets at least.

8.3.4.3 RQ3: Efficiency

Table 8.3 (from the fourth column) lists all relevant costs that DistEA incurred for producing the impact sets analyzed above, including the execution time of the static analyzer (*Static analysis*), overhead of the runtime monitors (*Runtime overhead*) measured as ratios of the runtime of original code (*Nr. run*) over that of instrumented code (*Inst. run*), and impact-computation time of the post-processor (*Querying*).

Individually, the static analyzer took more time for larger subjects as expected, yet within 2.2 minutes even on the largest system Voldemort. Note that this is a one-time cost (for a single program version) as the instrumented code can be executed on any inputs and used for computing any queries afterwards. Runtime and querying costs are constantly

TABLE 8.3

EFFECTIVENESS (MEANS) AND TIME-COST BREAKDOWN OF DistEA

Subject & input	**Mean effectiveness**			**Time costs (ms)**				
	Local	Remote	All	Static analysis	Nr. run	Inst. run	Runtime overhead	Querying (with stdev)
MultiChat	85.33%	85.03%	86.08%	12,817	5,461	5,738	5.07%	3 (2)
NioEcho	77.07%	76.42%	83.73%	13,365	3,213	3,623	12.76%	3 (3)
ZooKeeper-integration	55.85%	53.83%	70.73%		37,239	38,416	3.16%	10 (3)
ZooKeeper-system	53.54%	62.22%	81.17%		15,385	18,578	20.75%	16 (8)
ZooKeeper-load	50.14%	55.07%	76.17%	39,124	94,187	98,930	5.04%	15 (7)
ZooKeeper-unit	28.56%	43.26%	51.44%		1,109,146	1,370,143	23.53%	14,619 (87,056)
Voldemort-integration	55.86%	40.12%	69.72%		17,755	18,697	5.31%	27 (9)
Voldemort-system	45.71%	42.36%	67.12%		11,136	12,253	10.03%	19 (7)
Voldemort-load	30.90%	37.50%	61.68%	132,536	21,066	21,253	0.89%	122 (190)
Voldemort-unit	31.77%	37.55%	57.50%		132,676	167,861	26.52%	403 (835)
Overall average	41.42%	43.99%	63.88%	49,460.5	144,726.4	175,549.2	11.31%	3,228.9 (40,752.1)

correlated to subject sizes as well as the sizes of input sets, with the worst case seen by ZooKeeper on its unit-test input set, which is by far the largest among all subjects and input sets studied. Nevertheless, the runtime overhead is at worst 26% and longest querying time is no more than 15 seconds.

Storage costs are also tightly connected to the number of inputs in addition to subject sizes, of which the largest is 188MB total for the 195 event traces of ZooKeeper. In other cases, however, this cost is no more than 7MB; and the overall average is below 20MB.

Overall, the results suggest that DistEA is highly efficient in both time and space dimensions, and that it seems to be readily scalable up to large systems. As shown in the bottom row, it costs on average less than one minute for static analysis and a couple seconds for computing the impact set per query, with the mean runtime overhead of about 10%.

8.3.5 Threats to Validity

The main threat to *internal* validity lies in possible implementation errors in DistEA and our experimentation scripts. To reduce this threat, we did a careful code review for our tools and used the two small subject programs with the inputs to manually validate their functionalities and analysis results. An additional such threat concerns possible missing (remote) impacts due to network I/Os that were not monitored at runtime. However, we checked the code of all subjects and confirmed that they only used the most common message-passing APIs monitored by our tool with respect to the input sets we used.

The main threat to *external* validity is that our study results may not generalize to other distributed programs and input sets. In this study, we considered only limited number of subjects, which may not represent all real-world programs, and only subsets of inputs, which do not necessarily represent all behaviours of the studied programs. To reduce this threat, we chose programs of various sizes and functionality types, including the two industry-scale distributed systems from different application domains. In addition, we considered different types of inputs, including integration, system, load, and unit tests. Most of these tests came as part of the subjects except for the integration tests, which we manually created as per their system-level functionalities documented on their official websites.

The main threat to *construct* validity is the metrics used to evaluate our technique. Without closely related peer techniques in the literature, we assumed that developers would use coverage-based approach like *MCov*, as a representative alternative to DistEA, to narrow down the search space of potential impacts in the context of distributed multiprocess executions. To mitigate this threat, we examined the composition of each impact set and analyzed the effectiveness with respect to its local and remote subsets in addition to that of the holistic impact set to help demonstrate the usefulness of DistEA.

Finally, a *conclusion* threat concerns about the data points analyzed: We applied the statistical analyses only to methods for which impact sets could be queried (i.e., methods executed at least once). Also, the present study only considered potential changes in single

methods for a query, while in practice developers may plan for changes in multiple methods at a time, which possibly leads to different study results. To minimize this threat, we adopted this strategy for all experiments and calculated experimental metrics with respect to those methods only. We also considered every such method as a query in our study. Investigating multiple-method queries, with real changes, is our future work, among others.

8.4 Case Studies

To further investigate the effectiveness and practical utility of DistEA, we conducted two case studies. In contrast to the foregoing empirical evaluation, the case studies were focused on sample subjects and inputs against only a small number of queries. The small scale of these studies makes possible certain in-depth investigation of DistEA with its results on the samples. Also, it enables sample-based assessment of our technique with respect to metrics that could not be addressed in the extensive empirical experiments.

8.4.1 Study I: Precision and Usefulness of DistEA Impact Sets

8.4.1.1 Methodology

As we discussed earlier, DistEA is not expected to be very precise. Yet, currently there is no automatic means available for us to assess exactly how imprecise it would be. Thus, our first case study attempts to examine the precision of DistEA. To that end, we randomly chose two queries from a small subject MultiChat and three from a large one Voldemort. For each program, we also randomly picked an input; next, we run the original system on that input and manually determine the ground-truth set of *potential* impacts of the chosen queries. We then examine the precision of DistEA for those queries relative to the manual ground truth, which was the set of methods that would be impacted by any change in the query based on our understanding of the system's runtime behaviour relative to the input. Since manual inspection, with the two large and complex systems in particular,

TABLE 8.4

RESULTS FOR FIVE CASES OF USING DistEA

Subject & input	DistEA IS (precision)		Manual true IS		*MCov* IS	
	Local	Remote	Local	Remote	Local	Remote
MultiChat	1 (100%)	13 (69.2%)	1	9	3	21
MultiChat	13 (76.9%)	2 (50%)	10	1	22	3
Voldemort-system	4 (100%)	23 (56.5%)	4	13	740	809
Voldemort-system	3 (33.3%)	0 (-)	1	0	811	440
Voldemort-load	13 (46.1%)	41 (41.4%)	6	17	288	500
Overall average	6.8 (71.2%)	15.8 (51.7%)	4.7	8	373	354.6

is exhaustive, we limited our choices of queries and inputs to those for which the DistEA impact sets had no more than 50 methods.

8.4.1.2 Results

Table 8.4 lists the results for the five cases we studied, including two cases (queries) with respect to the system test for Voldemort. For each case, the table summarizes the impact-set sizes (*IS*) from DistEA, manual inspection, and the tentative baseline approach (*MCov*), all separated as the two subsets (*local* and *remote*). The numbers in parentheses are the precision per DistEA impact set relative to the manually determined true impacts (recall was constantly 100%).

In most of these cases, a considerable portion of the impact sets was methods executed after but not to be impacted by the query, as we expected. For instance, the first query in MultiChat was the `run` method of the main client thread, which executed at the end of the client process to iteratively send user input to the server. As a result, in the local process the query could only impact itself trivially; 13 methods executed after it in the server process,

of which four were false positives as they dealt with only general network connections independently of any message received from clients. For another example, the last query in Voldemort, an error-handling utility method, is defined in a common module executed by both the Voldemort master and its clients. Among six common impacts reported by DistEA, only two were possibly to be impacted by the query. The other four, with three of the seven unique impacts in the local process, never involved error handling. These methods, as 24 out of 41 methods devoted to network service maintenance only in the remote process, were falsely identified as impacted.

In all, DistEA had an overall average precision of 56.9% for the five randomly selected queries, and for remote impact sets only the number was 51.7%. These are very close to the precision of EAS obtained from an extensive study using various types of changes in [36]. In contrast, precision of local impact sets was considerably higher, not only on overall average but constantly in every single case. This may be due to the even looser coupling, via message passing only, among methods across components than within components.

Since we only studied five cases, these results are by no means conclusive. Yet, it seems to suggest that developers could expect impact sets about 50% precise from DistEA in an average case. Note that although this precision may not be sufficient in some situations, such as when reported impact sets are extremely large, DistEA is reasonably effective and useful compared to the much larger sets of covered methods (*MCov* results). For instance, checking on average 16 methods only with DistEA, instead of 354 with *MCov*, for impacts propagated to remote components implies significant reduction in impact-inspection effort for the studied cases.

8.4.2 Study II: Utility for Distributed-Program Understanding

8.4.2.1 Methodology

Dynamic analysis is an important avenue toward program comprehension relative to concrete executions, a process for which developers often spend great deal of effort [50].

However, understanding distributed system executions is challenging because of the complex interactions among concurrent component executions in such systems [121], even more so in the presence of selector-based non-blocking communications [16].

Thus, our second case study aimed to explore the utility of DistEA for program comprehension of distributed systems. In particular, we intended to see if DistEA can help understand interfaces of the distributed components and interprocess communications (IPC) among them. For this study, we chose NioEcho and ZooKeeper, both of which utilize selector-based network I/Os (Java NIO [16]) for IPC. For each of these two subjects, without prior knowledge about internals of either, we first picked a few important-looking queries (simply based on names) from each component (process), and then executed the instrumented program on its integration test as used in the empirical study. Next, we used the DistEA impact sets of the selected queries to learn about its runtime IPC semantics.

8.4.2.2 Results

For NioEcho, despite of its small source size and simple high-level functionality, the non-blocking IPCs between the server and client made it much harder than expected to fully understand the program, interactions between the two components in particular, by just reading the code. To use our tool, we picked two queries from the client and three from the server that all *looked* closely relevant to messaging. It turned out that using DistEA was quite instrumental in this case: the local and remote impacts, when listed together in the ascending order of associated timestamps, clearly show how the client initiates a response handler for a message before sending the message out and, before it gets to wait for response in the handler by checking a selector, the server already received the message and started its echo service, after which the client reads the server response.

In the case of ZooKeeper, given the large size and complexity of the entire system, we targeted only the particular IPCs for one fundamental operation `getData` [86]. We started with the entry methods of both the server and client modules, and then by searching

in their impact sets we located one most relevant-looking query from the client. Next, examining local and remote impacts of that query revealed the major transaction steps: the client first prepared a data request using an external library and then forwards the request to a client thread which spawned another child thread to actually connect to the server; next, when the client proceeded with some bookkeeping routines while waiting for response, the execute-after sequence in the remote process shows that the server has accepted the request and initiated a thread to access the database, then another thread to send the retrieved data back, followed by client's taking the response and processing the data.

In both cases, we also found the ordering of impacts (by their timestamps) fairly useful for following component-level interactions step by step, and that the common impacts, which reveal code reuse across components, also facilitated the comprehension process. On the other hand, we noticed that navigating in the textual impact sets can be tedious when the results become large, for which an effective visualization [105, 121] would be helpful. One solution is to visualize the partially ordered impacts as an *interprocess* call graph, which can complement or collaborate with other distributed-program comprehension approaches such as space-time diagrams and communicating finite state machines [22].

In sum, although the outcome from this exploratory study may not generalize to other cases and systems, our experience suggests that DistEA can greatly assist with understanding distributed programs and their executions. Note that the main source of this benefit comes from the remote impacts it produces.

8.5 Discussion

The core technique of DistEA is to simply partially order distributed method-execution events in order to discover interprocess execution-after relations among methods in multiple concurrent processes. Despite of this simplicity, we have demonstrated that the technique can be an important first step in developing practical tool supports for evolving distributed programs. Yet, the conservative nature of this technique, while makes it safe mod-

ulo the concrete executions utilized, causes its imprecision, especially in remote impact sets as our case study shows. This limitation may hinder applications of DistEA in tasks where highly precise results are crucial.

The present DistEA tool may not fully work for arbitrary distributed systems in Java, because it now considers only the two common cases of message-passing APIs as default and, even with user-specified list of such APIs, the text matching by API prototypes it uses for locating instrumentation points may lead to incomplete communication-event monitoring. However, this is the limitation of current implementation, not of the technique. In addition, the instrumentation for monitoring method events and extra network traffic for exchanging logic clocks may affect the performance of original systems [22], although we expect such effect to be minor in general according to studies.

8.6 Related Work

The execute-after sequences (EAS) [12] which DistEA utilized for impact prediction was a performance optimization of its predecessor PATHIMPACT introduced by Law and Rothermel in [108]. A large body of other work on dynamic impact analysis exists as well [111], aiming at improving precision [31, 35, 81, 84], recall [116], efficiency [29, 30], and cost-effectiveness [37], over PATHIMPACT and EAS. In this work, we leveraged the order of method executions for efficient impact analysis as well as these previous approaches did. However, none of these techniques can fully work with distributed multiprocess programs, computing impacts both with single processes and across multiple asynchronous processes, as DistEA is devoted to doing.

With finest-grained code-based dependency analysis, a great amount of prior work attempted extending traditional slicing techniques to concurrent programs [69, 104, 128, 194, 196]. The majority of these concurrent-program analysis approaches, however, target either multithreaded or multiprocess but *centralized* software. For such programs, traditional data-flow and control-flow analyses can be expanded to handle additional data and control

dependencies due to shared variable accesses, synchronization, and communication between threads and/or processes (e.g., [104, 128]). While DistEA also handles multithreaded executions as EAS did, it particularly addresses the needs for analyzing multiprocess programs running on distributed machines to support their evolution and maintenance.

For systems consisting of multiple processes where interprocess communication is realized through network-based message passing, an approximation for static slicing was discussed in [104]. Various dynamic slicing algorithms have been proposed too, earlier for procedural programs only [46, 58, 72, 99, 102] and recently for object-oriented software too [20, 123, 137]. A more complete and detailed summary of slicing techniques for distributed programs can be found in [196] and [20]. While these slicing algorithms were rarely evaluated against real-world distributed systems, it is expected that scalability issues would be quickly encountered by them with large systems based on the limited empirical results presented and the heavyweight nature of their design. In contrast to those whole-program finest-grained (statement-level) analysis, DistEA aims at a highly efficient method-level dynamic impact analysis that can readily scale up to large distributed programs. A few static analysis algorithms for distributed software exist as well but focus on different types of systems, such as RMI-based Java [166] and Android [130] applications.

At coarser levels, other researchers address dependency analysis of distributed systems too but for the purpose of enhancing parallelization [144], system configuration [100], or high-level understanding of system behaviour [1, 22]. A static analysis, the LSME (lexical source model extraction) [126] also computes inter-component dependencies due to implicit invocations, but has been shown both imprecise and incomplete [67, 142]. In [67], another static analysis is proposed to infer inter-component dependencies based on messaging interface matching. In contrast, DistEA performs code-based analysis while providing more focused impacts relative to program executions than static analysis approaches.

An impact analysis dedicated to distributed systems, Helios [142] can predict impacts of potential changes to support evolution tasks for DEBS. This technique, however, re-

lies on particular message type filtering and manual annotations by developers in addition to a few other constraints. Although these limitations are largely lifted by its successor Eos [67], the analysis is static and still limited to DEBS only, as is the latest approach [184] which identifies impacts based on change-type classification yet ignores intra-component dependencies hence provides merely incomplete results. While sharing similar goals, DistEA targets a broader range of distributed systems than DEBS without relying on special source-code information (e.g., interface patterns) as these techniques do.

To facilitate high-level comprehension and modeling of distributed systems, mining execution logs [22, 113] has been explored recently. Such techniques usually utilize textual analysis and log message correlations to infer interdependencies among system components, given the availability of underlying data required such as informative logs or certain patterns in them. In DistEA, we also utilize similar information (i.e., the Lamport timestamps transferred among processes) but for code-level impact analysis by inferring the happens-before relations among method-execution events. Also, DistEA automatically generates such information it requires rather than relying on existing information in the programs under analysis (e.g., logging statements).

In terms of partially ordering distributed events, the Lamport timestamps leveraged by DistEA is closely related to the vector clocks [64, 120] used by other tools, such as ShiVector [1] for ordering distributed logs, and Poet [105] for visualizing distributed systems executions. While we could utilize vector clocks in DistEA as well, we chose using Lamport timestamps as it is of lighter weight and simpler yet well suffices in our case. In addition, ShiVector requires accesses to source code and recompilation using the AspectJ compiler. DistEA does not have such constraints as it directly works on Java bytecode.

CHAPTER 9

CONCLUSION

Successful software always evolves, not only to mature itself but also to accommodate the volatility of requirements and, accordingly, the source code of programs often undergoes constant changes. Yet, these changes also impose threats to virtually every aspect of the program, ranging from quality and reliability to security and performance. To reduce such threats, a *preventive* strategy must be taken analyzing *potential* consequences of code changes so that the developer can assess and understand them before planning and designing changes for evolving the program. To that end, a fundamental approach is to analyze program dependencies, which models the interactions among entities of the program and characterizes its behaviour. This chapter summarizes the main contributions of the dissertation, and outlines ongoing and future research directions.

9.1 Summary

This dissertation is focused on developing advanced program dependency analysis techniques that aim at superior and more flexible cost-effectiveness tradeoffs over prior approaches and, as an example application with respect to program changes, using the techniques to address impact analysis, a crucial step in software evolution. In all, these techniques are subsumed in two major categories: (1) semantic-dependency quantification for prioritizing fine-grained dependencies, and (2) method-level dependency abstraction for balancing between the effectiveness and efficiency of dependency analysis and its clients.

9.1.1 Semantic-Dependency Quantification

Semantic dependencies of a program exactly capture the interactions among entities (statements) of the program. Since accurate computation of such dependencies is an undecidable problem, they are often over-approximated through syntactic (control and data) dependencies. A traditional fine-grained form of syntactic dependency analysis is program slicing, which identifies statements that may influence (forwardly) or be influenced (backwardly) by a given statement (criterion). Unfortunately, in many practical scenarios, slicing faces difficult balancing between the cost and effectiveness of the analysis: Static slicing is generally sound and efficient but can be egregiously imprecise; dynamic slicing is relatively more precise but not as efficient and scalable, and is only sound modulo to the concrete executions utilized by the analysis. And what is more, both forms of slicing often produce results that are too large to be realistically usable.

The main problem with slicing is that it simply includes all statements possibly influenced by the criterion yet does not distinguish them by the *extent* of influence, whereas intuitively not all the statements are equally relevant. For developers to optimize their inspection efforts with respect to budget constraints, I developed SENSA [40, 42], a novel dependency quantification technique which separates statements by the *strength* (or likelihood) of their influence and *prioritizes* them according to such strengths. This ability of SENSA is underlaid by the discovery and quantification of semantic program dependencies through its two core components: sensitivity analysis and execution differencing. Given a program P, a statement location C, and an input set I, SENSA repeatedly modifies the value computed at C when running P on I. It also tracks execution histories of the forward static slice of C during all the modified runs, and then calculates the times, hence the *frequency*, each statement from the slice appeared in the execution-history differences. Such frequencies are later taken as the strength measure to rank statements in the slice.

Although SENSA only detects a subset of semantic dependencies, the subset (referred to as *dynamic semantic dependencies*) is safe modulo to I. Therefore, for the concrete

operational profiles of P with respect to I and the extended inputs (through the runtime modifications), the approach is sound. Extensive empirical studies confirmed the advantageous cost-effectiveness of SENSA over both static and dynamic slicing in largely reducing potential efforts of inspecting dynamic impacts (dependencies) of given program locations, and several case studies demonstrated its additional application and benefits in fault localization [42]. In addition, SENSA has been applied to assess the practical accuracy of forward dynamic slicing relative to the true semantic dependencies with respect to the runtime changes made by SENSA.

9.1.2 Cost-Effective Dependency Abstraction

A fine-grained (statement-level) dependency analysis such as SENSA empowers developers to examine in detail the exact code regions that might be influenced by a criterion. Yet, in application tasks such as impact analysis, dependencies are more often commonly requested at method level. While lifting up statement-level dependency set to method level is straightforward, wielding the heavyweight fine-grained analysis for coarser results is apparently not cost-optimal, even more so when dynamic analysis is demanded to obtain dependencies relevant to concrete executions only. To address this problem, one natural approach is to abstract dependencies from statement to method level, and a representative form of such approaches is method-level dependency-based impact prediction: The predicted impact set is essentially equivalent to the forward dependency set for a same criterion (albeit we referred it to *query* in the context of impact analysis).

However, the most cost-effective prior approaches of this kind are intuitively very imprecise due to their overly conservative nature: They capture control flows only to roughly over-approximate syntactic dependencies. To formulate this imprecision problem, I conducted a comprehensive study of dynamic impact analysis [36, 41] targeting two representative techniques against synthesized, manually-curated, and repository changes, all in large scales. For this study, I also developed an experimentation framework DEAM [36] that can

be more generally used in evaluating predictive method-level dynamic dependency analysis. DEAM shares the two core components with SENSA but is capable of more generic execution differencing and provides automatic supports for compiling, instrumenting, and monitoring a large codebase consisting of many program versions in online repositories.

The study reveals that prior techniques are highly imprecise, producing one false positive out of every two dependencies (impacts). Such level of imprecision implies excessive waste of inspection efforts if using those techniques, and even worse consequences may ensue as false-positive results can be not only wasteful but misleading. To overcome this issue, I developed DIVER [35], a hybrid dynamic impact analysis which in essence computes method-level forward dynamic dependencies of a given method by combining context-insensitive *statement-level* static dependencies and a single form of *method-level* dynamic data (method-execution events). The central rationale of DIVER is to *safely* prune static dependencies that are definitely not exercised by the method events so as to reduce the false positives suffered from by prior approaches which either use the method events only or attempted hybrid approach but not succeeded (did not achieve significant improvements). Extensive empirical evaluation showed that DIVER is able to drastically improve the precision of dynamic impact prediction over previous alternatives.

As several industries confirmed, developers actually need different cost-effectiveness tradeoffs for impact analysis to best fit different task scenarios. Motivation for meeting such needs drove my further exploration of the design space of cost-effective dynamic dependency abstraction: Between DIVER and (statement-level) forward dynamic slicing, there should be other useful options providing more cost-effectiveness tradeoffs. In this context, I developed an unified framework of dynamic dependency abstraction [37], where multiple combinations of static dependencies with one or more forms of dynamic data are exploited, including statement coverage and dynamic points-to data in addition to the method-execution events. Empirical studies show that this framework, from which I investigated the two representative prior analyses and three new instances this far, can offer

multiple levels of cost-effectiveness. It is also shown that statement coverage seems to have stronger effects on performance of dynamic impact prediction than dynamic points-to sets.

When the concrete set of executions required by dynamic analysis are unavailable, static approaches would be in demand, which may also be more preferable in case of strict requirements for soundness. Therefore, I developed a static dependency abstraction, also at method level, called the method dependency graph (MDG) [38] as an alternative to a recent approach of the same kind which approximates method-level dependencies based on control flows only. The main goal of MDG is to directly model program dependencies at the method level, including both data and control dependencies but focusing on only *interprocedural* ones, with intraprocedural dependencies abstracted using summaries. A set of comparative studies has shown that MDG is significantly more precise than the control-flow-based alternative with few extra overheads, thus it offers a more cost-effective option for *static* dependency abstraction and, as a direct application, method-level static impact analysis. I am also investigating benefits of this abstraction to dynamic analysis.

9.2 Future Work

In light of the novel techniques and supporting evidences presented in this dissertation, immediate future work can be undertaken along two major directions in synergy: (1) continuing to probe for fundamental approaches, including dependency analysis and beyond, to understanding and reasoning about program properties and semantics, and (2) developing practical techniques and tools based on those foundations to address real-world tasks concerning the quality, security, and reliability of evolving software.

9.2.1 Analysis of Distributed Programs

Driven by growing demands for high-performance computing, an increasing number of distributed systems are being developed and deployed. Yet, traditional dependency-analysis approaches have very limited utility for such systems due to the lack of explicit

invocations or references among loosely coupled or entirely decoupled components in distributed systems, where the components run in concurrent processes and communicate via message passing without a global clock. Lately, I started exploring the dependency discovery in distributed systems through dynamic analysis. As the first step, I have developed DistEA, a dynamic impact analysis which instruments distributed programs and synchronizes per-process logical clocks through an online timestamping algorithm to identify the partial ordering of method-execution events so as to infer method-level dynamic dependencies between methods in distributed components from the partial ordering. However, inferring dependencies purely based on method execution order limits the effectiveness of DistEA. In the next step, I plan to improve the precision of this approach by exploiting message passing semantics and incorporating static analysis. For example, one immediate potential project on top of DistEA could be building an *interprocess* dynamic call graph to facilitate understanding and debugging of distributed programs and their executions.

9.2.2 Software Security Analysis

Dependency analysis can be naturally exploited for security analysis as well, with both static and dynamic approaches. I am interested in exploring vulnerability prediction and prevention techniques with the power of program analysis, for secure software systems, online trust management, and mobile applications. Nowadays, developers need to validate their programs for not only quality assurance but also security and privacy guarantees while evolving the programs. In order to assist developers with such tasks, one potential work is to propagate security/privacy-policy violations from the origins to identify affected code regions with dependency *chaining*. In addition, tracking and measuring information flow patterns and strengths through hybrid analysis can be utilized to detect security leaks. Previously, a similar approach has been attempted using dynamic dependency analysis only. I am planning to incorporate static analysis in hybrid approaches to help detect malicious artifacts in programs with enhanced cost-effectiveness. For instance, in a hybrid scheme, the

static analysis will first compute a conservative set of suspicious locations and then insert probes for monitoring and checking executions against security constraints; the dynamic analysis part takes charge of the runtime monitoring and logging to refine the initial suspicious set to narrow down the inspection space of malicious code. For this line of work, I will utilize Tracer [39], a tracing infrastructure I developed that provides generic utilities for dynamic dependency analysis through fine-grained logging of program states.

9.2.3 Autonomic Code Evolution

More broadly, technical supports for software evolution concerning either quality or security are mostly far from fully automatic, often relying on a large amount of manual work by developers. For example, impact analysis assists with evolution tasks by identifying where to be examined, yet it leaves other laborious steps to developers for figuring out where and how to apply changes in affected code areas so that the original change can be completely fulfilled. To help automate these later steps, a systematic approach is needed to propagate the original changes, updating impacted code where necessary. Similar strategies would also be desirable for ensuring the conformance of security and privacy properties upon the realization of code changes. To make these technical aids more cost-effective, incremental analysis will be employed, identifying candidate change locations on top of previously found ones, inferring updates using historical changes, and avoiding repetitive computations by pruning program points that can be safely skipped. I intend to explore various techniques as such to help automate reliable and secure code evolution.

BIBLIOGRAPHY

1. J. Abrahamson, I. Beschastnikh, Y. Brun, and M. D. Ernst. Shedding light on distributed system executions. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 598–599, 2014.

2. R. Abreu, P. Zoeteweij, and A. J. C. V. Gemund. On the accuracy of spectrum-based fault localization. In *Proceedings of Testing: Academic and Industrial Conference - Practice and Research Techniques*, pages 89–98, 2007.

3. M. Acharya and 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*, pages 746–765, 2011.

4. H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 246–256, 1990.

5. H. Agrawal, R. A. Demillo, B. Hathaway, W. Hsu, W. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur, and E. Spafford. Design of mutant operators for the C programming language. Technical report, Software Engineering Research Centre, 1989.

6. H. Agrawal, J. R. Horgan, E. W. Krauser, and S. London. Incremental regression testing. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 348–357, 1993.

7. A. V. Aho, M. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Prentice Hall, 2006.

8. M. Ajrnal Chaumun, H. Kabaili, R. K. Keller, and F. Lustman. A change impact model for changeability assessment in object-oriented software systems. In *Proceedings of European Conference on Software Maintainance and Reengineering*, pages 130–138, 1999.

9. Apache. Voldemort. `https://github.com/voldemort/voldemort`, 2008. Last accessed: July 2015.

10. Apache. ZooKeeper. `https://zookeeper.apache.org/`, 2012. Last accessed: July 2015.

11. Apache. PDFBox. `http://svn.apache.org/repos/asf/pdfbox/`, 2014. Last accessed: July 2015.

12. T. Apiwattanapong, A. Orso, and M. J. Harrold. Efficient and precise dynamic impact analysis using execute-after sequences. In *Proceedings of IEEE/ACM International Conference on Software Engineering*, pages 432–441, 2005.

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

14. A. Arcuri and L. Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Journal of Software Testing, Verification and Reliability*, 24(3):219–250, 2014.

15. R. S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

16. C. Artho, M. Hagiya, R. Potter, Y. Tanabe, F. Weitl, and M. Yamamoto. Software model checking for distributed systems with selector-based, non-blocking communication. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering*, pages 169–179, 2013.

17. S. Artzi, J. Dolby, F. Tip, and M. Pistoia. Directed test generation for effective fault localization. In *Proceedings of ACM International Symposium on Software Testing and Analysis*, pages 49–60, 2010.

18. G. K. Baah, A. Podgurski, and M. J. Harrold. The probabilistic program dependence graph and its application to fault diagnosis. *IEEE Transactions on Software Engineering*, 36(4):528–545, 2010.

19. L. Badri, M. Badri, and D. St-Yves. Supporting predictive change impact analysis: A control call graph based technique. In *Proceedings of Asia-Pacific Software Engineering Conference*, pages 167–175, 2005.

20. S. S. Barpanda and D. P. Mohapatra. Dynamic slicing of distributed object-oriented programs. *IET software*, 5(5):425–433, 2011.

21. S. Bekrar, C. Bekrar, R. Groz, and L. Mounier. Finding software vulnerabilities by smart fuzzing. In *Proceedings of IEEE International Conference on Software Testing, Verification and Validation*, pages 427–430, 2011.

22. I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy. Inferring models of concurrent systems from logs of their behavior with CSight. In *Proceedings of IEEE/ACM International Conference on Software Engineering*, pages 468–479, 2014.

23. A. Beszedes, C. Farago, Z. Mihaly Szabo, J. Csirik, and T. Gyimothy. Union slices for program maintenance. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 12–21, 2002.

24. A. Beszédes, T. Gergely, J. Jász, G. Tóth, T. Gyimóthy, and V. Rajlich. Computation of static execute after relation with applications to software maintenance. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 295–304, 2007.

25. D. Binkley, S. Danicic, T. Gyimóthy, M. Harman, A. Kiss, and B. Korel. Theoretical foundations of dynamic program slicing. *Theoretical Computer Science*, 360(1–3): 23–41, 2006.

26. D. Binkley, N. Gold, and M. Harman. An empirical study of static program slice size. *ACM Transactions on Software Engineering and Methodology*, 16(2), 2007.

27. S. Black. Computing ripple effect for software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 13(4):263–279, 2001.

28. M. Bohme, B. Oliveira, and A. Roychoudhury. Partition-based regression verification. In *Proceedings of IEEE/ACM International Conference on Software Engineering*, pages 302–311, 2013.

29. B. Breech, A. Danalis, S. Shindo, and L. Pollock. Online impact analysis via dynamic compilation technology. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 453–457, 2004.

30. B. Breech, M. Tegtmeyer, and L. Pollock. A comparison of online and dynamic impact analysis algorithms. In *Proceedings of European Conference on Software Maintainance and Reengineering*, pages 143–152, 2005.

31. B. Breech, M. Tegtmeyer, and L. Pollock. Integrating influence mechanisms into impact analysis for increased precision. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 55–65, 2006.

32. L. C. Briand, J. Wuest, and H. Lounis. Using coupling measurement for impact analysis in object-oriented systems. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 475–482, 1999.

33. L. C. Briand, Y. Labiche, and G. Soccar. Automating impact analysis and regression test selection based on UML designs. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 252–261, 2002.

34. J. Buckner, J. Buchta, M. Petrenko, and V. Rajlich. JRipples: A tool for program comprehension during incremental change. In *Proceedings of International Workshop on Program Comprehension*, pages 149–152, 2005.

35. H. Cai and R. Santelices. DIVER: Precise dynamic impact analysis using dependence-based trace pruning. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering*, pages 343–348, 2014.

36. H. Cai and R. Santelices. A comprehensive study of the predictive accuracy of dynamic change-impact analysis. *Journal of Systems and Software*, 103:248–265, 2015.

37. H. Cai and R. Santelices. A framework for cost-effective dependence-based dynamic impact analysis. In *Proceedings of IEEE International Conference on Software Analysis, Evolution, and Reengineering*, pages 231–240, 2015.

38. H. Cai and R. Santelices. Abstracting program dependencies using the method dependence graph. In *Proceedings of IEEE International Conference on Software Quality, Reliability, and Security*, 2015.

39. H. Cai and R. Santelices. TracerJD: Generic trace-based dynamic dependence analysis with fine-grained logging. In *Proceedings of IEEE International Conference on Software Analysis, Evolution, and Reengineering*, pages 489–493, 2015.

40. H. Cai, S. Jiang, R. Santelices, Y. Zhang, and Y. Zhang. SENSA: Sensitivity analysis for quantitative change-impact prediction. In *Proceedings of IEEE Working Conference on Source Code Analysis and Manipulation*, pages 165–174, 2014.

41. H. Cai, R. Santelices, and T. Xu. Estimating the accuracy of dynamic change-impact analysis using sensitivity analysis. In *Proceedings of IEEE International Conference on Software Security and Reliability*, pages 48–57, 2014.

42. H. Cai, R. Santelices, and S. Jiang. Prioritizing change impacts via semantic dependence quantification. *IEEE Transactions on Reliability*, 2015.

43. D. Callahan. The program summary graph and flow-sensitive interprocedual data flow analysis. In *Proceedings of ACM Conference on Programming Language Design and Implementation*, pages 47–56, 1988.

44. M. Ceccarelli, L. Cerulo, G. Canfora, and M. Di Penta. An eclectic approach for change impact analysis. In *Proceedings of IEEE/ACM International Conference on Software Engineering*, pages 163–166, 2010.

45. K. Chen and V. Rajich. Ripples: Tool for change in legacy software. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 230–239, 2001.

46. J. Cheng. Dependence analysis of parallel and distributed programs and its applications. In *Proceedings of International Conference on Advances in Parallel and Distributed Computing*, pages 370–377, 1997.

47. H. Cleve and A. Zeller. Finding failure causes through automated testing. In *Proceedings of International Workshop on Automated Debugging*, pages 254–259, 2000.

48. N. Cliff. *Ordinal Methods for Behavioral Data Analysis*. Psychology Press, 1996.

49. R. Coe. It's the effect size, stupid: What effect size is and why it is important. In *the Proceedings of Annual Conference of the British Educational Research Association*. Education-line, 2002.

50. B. Cornelissen, A. Zaidman, A. Van Deursen, L. Moonen, and R. Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702, 2009.

51. A. De Lucia, F. Fasano, and R. Oliveto. Traceability management for impact analysis. In *Frontiers of Software Maintenance*, pages 21–30, 2008.

52. C. R. de Souza and D. F. Redmiles. An empirical study of software developers' management of dependencies and changes. In *Proceedings of IEEE/ACM International Conference on Software Engineering*, pages 241–250, 2008.

53. V. Debroy and W. E. Wong. A consensus-based strategy to improve the quality of fault localization. *Software: Practice and Experience*, 43(8):989–1011, 2013.

54. G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of ACM SIGOPS Symposium on Operating Systems Principles*, pages 205–220, 2007.

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

56. R. A. DeMillo, H. Pan, and E. H. Spafford. Critical slicing for software fault localization. In *Proceedings of ACM International Symposium on Software Testing and Analysis*, pages 121–134, 1996.

57. H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.

58. E. Duesterwald, R. Gupta, and M. Soffa. Distributed slicing and partial re-execution for distributed programs. In *Proceedings of International Workshop on Languages and Compilers for Parallel Computing*, pages 497–511, 1993.

59. S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2):159–182, 2002.

60. M. Emami. A practical interprocedural alias analysis for an optimizing/parallelizing C compiler. Master's thesis, McGill University, 1993.

61. M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001.

62. P. Eugster and K. Jayaram. EventJava: An extension of java for event correlation. In *Proceedings of European Conference on Object-Oriented Programming*, pages 570–594, 2009.

63. J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319-349, 1987.

64. C. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of Australian Computer Science Conference*, pages 56–66, 1988.

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

66. K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, 1991.

67. J. Garcia, D. Popescu, G. Safi, W. G. Halfond, and N. Medvidovic. Identifying message flow in distributed event-based systems. In *Proceedings of ACM International Symposium on the Foundations of Software Engineering*, pages 367–377, 2013.

68. M. Gethers, B. Dit, H. Kagdi, and D. Poshyvanyk. Integrated impact analysis for managing software changes. In *Proceedings of IEEE/ACM International Conference on Software Engineering*, pages 430–440, 2012.

69. D. Giffhorn and C. Hammer. Precise slicing of concurrent programs. *Automated Software Engineering*, 16(2):197–234, 2009.

70. GoogleCode. MultiChat. `https://code.google.com/p/multithread-chat-server/`, 2009. Last accessed: July 2015.

71. T. Goradia. Dynamic impact analysis: A cost-effective technique to enforce error-propagation. In *Proceedings of ACM International Symposium on Software Testing and Analysis*, pages 171–181, 1993.

72. D. Goswami and R. Mall. Dynamic slicing of concurrent programs. In *Proceedings of International Conference on High Performance Computing*, pages 15–26, 2000.

73. A. R. Group. Java Architecture for Bytecode Analysis. `http://www.cc.gatech.edu/aristotle/Tools/jaba.html`, 2003. Last accessed: July 2015.

74. C. Gupta, Y. Singh, and D. S. Chauhan. An efficient dynamic impact analysis using definition and usage information. *International Journal of Digital Content Technology and its Applications*, 3(4):112–115, 2009.

75. N. Gupta and Z. Heidepriem. A new structural coverage criterion for dynamic detection of program invariants. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering*, pages 49–58, 2003.

76. R. Gupta, M. Harrold, and M. Soffa. Program slicing-based regression testing techniques. *Journal of Software Testing, Verification, and Reliability*, 6(2):83–111, 1996.

77. T. Gyimóthy, A. Beszédes, and I. Forgács. An efficient relevant slicing method for debugging. In *Proceedings of ACM International Symposium on the Foundations of Software Engineering*, pages 303–321, 1999.

78. R. J. Hall. Automatic extraction of executable program subsets by simultaneous dynamic program slicing. *Automated Software Engineering*, 2(1):33–53, 1995.

79. M. Harman. Why source code analysis and manipulation will always be important. In *Proceedings of IEEE Working Conference on Source Code Analysis and Manipulation*, pages 7–19, 2010.

80. M. Harman, J. Krinke, J. Ren, and S. Yoo. Search based data sensitivity analysis applied to requirement engineering. In *Proceedings of Genetic and Evolutionary Computation Conference*, pages 1681–1688, 2009.

81. L. Hattori, D. Guerrero, J. Figueiredo, J. Brunet, and J. Damasio. On the precision and accuracy of impact analysis techniques. In *Proceedings of IEEE/ACIS International Conference on Computer and Information Science*, pages 513–518, 2008.

82. K. J. Hoffman, P. Eugster, and S. Jagannathan. Semantics-aware trace analysis. In *Proceedings of ACM Conference on Programming Language Design and Implementation*, pages 453–464, 2009.

83. S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, 1990.

84. L. Huang and Y.-T. Song. Precise dynamic impact analysis with dependency analysis for object-oriented programs. In *International Conference on Software Engineering Research, Management & Applications*, pages 374–384, 2007.

85. L. Huang and Y.-T. Song. A dynamic impact analysis approach for object-oriented programs. *Advanced Software Engineering and Its Applications*, 0:217–220, 2008.

86. P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of USENIX Annual Technical Conference*, volume 8, pages 1–14, 2010.

87. D. Jackson and D. A. Ladd. Semantic Diff: A tool for summarizing the effects of modifications. In *International Conference on Software Maintenance*, pages 243–252, 1994.

88. D. Jackson and M. Rinard. Software analysis: A roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 133–145, 2000.

89. J. Jász. Static execute after algorithms as alternatives for impact analysis. *Electrical Engineering*, 52(3-4):163–176, 2010.

90. J. Jász, Á. Beszédes, T. Gyimóthy, and V. Rajlich. Static execute after/before as a replacement of traditional software dependencies. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 137–146, 2008.

91. J. Jász, L. Schrettner, Á. Beszédes, C. Osztrogonác, and T. Gyimóthy. Impact analysis using static execute after in WebKit. In *Proceedings of European Conference on Software Maintainance and Reengineering*, pages 95–104, 2012.

92. K. Jayaram and P. Eugster. Program analysis for event-based distributed systems. In *Proceedings of ACM International Conference on Distributed Event-Based System*, pages 113–124, 2011.

93. Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.

94. S. Jiang, R. Santelices, M. Grechanik, and 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*, pages 145–154, 2014.

95. W. Jin, A. Orso, and T. Xie. Automated behavioral regression testing. In *Proceedings of IEEE International Conference on Software Testing, Verification and Validation*, pages 137–146, 2010.

96. J. A. Jones and M. J. Harrold. Empirical evaluation of the Tarantula automatic fault-localization technique. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering*, pages 273–282, Nov. 2005.

97. P. Jönsson and M. Lindvall. Impact analysis. In *Engineering and managing software requirements*, pages 117–142. Springer Berlin Heidelberg, 2005.

98. M. Kamkar. An overview and comparative classification of program slicing techniques. *Journal of Systems and Software*, 31(3):197–214, 1995.

99. M. Kamkar and P. Krajina. Dynamic slicing of distributed programs. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 222–229, 1995.

100. F. Kon and R. H. Campbell. Dependence management in component-based distributed systems. *IEEE concurrency*, 8(1):26–36, 2000.

101. B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, 1990.

102. B. Korel and R. Ferguson. Dynamic slicing of distributed programs. *Applied Mathematics and Computer Science*, 2(2):199–215, 1992.

103. B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29 (3):155–163, 1988.

104. J. Krinke. Context-sensitive slicing of concurrent programs. In *Proceedings of ACM International Symposium on the Foundations of Software Engineering*, pages 178–187, 2003.

105. T. Kunz, J. P. Black, D. J. Taylor, and T. Basten. Poet: Target-system independent visualizations of complex distributed-application executions. *The Computer Journal*, 40(8):499–512, 1997.

106. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

107. J. Law and G. Rothermel. Incremental dynamic impact analysis for evolving software systems. In *Proceedings of the 14th International Symposium on Software Reliability Engineering*, pages 430–441, 2003.

108. J. Law and G. Rothermel. Whole program path-based dynamic impact analysis. In *Proceedings of IEEE/ACM International Conference on Software Engineering*, pages 308–318, 2003.

109. S. Lehnert. A review of software change impact analysis. Technical report, Ilmenau University of Technology, 2011.

110. B. Li, X. Sun, and J. Keung. FCA–CIA: An approach of using FCA to support cross-level change impact analysis for object oriented java programs. *Information and Software Technology*, 55(8):1437–1449, 2013.

111. B. Li, X. Sun, H. Leung, and S. Zhang. A survey of code-based change impact analysis techniques. *Software Testing, Verification and Reliability*, 23:613–646, 2013.

112. L. Li and A. J. Offutt. Algorithmic analysis of the impact of changes to object-oriented software. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 171–184, 1996.

113. J.-G. Lou, Q. Fu, Y. Wang, and J. Li. Mining dependency in distributed systems through unstructured logs analysis. *ACM SIGOPS Operating Systems Review*, 44(1): 91–96, 2010.

114. J. P. Loyall and S. A. Mathisen. Using dependence analysis to support the software maintenance process. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 282–291, 1993.

115. Y.-S. Ma, J. Offutt, and Y. R. Kwon. MuJava: An automated class mutation system. *Software Testing, Verififcation and Reliability*, 15(2):97–133, 2005.

116. M. C. O. Maia, R. A. Bittencourt, J. C. A. de Figueiredo, and D. D. S. Guerrero. The hybrid technique for object-oriented software change impact analysis. In *Proceedings of European Conference on Software Maintainance and Reengineering*, pages 252–255, 2010.

117. J. I. Maletic and M. L. Collard. Supporting source code difference analysis. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 210–219, 2004.

118. W. Masri and A. Podgurski. An empirical study of the strength of information flows in programs. In *Proceedings of International Workshop on Dynamic Systems Analysis*, pages 73–80, 2006.

119. W. Masri and A. Podgurski. Measuring the strength of information flows in programs. *ACM Transactions on Software Engineering and Methodology*, 19(2):1–33, 2009.

120. F. Mattern. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms*, 1(23):215–226, 1989.

121. J. Moc and D. A. Carr. Understanding distributed systems via execution trace data. In *Proceedings of International Workshop on Program Comprehension*, pages 60–67, 2001.

122. M. Mock, D. C. Atkinson, C. Chambers, and S. J. Eggers. Program slicing with dynamic points-to sets. *IEEE Transactions on Software Engineering*, 31(8):657–678, 2005.

123. D. P. Mohapatra, R. Kumar, R. Mall, D. Kumar, and M. Bhasin. Distributed dynamic slicing of java programs. *Journal of Systems and Software*, 79(12):1661–1678, 2006.

124. F. Mosteller and R. A. Fisher. Questions and answers. *The American Statistician*, 2 (5):pp. 30–31, 1948.

125. G. Mühl, L. Fiege, and P. Pietzuch. *Distributed Event-Based Systems*. Springer, 2006.

126. G. C. Murphy and D. Notkin. Lightweight lexical source model extraction. *ACM Transactions on Software Engineering and Methodology*, 5(3):262–292, 1996.

127. E. M. Myers. A precise inter-procedural data flow algorithm. In *Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 219–230, 1981.

128. M. G. Nanda and S. Ramesh. Interprocedural slicing of multithreaded programs with applications to java. *ACM Transactions on Programming Languages and Systems*, 28 (6):1088–1144, 2006.

129. J. C. Ng. Context-sensitive control flow graph. Master's thesis, Iowa State University, 2004.

130. D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon. Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. In *Proceedings of USENIX Security Symposium*, pages 543–558, 2013.

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

132. Oracle. Java NIO. `http://docs.oracle.com/javase/7/docs/api/java/nio/package-summary.html`, 2014. Last accessed: July 2015.

133. Oracle. Java Socket I/O. `http://docs.oracle.com/javase/7/docs/api/java/net/Socket.html`, 2014. Last accessed: July 2015.

134. A. Orso, T. Apiwattanapong, and M. J. Harrold. Leveraging field data for impact analysis and regression testing. In *Proceedings of ACM International Symposium on the Foundations of Software Engineering*, pages 128–137, 2003.

135. A. Orso, T. Apiwattanapong, J. B. Law, G. Rothermel, and M. J. Harrold. An empirical comparison of dynamic impact analysis algorithms. In *Proceedings of IEEE/ACM International Conference on Software Engineering*, pages 491–500, 2004.

136. C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *Proceedings of IEEE/ACM International Conference on Software Engineering*, pages 75–84, 2007.

137. S. Pani, S. M. Satapathy, and G. Mund. Slicing of programs dynamically under distributed environment. In *Proceedings of International Conference on Advances in Computing*, pages 601–609, 2012.

138. S. Park, R. W. Vuduc, and M. J. Harrold. Falcon: fault localization in concurrent programs. In *Proceedings of IEEE/ACM International Conference on Software Engineering*, pages 245–254, 2010.

139. S. Park, B. M. M. Hossain, I. Hussain, C. Csallner, M. Grechanik, K. Taneja, C. Fu, and Q. Xie. Carfast: Achieving higher statement coverage faster. In *Proceedings of ACM International Symposium on the Foundations of Software Engineering*, pages 35:1–35:11, 2012.

140. M. Petrenko and V. Rajlich. Variable granularity for improving precision of impact analysis. In *Proceedings of International Conference on Program Comprehension*, pages 10–19, 2009.

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

142. D. Popescu, J. Garcia, K. Bierhoff, and N. Medvidovic. Impact analysis for distributed event-based systems. In *Proceedings of ACM International Conference on Distributed Event-Based Systems*, pages 241–251, 2012.

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

144. K. Psarris and K. Kyriakopoulos. An experimental evaluation of data dependence analysis techniques. *IEEE Transactions on Parallel and Distributed Systems*, 15(3): 196–213, 2004.

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

146. V. Rajlich. Changing the paradigm of software engineering. *Communications of the ACM*, 49(8):67–70, 2006.

147. V. Rajlich. *Software Engineering: The Current Practice*. Chapman and Hall/CRC, 2011.

148. V. Rajlich. Software evolution and maintenance. In *Proceedings of the Conference on the Future of Software Engineering*, pages 133–144, 2014.

149. M. K. Ramanathan, A. Grama, and S. Jagannathan. Sieve: A tool for automatically detecting variations across program versions. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering*, pages 241–252, 2006.

150. V. P. Ranganath, T. Amtoft, A. Banerjee, J. Hatcliff, and M. B. Dwyer. A new foundation for control dependence and slicing for modern program structures. *ACM Transactions on Programming Languages and Systems*, 29(5), 2007.

151. X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: A tool for change impact analysis of Java programs. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 432–448, 2004.

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

153. G. N. Rodrigues, D. S. Rosenblum, and S. Uchitel. Sensitivity analysis for a scenario-based reliability prediction model. In *Proceedings of Workshop on Architecting Dependable Systems*, pages 1–5, 2005.

154. G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, 1996.

155. G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, 2001.

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

157. N. Rungta, S. Person, and J. Branchaud. A change impact analysis to characterize evolving program behaviors. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 109–118, 2012.

158. A. Saltelli, K. Chan, and E. M. Scott. *Sentitivity Analysis*. John Wiley & Sons, 2009.

159. R. Santelices and M. J. Harrold. Efficiently monitoring data-flow test coverage. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering*, pages 343–352, 2007.

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

161. R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold. Test-suite augmentation for evolving software. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering*, pages 218–227, 2008.

162. R. Santelices, M. J. Harrold, and A. Orso. Precisely detecting runtime change interactions for evolving software. In *Proceedings of IEEE International Conference on Software Testing, Verification and Validation*, pages 429–438, 2010.

163. R. Santelices, Y. Zhang, H. Cai, and S. Jiang. DUA-Forensics: A fine-grained dependence analysis and instrumentation framework based on Soot. In *Proceedings of ACM SIGPLAN International Workshop on State Of the Art in Java Program Analysis*, pages 13–18, 2013.

164. R. Santelices, Y. Zhang, S. Jiang, H. Cai, , and Y. Zhang. Quantitative program slicing: Separating statements by relevance. In *Proceedings of IEEE/ACM International Conference on Software Engineering – New Ideas and Emerging Results track*, pages 1269–1272, 2013.

165. L. Schrettner, J. Jász, T. Gergely, Á. Beszédes, and T. Gyimóthy. Impact analysis in the presence of dependence clusters using static execute after in WebKit. *Journal of Software: Evolution and Process*, 26(6):569–588, 2013.

166. M. Sharp and A. Rountev. Static analysis of object references in RMI-based Java software. *IEEE Transactions on Software Engineering*, 32(9):664–681, 2006.

167. M. Sherriff and L. Williams. Empirical software change impact analysis using singular value decomposition. In *Proceedings of IEEE International Conference on Software Testing, Verification and Validation*, pages 268–277, 2008.

168. S. Sinha and M. J. Harrold. Analysis and testing of programs with exception handling constructs. *IEEE Transactions on Software Engineering*, 26(9):849–871, 2000.

169. S. Sinha, M. J. Harrold, and G. Rothermel. Interprocedural control dependence. *ACM Transactions on Software Engineering and Methodology*, 10(2):209–254, 2001.

170. SourceForge. NioEcho. `http://rox-xmlrpc.sourceforge.net/niotut/index.html#Thecode`, 2013. Last accessed: July 2015.

171. M. Sridharan, S. J. Fink, and R. Bodik. Thin slicing. In *Proceedings of ACM Conference on Programming Language Design and Implementation*, pages 112–122, 2007.

172. W. N. Sumner and X. Zhang. Comparative causality: Explaining the differences between executions. In *Proceedings of IEEE/ACM International Conference on Software Engineering*, pages 272–281, 2013.

173. W. N. Sumner, T. Bao, and X. Zhang. Selecting peers for execution comparison. In *Proceedings of ACM International Symposium on Software Testing and Analysis*, pages 309–319, 2011.

174. X. Sun, B. Li, C. Tao, W. Wen, and S. Zhang. Change impact analysis based on a taxonomy of change types. In *IEEE Computer Software and Applications Conference*, pages 373–382, 2010.

175. X. Sun, B. Li, C. Tao, and S. Zhang. HSM-based change impact analysis of object-oriented Java programs. *Chinese Journal of Electronics*, 20(2):247–251, 2011.

176. X. Sun, B. Li, S. Zhang, C. Tao, X. Chen, and W. Wen. Using lattice of class and method dependence for change impact analysis of object oriented programs. In *ACM Symposium on Applied Computing*, pages 1439–1444, 2011.

177. X. Sun, B. Li, B. Li, and W. Wen. A comparative study of static CIA techniques. In *Proceedings of Asia-Pacific Symposium on Internetware*, pages 23:1–23:8, 2012.

178. A. Takanen, J. D. Demott, and C. Miller. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, 2008.

179. K. Taneja, T. Xie, N. Tillmann, and J. de Halleux. eXpress: Guided Path Exploration for Efficient Regression Test Generation. In *Proceedings of ACM International Symposium on Software Testing and Analysis*, pages 1–11, 2011.

180. Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim. How do software engineers understand code changes?: an exploratory study in industry. In *Proceedings of ACM International Symposium on the Foundations of Software Engineering*, pages 51:1–51:11, 2012.

181. F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

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

183. G. Tóth, P. Hegedűs, Á. Beszédes, T. Gyimóthy, and J. Jász. Comparison of different impact analysis methods and programmer's opinion: An empirical study. In *Proceedings of International Conference on the Principles and Practice of Programming in Java*, pages 109–118, 2010.

184. S. Tragatschnig, H. Tran, and U. Zdun. Impact analysis for event-based systems using change patterns. In *Proceedings of Annual ACM Symposium on Applied Computing*, pages 763–768, 2014.

185. R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of Conference of the Centre for Advanced Studies on Collaborative Research*, pages 1–11, 1999.

186. R. Vanciu and V. Rajlich. Hidden dependencies in software systems. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 1–10, 2010.

187. G. A. Venkatesh. The semantic approach to program slicing. *SIGPLAN Notice*, 26 (6):107–119, 1991.

188. J. Voas. PIE: A dynamic failure-based technique. *IEEE Transactions on Software Engineering*, 18(8):717–727, 1992.

189. D. Vysochanskij and Y. Petunin. Justification of the three-sigma rule for unimodal distributions. *Theory of Probability and Mathematical Statistics*, 21:25–36, 1980.

190. S. Wagner. Global sensitivity analysis of predictor models in software engineering. In *Proceedings of IEEE International Workshop on Predictor Models in Software Engineering*, pages 3–10, 2007.

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

192. M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4): 352–357, 1984.

193. M. Weiser, J. D. Gannon, and P. R. McMullin. Comparison of structural test coverage metrics. *IEEE Software*, 2(2):80–85, 1985.

194. J. Xiao, D. Zhang, H. Chen, and H. Dong. Improved program dependence graph and algorithm for static slicing concurrent programs. In *Advanced Parallel Processing Technologies*, pages 121–130. Springer Berlin Heidelberg, 2005.

195. Z. Xing and E. Stroulia. UMLDiff: An algorithm for object-oriented design differencing. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 54–65, 2005.

196. B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. A brief survey of program slicing. *ACM SIGSOFT Software Engineering Notes*, 30(2):1–36, 2005.

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

198. S. Yau, J. Collofello, and T. MacGregor. Ripple effect analysis of software maintenance. In *Proceedings of IEEE International Computer Software and Applications Conference*, pages 60–65, 1978.

199. S. Yoo, D. Binkley, and R. Eastman. Seeing is slicing: Observation based slicing of picture description languages. In *Proceedings of IEEE Working Conference on Source Code Analysis and Manipulation*, pages 175–184, 2014.

200. Z. Yu and V. Rajlich. Hidden dependencies in program comprehension and change propagation. In *Proceedings of International Workshop on Program Comprehension*, pages 293–299, 2001.

201. A. Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of ACM International Symposium on the Foundations of Software Engineering*, pages 1–10, 2002.

202. L. Zhang, M. Gligoric, D. Marinov, and S. Khurshid. Operator-based and random mutant selection: Better together. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering*, pages 92–102, 2013.

203. X. Zhang and R. Gupta. Matching execution histories of program versions. *ACM SIGSOFT Software Engineering Notes*, 30(5):197–206, 2005.

204. X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. In *Proceedings of IEEE/ACM International Conference on Software Engineering*, pages 319–329, 2003.

205. X. Zhang, N. Gupta, and R. Gupta. Pruning dynamic slices with confidence. In *Proceedings of ACM Conference on Programming Language Design and Implementation*, pages 169–180, 2006.

206. X. Zhang, S. Tallam, and R. Gupta. Dynamic slicing long running programs through execution fast forwarding. In *Proceedings of ACM International Symposium on the Foundations of Software Engineering*, pages 81–91, 2006.

This document was prepared & typeset with pdfLaTeX*, and formatted with* NDdiss2*" classfile (v3.2013[2013/04/16]) provided by Sameer Vijay and updated by Megan Patnott.*