Discriminating Influences among Instructions in a Dynamic Slice

Vijay Krishna Palepu and James A. Jones
University of California, Irvine
{vpalepu, jajones}@uci.edu

ABSTRACT

Dynamic slicing is an analysis that operates on program execution models (*e.g.,* dynamic dependence graphs) to support the interpreation of program-execution traces. Given an execution event of interest (*i.e.,* the slicing criterion), it solves for all instruction-execution events that either *affect* or *are affected by* that slicing criterion, and thereby reduces the search space to find influences within execution traces. Unfortunately, the resulting dynamic slices are still often prohibitively large for many uses. Despite this reduction search space, the dynamic slices are often still prohibitively large for many uses, and moreover, are provided without guidance of which and to what degree those influences are exerted. In this work, we present a novel approach to quantify the relevance of each instruction-execution event within a dynamic slice by its degree of relative influence on the slicing criterion. As such, we augment the dynamic slice with dynamic-relevance measures for each event in the slice, which can be used to guide and prioritize inspection of the events in the slice. We conducted an experiment that evaluates the ability of existing dynamic slicing and our approach, using dynamic relevance, to correctly identify sources of execution influence and state propagation. The results of the experiment show that inspections that were guided by traditional dynamic slicing to find the root cause for a failure reduced the search space by, on average, 61.3%. Further, inspections guided with the assistance of the new dynamic relevance reduced the search space by 96.2%.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

Keywords

Program Analysis, Dependence Analysis, Dynamic Slicing

1. INTRODUCTION

Dynamic slicing is a well known technique that is used for describing program behavior and for several software comprehension and maintenance tasks. In use, its user (*e.g.,* a developer or another automated analysis technique) expresses an interest in an execution event, which is referred to as the *slicing criterion.* The dynamic-slicing approach then computes the set of all instruction-execution events[1] that influence (or are influenced by) the slicing criterion. The result of the analysis is a *dynamic slice*, and is comprised of a undifferentiated set of either static instructions or dynamic instruction-execution events that are influential for the slicing criterion. Such dynamic slices could support development tasks such as debugging (*e.g.,* [6, 23]), for which the criterion may represent an observed error state at a breakpoint, or change-impact analysis (*e.g.,* [16]), for which the criterion may represent points in an execution that new logic may be introduced.

Despite its importance and widespread use in software engineering tools and techniques, there exists a general consensus that dynamic slices are often intractable for programmer inspection and navigation due to their size and complexity. As examples, LaToza and Myers [10] note that developers comprehend program behavior by navigating dependence relations within a dynamic slice that can often be prohibitively challenging, and Zhang *et al.* [23] found backward dynamic slices as large as 49% of the program execution trace and forward dynamic slices as large as 63% of the traces. Moreover, such resulting dynamic slices are typically presented an unordered sets of instruction instances, without any auxiliary information that may assist the developer or client analysis in their interpretation. And, although some existing work has been applied to "pruning" dynamic slices to further reduce their size [20, 22], the result is still a set of instances without information to help assess the relative influence of each such instruction instance on the criterion.

Given the fundamental difficulties of size and complexity in inspecting dynamic slices to understand program behavior, in this work we propose a mechanism that is able to differentiate the degree of influence between instruction-execution events. As a result, slicing users can perform prioritized inspections of the slice to more quickly identify most-relevant events, and as such, further reduce the search for influences. We also conducted an experiment to determine the degree to which the additional relevance information can further support dynamic-slice interpretation and

ASE'14, September 15-19, 2014, Vasteras, Sweden.
Copyright 2014 ACM 978-1-4503-3013-8/14/09 ...$15.00.
http://dx.doi.org/10.1145/2642937.2642962 .

[1]An *instruction-execution event* is a specific instance of an execution of an instruction within a single execution of the program. For example, an instruction may be executed many times during an execution — each such instruction execution constitutes an instruction-execution event. We also refer to these as *instruction instances.*

search reduction. Our results found that although dynamic slicing was effective in reducing the search for a debugging task (on average, reducing the cost by 61.3% from the original execution), using the additional relevance information to augment the dynamic slice substantially further reduced the search (on average, reducing the cost by 96.2%, or an additional cost savings of 34.9%).

The main contributions of this work are:

- A new measure, that we are calling *dynamic relevance*, that represents the degree to which an instruction's execution distinctively influences the slicing criterion.
- A novel, augmented model of a dynamic slice that encodes a degree of influence on the dynamic-substantially further slicing criterion, *i.e.,* our dynamic relevance measure, for each instruction-execution event in the dynamic slice. These measures may provide valuable information to direct users and client analyses toward instruction-execution events that strongly affect the state at the slicing criterion.
- An experimental design that evaluates the ability of dynamic slicing to narrow the search space for root-cause program-state infection, and further, the potential additional benefit of augmenting the dynamic slicing result with dynamic relevance measures.

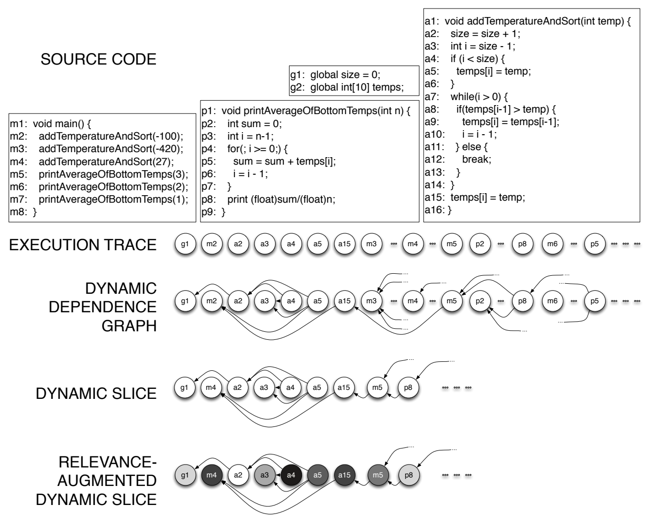
2. APPROACH

In this section, we describe the dynamic-relevance measure and the approach to generate it. The first three steps are shared with traditional dynamic slicing. First, we instrument a program to record an execution trace that captures all instruction-execution instances, along with their reads and writes to memory. Then, using control-dependence and data-dependence analyses, we compute the dynamic dependence graph for the execution trace. Next, we perform a traversal of the graph while tracking memory objects and their definitions and uses. Finally, we perform an analysis to determine the degree to which each instruction instance in the dynamic slice is relevant to the slicing criterion.

Motivating Example. In Figure 1 we present a simple program that simulates three temperature readings from a thermometer sensor that gives one erroneous reading. The program reads the sensor the values, records them, sorts them, and computes the mean temperature of the lowest of one, two, and three readings.

We created and use this simple example program to demonstrate the process and goals of the dynamic relevance measure to augment dynamic slicing. Although the program is simple and the program-model depictions are merely representative, we nevertheless implemented the example program, monitored its execution to gather its execution trace, computed its dynamic dependence graph and dynamic slice, and calculated the dynamic relevance values for each instruction instance. Despite the need to abstract the full trace, graph, slice, and relevance to fit in this paper, we provide the actual artifacts of these analyses online.[2]

As shown in the top part of Figure 1, we start with analyzing the most basic model of program behavior *i.e.,*, the source code of the software program. The actual source code for this example is compose of 74 lines of code (some of which



```
m1:  void main() {
m2:    addTemperatureAndSort(-100);
m3:    addTemperatureAndSort(-420);
m4:    addTemperatureAndSort(27);
m5:    printAverageOfBottomTemps(3);
m6:    printAverageOfBottomTemps(2);
m7:    printAverageOfBottomTemps(1);
m8:  }
```

```
g1:  global size = 0;
g2:  global int[10] temps;
```

```
p1:  void printAverageOfBottomTemps(int n) {
p2:    int sum = 0;
p3:    int i = n-1;
p4:    for(; i >= 0;) {
p5:      sum = sum + temps[i];
p6:      i = i - 1;
p7:    }
p8:    print (float)sum/(float)n;
p9:  }
```

```
a1:  void addTemperatureAndSort(int temp) {
a2:    size = size + 1;
a3:    int i = size - 1;
a4:    if (i < size) {
a5:      temps[i] = temp;
a6:    }
a7:    while(i > 0) {
a8:      if(temps[i-1] > temp) {
a9:        temps[i] = temps[i-1];
a10:       i = i - 1;
a11:     } else {
a12:       break;
a13:     }
a14:  }
a15:  temps[i] = temp;
a16: }
```

SOURCE CODE
EXECUTION TRACE
g1 m2 a2 a3 a4 a5 a15 m3 m4 m5 p2 p8 m6 p5
DYNAMIC DEPENDENCE GRAPH
g1 m2 a2 a3 a4 a5 a15 m3 m4 m5 p2 p8 m6 p5
DYNAMIC SLICE
g1 m4 a2 a3 a4 a5 a15 m5 p8
RELEVANCE-AUGMENTED DYNAMIC SLICE
g1 m4 a2 a3 a4 a5 a15 m5 p8

Figure 1: Models of program behavior at each step.

are condensed in the figure). The source code is processed into an instrumented executable binary that is able to produce execution traces for the software program's execution. The compiled bytecode files contain 22 executable bytecode instructions.

As depicted in Figure 1, execution traces represent a long sequence of instruction-execution instances (represented by nodes in the figure), *i.e.,* each distinct runtime execution of each static instruction from the program, and each static instruction may be executed multiple times throughout the execution. In our example, the execution trace contained 224 instruction-execution instances.

We then analyze the execution trace to deduce dependence relations between the instruction-execution instances for both control-dependence and data-dependence. The result of this analysis is a dynamic dependence graph, as shown in Figure 1, with directed edges between different instruction instances, depicted as nodes. The dynamic dependence graph contains an equal number of nodes as the execution trace — for our example program, this was 224 nodes that represent instruction-execution instances.

In order to find the root-cause of an anomalous output, we compute the backward dynamic slice of that anomalous output. In our example, the anomalous output is represented as a higher-than-usual three-day average, and the root cause is the erroneous thermometer sensor reading. As such, we use the anomalous print instruction instance as the criteria to prune the dynamic dependence graph by performing dynamic slicing. As highlighted in Figure 1, the dynamic slice is smaller, now containing 80 instruction instances (from the original 224 in the trace).

To further differentiate the instruction-execution events within the dynamic slice, we then augment its constituent instruction-execution instances with relevance measures with regard to the anomalous print instruction instance as the slicing criteria. Such relevance measures provide a nuanced view of the dynamic slice, drawing attention to those instruction instances, over others, with high relevance to the anomalous print instruction instance, potentially leading to the reading of the anomalous temperature without investigating each of the 80 undifferentiated instruction instances

[2]http://www.ics.uci.edu/~vpalepu/ase14/approachExample

in the dynamic slice. Figure 1 depicts instruction-execution instances with a higher relevance measures with darker nodes, while those with lower relevance measure are indicated with lighter nodes. When considering all instruction-execution instances in the dynamic slice (*i.e.,* 80 nodes) the actual root-cause thermometer-sensor reading is ranked at position 25 out of all instruction-execution instances. The other highly ranked instruction-execution instances are also transitive dependencies that strongly and distinctively influence the anomalous output, which may also lead to the root cause. Furthermore, when searching among all thermometer readings, the erroneous reading is ranked at the #1 position, and thus the dynamic-relevance measure is able to help pinpoint the root-cause infection.

Computing Dynamic Relevance.
The computation of dynamic relevance between instruction instances relies on the observation that some instruction instances influence fewer instruction instances than others. Consider the instruction instances s_i, t_j and v_k for the static instructions s, t and v, respectively. The instruction instance s_i that exclusively influences the instruction instance t_j, is likely to be more relevant to t_j, than an instruction instance v_k that influences multiple instruction instances; only one of which is t_j. In other words we posit that, the instance s_i, is relevant to t_j, because the sole reason for the execution, s_i, of s is to influence the forthcoming execution, t_j, of the instruction t.

We extend this notion of relevance to instructions instances that are related by direct as well as transitive dynamic influences as follows. Consider the instruction instances s_i and t_j, such that s_i transitively influences t_j. We say that s_i is highly relevant to t_j if s_i distinctively influences (directly or transitively) those instruction instances on which t_j is dependent.

The following steps elucidate the computation of the dynamic relevance of an instruction instance s_i for another instruction instance t_j, where t_j is an instruction instance of interest, and together t_j and s_i are part of the dynamic dependence graph $\mathcal{G}$.

Step 1. We obtain the instructions that directly or transitively influence t_j. A backward dynamic slice with t_j as the slicing criterion, *i.e.,* $\mathcal{S}_{\text{bak}}(\mathcal{G}, t_j)$ will constitute the instructions that influence t_j.

Step 2. We obtain all instruction instances that s_i influences. The forward dynamic slice with s_i as the slicing criterion, *i.e.,* $\mathcal{S}_{\text{fwd}}(\mathcal{G}, s_i)$, will constitute the instructions that s_i influences.

Step 3. To compute the relevance of s_i to t_j, we compare the number of instruction instances in $\mathcal{S}_{\text{fwd}}(\mathcal{G}, s_i)$ (forward slice from s_i) that are also present in $\mathcal{S}_{\text{bak}}(\mathcal{G}, t_j)$ (backward slice from t_j), to the total number of instruction instances that are present in $\mathcal{S}_{\text{fwd}}(\mathcal{G}, s_i)$ (forward slice from s_i).

We denote the relevance of instruction instance s_i for another instruction instance t_j as *DynRel* (s_i, t_j).

$DynRel\,(s_i, t_j) = \frac{|\mathcal{S}_{\text{bak}}(\mathcal{G}, t_j) \cap \mathcal{S}_{\text{fwd}}(\mathcal{G}, s_i)|}{|\mathcal{S}_{\text{fwd}}(\mathcal{G}, s_i)|}$, where,

· t_j is the instruction instance of interest (*i.e.,* the slicing criterion);

· s_i is an instruction instance that has a dynamic influence on t_j (*i.e.,* s_i is included in the backward dynamic slice of t_j);

· $DynRel\,(s_i, t_j)$ is the measure of relevance of s_i for t_j;

· $\mathcal{S}_{\text{bak}}(\mathcal{G}, t_j)$ is the backward slice from t_j;

· $\mathcal{S}_{\text{fwd}}(\mathcal{G}, s_i)$ is the forward slice from s_i;

· $|\mathcal{S}_{\text{bak}}(\mathcal{G}, t_j) \cap \mathcal{S}_{\text{fwd}}(\mathcal{G}, s_i)|$ is the number of instruction instances common to both $\mathcal{S}_{\text{bak}}(\mathcal{G}, t_j)$ and $\mathcal{S}_{\text{fwd}}(\mathcal{G}, s_i)$; and

· $|\mathcal{S}_{\text{fwd}}(\mathcal{G}, s_i)|$ is the number of instruction instances in $\mathcal{S}_{\text{fwd}}(\mathcal{G}, s_i)$

If, however, the instruction instance s_i, whose dynamic relevance measure is being computed, is not a part of the backward dynamic slice with t_j as the slicing criteria, the dynamic relevance measure of s_i is set to 0. In contrast, instruction instances that do not influence (directly or transitively) any other instruction instance that has no influence on the slicing criteria would have a dynamic relevance measure of 1.0, with regard to the slicing criteria.

In summation, such a formulation of dynamic relevance, as stated above, penalizes the dynamic relevance measure for an instruction instance if the instruction instance were to directly or transitively influence any instruction instance that has no forward influence on the slicing criteria. In effect, for an instruction instance s_i, a increasing number of instruction instances that are influenced by s_i and have no influence on the slicing criteria, will decrease the dynamic relevance measure of s_i with regard to the slicing criteria.

3. EVALUATION

We evaluated the degree to which the dynamic-relevance measure can assist in distinguishing relevant execution events. As such, we conducted a controlled experiment on a set of executions for which program state was infected early in an execution, and the first external manifestation of failure (*i.e.,* the output instruction-execution event that first revealed the infection) is selected as the slicing criterion. We then compare the search reduction that is provided by traditional dynamic slicing with that afforded by our dynamic-relevance-augmented dynamic slices by examining the slice in order of decreasing relevance.

3.1 Experimental Setup

The subject of our experiment was the Java program NanoXML (>7,000 LOCs). The dynamic-slicing and relevance technique were implemented in Java and analyzed at the fine-grained bytecode-level.

Because the goal of our experiment is to evaluate the validity of the relevance measures assigned to instruction-execution events in a dynamic slice, we established the ground truth for the targeted root-cause instruction-execution event. To do this, we introduced a data mutation on a small input element within each large XML input to our subject program. We then executed the program with this mutated input, which then infected the program state, propagated throughout the execution of the program, and finally producing mutated output as a result of the program state infection and propagation. For each, we captured the execution trace and recorded the specific instruction-execution instance in the beginning of the execution that input the specific mutated elements. We also monitored the output to determine the first external symptom of the imposed mutation infection — the outputting instruction-execution instance that was responsible for producing that symptom was defined as our slicing criterion. As a result, we establish the evaluation ground truth for which input instruction-execution events are most relevant to the symptom-producing output instruction-execution event.

3.2 Experimental Variables

We assess effectiveness according to the search-space size of identifying the source of the program state infection — in this case, the input read instruction instances responsible for reading the mutated input elements. In our experiment, we control for all variables and vary only the technique (*i.e.,* the *treatment* or the independent variable) to determine the search size (*i.e.,* the dependent variable) to find the ground-truth-established, input instruction-execution instance that originally infected the program state.

As such, we compare three treatments:

- **Execution Trace.** This treatment serves as a baseline upon which the others can be compared. For it, we record the number of input instruction-execution instances throughout each execution, which constitutes the initial search-space of all inputting instructions and all their individual execution instances.
- **Traditional Dynamic Slice.** With this treatment, we determine the extent to which dynamic slicing can reduce the search space for the input instruction-execution instances according to the number of input instructionexecution events by backward slicing from the slicing criterion to all transitively reachable input instances.
- **Dynamic-Relevance-Augmented Dynamic Slice.** With this treatment, we determine the relative dynamic relevance values that we assigned to each input instruction-execution instance, and assess the degree to which we effectively distinguish the ground-truth root-cause instances. For this, we assess the number of input instruction-execution instances that need to be inspected in decreasing order of dynamic relevance.

3.3 Experimental Results

We present the experimental results in Table 1 and Figure 2. Table 1 shows the data for all 20 executions of NanoXML that were infected with mutations in their inputs. Each row in Table 1 presents the information for a single infected execution of NanoXML. Figure 2 depicts the mean values of the data for all 20 executions as shown in last row of Table 1.

For example, for `execution-1` there were a total of 2,993 input instruction-execution instances in the entire execution, out of which only 224 instances (*i.e.,* 7.48%) were identified as relevant for inspection after dynamic slicing. Of these, only 9 input instruction-execution instances (*i.e.,* 0.30%) required inspection when inspecting in order of decreasing dynamic relevance.

In each of the 20 executions, using the additional relevance measure substantially reduced the search for the infection source, as against the number of instruction-execution instances that required inspection as a consequence of being included in the dynamic slice. In the best case of execution-8, dynamic slicing reduced the instruction-execution instances for inspection to 62.4% of the total input instances (*i.e.,* from 362 to 226). In addition, after utilizing the relevance

Execution	# of input instruction-execution instances in trace	# (and %) instances to inspect in traditional dynamic slice		# (and %) instances to inspect in augmented dynamic slice	
execution-1	2993	224	7.48%	9	0.30%
execution-2	2994	245	8.18%	4	0.13%
execution-3	536	106	19.78%	2	0.37%
execution-4	423	93	21.99%	6	1.42%
execution-5	92	80	86.96%	20	21.74%
execution-6	89	64	71.91%	34	38.20%
execution-7	415	282	67.95%	5	1.20%
execution-8	362	226	62.43%	1	0.28%
execution-9	3075	875	28.46%	11	0.36%
execution-10	2964	390	13.16%	13	0.44%
execution-11	2195	246	11.21%	5	0.23%
execution-12	1482	222	14.98%	5	0.34%
execution-13	1501	116	7.73%	5	0.33%
execution-14	1500	445	29.67%	9	0.60%
execution-15	625	212	33.92%	6	0.96%
execution-16	498	178	35.74%	6	1.20%
execution-17	411	266	64.72%	6	1.46%
execution-18	452	283	62.61%	5	1.11%
execution-19	250	148	59.20%	9	3.60%
execution-20	413	273	66.10%	6	1.45%
(Mean)	1163.5	248.7	38.7%	8.35	3.8%

Table 1: Results for 20 mutated executions of NanoXML

measures, exactly 1 instruction-execution instance required inspection, *i.e.,* the relevance measure was able to pinpoint the input instance most relevant among all 226 input instances that dynamic slicing determined truly affected the slicing criterion. Even in the worst case of execution-6, dynamic slicing reduced the search space of input instruction instances to 71.9% of the total input instances, leaving 64 instruction instances to inspect; this was further reduced to 38.2% of the total input instances leaving 34 instruction instances requiring inspection, which is a further reduction of the search by 46.9% from what the dynamic slice provided.

On average, for all 20 executions, dynamic slicing reduced the search space for the input instruction instances to 38.7%, thus providing an average cost savings of 61.3%. In addition, augmenting the dynamic slicing with the relevance measure, reduced the search for the read instructions, on average, to 3.8% of the total number of input instruction instances in the execution, thus providing an average cost savings of 96.2%.

3.4 Discussion

Overall, the results of our evaluation suggest that that augmenting the dynamic slice with relevance measures can substantially reduce the search for instruction-execution instances and thus lead to potential cost savings. The traditional dynamic slice was surprisingly effective at reducing the search space, in terms of the input instruction instances (61% reduction) that required inspection. However, the dynamic relevance information provides even further reduction in the search (96% reduction) to identify the mutated input-execution instance.

It was interesting to us the number of input-execution instances (*i.e.,* between 64 and 875) that dynamic slicing determined to be influencing the infection-revealing output instruction instance (*i.e.,* the slicing criterion). To verify the correctness of our analysis, we performed manual investigation to determine whether the inclusion of the input instruction instances that were reading un-mutated data were

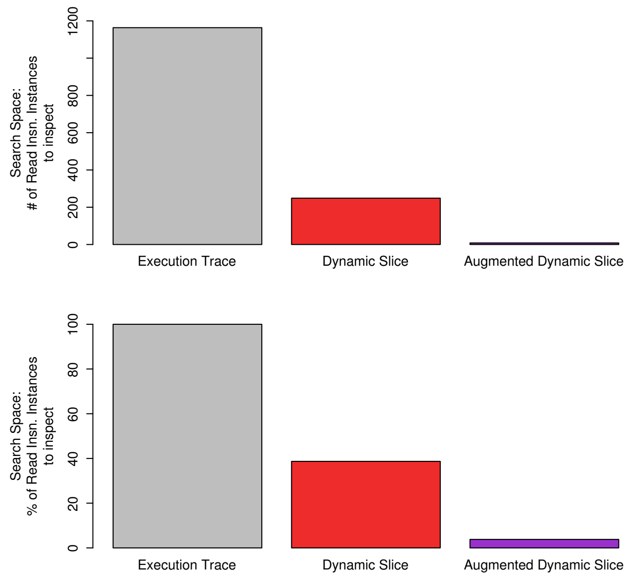


Figure 2: Mean search space to locate the source of program-state infection. (*N.B.,* Lower values are better.)

correctly included in the slices. We found that indeed these reads do influence the control and data dependencies that propagate to the propagated and mutated output state. We found that many of these relationships arise due to the relationships in the structured input (*e.g.,* hierarchical XML elements, attributes of elements, and element types that prescribe certain attributes). Moreover, we found that the most relevant among these according to our dynamic-relevance measure were those that had the most direct relationships in the structural input.

Finally, we note threats to the validity of our evaluation results. Most notably, the results of our evaluation were subject to the behavior of a single software program and thus neither confirm nor reject generalizability, warranting further inquiry with additional software programs as subjects for studying program behavior. Also, we evaluated using one application of dynamic slicing: finding the root-cause infection in the input space. We think that the dynamic relevance measure can be helpful in distinguishing influences in dynamic slices for many other tasks, and those too should be evaluated in the future.

4. RELATED WORK

Dynamic Slicing & Dependence Analysis. Dynamic Slicing was first proposed by Korel and Laski [8], and has inspired various bodies of work ranging from efficient computation of dynamic slices to numerous applications in the field of software engineering. One of the earliest such contributions to dynamic slicing was made by Agrawal and Horgan [1] by first proposing and levering the dynamic dependence graph to compute dynamic slices.

A general description of slicing technology and challenges can be found in Tip's survey [18], Krinke's thesis [9], and with a more recent survey by Xu *et al.* [21]. Works by Binkley *et al.* [3, 4] further provided theoretical frameworks for distinguishing different types of slices.

A noteworthy amount of dynamic slicing research has focused on pruning, compressing and summarizing dynamic slices or dynamic dependence graphs upon which the slicing is performed to aid comprehension (*e.g.,* [7, 11, 14, 15, 19, 20, 22]). An underlying motivation for such works has been to eventually support efficient navigation and inspection of dynamic slices both computationally and for manual inspection by software engineers. Our work is similar to such works as it motivated by the final goal of aiding inspection and navigation of dynamic slices for software engineers. However, in contrast, our technique attempts to achieve improved navigation by deciphering the more important sections in a dynamic slice, instead of pruning or summarizing it. Indeed, it would be interesting to study the effects of summarization and/or pruning in conjunction with our approach.

Gupta *et al.* [6] and Zhang *et al.* [23] employ multiple-points slicing to minimize the program execution search space for localizing faults in a software program. This is similar to our approach as we employ dynamic slices from multiple points and directions to compute relevance measures. However, in contrast, our approach does not minimize the dynamic slice itself; instead it highlights certain relevant points in the slice based on the underlying influences.

Masri and Podgurski [12, 13] present a forward computing approach, called dynamic information flow analysis (DIFA) to track influences within instruction instances during a program's runtime. They use dynamic dependences to compute influence. Further, they measure the strength of information flows modeled by program dependencies using information theory and statistical correlation between empirically observed variable values during software program executions. Our model also captures the influence of instruction instances on each other and moreover computes a measure of relevance for such influences. However, in contrast, our approach analyzes the underlying influences and dependencies themselves to arrive at such relevance measures, instead of using empirically observed values for different variables in a program execution.

Quantified Dependence Models. Recent works have looked at augmenting runtime information with system and program dependence graphs for weighting different elements, *i.e.,* nodes and edges in such dependence models towards improved applicability for various software engineering tasks. Baah *et al.* [2] propose the probabilistic program dependence graph that composes statistical dependencies deduced from test runs to finally enable improved fault localization and comprehension. Santelices *et al.* [17] propose quantitative slicing that weights statements in a static forward slice using differential executions to improve change impact analysis. Deng and Jones [5] propose the weighted system dependence graph that weights the edges of the system dependence graph using test execution data to better enable debugging and program comprehension. Our work is similar to such hybrid models of program behavior, as we weight dependencies between different instruction instances to highlight certain influences between instruction instances over others. However, in contrast to such techniques that are based on static dependence models, our approach models program behavior based on dynamically observed dependencies; and thus, is likely to be more precise.

5. CONCLUSION AND FUTURE WORK

In this work, we presented a novel dynamic dependence and slicing model that incorporates a new measure to quantify dynamic influence among instruction execution events within executions traces and dynamic slices. Our augmented dynamic slice model represents not only the traditional inclusion and identification of influencing instruction instances, but also includes the dynamic dependence relationships among them, as well as a newly created measure that quantifies relationships. Our new measure, which we are calling dynamic relevance, represents the degree of distinctive influence among instruction instances, whether through direct or transitive dependencies.

We conducted evaluations to determine the degree to which traditional dynamic slicing and our new augmented model can reduce the search space for a set of debugging tasks. Our results show strong promise the new augmented model, and lend confidence to the intuitions that gave rise to the dynamic relevance measure. For our experimental subject and protocol, search-space costs were reduced to 3.8% of all read instruction instances within an execution trace, which bested the search-space reduction of traditional dynamic slicing (of 38.7%).

Although these results are promising, future work is needed to further lend confidence in these benefits and to explore the full range of benefits and limitations of all such approaches. First, we will perform larger studies on more subject programs and a variety of software-engineering tasks. Also, we envision future client analyses that can benefit from dynamic relevance measures, and we also envision user interfaces that will allow developers to explore and query dynamic slices.

6. ACKNOWLEDGEMENTS

This work is supported by the National Science Foundation under awards CAREER CCF-1350837 and CCF-1116943.

7. REFERENCES

[1] H. Agrawal and J. R. Horgan. Dynamic program slicing. *SIGPLAN Not.*, 25(6):246–256, June 1990.

[2] G. Baah, A. Podgurski, and M. Harrold. The probabilistic program dependence graph and its application to fault diagnosis. *Software Engineering, IEEE Transactions on*, 2010.

[3] D. Binkley, S. Danicic, T. Gyimothy, M. Harman, A. Kiss, and B. Korel. Minimal slicing and the relationships between forms of slicing. In *Source Code Analysis and Manipulation, 2005. Fifth IEEE International Workshop on*, pages 45–54, Sept 2005.

[4] D. Binkley, S. Danicic, T. Gyimothy, M. Harman, A. Kiss, and L. Ouarbya. Formalizing executable dynamic and forward slicing. In *Source Code Analysis and Manipulation, 2004. Fourth IEEE International Workshop on*, pages 43–52, Sept 2004.

[5] F. Deng and J. A. Jones. Weighted system dependence graph. In *International Conference on Software Testing, Verification, and Validation*, 2012.

[6] N. Gupta, H. He, X. Zhang, and R. Gupta. Locating faulty code using failure-inducing chops. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 263–272, 2005.

[7] M. Kamkar, N. Shahmehri, and P. Fritzson. Interprocedural dynamic slicing. In *Programming Language Implementation and Logic Programming*, volume 631, pages 370–384. 1992.

[8] B. Korel and J. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29:155–163, October 1988.

[9] J. Krinke. *Advanced Slicing of Sequential and Concurrent Programs*. PhD thesis, University of Passau, 2003.

[10] T. D. LaToza and B. A. Myers. Developers ask reachability questions. In *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, volume 1, pages 185–194. IEEE, 2010.

[11] W. Masri, N. Nahas, and A. Podgurski. Memoized forward computation of dynamic slices. In *Proceedings of International Symposium on Software Reliability Engineering*, pages 23–32, 2006.

[12] W. Masri and A. Podgurski. Algorithms and tool support for dynamic information flow analysis. *Information and Software Technology*, 51(2):385–404, 2009.

[13] W. Masri and A. Podgurski. Measuring the strength of information flows in programs. *ACM Trans. Softw. Eng. Methodol.*, 19(2):5:1–5:33, Oct. 2009.

[14] V. K. Palepu and J. A. Jones. Visualizing constituent behaviors within executions. In *Software Visualization (VISSOFT), 2013 First IEEE Working Conference on*, pages 1–4. IEEE, 2013.

[15] V. K. Palepu, G. Xu, and J. A. Jones. Improving efficiency of dynamic analysis with dynamic dependence summaries. In *ASE '13*, pages 59–69, 2013.

[16] R. Santelices, M. J. Harrold, and A. Orso. Precisely detecting runtime change interactions for evolving software. *Software Testing, Verification, and Validation, 2008 International Conference on*, 0:429–438, 2010.

[17] R. Santelices, Y. Zhang, S. Jiang, H. Cai, and Y.-j. Zhang. Quantitative program slicing: separating statements by relevance. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 1269–1272. IEEE Press, 2013.

[18] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[19] C. Wang and A. Roychoudhury. Dynamic slicing on Java bytecode traces. *ACM Trans. Prog. Lang. Syst.*, 30(2):1–49, 2008.

[20] T. Wang and A. Roychoudhury. Hierarchical dynamic slicing. In *Proceedings of the 2007 international symposium on Software testing and analysis*, ISSTA '07, pages 228–238, 2007.

[21] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, pages 1–36, Mar. 2005.

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

[23] X. Zhang, N. Gupta, and R. Gupta. Locating faulty code by multiple points slicing. *Softw. Pract. Exper.*, 37(9):935–961, July 2007.