Advanced Dependence Analysis for Software Testing, Debugging, and Evolution

Raul Santelices, Haipeng Cai, Siyuan Jiang, and Yiji Zhang
University of Notre Dame
Notre Dame, Indiana, USA
email: {rsanteli|hcai|sjiang1|yzhang20}@nd.edu

Abstract

Software is more than just a collection of individual entities (e.g., components, statements). The behavior of software is the result of the interactions among those entities. To analyze and improve the functional behavior of software for software reliability and engineering tasks such as testing, debugging, and evolution, we need dependence analysis to identify which entities affect which other entities, and in which ways.

In this article, we briefly review classical concepts and applications of dependence analysis and we present new directions and results on dependence quantification and abstraction, which extend the utility of classical dependence analysis for modern software.

1 Introduction

Software defects have enormous economical and human costs for our modern society, which increasingly depends on software. Defects can affect critical qualities of software and information systems such as correctness, reliability, security, and safety. Developers, however, struggle daily with market pressures, resource constraints, and increasing complexity, all of which limits their ability to detect and fix serious software defects. Moreover, modern software *evolves* (i.e., it changes constantly), which adds a new dimension of complexity to software assurance tasks. For example, changing the type of a collection of items from a set to a list might lead to undesired duplicate items. For another example, adding minimum-age constraints to a banking system might prevent children from monitoring their accounts even if only parents can withdraw funds.

To ensure the reliability of a version or series of versions of the software, developers create test suites to automate its testing. When a test case fails (e.g., the system crashes or allows unauthorized access), developers perform debugging to identify and fix the underlying defect. This test-and-debug process continues for each new version that is produced as the software evolves. Each such version is, in fact, the result of a change process. Because software entities interact and affect each other, before making any change, developers must understand the impacts and risks of their proposed changes and take appropriate actions (e.g., adapt impacted entities, update the test suite).

These testing, debugging, and evolution tasks in real-world software are far from trivial. To complete these tasks effectively and timely, automated support is crucial. Program-analysis techniques and, in particular, *dependence analysis* [7], have arisen to automatically identify relationships among software entities (e.g., components, methods, statements) and to reason about those relationships. Dependence analysis underlies most white-box testing techniques [9] as well as (semi-)automated debugging [14] and change-impact analysis [2]. Thus, dependence analysis is crucial for a large class of software-engineering and assurance activities.

For example, it is possible to identify the defect that causes a sensor to output an erroneous value v because that value is computed by (i.e., is *data* dependent on) a statement s which, while correct, executes when decided by (i.e., is *control* dependent on) a condition c that compares two temperature values which are in different units, making c the source of the defect. For this example, testing would exercise these dependencies [9] by making c lead to the execution of s, which affects the output v whose value is not the one expected. Then, debugging traverses those dependencies backwards [14] from the output v until identifying c as the cause (unit mismatch). When preparing to fix this defect and make other changes, impact analysis [2] would identify which code, such as other computations also dependent on c, might be affected. Impacted code would have to be inspected and also changed if necessary. Finally, the interactions among changes should be tested [10].

Data and control dependencies, however, do not capture all aspects of the interactions among software entities. Remarkably, classical analyses do not indicate under which program states a bug or change propagates its effects—they only indicate which dependencies they propagate through [11]. Although other types of dependencies, such as structural (e.g., call graphs) and socio-technical (e.g., developers and components) [13], are used to model relationships in software projects at higher levels of abstraction to capture distinct information, the essential information they convey is still *whether* entities depend on other entities, but not *how*. Moreover, at abstract levels, commonly-used structures such as call graphs only represent *control flow* rather than behavioral dependencies: a method that calls another method might or might not affect its behavior.

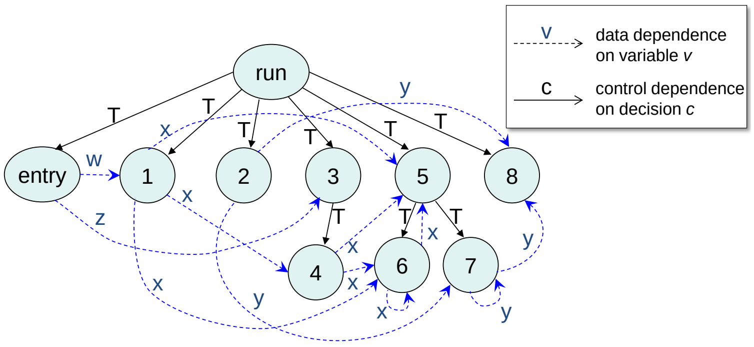
To overcome these limitations of classical dependence analysis, two directions have emerged recently which complement and enrich the information given by those analyses: information carried by dependencies *describing* or *quantifying* the states in which they propagate effects [9,12,15] and higher-level *behavioral* definitions of dependencies that supersede simpler control-flow relationships [4,5]. For software evolution, the first direction identifies the exact *conditions* on the program state under which a change propagates its effects through sequences of dependencies [9]. Because executing sequences of dependencies is not enough to guarantee that the effects of changes will be observed, these conditions are needed. In this same direction, as a more scalable alternative to computing and monitoring complex state conditions, a newer trend is to *quantify* dependencies with the probabilities that the effects of bugs or changes propagate through them [12,15].

The second recent direction addresses the imprecision of method-level structural dependence analyses [1] which are based on execution orders and scales better than statement-level analyses. Studies for the *dynamic* (i.e,, execution-based) form of this analysis have identified a large degree of imprecision in these traditional techniques [5]. Consequently, a new technique was developed which uses behavioral definitions of *method-level dependencies* based on data- and control-flow analysis. Based on initial results and further ongoing research, the new technique appears to *triple* the precision of classical results without significant increases in cost [4], to make method-level dependence analysis a truly cost-effective alternative to statement-level analysis.

2 Classical Dependence Analysis

This section reviews core concepts of classical dependence analysis and illustrates these concepts using the example program of Figure 1. In this example, program `P` takes a boolean `z` and an integer `w` as inputs, initializes local variables `x` and `y`, conditionally increments `x` based on `z`, updates `y` in a loop controlled by `x`, and finally prints the value of `y`. The graph on the right shows the dependence structure of this program, as explained next.

```
P(bool z, int w) // entry
1:  x = w;
2:  y = 2;
3:  if (z)
4:      x++;
5:  while(x > 0) {
6:      x--;
7:      y += 2;
    }
8:  printf("%d", y);
```



v
data dependence on variable *v*
c
control dependence on decision *c*
run
entry
1
2
3
4
5
6
7
8
T
w
x
y
z

Figure 1. Example program and its program dependence graph (PDG).

2.1 Syntactic Dependencies

Syntactic program dependencies [7] are derived directly from the program's syntax. These dependencies are classified as control or data dependencies. A statement s_1 is *control dependent* on a statement s_2 if a branching decision at s_2 determines whether s_1 necessarily executes. In Figure 1, for example, statement 4 is control dependent on statement 3 because the decision taken at 3 determines whether statement 4 executes or not.

A statement s_1 is *data dependent* 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 1, statement 8 is data dependent on statement 7 because 7 defines `y`, 8 uses `y`, and there is a path $\langle$7,5,8$\rangle$ that does not re-define `y` after 7 in the last iteration of the loop. Statement 8 is also data dependent on 2 because an execution might not enter the loop (the loop re-defines `y` at 7).

The parameters `z` and `w` at the *entry* line of `P` are inputs and thus are not data dependent on any statement. However, these parameters are treated as definitions of `z` and `w`. Thus, the statements 1 and 3, which use those variables, are data dependent on the entry of `P`.

On the right, Figure 1 shows the *program dependence graph* (PDG) of `P`. PDGs and their *interprocedural* (all-functions) extensions depict the dependence structure of entire programs. In the figure, the nodes are the statements of the program (1 to 8) and two special nodes: *run* which represents the decision of executing the program and *entry* for the entry of `P` where the parameters are defined. Solid edges represent control dependencies labeled with the corresponding control decision (e.g., T or F). The solid edges T from *run* represent the consequences of deciding to run `P`. Dashed edges represent data dependencies and are labeled with the corresponding variables.

2.2 Semantic Dependencies

Semantic dependencies represent the *actual behaviors* that the program can exhibit. Syntactic dependencies, in contrast, can only (over-)approximate those behaviors because a sequence of data and control dependencies between two statements is a necessary but not sufficient condition for those statements to be semantically dependent. Informally, a statement s is *semantically dependent* on a statement t if there is any change that can be made to t that affects the behavior of s.

For example, in Figure 1, the statements in the loop (5, 6, and 7) are semantically dependent on statements 1, 2, 3, and 4 because the latter could be changed so that the execution of the loop changes (e.g., by iterating more or fewer times) or the state of the variables computed in the loop (`x` and `y`) changes. In this case, the semantic dependencies of the loop statements coincide with their direct and transitive syntactic dependencies—as represented by the PDG.

However, for this same example, if `w` is guaranteed to be always negative (e.g., `P` is a function in a larger program whose only caller passes -1 for `w`), the loop head (statement 5) will always behave in the same way by evaluating to false and the loop body (statements 6 and 7) will never execute. In that case, despite being syntactically dependent on statements 1–4, the loop statements 5–7 are not semantically dependent on 1–4.

This case illustrates the caveats of classical dependence analysis, which in is most precise form is based on syntactic (control and data) dependencies. Unfortunately, computing semantic dependencies is an undecidable problem [7]. In between these two types of dependencies, however, there is room for incorporating state information not captured by classical data- and control-dependence analysis, as discussed in the next sections.

3 Dependence Quantification

Recently, to address the limitations of classical dependence analysis, we introduced the concept of *dependence quantification* [12]. This approach enriches dependencies by assigning them quantities that represent properties such as likelihood or strength. Such quantities denote the relative relevance of dependencies in programs, which help users and tools focus their attention first on the dependencies that—likely—matter the most. Quantification has been found to effectively direct users first to the areas most affected by a change [3] or most likely to contain bugs [15].

3.1 Example

Consider the code fragment in Figure 2 for finding tangents between circles. The forward *static slice* (transitive syntactic dependencies) from `c` at line 2 contains lines 2–11, which suggests that they might be affected by `c`. However, `c` at line 2 strongly affects `c*c` at line 3 but less strongly affects the branching decision in that line—variations in `c` may or may not flip the branch taken. Therefore, because `c` may or may not affect this decision, the remaining lines are "less affected" than lines 2 and 3. Lines 6–11, however, also use the value of `c`, which makes them "more affected" than lines 4 and 5 (but still less affected than lines 2 and 3).

Quantification identifies these differences among lines in the slice. This kinds of approach can give, for example, a score of 1.0 to lines 2 and 3, 0.5 to lines 4 and 5, and an intermediate value 0.75 to the rest. The actual scores for these lines will depend on the specific technique used to quantify the slice. We present two such techniques next.

3.2 Static Analysis

Static quantification of program *slices* (i.e., transitive syntactic dependencies) can be achieved by analyzing the control- and data-flow structure of programs. The advantages of this approach to quantification are that no execution data is required and the results represent all behaviors of the program. The latter advantage is quite attractive for tasks like testing because, no matter how many test cases have been created, this approach points users to behaviors not tested yet.

For this static approach to dependence quantification, we used two key insights:

1. Some data dependencies are less likely to occur than others because the conditions to reach the target from the source of the dependence vary.

```
for (sign1 = +1; sign1 >= -1; sign1 -= 2) {
   c = (r1 - sign1 * r2) / d;
   if (c*c <= 1.0) {
      h = sqrt(d*d - pow(r1-sign1*r2),2)) / d;
      for (sign2=+1; sign2>=-1; sign2-=2) {
         nx = vx * c - sign2 * h * vy;
         ny = vy * c + sign2 * h * vx;
         print x1 + r1 * nx;
         print y1 + r1 * ny;
         print x2 + sign1 * r2 * nx;
         print y2 + sign1 * r2 * ny;   }}}
```

Figure 2. Excerpt from a program that computes the tangents among circles.

2. Data dependencies are more likely to propagate information than control dependencies, yet control dependencies should not be ignored either as in classical analyses.

Using the first insight, a reachability and alias analysis of the control flow of the program have been created. This analysis estimates the probability that the target of a dependence is reached from its source and that both points access the same memory location. Using the second insight, the approach performs another reachability analysis, this time on the dependence graph, that gives a lower but non-zero score to control dependencies.

We estimate the probability that a statement a affects a statement b by computing two components: (1) the probability that a sequence of dependencies from a to b occurs when the program executes and (2) the probability that information flows through that sequence. We present more details of this static approach and initial positive results in [11].

3.3 Dynamic Analysis for Quantification

Given a representative test suite, we can quantify slices via differential execution analysis [10] and sensitivity analysis [8].

3.3.1 Differential Execution Analysis

Differential execution analysis (DEA) is designed specifically for forward slicing from changes to identify the *runtime* semantic dependencies [7] of statements on changes. Semantic dependencies tell which statements are truly affected by which other statements or changes. Although finding semantic dependencies is an undecidable problem, DEA detects such dependencies on changes when they occur at runtime to under-approximate the set of semantic dependencies in the program. Therefore, DEA does not guarantee 100% recall of semantic dependencies but it achieves 100% precision. This is much better than what dynamic slicing normally achieves [6, 10].

DEA works by executing a program before and after the change, collecting the *augmented execution history* [10] of each execution, and then comparing both histories. The execution history of a program for an input is the sequence of statements executed for that input. The *augmented* execution history is the execution history annotated with the values read and written by each statement occurrence. The differences between two such histories reveal which statements had their occurrences or values altered by a change—the conditions for semantic dependence. A formal definition of DEA is given in [10].

3.3.2 Sensitivity Analysis

DEA can be used to quantify *static* forward slices when the change is known—for *post-change* impact analysis. To do this, a DEA-based quantification technique can execute the program repeatedly with and without the change for many inputs and find, for each statement, the frequency with which it is impacted by the change. If the inputs are sufficiently representative of the program's behavior, we can use these frequencies as the quantities for the statements in a slice.

More generally, however, the specifics of a change might not be known when a user asks for the impacts of modifying a statement or when the slicing task does not involve a change (e.g., debugging, information-flow analysis). For such situations, we created SENSA, a new sensitivity-analysis technique and tool for slice quantification and other applications [3]. Sensitivity analysis is used in many fields to determine how modifications to some aspect of a system (e.g., an input) affect other aspects of that system (e.g., the outputs) [8].

We designed SENSA as a generic modifier of program states at given locations, such as changes or failing points. SENSA inputs a program P, a test suite T, and a statement c. For each test case t in T, SENSA executes t repeatedly, replaces each time the value(s) computed by c with a different value, and uses DEA to find which statements were affected by these modifications. With this information for all test cases in T, SENSA computes the sensitivity of each statement s in P to the behavior of c by measuring the frequency with which s is affected by c. These frequencies are the degree of dependence on statement c of all statements s in P, given T.

For a *forward* static slice from statement c in program P, SENSA uses T to quantify the dependence on c of the statements in that slice. For a *backward* static slice from s, SENSA can be used in a similar fashion to quantify the dependence of s on selected statements c from that slice.

SENSA is highly configurable. In addition to parameters such as the number of times to re-run each test case with a different modification (the default is 20), SENSA lets users choose among built-in modification strategies for picking new values for c at runtime. Furthermore, users can add their own strategies. SENSA ensures that each new value picked for c is unique, to maximize diversity while minimizing bias. Whenever a strategy runs out of possible values for a test case, SENSA stops and moves on to the next test case. SENSA offers two modification strategies from which the user can choose:

1. *Random*: A random value is picked within a specified range. By default, the range covers all elements of the value's type except for *char*, for which only readable characters are picked. For some reference types such as *String*, objects with random states are picked. For all other reference types, the strategy currently picks *null*.

2. *Incremental*: A value is picked that diverges from the original value by increments of i (the default is 1.0). For example, for a value v, the strategy first picks $v + i$ and then picks $v - i$, $v + 2i$, $v - 2i$, etc. For common non-numeric types, the same idea is used. For example, for string *foo*, the strategy picks *fooo*, *fo*, *foof*, *oo*, etc.

3.3.3 Empirical Results

To assess the effectiveness of SENSA, we implemented it to analyze Java-bytecode programs and applied it to the task of predicting change impacts at various program locations. The scenario is *early change-impact analysis*, in which users query for the potential impacts of changing a location without necessarily knowing the details of the change yet.

For this study, we chose four Java subjects for which many test cases and changes (bug fixes) are provided for research studies. Table 1 in columns 1–4 lists these subjects along with their sizes,

Table 1. Average inspection efforts using predictions of slicing and SENSA

Subject name	Lines of code	Test cases	Changes studied	Ideal (best) case	Average effort: Static slicing	Average effort: Dynamic slicing	Average effort: SENSA Rand	Average effort: SENSA Inc
Schedule1	301	2650	7	47.90	50.14	48.34	48.01	48.01
NanoXML	3521	214	7	8.84	22.71	27.09	20.27	22.37
XML-security	22361	92	7	5.00	31.94	45.37	13.15	21.49
Ant	44862	205	7	3.21	39.16	41.55	29.84	23.76
			average:	**16.24**	**35.99**	**40.59**	**27.82**	**28.91**
			standard deviation:	19.36	13.22	13.08	19.20	20.59
						p-value w.r.t. static slicing:	**0.0098**	**0.0237**

test suites and changes used. We compared the results of SENSA for this task with those of the two main techniques from classical dependence analysis: *static* slicing (code-based) and *dynamic* (execution-based) slicing.

We first applied SENSA and static forward slicing to the locations of the changes to reproduce the scenario in which users query for the consequences that changing those locations would have. Then, for each result of SENSA, we ranked the statements from greatest to lowest score. For comparison, we also ranked the static and dynamic slices using Weiser's approach [14] by visiting statements breadth-first from the change location and sorting them by increasing visit depth. For tied statements in a ranking, we used as their rank the average of their positions in that ranking.

To compare the predictive power of the rankings given by SENSA and slicing, we applied the changes, one at a time, to the corresponding location in its subject. Then, for each change, we used DEA on the unchanged and changed subject to find the statements actually impacted when running all test cases for that subject. Using these actual impacts, we calculated how closely each ranking predicted those impacts. For each ranking and each impacted statement found by DEA, we determined the percentage of the *static slice* that would have to be traversed, in the ranking's order, to reach that statement. We call this percentage the *cost* of finding an actually-impacted statement using that ranking. Then, we computed the average cost of finding all actual impacts for each ranking—the lower this cost is, the better the technique that produced that ranking is at predicting impacts. Also, to assess how close to the best possible result each ranking was, we created the *ideal* ranking by placing all actually-impacted statements at the top of that ideal ranking.

The last five columns of Table 1 present the average cost, for each subject and the seven changes in that subject, of the following rankings: the *ideal* ranking, the static and dynamic *slicing* rankings (using Weiser's traversal), and the rankings for SENSA and its strategies *Rand* (Random) and *Inc* (Incremental).

The *ideal* (best possible) costs in Table 1 reveal that, on average, most of the statements in Schedule1 were *actually* impacted, in contrast with the other three subjects, for which low numbers of statements were impacted. Overall, for static and dynamic *slicing*, the prediction costs ranged between 23–50% with average 36% and 41%, respectively, whereas the two strategies of SENSA were much closer to the ideal predictions at 13–48% and averages of 28% and 29%, respectively.

In all, SENSA seems much better overall than classical slicing (Weiser's traversal) at predicting actual impacts. A *non-parametric* (i.e., no data distribution assumptions) Wilcoxon signed-rank test gives p-values below 0.05, as shown at the bottom of Table 1, which indicate that the superiority of SENSA over slicing for predicting impacts is statistically significant.

$$\boxed{\mathrm{M0}_e\ \mathrm{M1}_e\ \mathrm{M2}_e\ \mathrm{M5}_e\ \mathrm{M2}_i\ \mathrm{M1}_i\ \mathrm{M3}_e\ \mathrm{M1}_i\ \mathrm{M0}_i\ \mathrm{M4}_e\ \mathrm{M4}_i\ \mathrm{M0}_i\ \mathrm{x}}$$

(Subscript e for entry event and i for returned-into event; entry method M0; exit event x)

Figure 3. Example execution trace of a program with six methods M0–M5.

4 Method-level Dependencies

4.1 Problem

For scalability of dependence analysis, it is often desirable to switch the granularity of an analysis from the statement level to the method level. Although important information can be missed by doing so, the analysis can provide a higher-level picture of larger software systems.

Unfortunately, at the method and coarser levels, classical analyses use simple structural dependencies such as method calls. For example, a method m called by a method m', directly or transitively, is considered as dependent on m'. This can be quite imprecise. In fact, we found in recent studies that classical method-level *dynamic impact analyses* such as PathImpact [1] can be too imprecise [5] with large numbers of false positives. The reason is that, at its core, PathImpact simply marks as impacted by a method m all methods called or returned into after m executes.

Figure 3 shows an example execution trace where the e and i subscripts for a method represent the entry and return-into events for that method. When queried for the impact set of M2, in addition to M2 itself, PathImpact first finds M5, M3, and M4 as impacted because they were entered after M2 was entered and then finds M0 and M1 because they returned after M2 was entered (i.e., parts of them executed after parts or all of M2). Thus, the resulting dynamic impact set of M2 is {M0, M1, M2, M3, M4, M5} for this trace. For multiple traces, the results are unioned.

4.2 Solution

Reaching a method m' after a method m at runtime is a necessary condition for m to impact m', but not all such methods m' necessarily depend on m. To fix this problem, we recently proposed a technique called Diver [4] which builds first a *method-level* dependence graph of the program—based on statement-level dependencies—which is then used to prune execution traces to find which methods really depend on m. For a reasonable cost, these method-level dependencies can be much more precise for impact analysis than simple call information.

Diver works in three phases: static analysis, runtime, and post-processing. The static-analysis phase computes method-level data and control dependencies for the entire program. This is done only one time, regardless of the executions to be considered for the runtime phase and the queries for impact sets performed at post-processing. At the method level, a method m is dependent on a method m' if there is a statement s is m that is dependent on a statement s' via a sequence of one or more statement-level data and control dependencies. The runtime phase is identical to PathImpact by simply collecting the method traces. The post-processing phase, however, uses the static dependencies to determine whether a method m that executed after m' did so along a dependence path from m' to m. If not, unlike PathImpact, Diver does not report m as impacted.

To illustrate, consider again the trace in Figure 3. It is easy to imagine a program with these methods in which not all methods depend on each other—see [4] for an example. For such a program and query M2, for example, Diver traverses the trace to find which dependencies were exercised in it 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 {M2}. Then, there

Table 2. Relative precision as ratios of impact set sizes of DIVER to PATHIMPACT (PI).

Subject	PI IS Size mean	PI IS Size stdev	DIVER IS Size mean	DIVER IS Size stdev	IS Size Ratio mean	IS Size Ratio stdev	Wilcoxon p-value
Schedule1	18.0	1.6	12.8	4.7	71.3%	24.5%	6.65E-05
NanoXML	82.6	48.1	37.1	28.9	51.7%	33.1%	2.40E-30
XML-security	199.8	168.4	45.1	68.1	28.8%	30.3%	4.79E-102
Ant	159.5	173.4	17.9	34.3	25.7%	33.6%	2.94E-100
Average	166.2	164.9	32.2	53.1	**30.8%**	**33.3%**	**9.29E-07**

might be only one outgoing dependence from M2 in the graph—to method M5 [4]—in which case the impact set is $\{M2, M5\}$, in contrast with PATHIMPACT which reports all six methods.

4.3 Empirical Results

Table 2 presents comparative precision results for DIVER and PATHIMPACT for the same four subjects used in Section 3.3.3, with two statistics per subject and overall for all queries (last row) for the corresponding data points: the *mean* and the standard deviation (*stdev*) of the impact set sizes and ratios.

The results in the table show that, on average, the DIVER impact sets were much smaller than for PATHIMPACT, especially for the two largest subjects. Large numbers of false positives for PATHIMPACT were identified as such and pruned by DIVER. For example, PATHIMPACT identified 160 methods on average in its impact sets for Ant, whereas DIVER reported only 18 for a mean ratio of 25.7%. (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 results suggest that DIVER is even stronger with respect to PATHIMPACT for larger subjects, which are more representative of modern software. For the smaller subjects Schedule1 and NanoXML, DIVER provides smaller gains possibly due to the proximity and interdependence of the few methods they contain.

We applied the Wilcoxon signed-rank one-tailed test 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 2 shows the resulting p-values. For $\alpha = .05$, the null hypothesis is that DIVER is not more precise than PATHIMPACT. The 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.

In all, DIVER can *safely* prune 70% of the impact sets computed by PATHIMPACT, which amounts to a precision increase by a factor of 3.33 (i.e., by 200%). Thus, method-level data and control dependencies should replace method calls as the common practice for high-level analysis. For more technical and empirical details on DIVER, we refer the reader to [4].

5 Conclusion

Dependence analysis for software testing, debugging, evolution, and many other tasks has a long history [1, 7, 14]. However, there are still under-explored dimensions of dependence analysis which can significantly increase the cost-effectiveness of its applications to these tasks [3, 4, 12]. This ongoing work shows that fundamental advances are still possible and necessary.

In this article, we revisited the classical concepts of data and control dependence analysis at the statement level and structural (call) dependencies at the method and higher levels. Then, we showed some recent advances and results on two particular dimensions—quantification and abstraction—which considerably improve upon the results of classical analyses. Through this exposition, we expect the reader to have gained new insights into the fundamentals and some recent advances of this field, which underlies the description of software behavior for virtually any engineering task.

Acknowledgment

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

References

[1] 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, May 2005.

[2] S. A. Bohner and R. S. Arnold. *An Introduction to Software Change Impact Analysis*. In Software Change Impact Analysis, Bohner & Arnold, Eds. IEEE Computer Society Press, pages 1–26, June 1996.

[3] H. Cai, S. Jiang, R. Santelices, Y. jie Zhang, and Y. Zhang. SENSA: Sensitivity Analysis for Quantitative Change-impact Prediction. In *Proc. of IEEE Int'l Working Conf. on Source Code Analysis and Manipulation*, Sept. 2014. 10pp, to appear.

[4] H. Cai and R. Santelices. DIVER: Precise Dynamic Impact Analysis Using Dependence-based Trace Pruning. In *Proc. of IEEE/ACM Int'l Conf. on Automated Software Engineering, New Ideas track*, Sept. 2014. 6pp, to appear.

[5] H. Cai, R. Santelices, and T. Xu. Estimating the Accuracy of Dynamic Change-Impact Analysis using Sensitivity Analysis. In *Proc. of Int'l Conf. on Software Security and Reliability (IEEE Reliability Society)*, pages 48–57, June 2014.

[6] 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.

[7] A. Podgurski and L. A. 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.

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

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

[10] 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, Apr. 2010.

[11] R. Santelices, Y. Zhang, H. Cai, and S. Jiang. Change-Effects Analysis for Evolving Software. *Advances in Computers*, 93:227–285, Mar. 2014.

[12] R. Santelices, Y. Zhang, S. Jiang, H. Cai, , and Y. jie 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, May 2013.

[13] A. Sarma, L. Maccherone, P. Wagstrom, and J. Herbsleb. Tesseract: Interactive visual exploration of socio-technical relationships in software development. In *Proc. of IEEE/ACM Int'l Conf. on Softw. Eng.*, pages 23–33, May 2009.

[14] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[15] Y. Zhang and R. Santelices. Prioritized Static Slicing for Effective Fault Localization in the Absence of Runtime Information. *Technical Report TR 2013-06*, CSE, U. of Notre Dame, Nov. 2013. 12pp, in revision for J. of Systems and Software.