Tree-Oriented vs. Line-Oriented Observation-Based Slicing

David Binkley*, Nicolas Gold†, Syed Islam‡, Jens Krinke†, and Shin Yoo§
*Loyola University Maryland, 4501 N. Charles St., Baltimore, MD 21210-2699, USA
†University College London, Gower Street, London, WC1E 6BT, UK
‡University of East London, University Way, London E16 2RD
§KAIST, 291 Daehak-ro, Yuseong-gu, Daejeon, 305-701, Republic of Korea

***Abstract*—Observation-based slicing is a recently-introduced, language-independent slicing technique based on the dependencies observable from program behavior. The original algorithm processed traditional source code at the line-of-text level. A recent variation was developed to slice the tree-based XML representation of executable models. We ported the model slicer to source code using srcML to construct a tree-based representation of traditional source code. We present the results of a comparison of the two slicers using four experiments involving seventeen different programs, including classic benchmarks and larger production systems. The resulting slices had essentially the same size and quite often the same content. Where they differ, the use of tree structure traded an ability to remove unnecessary parts of a statement for the requirement of maintaining aspect of the code structure. Comparing the slicers finds that each has its advantages. For example, when the tree representation facilitates the deletion of large chunks of code, the tree slicer was over eight times faster. In contrast, when slicing C++ code it was over nine times slower because of the multitude of small trees created to support C++ syntax. Given the pros and cons of the two, the results suggest the value of their hybrid combination.**

I. Introduction

Observation-based slicing is a recently introduced technique that handles two long-standing slicing challenges: slicing systems consisting of components written in different programming languages and slicing systems that include binary components or libraries [1]. In addition, observation-based slicing obviates the need to replicate much of the compiler's infrastructure (e.g., parsing the code). Instead the approach leverages the existing build tool-chain and thus provides a way to construct a slicer out of the existing build tools. Doing so circumvents the need for costly development of new language-specific toolsets. Operationally, it speculatively deletes part of the code and then observes the program's behavior, committing to the deletion if a desired behavior is still observed.

While similar to dynamic slices, in their reliance on a selected set of inputs, observation-based slices are based on *observed dependencies*, rather than the *statically determined* but dynamically *witnessed* dependencies used by dynamic slicers. That is, a dynamic slice contains a statement if a (statically determined) dependence is witnessed during some execution. By contrast, an observation-based slice contains a statement if its deletion has an observable effect on the slicing criterion.

The original implementation of observation-based slicing processed traditional source code at the line-of-text level. Its direct application to executable models, which are widely used in software engineering as well as other engineering domains to prototype, communicate, reason about, and simulate complex systems, was hampered by the tree-based XML representation of model source code. This led to the creation of an algorithm for observation-based slicing of tree-represented modeling languages [2]. The initial experiment considered executable models written using Mathworks' Simulink [3], which is part of the MATLAB software suite.

This paper compares and contrasts implementations of the two algorithms in the domain of the original algorithm. This is done by transforming traditional source code from lines of text into XML using srcML [4]. Doing so should retain the slicers' language independence (within the limits of the languages supported by srcML) while allowing it to exploit the more natural organization of the source code; for example, deleting the entire body of a function in a single step rather than having to consider each of the function's lines. The remainder of the paper is structured as follows. Section II provides basic slicing definitions including that of observation-based slicing, while Section III describes the two implementations of observation-based slicing. The results are then setup by Section IV, which states the three research questions considered, and Section V, which provides demographics for the systems studied. Results of the empirical comparison are presented in Section VI. Finally, related work is discussed in Section VII and Section VIII summarizes the contributions of the paper.

II. Slicing Definitions

Informally, Weiser defined a slice as a subset of a program that preserves the behavior of the program for a specific slicing criterion. This section briefly describes traditional static and dynamic slicing before considering observation-based slicing.

Static [5] and Dynamic [6] slicing seek to find an executable subset of a program's statements that exhibits the same behavior as the original program for a specified variable at a specified location (referred to as a slicing criterion). A static slice does so *for all possible inputs*, while a dynamic slice does so for a selected set of inputs.

It is interesting to note that while Weiser's original definition of program slicing [5] is based on statement deletion, static and dynamic slicers tend to use dependency analysis to determine which statements cannot be deleted. In contrast observation-

based slicing actually *deletes* statements and then *observes* the behavior at the slicing criterion.

Static and Dynamic Slice: A slice S of program P taken with respect to slicing criterion C (composed of variable v and line l) and set of inputs $\mathcal{I}$ is any executable program with the following two properties:

1) S can be obtained from P by deleting zero or more statements from P.
2) Whenever P halts on input I *from* $\mathcal{I}$ with state trajectory T, then S also halts on input I with state trajectory T' and $\mathit{PROJ}_C(T) = \mathit{PROJ}_C(T')$.

The projection function $\mathit{PROJ}_C(T)$ [5] returns the elements of trajectory T produced at C. For a static slice the set $\mathcal{I}$ is the set of all possible inputs to the program, while for a dynamic slice it is a subset of this set. Usually, the criterion for a dynamic slice explicitly includes $\mathcal{I}$ and is thus given as $(v, l, \mathcal{I})$ denoting variable v at location l for all occurrences in the trajectory, or as $(v_i, l, \mathcal{I})$ where v_i is the ith occurrence of variable v in the trajectory.

Observation-Based Slicing is a recently-introduced alternative to dependence-based program slicing [1]: rather than relying on dependency analysis to identify allowed deletions, observation-based slicing uses observation to preserve the relevant part of the state trajectory. Operationally, it does this by tentatively deleting some portion of the program. Only if the result of the deletion compiles and yields the correct output is the deletion made permanent. Because certain lines are only deletable after other lines have been deleted, multiple passes are performed until a pass performs no deletions. One advantage that observation-based slicing brings is the ability to slice any system where it is possible to delete components and then observe the computation at the criterion.

While similar, the definition of static and dynamic slicing projects elements from the complete state trajectory. In contrast observation-based slicing does not require the complete trajectory. Instead it observes only the relevant values [1]:

Observation-Based Slice: An *observation-based* slice S of a program P taken with respect to slicing criterion $C = (v, l, \mathcal{I})$ composed of variable v, line l, and set of inputs $\mathcal{I}$, is any executable program with the following properties:

1) S can be obtained from P by deleting zero or more components from P.
2) The execution of P for every input I in $\mathcal{I}$ halts and produces a sequence of values $V(P, I, v, l)$ for variable v at line l.
3) The execution of S for every input I in $\mathcal{I}$ halts and produces a sequence of values $V(S, I, v, l)$ for variable v at line l.
4) $\forall_{I \in \mathcal{I}} V(P, I, v, l) = V(S, I, v, l)$.

In practice, the sequence of values produced is observed by injecting a statement that outputs the value of v, just before line l. Furthermore, while the definition of the *components* deleted can simply be "statements" to match the definition used with static and dynamic slicing, it can also be entirely language independent. For example, by deleting *lines of text* or *white-space-delimited tokens*, it is possible to effectively slice multi-language systems [1].

Algorithm 1: Core of the ORBS slicer

ORBS_CORE($S, cl, \mathcal{I}, max\_ws$)
Input: Current slice S, input set $\mathcal{I}$, and maximum deletion window size, max_ws
Output: Updated slice, S

```
(1)  cl ← 1 // for each current line
(2)  while cl ≤ length(S)
(3)      if s_cl ∉ S    // i.e., if s_cl has been deleted
(4)          cl ← cl + 1
(5)          continue
(6)      builds ← False
(7)      for ws = 1 to max_ws
(8)          S' ← S − {s_cl, ..., s_min(length(S),cl+ws−1)}
(9)          B' ← BUILD(S')
(10)         if B' built successfully
(11)             builds ← True
(12)             break
(13)     if builds
(14)         V' ← EXECUTE(B', I)
(15)         if V = V'
(16)             S ← S'
(17)             cl ← cl + ws
(18)     return S
```

III. Two Observation-Based Slicers

This section describes two implementations of observation-based slicing: the original text-line based slicer and the more recent tree-based slicer. The components considered by the original observation-based slicer, ORBS, are lines of text [1]. If source files are formatted with one statement per line, then ORBS can produce 1-minimal statement slices from which it is not possible to delete any single statement, however, it may be possible to delete a combination of multiple statements; consequently the slices are not necessarily n-*minimal*. Unfortunately, finding such slices is computationally intractable.

The core of the ORBS algorithm, shown as Algorithm 1, loops through each undeleted line in S. For each current line cl, the algorithm attempts to delete a sequence of lines up to the maximum windows size, *max_ws* [1]. This enables mutually dependent lines (e.g., opening and closing braces on successive lines) to be deleted. The maximum deletion window size places an upper bound on the number of lines that can be deleted together in one deletion. Higher values offer potentially smaller slices at the cost of increased slicing time. To improve efficiency, ORBS caches results from previous BUILD and EXECUTE steps. If a subsequent build or execution produces a cache hit then the cached result is used.

For example, consider the code segment shown in Figure 1. ORBS cannot produce the minimal slice (i.e., Line 4) by attempting to delete only a single line at a time. While deleting Line 2 alone *is* a legitimate slicing action, Lines 1 and 3 can only be deleted in tandem because deleting only one of

```
1 if (x < 0) {
2   print x;
3 }
4 y = 42; // Slice taken w.r.t. y
```

Fig. 1. Deletion Window Motivation

Algorithm 2: Core of the Tree-ORBS Slicer

T-ORBS_CORE($T, O, \mathcal{I}$)
Input: Current Tree T, the criterion consisting of observer O, and input set $\mathcal{I}$
Output: Updated Tree, T

```
(1)  q ← APPEND(empty_queue, start_node(T))
(2)  while ¬ EMPTY(q)
(3)      c ← DEQUEUE(q)
(4)      T' ← DELETE(T, c)
(5)      V' ← O(T', I)
(6)      if V = V'
(7)          T ← T'
(8)      else
(9)          q ← APPEND(q, CHILDREN(c))
(10) return T
```

them results in a syntax error. ORBS avoids this issue by increasing the deletion window until the result compiles. Using a maximum deletion window size *max_ws* of two or more, ORBS produces the desired slice.

T-ORBS, the second implementation of observation-based slicing, was built to slice Simulink models including any embedded Stateflow, both of which are stored using XML [2]. In the same paper, observation-based slicing was generalized to observational slicing. The original definition as given in Section II compared sequences of values observed during execution. *Observational slicing* generalizes this comparison by introducing an observer O and a matching relation R as part of the criterion. In this paper, only traditional observations and matching are used, and therefore, the term observation-based is used. It also permits the simplified version of T-ORBS core shown as Algorithm 2. Rather than line-by-line, the loop on Line (2) performs a breadth-first tree traversal. During each iteration, T-ORBS attempts to delete the subtree rooted at current node, c. If the resulting system produces the correct sequence of values then c is permanently deleted. Otherwise c's children are placed on the worklist.

The T-ORBS implementation was constructed to slice MATLAB's Simulink models, which are stored using XML [2]. Thus to slice traditional source code such as C or Java code, the code must first be transformed into XML. For this we use srcML [4]. In theory, T-ORBS should be able to slice the resulting XML tree-based source code representation without modification. In practice, this came close to being true. Unlike Simulink's XML representation srcML includes XML name spaces. Thus it was necessary to generalize T-ORBS' command-line arguments to include a name-space specification. The only other change necessary was to transform srcML's output from mixed content, where (source) text is intermixed with tags, to element content. In greater detail, the output from srcML uses mixed content (much like HTML) where an element may contain text and other elements. For example, the <if> tag includes the *text* "if" and several elements including the *element* for the (boolean) condition: <if>if <condition> ... </condition> ... </if>. The transformation to element content moves the "free" text "if" to be an attribute of an element, resulting in the XML <if text="if"> <condition> ... </condition> ... </if>. This transformation avoids ambiguities concerning to which element the intermixed text belongs. The resulting T-ORBS slicer is capable of slicing any language supported by srcML or any other XML creation tool. For example, it was initially developed using C code, but was able to slice C++ and Java code without the need for a single modification.

IV. Research Questions

Prior work [1] compared ORBS with various forms of dynamic slicing, all of which are its 'algorithmic cousins' because they all have common roots in dynamic analysis. Subsequently, ORBS slices were compared to static slices in order to explore the subtleties and limits of static analysis [7]. This paper studies the two implementations of observation-based slicing, ORBS, and T-ORBS, using the following research questions.

RQ1: How do ORBS and T-ORBS slices compare quantitatively? This quantitative question considers the sizes of the slices produced by the two implementations.

RQ2: How do the slices produced by ORBS and T-ORBS compare qualitatively? This qualitative question considers differences in the slices produced by the two implementations.

RQ3: What impact does implementation have on the time taken to compute a slice? This quantitative question asks if T-ORBS' ability to delete large sub-trees pays for its having to consider a multitude of small subtrees (e.g., each token of an expression such as a * b + c). It also compares the scalability of the two implementations by slicing three production systems as well as the impact of programming language on the slicers.

V. Subject Demographics

Our experiments concern the seventeen programs shown in Table I. These are split into four sets, each of which is specifically chosen to help address various aspects of the comparison. The first set includes four widely-studied (tiny) benchmark programs taken from the literature because they have been used to exemplify slicing challenges and techniques. While not large, the programs of the second set are small enough that it is feasible to compute all slices for all computations of scalar values (e.g., values of types int, char, double). The third set includes the three production systems (byacc, ed, and the shell bash) and is used to study the scalability of observation-based slicing. Finally, the fourth set includes two Java programs and one C++ program. It is used to consider the impact of programming language by slicing non-C code.

The first of the tiny programs, sumprod computes the sum and product of the first ten integers. It is commonly used to illustrate slicing's ability to separate the computation of the sum from that of the product. The second tiny program, word count, is shown in Figure 2. It computes the number

TABLE I
SUBJECTS CONSIDERED IN THE EMPIRICAL INVESTIGATION

Program	LoC	SLoC	Slices
Known Semantics			
sumprod	20	16	8
wc	128	70	17
mug	73	62	16
mbe	82	62	12
Exhaustively Sliced			
tcas	185	141	43
schedule2	368	291	78
schedule	465	313	58
totinfo	573	347	54
printtokens2	638	407	75
replace	658	541	309
printtokens	895	569	81
Production Systems			
ed	3 062	2 393	1
byacc	7 760	6 615	1
bash	68 230	48 339	1
Non-C Systems			
Hanoi.java	171	158	1
permutation.java	658	3091	1
concordance.c++	1490	1033	1

```
1  word_count()
2  {
3    while (scanf("%c", &c) == 1)
4    {
5      characters = characters + 1;
6
7      if (c == '\n')
8      {
9        lines = lines + 1;
10     }
11
12     if (isletter(c))
13     {
14       if (inword == 0)
15       {
16         words = words + 1;
17         inword = 1;
18       }
19     }
20     else
21     {
22       inword = 0;
23     }
24   }
25 }
26
27 int isletter(char c)
28 {
29   printf("\norbs: %c\n", c); //slice here
30   if (((c >= 'A' ) && (c <= 'Z'))
31       || ((c >= 'a' ) && (c <= 'z')))
32   {
33     return 1;
34   }
35   else
36   {
37     return 0;
38   }
39 }
```

Fig. 2. The word count program with a printf added to slice with respect to variable c at the top of the function isletter.

of lines, words, and characters in an input text. Its slices are used in many papers on slicing [8], [9], as trivial examples of static slices. It is implicit in all treatments of this example, that the slices are trivial, and present few interesting issues, hence its widespread use as an illustrative example. As we shall see, observation-based slicing reveals that there *are*, in fact, subtleties, even in this simplest of examples.

Third, *the SCAM mug* example, shown in Figure 3, appeared on the souvenir mug given to delegates of the first incarnation of the SCAM conference in Florence, 2001. It has subsequently been used as a 'challenge' example for slicing algorithms [10],

```
1  int mug(int i, int c, int x)
2  {
3     while (p(i))
4     {
5        if (q(c))
6        {
7           x = f();
8           c = g();
9        }
10       i = h(i);
11    }
12    printf("\norbs:%d\n", x); //slice here
13 }
```

Fig. 3. The SCAM'01 Mug Example. Predicates p and q, and function h depend only on their single formal parameter while functions f and g return (unknown) constant values. The key point in this code is that in any terminating execution the final value of x is independent of Line 8: if q(c) is initially false, it remains false and thus x retains its initial value. On the other hand, if q(c) is true one or more times then x will have the value assigned at Line 7. In the latter case, it does not matter how often q(c) is true and thus the assignment at Line 8 does not impact the value of x at Line 12.

```
1  int mbe(int j, int k)
2  {
3     while (p(j))
4     {
5        if (q(k))
6        {
7           k = f1(k);
8        }
9        else
10       {
11          k = f2(k);
12          j = f3(j);
13       }
14    }
15    printf("\norbs:%d\n", j); //slice here
16 }
```

Fig. 4. The Montréal Boat Example. Predicates p and q, and functions f1, f2, and f3 are unshown. They depend only on their single formal parameter. The relevant observation is that in any terminating execution, the computation of k is irrelevant to the computation of j.

due to its subtle semantics and the difficulty in obtaining a minimal slice, even using very sophisticated algorithmic techniques.

The Montréal Boat Example, mbe, shown in Figure 4, was formulated by Sebastian Danicic and John Howroyd during a boat excursion at the 2^{nd} incarnation of the SCAM conference in Montréal, 2002. It was discussed at length at the conference as an example of the subtleties of producing minimal slices [11].

In addition to having been used in prior slicing research [1], [12]–[14], the next set of programs was chosen because it is possible to compute all slices for assignments involving basic scalar types (e.g., ints). Doing so supports the comparison over a large number of slices that have a wide range of complexity (from slices taken with respect to variable initializations all the way through to slices taking with respect to final outputs).

The six remaining systems include three larger systems that are in production use and three systems written in programming language other than C. For example, these include as a real-world case study, the often-used non-trivial application: bash (version 4.2), a Unix shell that is the default on Linux and Mac OS X. The bash source package includes various tools and libraries required to build the executable. The build is complex from a slicing perspective because, during the build, source code is generated from a grammar and the build itself is strongly tied to the target operating system. Together with its size, this makes bash a challenge to statically or dynamically slice (we

are not aware of any non-observational slicer that is capable of slicing bash). These properties make bash an excellent example to study the characteristics of observation-based slicing.

For the larger systems computing all possibles slices is infeasible, and thus a single representative slice of each was chosen. These slices attempt to capture the kind of questions a software engineer working on the program might have. For the first production system, ed, this is the value of *addr_cnt at Line 186 of the file main_loop.c. This line is at the top of the function next_addr() which returns "the next line address from the command buffer." The test suite for ed consists of 80 different input command sequences. From the 80 inputs, we have selected 52 and added three more (smaller) inputs: (1) an empty command sequence, (2) a single command to enable error explanations, and (3) a command to read a file.

For the second production system, byacc, the criteria chosen was the value of the variable symbol at Line 252 of the file lalr.c. This line is at the top of the function map_goto, which "maps a (state, symbol) pair to its numeric representation." The test suite of byacc is a set of 10 different grammar files.

Third, the slicing criterion we chose for bash is the variable val in Line 1393 of file expr.c with the input given by the test file arith.tests. At Line 1393 the result of converting a string to an integer is returned to the caller of the function, which an engineer debugging a value error might employ. It is expected that this function is called frequently while processing the test cases of arith.tests, because these tests focus on the arithmetic functions of bash. This expectation is confirmed by measuring the statement coverage with *gcov*: the conversion function strlong is invoked 80 425 times causing 80 425 occurrences of the criterion in the trajectory.

The slice focuses on the file expr.c, which starts with 1111 executable lines of code. Bash is also unique among the programs considered in that it required a modification to its build steps. The Makefile for bash automatically increments a minor version number. While the increment did not affect the slice, it did interfere with the executable cache because each binary has at least the minor version number difference.

Finally, while in theory observation-based slicing is language independent, in practice it is interesting to consider the impact of several programming languages. Three non-C codes are sliced, two Java programs and one C++ program. A representative slice was used for each of these systems. For the first Java program, Hanoi, the value of nDisks (number of disks to move) at the top of the function SolveTOH (solve Towers of Hanoi) was chosen. This criteria captures the call structure of the code.

The second Java program computes permutations of a string using an algorithm that sorts substrings as arrays of characters. The slicing criterion is the length of the sub-array at the start of the method sortchar. For this subject, the test input is the string "orbs".

Finally, the C++ program concordance, creates an index of back pointers for a set of index terms. The slicing criterion considers the three points that a word is manipulated in the concordance: Word:Word, which creates a new word, Word:incWord, which adds a new text location to a word, and Word:addWord, which adds a new word to the concordance. The test case forms a concordance from a test set of 19 documents totaling 2447 words.

VI. Results

A. RQ1: How do ORBS and T-ORBS slices compare quantitatively?

To answer RQ1, the quantitative slice-size comparison looks at two sets of slices. The first set aims to determine if, like ORBS, T-ORBS can produce the minimal static slices of the tiny, well studied benchmark programs. The second set includes the exhaustively sliced programs. Exhaustive slicing avoids potential experimenter bias when selecting which slices to consider. We constructed 751 slices in total including, for completeness, 53 slices of the known semantics benchmark programs. The expectation here is that T-ORBS will occasionally produce larger slices because it respects the tree structure of the syntax while slicing.

For the semantic challenge benchmarks, mbe and mug, ORBS and T-ORBS produce the same slices with two exceptions. First, there are layout differences (these are removed, for example, by pretty printing). The more interesting exception occurs in two slices where ORBS removes lines of text that are part of an if statement, while T-ORBS retains the predicate and empty true branch. This statement is found on Lines 5-13 of Figure 4. In the slice, k's value does not actually affect the value of j and thus only the assignment to j in the false branch is semantically necessary. This enables ORBS to delete Lines 5-9. The following is the T-ORBS slice:

```
5       if ((k))
6       {
7
8       }
9       else
10      {
11
12          j = f3();
13      }
```

The T-ORBS slice correctly untangles the computations of k and j. However, it retains Lines 5-9 because in the tree of the if statement, the keyword if is part of a parent node that has three subtrees representing the condition, then-part, and else-part of the if statement. Thus its removal is only possible if the entire if statement can be deleted. Future work will consider the possibility of replacing a parent (e.g., the if node) with one of its children (e.g., the else branch in the example). These two exceptions account for the minor average-slice-size differences seen in Table II.

Turning to the 751 slices, Figure 5 graphs slice-size differences. Each point is the size of the ORBS slice minus the size of the T-ORBS slice. Most of the differences hover around zero. For example, 53% (398 or 751) differ by less than 10 lines. ORBS produces smaller slices about 60% of the time. Inspecting a random sample of these slices they are dominated by T-ORBS preserving structure as illustrated in the mbe slice above as well as several slices considered in the discussion of RQ2. On the other hand, T-ORBS produces

smaller slices when ORBS deletes an initialization because of *fortuitous placement*, which later inhibits the deletion of lines elsewhere in the code. For example, consider two subsequent function calls to functions f and then g each having a single local variable, l_f and l_g, respectively. In C, local variables are not automatically initialized and thus end up with the value found in the memory they are assigned. Unless overwritten, the value of l_g during the call to g will be the final value of l_f from the call to f (assuming that the activation records for the two have a similar layouts). If ORBS may delete the initialization of l_g because it fortuitously has the correct value, it will be unable to later delete f and the code that gives l_f its final value because it is needed to maintain the initial value of l_g. While T-ORBS is susceptible to the same issue, it deletes components in a different order and thus can delete f before attempting to delete l_g.

Fig. 5. Slice Size Comparison (positive values indicate that the T-ORBS slice is smaller)

Table II takes a numeric look at slice size. Using the same groups as shown Table I, it presents the average slice sizes produced by the two slicers. The average percent reduction is fairly stable and ranges from just over 50% to almost 80%. There is more variation for programs where only a single slice was taken.

Comparing the two slicers, for most systems their performance is similar. Only three programs show more than a five percentage point difference in the percent reduction. For both wc and printtokens2 this is caused by T-ORBS maintaining structure when ORBS does not. As illustrated in the next section, this can occur because T-ORBS can only remove the predicate of an if statement when it can remove both the then and else branches. ORBS has no such restriction.

Finally, the only time that T-ORBS significantly out performs ORBS is for the single slice of permutation. This is caused by ORBS removing a predicate that has two effects: it causes the function sortchar to be called more often and it causes one of those calls to abort the program. As it turns out, these two offset each other and the slice retains the correct behavior. However, it forces ORBS to retain the code that cause execution to abort. This is a considerable amount of code and thus it attains less of a reduction.

In summary, for RQ1 the slices produced by the two algorithms are similar in size. ORBS was seen to have one structure advantage in that T-ORBS is forced to retain elements to maintain the tree structure with which it represents code.

TABLE II
AVERAGE SLICE SIZES

Program	Original (SLoC)	Average Slice ORBS (SLoC)	Average Slice T-ORBS (SLoC)	Percent Reduction ORBS	Percent Reduction T-ORBS
Known Semantics					
sumprod	16	9.0	9.1	44%	43%
wc	70	15.9	21.9	77%	69%
mbe	62	29.8	30.7	52%	51%
mug	62	20.1	19.8	68%	68%
Exhaustively Sliced					
tcas	141	20.5	21.4	85%	85%
schedule2	291	105.7	101.6	64%	65%
schedule	313	110.7	114.1	65%	64%
totinfo	347	85.6	78.1	75%	77%
printtokens2	407	98.7	142.0	76%	65%
replace	541	193.8	199.4	64%	63%
printtokens	569	212.1	241.8	63%	58%
Production Systems					
ed/main_loop.c	641	269.0	234.0	58%	63%
byacc/lalr.c	546	164.0	149.0	70%	73%
bash/expr.c	1111	685.0	666.0	38%	40%
Non-C Systems					
Hanoi	158	44.0	45.0	72%	72%
permutation	129	98.0	55.0	24%	57%
concordance	1033	225.0	186.0	78%	82%
Average	379	140.4	136.2	63%	64%

B. RQ2: Qualitatively how do the slices produced by ORBS and T-ORBS compare?

RQ2 provides a qualitative look at the slices. The analysis focuses on the tiny programs where knowing the ground truth facilitates comparison. First and foremost, except for the mbe slice described in Section VI-A, like ORBS, T-ORBS computes minimal slices for the challenge problems mug and mbe. And even when not minimal, T-ORBS untangles the complex control and data dependence interactions found in the code. This observation and the representative examples considered in this section point to T-ORBS structure preservation as being the one substantial difference between the two implementations.

Another example where T-ORBS structure preservation is a detriment is in the tcas slice taken with respect to need_downward_RA. It turns out that the test suite includes only tests that make the predicate of the if statement on Line 4 true. ORBS "discovers" this and thus its slice omits the predicate. More importantly it also omits those definitions upon which the predicate depends. Thus the ORBS slice retains only Line 6 of the following code. In contrast, T-ORBS retains all of the code because it cannot remove the predicate of an if statement without removing both its then and else subtrees.

```
1 #define OLEV 600 /* in feets/minute */
2 ...
3   enabled = High_Confidence && (Own_Tracked_Alt_Rate <=
        OLEV) && (Cur_Vertical_Sep > MAXALTDIFF);
4   if (enabled && ((tcas_equipped && intent_not_known) || !
        tcas_equipped))
5   {
6     need_downward_RA = Non_Crossing_Biased_Descend &&
          Own_Above_Threat();
```

A final example of structure preservation being a detriment is T-ORBS' inability to delete the lines #ifdef DEBUG and #endif. In the srcML representation, each of these is a separate subtree and thus T-ORBS cannot remove them, because it attempts to do so one at a time.

In these three examples, T-ORBS use of a tree-based structure causes it to include parent structures (e.g., if statements) when

only a child structure (e.g., the else block) is required, as well as preprocessor directives such as #ifdef. However, the use of a tree-base structure also enables T-ORBS to "dissect" individual lines of text. One example from concordance.cc is the replacement of the lines

```
typedef enum Boolean
{ FALSE = 0, TRUE = 1, FAIL = 0, SUCCEED = 1, OK = 1, NO
    = 0, YES = 1, NOMSG = 0,
  MSG = 1, OFF = 0, ON = 1 } BOOLEAN;}
```

with

```
typedef enum { OK = 1, NO = 0, YES } BOOLEAN;}
```

In a similar example, consider the function header int h(int i) and note that int is C's implicit default type. ORBS is unable to delete the text line containing the function header because the function h is part of the slice. However, T-ORBS reduces this line to h(i), because the srcML for a function includes four subtrees: (return) type, (function) name, parameters, and body:

```
<function>
  <type><name>int</name></type>
  <name>h</name>
  <parameter_list> (<parameter><decl><type><name>int</name
      ></type> <name>i</name> </decl></parameter>)</
      parameter_list>
  <block>{}</block>
</function>
```

A related example occurs when T-ORBS removes parameters because a proceeding call has placed the same value at the correct stack location. For example, in the following code (a fragment of the slice for the Montréal Boat Example in Fig. 4), the call q(k) places k on the stack in the first parameter position, thus the call f1() effectively also passes k to f1() (because k is still on the stack).

```
if (q(k))
{
  k = f1();
  printf("\norbs:%d\n", k); //slice here w.r.t. k
```

A final similar example comes from the SCAM mug example. This program is really a schema as it involves several unspecified constants. In the concrete implementation, these constants are assigned values using command-line arguments such as x = (int) strtol(argv[3], NULL, 10). Because other than degenerate values such as zero, the actual value chosen is uninteresting, various constants were chosen. The ORBS test suite, which include no degenerate values and is thus sufficient to ensure that ORBS produces the minimal static slice was initially used with T-ORBS. T-ORBS replaced the initialization with x = (int) (10). Updating the test suite to include a value other than 10 enabled T-ORBS to generate the expected slice.

The next two examples illustrate a form of "capture" in which ORBS is able to combine parts of different syntactic units. The first if from the sumprod program, the first seven lines of which are as follows:

```
for(i=1; i<=10; i++)
{
  sum = sum + i;
  prod = prod * i;
}
printf("at end i = %d\n", i);
printf("\norbs:%d\n", i); //slice here w.r.t. i
```

T-ORBS produces the expected slice by removing the body of the loop and the first call to printf (Lines 3, 4, and 6). In contrast, with a window-size of four, ORBS deletes Lines 2-5. In the resulting code, the first of the two printf calls gets "captured" by the loop header leading to the following code.

```
1 for(i=1; i<=10; i++)
2   printf("at end i = %d\n", i); // indentation added for
        clarity
3 printf("\norbs:%d\n", i); //slice here w.r.t. i
```

Because this syntactically correct program computes the correct values for i, it is a slice of the original. Here ORBS produces the smaller slice (of only three lines), while T-ORBS produces the more natural slice (the one that preserves more of the original structure).

A second capture example is one of the more interesting ORBS slices where the slice combines statements from two (adjacent) functions. The following code is from the word count program. The slice was taken with respect to the value of c at the top of the function isletter. It just so happens that the same variable name is used by the caller.

```
1  while (scanf("%c", &c) == 1)
2  {
3    if (isletter(c))
4    {
5  ...
6
7  int isletter(char c)
8  {
9    printf("\norbs:%c\n",c); //slice here w.r.t. c
10 ...
```

ORBS discovers that it is possible to merge code from these two functions. The resulting slice includes the while loop from Line 1 and the call to printf from Line 9.

```
1 while (scanf("%c", &c) == 1)
2   printf("\norbs:%c\n",c); //slice here w.r.t. c
```

T-ORBS is unable to produce such a slice because it cannot merge subtrees.

The final example considers a difference caused by the order in which deletions are attempted. The program tcas initializes the variable alt_sep to zero at the top of the function alt_sep_test. The slices taken with respect to this initialization must preserve the call which ORBS does by retaining the entire line fprintf(stdout, "alt_sep_test()). In contrast, T-ORBS reduces this line to (alt_sep_test()), dropping the call to fprintf and replacing it with an expression list. Another interesting aspect of this slice comes from the test suite including a test with insufficient command-line arguments. For this test case no output should be generated. The following is the relevant part of the code.

```
1  if(argc < 13)
2  {
3     fprintf(stdout, "Error: Command line arguments are
          ...\n");
4     exit(1);
5  }
6  ...
7  Climb_Inhibit = atoi(argv[12]);
8  ...
9  fprintf(stdout, "%d\n", alt_sep_test());
```

Because ORBS and T-ORBS involve different deletion orders, T-ORBS retains the if statement and the call exit(1) (but not the call to fprintf on Line 3). ORBS on the other hand deletes Lines 1-5 including the call exit(1). It thus is forced to retain

the call atoi(argv[12]), which causes the program to abort when there are insufficient arguments – effectively preventing the program from calling alt_sep_test(). In this case, again, T-ORBS produces the more natural slice.

In summary, for RQ2 the differences in the slices produced by the two slicers fall into four categories. ORBS produces smaller slices when T-ORBS by its very nature is forced to retain more of the structure of the underlying code. In contrast, T-ORBS naturally performs "sub-line" deletions, which in one case helped to focus an enum on only those entries relevant to the slice. Third, ORBS is more prone to capture lines. While this can produce smaller slices, they are often harder to comprehend. On the other hand, in the final group T-ORBS produces several *more intuitive* slices. It is clear from these examples that each slicer brings pros and cons to the qualitative comparison.

C. RQ3: What impact does implementation have on the time taken to compute a slice?

RQ3 takes a quantitative look at slicing time. In the broad context, the expectation is that T-ORBS will be faster when large chunks of code can be deleted in a single deletion (e.g., an entire function body), but must pay for this as it considers all subtrees. This is particularly costly when a statement is required by the slice and has lots of subtrees. For example, T-ORBS attempts the independent deletion of a, =, b, +, and c from the statement a = b + c.

The experiments actually employ two variants of the ORBS slicer. The original version is used with the smaller C codes while a more recent parallel version, referred to as P-ORBS [15], is used with the production C code and the C++ and Java codes. These implementations were not coded with this comparison in mind, so the analysis necessarily focuses on larger trends. For example, we largely ignore cases where the difference is less than a factor of two. The implementation of P-ORBS is similar to that of ORBS except that it attempts to delete a set of windows sizes in parallel and then selects the largest deletion that compiled and produced the correct execution semantics. This enables the slicer to more quickly delete large blocks of lines, but it also uses considerably more user time because of the multiple deletion attempts.

Table III shows the CPU and wall-clock times for the seventeen programs of Table I. Looking at these times several patterns are evident. For example, in general ORBS takes less user time than T-ORBS, but, as expected, the same is not true of P-ORBS. The six programs where T-ORBS consumes less user time include the two semantic-challenge problems mug and mbe, the two largest systems byacc and bash, and the two Java programs Hanoi and permutation. For the semantic-challenge problems, T-ORBS takes about half the user time and one third of the wall-clock time relative to ORBS. These problems might be described as having *dense semantics*, which from ORBS perspective means that it tends to have to use smaller window sizes and thus attempt more deletions. Of the remaining four programs, only byacc shows a notable difference where T-ORBS computes the slice over eight times faster. In this case T-ORBS ability to remove large portions of code in a single deletion has its greatest impact.

TABLE III
SLICE TIMES (SMALLER TIMES SHOWN IN **bold**

	ORBS		T-ORBS	
Program	User Time (h:m:s)	Wall Clock (h:m:s)	User Time (h:m:s)	Wall Clock (h:m:s)
sumprod	**0:07**	**0:49**	0:08	1:03
wc	**0:24**	3:43	0:43	**3:26**
mbe	1:40	16:16	**0:54**	**5:41**
mug	1:42	18:08	**1:05**	**5:30**
tcas	**4:54**	30:09	8:57	**28:20**
schedule2	**23:19**	3:58:19	1:36:59	**3:42:21**
schedule	**1:11:22**	**3:15:01**	1:38:21	3:32:12
totinfo	**28:00**	**1:27:31**	1:12:14	3:26:45
printtokens2	**44:36**	7:52:25	2:19:48	**6:54:08**
replace	**7:14:56**	127:42:28	33:21:21	**56:40:22**
printtokens	**2:12:27**	25:31:03	6:07:44	**15:43:35**
	P-ORBS		T-ORBS	
ed	**1:08:22**	**1:11:53**	1:32:21	2:08:54
byacc	1:08:19	2:45:10	**8:23**	**21:17**
bash	2:33:03	**1:10:28**	**2:25:17**	2:41:03
Hanoi.java	15:56	**4:25**	**13:58**	6:55
permutation.java	15:18	**3:48**	**12:47**	7:18
concordance.c++	**10:00**	**16:16**	1:33:45	1:45:47

Considering the wall-clock times, ORBS has rather low CPU utilization. In the worst case, for schedule2, its CPU utilization is only 13.6%. For this program T-ORBS manages 45.5%. However as seen at in the bottom of the table, P-ORBS is able to put the processor to better use. These observations suggest that T-ORBS will see less advantage from parallelization given it is already making better use of the processor. An added challenge here is that it is less clear how one might parallelize T-ORBS.

An interesting utilization-related language effect is seen when slicing the Java programs. T-ORBS reports an average CPU utilization of 197% for the two Java systems while P-ORBS reports an average of 397%. P-ORBS highest values on the other C and C++ programs is 106%, while ORBS' best is 46% and T-ORBS' is 87%. Independently executing the Java programs it appears that at least some of the parallelism is coming from the JVM, likely at startup.

Another language effect also evident is that T-ORBS gets bogged down near the leaves of the tree. T-ORBS has particularly poor performance on the C++ program concordance. Compared to C and Java, C++ programs tend to involve denser low-level syntax.

In summary the investigation into RQ3 involving the impact of slicer implementation strategy on slice time suggests trends related to scalability and the impact of programming language. For one of the three larger systems, T-ORBS sliced dramatically faster. In contrast, for the C++ code it was dramatically slower. Finally, the Java slice times hint that T-ORBS would benefit from greater use of parallelism.

D. Summary

From the examples presented to study RQ2 and the data considered to address RQ1 and RQ3, is clear that each slicer has its own pros and cons. In general, the two produce similar slices where T-ORBS slices can be slightly larger because

they must maintain the XML tree structure. However, T-ORBS' larger slices are often the more intuitive option. On the other hand, T-ORBS can perform "sub-line" deletion, which as shown in Section VI-B, can be both a blessing and a curse. Finally, for one of the three production systems T-ORBS was dramatically faster, however, for the C++ program it was dramatically slower.

VII. Related Work

Static slicing was introduced by Weiser [16]. Ottenstein and Ottenstein [17] proposed that program slicing can be viewed as a graph reachability problem and noted that the program dependence graph (PDG) was the ideal structure for program slicing. Horwitz et al. [18] introduced an algorithm which extended the idea to slice entire programs (represented as System Dependence Graphs) and later [19] introduced a two-pass static slicing algorithm. This approach remains the most pre-dominantly used and variants are widely researched.

There are many other flavours of static slicing that attempt the reduce the size of the slice. Incremental Slicing [20] allows selection of the type of data dependencies that are to be included in a slice. Stop-list slicing [21] allows the programmer to define variables that are not of interest, which is used to purge the dependence graph before computing slices causing the slice to be smaller. Barrier Slicing [22] allows the programmer to specify which parts of the program can be traversed when constructing the slice and which parts cannot. A barrier is specified with a set of nodes (or edges) of the PDG that cannot be passed during the graph traversal, also resulting in focused and smaller slices.

Amorphous Slicing [23] is another approach that aims to preserve the semantics of the program but not the syntax. Amorphous slices use transformation to simplify programs, preserving the semantics of the program with respect to the slicing criterion. Although ORBS only deletes lines of code, this may cause merging and this could be regarded as a form of (very slightly) amorphous slicing (depending on the precise interpretation of the phrase 'syntax preserving').

To our knowledge no other slicing approach follows the observation-based statement-deletion approach used by ORBS. The ORBS algorithm [1] is a dynamic form of slicing but constructs slices using dynamically *observed* dependencies, rather than dynamically *occurring* (but statically determined) dependence (used in all other dynamic slicing approaches).

Dynamic slicing is a concept introduced by Korel and Laski [6], [24]. They considered several algorithms to compute dynamic slices based on their definition. In contrast, most later work on dynamic slicing 'defines' dynamic slicing based on the algorithms used to compute it (e.g., Agrawal et al. [25] and Demillo et al. [26]). Although many research prototypes and approaches exist [27]–[33], all approaches are for a single specific programming language whereas the observation based nature of ORBS allows it to slice programs constructed from multiple programming languages [1]. Of all previous dynamic slicing formulations, the closest to our observation-based slicing is Critical Slicing [26]. However, we have found that critical slices are significantly larger than observation-based slices and are often incorrect [1].

The idea to delete parts of a program or test input is also prominent in applications of delta debugging [34]–[36]. As delta debugging can be very expensive, some approaches have modified the original delta debugging formulation, so that it exploits programming language syntax and semantics. For example, Hierarchical Delta Debugging [37] exploits tree structures for a tree-based delta debugging approach. Delta [38] uses a separate tool to flatten tree structures found in programs before applying delta debugging. Regehr et al. [39] exploit the syntax and semantics of C for four delta-debugging based algorithms to minimize C programs that trigger compiler bugs.

Jiang et al. [40] presented a forward dynamic slicing approach similar in spirit with ORBS. They mutate the value of the variable at the location as given by the slicing criterion. They then observe the computed values in the state trajectory and the dynamic slice consist of all statements for which the computed values have changed compared to the trajectory of the original program. However, their forward dynamic slicing suffers from low recall of what they call *dynamic semantic dependencies* which can have serious effects on impact analysis.

Finally, union slicing [41] is also related to observation based slicing. Like ORBS, a union slice aims to approximate a static slice by unioning dynamic slices for a set of test inputs. However, union slicing shares the critical difference between dynamic and observation-based slicing: The dependencies considered by union slicing are dynamically *occurring* (but statically determined) dependencies, rather than dynamically *observed* dependencies.

VIII. Conclusion

Observation-based slicing is a new form of slicing in which dependencies observed during execution are used to construct slices. Previous work has compared observation-based slicing to traditional slicing techniques. It has also looked at applying the original observation-based slicing algorithm in alternate domains such as visual languages [42] and modeling languages [2]. The development of a slicer for modeling languages led to the creation of an observation-based slicer that worked with XML trees rather than lines of text. This paper, in essence, closes the loop, by re-targeting the XML tree slicer to source code using srcML to transform source code into XML.

ORBS uses statement deletion as its primary operation while T-ORBS uses sub-tree deletion. Both use observation as their validation criteria. The comparison of the two slicers, ORBS and T-ORBS, helps to better understand the pros and cons of each and thus in which applications each might be preferred. Overall, we believe that our results hint at the rich diversity of possible *language-independent* slicers and slicing strategies, and thus opens the door for the study of the impact these variations have when it comes to providing a natural complement to existing slicing techniques.

Looking forward, the results presented in this paper suggest several directions for future work. First, the application of

T-ORBS to traditional source code would benefit from some notion of scale. For example, early iterations might skip subtrees that fail to meet some requirement such as representing at least k lines (or k characters) of code. The retention of if statements when only one branch are needed by the slice suggests a "re-parenting" transformation in which, rather then deleting the subtree rooted at a node, the node is replaced by one of its required descendants.

Finally, it may be possible to combine and thus exploit the advantages of ORBS and T-ORBS. For example, by having T-ORBS make a pass over the code only considering subtrees that represent "large" amounts of code would enable the quick deletion of large blocks. This could be followed by one or more ORBS passes, which could delete elements that T-ORBS can't, such as #ifdef (because directives are each in their own subtree, T-ORBS never deletes matching pairs of #ifdef / #endif). Then a final T-ORBS pass that considers subtrees that represent only "small amounts of code," which would serve to simplify existing lines such as the typedefs simplification described in Section VI-B.

IX. Acknowledgements

A special thanks to Mark Harman for many interesting conversations on the use of observational slicing. Dave Binkley is supported by NSF grant 1626262.

References

[1] D. Binkley, N. Gold, M. Harman, S. Islam, J. Krinke, and S. Yoo, "ORBS: Language-independent program slicing," in *Proc. 22nd ACM SIGSOFT Intl. Symposium on Foundations of Software Engineering*, 2014.

[2] N. E. Gold, D. Binkley, M. Harman, S. Islam, J. Krinke, and S. Yoo, "Generalized observational slicing for tree-represented modelling languages," in *Proc. 25nd ACM SIGSOFT Intl. Symposium on Foundations of Software Engineering*, 2017.

[3] The Mathworks Inc. (2016) Simulink. Accessed 21 July 2016. [Online]. Available: http://uk.mathworks.com/products/simulink/

[4] M. Collard, "Addressing source code using srcml," in *IEEE International Workshop on Program Comprehension Working Session (IWPC'05)*, 2005.

[5] M. Weiser, "Programmers use slices when debugging," *Communications of the ACM*, vol. 25, no. 7, 1982.

[6] B. Korel and J. Laski, "Dynamic program slicing," *Information Processing Letters*, vol. 29, no. 3, 1988.

[7] D. Binkley, N. Gold, M. Harman, S. Islam, J. Krinke, and S. Yoo, "ORBS and the limits of static slicing," in *Intl. Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2015.

[8] K. B. Gallagher and J. R. Lyle, "Using program slicing in software maintenance," *IEEE Transactions on Software Engineering*, vol. 17, no. 8, 1991.

[9] T. Reps and T. Turnidge, "Program specialization via program slicing," in *Dagstuhl Seminar on Partial Evaluation*, O. Danvy, R. Glück, and P. Thiemann, Eds., vol. 1110, 1996.

[10] M. Ward, "Slicing the SCAM mug: A case study in semantic slicing," in *Intl. Workshop on Source Code Analysis and Manipulation (SCAM)*, 2003.

[11] S. Danicic and J. Howroyd, "Montréal boat example," in *Source Code Analysis and Manipulation (SCAM 2002) conference resources website*, 2002. [Online]. Available: http://www.ieee-scam.org/2002/Slides_ct.html

[12] D. Binkley, M. Harman, Y. Hassoun, S. Islam, and Z. Li, "Assessing the impact of global variables on program dependence and dependence clusters," *Journal of Systems and Software*, vol. 83, no. 1, 2009.

[13] M. Harman, D. Binkley, K. Gallagher, N. Gold, and J. Krinke, "Dependence clusters in source code," *ACM Transactions on Programming Languages and Systems*, vol. 32, no. 1, pp. 1:1–1:33, 2009.

[14] D. Binkley, R. Capellini, L. Raszewski, and C. Smith, "An implementation of and experiment with semantic differencing," in *Proceedings of the 2001 IEEE International Conference on Software Maintenance*, November 2001, pp. 82–91.

[15] S. Islam and D. Binkley, "PORBS: A parallel observation-based slicer," in *24th International Conference on Program Comprehension (ICPC)*. IEEE, 2016, pp. 1–3.

[16] M. Weiser, "Program slicing," in *Proc. of the 5th Intl. Conf. on Software Engineering*, 1981.

[17] K. J. Ottenstein and L. M. Ottenstein, "The program dependence graph in software development environments," in *Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environment*, 1984.

[18] S. Horwitz, T. Reps, and D. W. Binkley, "Interprocedural slicing using dependence graphs," in *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 1988.

[19] ——, "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 1, 1990.

[20] A. Orso, S. Sinha, and M. J. Harrold, "Incremental slicing based on data-dependences types," in *Proc. of the IEEE Intl. Conf. on Software Maintenance (ICSM)*, 2001.

[21] K. B. Gallagher, D. Binkley, and M. Harman, "Stop-list slicing," in *Intl. Workshop on Source Code Analysis and Manipulation (SCAM)*, 2006.

[22] J. Krinke, "Barrier slicing and chopping," in *Intl. Workshop on Source Code Analysis and Manipulation (SCAM)*, 2003.

[23] M. Harman and S. Danicic, "Amorphous program slicing," in 5^{th} *IEEE International Workshop on Program Comprenhesion (IWPC)*, 1997.

[24] B. Korel and J. Laski, "Dynamic slicing in computer programs," *Journal of Systems and Software*, vol. 13, no. 3, 1990.

[25] H. Agrawal and J. R. Horgan, "Dynamic program slicing," in *Proc. of the ACM SIGPLAN'90 Conf. on Programming Language Design and Implementation (PLDI)*, 1990.

[26] R. A. DeMillo, H. Pan, and E. H. Spafford, "Critical slicing for software fault localization," in *Proc. of the Intl. Symposium on Software Testing and Analysis (ISSTA)*, 1996.

[27] A. Beszedes, T. Gergely, Z. M. Szabó, J. Csirik, and T. Gyimothy, "Dynamic slicing method for maintenance of large C programs," in *Proc. of the 5th Conf. on Software Maintenance and Reengineering*, 2001.

[28] A. Beszedes, T. Gergely, and T. Gyimóthy, "Graph-less dynamic dependence-based dynamic slicing algorithms," in *Intl. Workshop on Source Code Analysis and Manipulation (SCAM)*, 2006.

[29] G. Mund and R. Mall, "An efficient interprocedural dynamic slicing method," *Journal of Systems and Software*, vol. 79, no. 6, 2006.

[30] A. Szegedi and T. Gyimóthy, "Dynamic slicing of Java bytecode programs," in *Intl. Workshop on Source Code Analysis and Manipulation (SCAM)*, 2005.

[31] X. Zhang and R. Gupta, "Cost effective dynamic program slicing," in *Proc. of the ACM SIGPLAN 2004 Conf. on Programming Language Design and Implementation*, 2004.

[32] X. Zhang, N. Gupta, and R. Gupta, "A study of effectiveness of dynamic slicing in locating real faults," *Empirical Software Engineering*, vol. 12, no. 2, 2007.

[33] S. S. Barpanda and D. P. Mohapatra, "Dynamic slicing of distributed object-oriented programs," *IET software*, vol. 5, no. 5, 2011.

[34] A. Zeller, "Yesterday, my program worked. today, it does not. Why?" in *European Software Engineering Conf. and Foundations of Software Engineering*, 1999.

[35] H. Cleve and A. Zeller, "Finding failure causes through automated testing," in *Intl. Workshop on Automated Debugging*, 2000.

[36] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, 2002.

[37] G. Misherghi and Z. Su, "HDD: hierarchical delta debugging," in *Proc. of the 28th Intl. Conf. on Software Engineering (ICSE)*, 2006.

[38] S. McPeak, D. S. Wilkerson, and S. Goldsmith. Delta. [Online]. Available: http://delta.tigris.org

[39] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for C compiler bugs," in *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2012.

[40] S. Jiang, R. Santelices, M. Grechanik, and H. Cai, "On the accuracy of forward dynamic slicing and its effects on software maintenance," in *Intl. Working Conf. on Source Code Analysis and Manipulation (SCAM)*, 2014.

[41] Á. Beszédes, C. Faragó, Z. M. Szabó, J. Csirik, and T. Gyimóthy, "Union slices for program maintenance," in *Proc. of the 18th Intl. Conf. on Software Maintenance (ICSM)*, 2002.

[42] S. Yoo, D. Binkley, and R. D. Eastman, "Seeing is slicing: Observation based slicing of picture description languages," in *Intl. Workshop on Source Code Analysis and Manipulation (SCAM)*, 2014, pp. 175–184.