Quantitative Program Slicing: Separating Statements by Relevance

Raul Santelices, Yiji Zhang, Siyuan Jiang, Haipeng Cai, and Ying-jie Zhang*
University of Notre Dame, Indiana, USA
*Tsinghua University, Beijing, China
{rsanteli|yzhang20|sjiang1|hcai}@nd.edu, *zhang-yj09@mails.tsinghua.edu.cn

***Abstract*—Program slicing is a popular but imprecise technique for identifying which parts of a program affect or are affected by a particular value. A major reason for this imprecision is that slicing reports all program statements possibly affected by a value, regardless of how relevant to that value they really are. In this paper, we introduce *quantitative slicing* (*q-slicing*), a novel approach that quantifies the relevance of each statement in a slice. Q-slicing helps users and tools focus their attention first on the parts of slices that matter the most. We present two methods for quantifying slices and we show the promise of q-slicing for a particular application: predicting the impacts of changes.**

I. Introduction

A major problem of program analyses is their imprecision, manifested in an excess of *false positives* (false alarms) or *false negatives* (missed targets) or both. Program slicing [1] is one such analysis that is popular but also quite imprecise [2]. *Static* (code-based) slicing often reports too many potentially-relevant statements, whereas *dynamic* (execution-based) slicing—a variant that trades completeness for precision—can also produce many false positives [2], [3].

To increase the precision of static slicing by reducing the size of slices, researchers have proposed combining static and dynamic slicing (e.g., [4]) and pruning slices based on some criterion (e.g., [5], [6]). However, these methods can suffer from false positives and even false negatives. Other attempts to improve this precision include better alias and points-to analyses, which can reduce slice sizes to some extent.

To address the limitations of slicing, especially its static version, we introduce in this paper *quantitative slicing* (*q-slicing*), our umbrella project for assigning quantities to statements in a slice, such as scores or probabilities. These quantities denote the relevance of each statement in that slice to help users and tools focus their attention first on the parts of the slice that matter the most. For example, for change-impact analysis, q-slicing points users to the most affected areas of the program first so they can take major corrective actions early. Q-slicing can also help tools prioritize regression-testing targets by their likelihood of interaction with changes.

As with program analysis in general, we use two approaches to computing quantitative slices (q-slices): a static and a dynamic approach. We have developed one technique of each type. Each technique has its own merits and has the potential to complement the other. The first technique uses a probabilistic model of dependence propagation to analyze the static structure of the program [7]. The second technique applies sensitivity analysis to program executions to dynamically measure the degree to which statements in the program depend on modifications to other statements.

In this paper, we discuss both techniques with an emphasis on our *new* dynamic approach that combines sensitivity analysis and execution differencing. The results are encouraging. Using our new sensitivity-analysis tool, we quantified the forward static slices from various locations. The resulting scores isolated with great precision, among all potentially-impacted statements found by static slicing, the actual impacts observed at runtime after making real changes in those locations.

The benefits of quantifying slices are many and can further the adoption of program slicing in development tools. We already found that *early* change-impact analysis, in which users assess the potential impact of changing a location before designing the change, can become highly effective. Quantitative *forward* slicing can also support regression testing, testability analysis, and information-flow analysis. Meanwhile, quantitative *backward* slicing can improve debugging, comprehension, reverse engineering, and parallelization. Even dynamic slicing can benefit from q-slicing, as dynamic slices are mediocre approximations of true runtime dependencies [2], [3].

II. Example

Consider the code fragment in Figure 1 for finding tangents between circles. The forward static slice 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).

Q-slicing quantifies these differences among lines in the slice. A q-slicing technique 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 q-slicing technique used to quantify the slice. We present two such techniques next.

III. Static Quantification

Static quantification of program slices can be achieved by analyzing the control- and data-flow structure of programs [7],

```
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;   }}}
```

Fig. 1. Excerpt from a program that computes the tangents among circles.

[8]. The advantages of this approach 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.

Our static approach for q-slicing uses 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.
2) Data dependencies are more likely to propagate information than control dependencies, yet control dependencies should not be ignored either as in some existing work.

Using the first insight, we created a reachability and alias analysis of the control flow of the program that 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, our 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. More details of this approach and initial positive results are presented in [7], [8].

IV. Dynamic Quantification

Given a representative test suite, we can quantify slices via differential execution analysis [2] and sensitivity analysis [9].

A. Differential Execution Analysis

Differential execution analysis (DEA) is designed specifically for forward slicing from changes to identify the runtime *semantic dependencies* [10] 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 [2], [3].

DEA works by executing a program before and after the change, collecting the *augmented execution history* [2] 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 [2].

B. 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 q-slicing 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 q-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,[1] a new sensitivity-analysis technique and tool for slice quantification and other applications. 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) [9].

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 three 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

[1]SensA is available for download at http://nd.edu/~hcai/sensa/html

TABLE I
SUBJECTS AND OVERALL RESULTS FOR STUDY OF CHANGE-IMPACT PREDICTION USING SENSA FOR Q-SLICING

Subject	**Description**	**Lines of code**	**Test cases**	**Changes studied**	**Average slice size**	**Average impact-prediction cost (% of ranking)**				
						ideal	**slicing**	**SENSA-rnd**	**SENSA-inc**	**SENSA-obs**
Schedule1	priority scheduler	290	2650	7	63%	37.1%	45.2%	38.4%	38.3%	38.4%
NanoXML	XML parser	3497	214	7	52%	7.8%	24.6%	9.9%	10.7%	—
XML-security	encryption library	21613	92	7	73%	5.7%	35.8%	12.0%	12.5%	—

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]

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) *Observed*: First, the strategy collects all values that c computes when running T on P. Then, the strategy picks iteratively from this pool the new value at c. The goal is to ensure that values remain meaningful to the program.

C. Preliminary Evaluation

We compared the q-slices produced by SENSA with context-insensitive static forward slices for 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. Our research questions were:

RQ1: How good is q-slicing at predicting impacts?

RQ2: How good is q-slicing for different usage budgets?

RQ3: How expensive is SENSA for q-slicing?

The first and second questions address the benefits of q-slicing overall and per slice-inspection effort. The third question targets the practicality of SENSA.

1) Setup: SENSA extends DUA-FORENSICS [11], our analysis and monitoring tool for Java bytecode programs. For this study, we chose three Java subjects from the SIR repository [12] for which many test cases and changes (bug fixes) are provided. Table I lists these subjects along with short descriptions, sizes, test suites, changes used, and average sizes of their slices as percentages of the respective subject sizes.

2) Methodology: 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 q-slice given by SENSA, we ranked its statements from greatest to lowest score. For comparison, we also ranked the static slices using Weiser's approach [1] by visiting statements in breadth-first order 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 assess and compare the predictive power of the rankings given by q-slicing via SENSA and Weiser's 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 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.

3) Results and Analysis: The last five columns of Table I present the average cost, for each subject and the seven changes in that subject, of five rankings: the *ideal* ranking, the *slicing* ranking (using Weiser's traversal), and the rankings for SENSA and its three strategies: *rnd* (Random), *inc* (Incremental), and *obs* (Observed). For some changes in NanoXML and XML-security, SENSA-*obs* was not applicable as only one value at the change was observed at runtime and, thus, SENSA could not find a different value to replace it. In consequence, we omitted the SENSA-*obs* results for those subjects.

RQ1: The *ideal* cost for Schedule1 in Table I reveals that, on average, more than a third of the statements in the slices were *actually* impacted, in contrast with the two other subjects, for which less than 10% of the statements in their slices were impacted. For Weiser's *slicing*, the prediction costs ranged between 25–45%, whereas SENSA was remarkably closer to the ideal predictions at 10–38%. All SENSA strategies exhibited similar costs when applicable. In all, SENSA for q-slicing seems much better overall than simple program slicing (Weiser's traversal) at predicting actual impacts.

RQ2: Because slices can be large, if users cannot explore the entire slice, we expect that they will inspect each ranked slice from the top. The graphs of Figures 2 for NanoXML and 3 for XML-security—the two largest subjects—show, on average for all changes in the subject, the percentage of the actual impacts found (Y axis) for each percentage of the ranked slice inspected (X axis). For example, at 10% of the traversal for NanoXML, on average, the ideal ranking finds 75% of the actual impacts while the *slicing* predictions find only 33%. The SENSA predictions, in contrast, find 66–67% of those impacts, suggesting that SENSA is not only better overall (RQ1) but is also especially good for the portions of the rankings that users and tools would inspect first.

[2]We are developing a method to instantiate most types with random states.

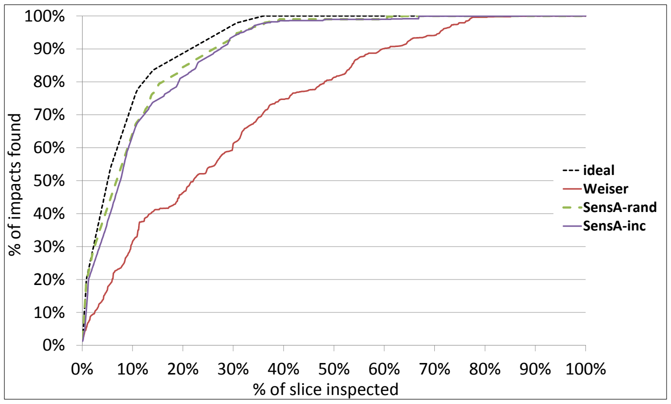

Fig. 2. Effectiveness of slicing per inspection effort for NanoXML.

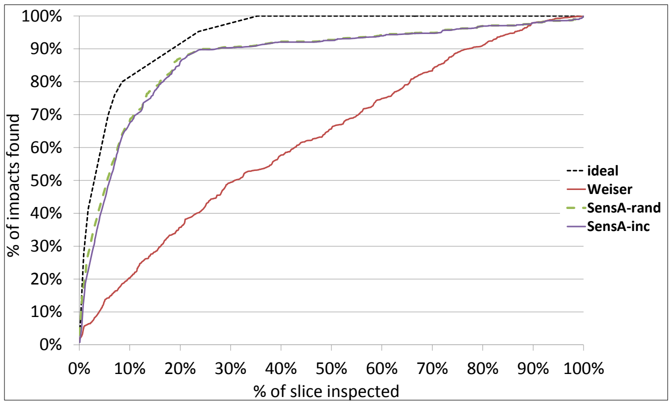

Fig. 3. Effectiveness of slicing per inspection effort for XML-security.

RQ3: We run our experiments on an Intel Core i5 quad-core machine with 8GB RAM and 64-bit Linux. The average time taken by SENSA for all changes in our subjects ranged from 303 to 493 seconds. The runtime overhead of executing the entire test suites for the instrumented subjects ranged between 32–63%. We consider these costs acceptable. There is also plenty of room for optimizations in our implementation.

V. Related Work

A few other techniques discriminate among dependencies within slices. Two of them [5], [13] work on dynamic backward slices to estimate influences on outputs, but do not consider coverage probabilities as we do. These techniques could eventually be compared with q-slicing after we develop backward variants. Also for backward analysis, the work of [14] models the behavior of dependencies statistically and thin slicing [6] prunes control dependencies incrementally.

Sensitivity analysis has been used in software engineering to analyze requirements and components (e.g., [15]). For other tasks, however, only restricted forms of sensitivity analysis have been used, such as *mutation analysis* and *mutation testing* [16]. Mutation testing changes code in many locations to simulate common programming errors for test-suite assessment, whereas our approach systematically modifies the program *state* at one point to find as many impacts as possible.

VI. Conclusion and Future Work

Quantitative slicing promises significant increases in the usefulness of program slices. Rather than pruning statements from slices, q-slicing grades statements according to their relevance in a slice. In this paper, we discussed two approaches, including a novel dynamic technique, for computing relevance scores. More techniques can be added under the umbrella of q-slicing. The possibilities are many for quantifying slices in better ways and for improving other applications, such as debugging. Q-slicing is an emerging, rich, and open field.

We are adding dynamic slicing to our comparisons. We are also extending our studies to more subjects and changes. Moreover, we are developing a visualization tool for q-slices to improve our own understanding of the overall approach and specific techniques. Using this tool, we will also study how developers take advantage in practice of quantitative slices.

Slightly farther in the future, we foresee adapting q-slicing to quantify other important tasks, such as debugging, comprehension, mutation and interaction testing, and information-flow analysis. More generally, we see q-slicing's scores as abstractions of states and interactions among those states. Those scores can be expanded to multi-dimensional values or data structures and reduced to discrete sets on demand.

References

[1] M. Weiser, "Program slicing," *IEEE Trans. on Softw. Eng.*, vol. 10, no. 4, pp. 352–357, 1984.

[2] R. Santelices, M. J. Harrold, and A. Orso, "Precisely detecting runtime change interactions for evolving software," in *Proc. of IEEE Int'l Conf. on Softw. Testing, Verif. and Validation*, Apr. 2010, pp. 429–438.

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

[4] J. Krinke, "Effects of context on program slicing," *J. of Systems and Software*, vol. 79, no. 9, pp. 1249–1260, 2006.

[5] X. Zhang, N. Gupta, and R. Gupta, "Pruning dynamic slices with confidence," in *Proc. of ACM Conf. on Prog. Lang. Design and Impl.*, Jun. 2006, pp. 169–180.

[6] M. Sridharan, S. J. Fink, and R. Bodik, "Thin slicing," in *Proc. of ACM Conf. on Prog. Lang. Design and Impl.*, Jun. 2007, pp. 112–122.

[7] R. Santelices and M. J. Harrold, "Probabilistic slicing for predictive impact analysis," *Tech. Rep. CERCS-10-10*, Georgia Tech, Nov. 2010.

[8] Y. Zhang and R. Santelices, "Predicting data dependences for slice inspection prioritization," in *IEEE Int'l Workshop on Program Debugging, co-loc. with ISSRE*, Nov. 2012, pp. 177–182.

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

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

[11] R. Santelices and M. J. Harrold, "Efficiently monitoring data-flow test coverage," in *Proc. of IEEE/ACM Int'l Conf. on Autom. Soft. Eng.*, Nov. 2007, pp. 343–352.

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

[13] T. Goradia, "Dynamic impact analysis: a cost-effective technique to enforce error-propagation," in *Proc. of ACM SIGSOFT Int'l Symp. on Softw. Testing and Analysis*, Jul. 1993, pp. 171–181.

[14] G. Baah, A. Podgurski, and M. Harrold, "The probabilistic program dependence graph and its application to fault diagnosis," *IEEE Trans. on Softw. Eng.*, vol. 36, no. 4, pp. 528–545, 2010.

[15] M. Harman, J. Krinke, J. Ren, and S. Yoo, "Search based data sensitivity analysis applied to requirement engineering," in *Proc. of Genetic and Evol. Comp. Conf.*, Jul. 2009, pp. 1681–1688.

[16] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, Apr. 1978.