ORBS: Language-Independent Program Slicing

David Binkley* Nicolas Gold† Mark Harman†
Syed Islam† Jens Krinke† Shin Yoo†

*Loyola University Maryland
Baltimore, USA

†University College London
London, UK

ABSTRACT

Current slicing techniques cannot handle systems written in multiple programming languages. Observation-Based Slicing (ORBS) is a language-independent slicing technique capable of slicing multi-language systems, including systems which contain (third party) binary components. A potential slice obtained through repeated statement deletion is validated by *observing* the behaviour of the program: if the slice and original program behave the same under the slicing criterion, the deletion is accepted. The resulting slice is similar to a dynamic slice. We evaluate five variants of ORBS on ten programs of different sizes and languages showing that it is less expensive than similar existing techniques. We also evaluate it on `bash` and four other systems to demonstrate feasible large-scale operation in which a parallelised ORBS needs up to 82% less time when using four threads. The results show that an ORBS slicer is simple to construct, effective at slicing, and able to handle systems written in multiple languages without specialist analysis tools.

Categories and Subject Descriptors

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

General Terms

Algorithms, Experimentation, Measurement

Keywords

Program Slicing, Delta Debugging

1. INTRODUCTION

Since Weiser introduced program slicing [40] hundreds of papers and research prototypes have appeared. Despite significant and sustained work, two long-standing challenges remain open: how to slice heterogeneous programs consisting of components written in different programming languages and how to slice systems that include binary components or libraries. Although one may be able to slice the components of the system written in a particular single language, the resulting slice has limited utility because one either has to ignore effects of the components written in other languages or use worst-case assumptions. It is typically too complicated to construct slicers for several languages and combine their analyses.

FSE '14, November 16–22, 2014, Hong Kong, China

Slicing has many proposed applications, including testing [7, 18], debugging [23, 42], maintenance [15, 16], re-engineering [8], re-use [3, 9], comprehension [11, 22, 39] and refactoring [14]. However, full development of these applications will not occur unless slicing can be used to slice programs written in multiple languages. To the authors' knowledge, no tool exists that can slice a such system.

Consider the code shown in Figure 1, which consists of three components: a Java and a C program connected by some logic implemented in Python. Assume you want to know which statements can influence the value of the variable `dots` just after Line 13 of `checker.java`. A slice would capture the set of influencing statements, but almost no program analysis approach can handle such systems without requiring complex abstract language models [31] or restrictions to particular fixed groups of languages [29].

The number of languages in a software system is often between two and fifteen [19, pp. 504-505] and the costs of development and maintenance rise with the number of languages. A multi-language approach is likely to have much greater impact and utility than one for a single-language alone, and since neither traditional static or dynamic slicing is suitable, an alternative approach is needed.

We present a technique that can compute slices for heterogeneous systems. In addition to handling multiple languages, it removes the need to replicate much of the compiler's work (e.g., parsing the code being analyzed) when developing a slicing system by leveraging the existing build tool-chain instead. Our approach thus provides a way to construct a slicer out of the build tools already being used by most programmers, rather than requiring the costly development of a new language-specific toolset. Figure 2 shows the slice for the above example as computed using our implementation.

Weiser defined a slice as a subset of a program that preserves the behaviour of the program for a specific criterion. Although he defined the slice subset in terms of statement deletion, most slicers compute slices by analysing dependencies to establish which statements must be retained. Our approach actually *deletes* statements, *executes* the candidate slice, and *observes* the behaviour for a given slicing criterion, i.e., it observes a variable's state at a location as given by the criterion. The resulting slices have the same observed behaviour for the criterion as the original program. Because we compute slices for a specific set of executions and inputs, our slices are similar to dynamic slices. However, dynamic slicing typically uses dependence analysis to extract some information from an execution of a program to decide which statements should be retained to form the slice (execute–observe–select). In contrast, our approach relies on observing the actual behaviour after deleting statements (delete–execute–observe) and is thus called *observation-based slicing*.

There are two program reduction techniques that operate in a similar deletion-oriented way: Critical Slicing [12] and delta-debugging [45] based approaches (e.g., STRIPE [10] or Delta [26]).

checker.java:

```
class checker {
  public static void main(String[] args) {
    int dots = 0;
    int chars = 0;
    for (int i = 0; i < args[0].length(); ++i) {
      if (args[0].charAt(i) == '.') {
        ++dots;
      } else if ((args[0].charAt(i) >= '0')
                && (args[0].charAt(i) <= '9')) {
        ++chars;
      }
    }
    System.out.println(dots); // Slice here
    System.out.println(chars);
  }
}
```

reader.c:

```
#include <stdlib.h>
#include <stdio.h>
#include <locale.h>

int main(int argc, char **argv) {
  setlocale(LC_ALL, "");
  struct lconv *cur_locale = localeconv();
  if (atoi(argv[1]))
  {
    printf("%s\n", cur_locale->decimal_point);
  }
  else
  {
    printf("%s\n", cur_locale->currency_symbol);
  }
  return 0;
}
```

glue.py:

```
# Glue reader and checker together.
import commands
import sys

use_locale = True
currency = "?"
decimal = ","

if use_locale:
  currency = commands.getoutput('./reader 0')
  decimal = commands.getoutput('./reader 1')

cmd = ('java checker ' + currency
      + sys.argv[1] + decimal + sys.argv[2])
print commands.getoutput(cmd)
```

Figure 1: Example Multi-Language Application

These techniques might also be considered as candidates for the observation-based slicing we propose and will be discussed in detail. The comparative study in Section 5 shows that Critical Slicing often produces incorrect slices (while observation-based slices are always correct by construction) and observation-based slicing using delta debugging is too expensive for practical use.

The contributions of this paper are

- A language-independent algorithm, ORBS, for computing observation-based slices in a serial and a parallel version,
- Empirical studies that demonstrate the application, operation, and comparability of the approach,
- An in-depth case study that explores and illustrates characteristics of our approach, and
- A parallelised implementation of ORBS that significantly decreases the runtime.

checker.java:

```
class checker {
  public static void main(String[] args) {
    int dots = 0;
    for (int i = 0; i < args[0].length(); ++i) {
      if (args[0].charAt(i) == '.') {
        ++dots;
      }
    }
  }
```

reader.c:

```
#include <locale.h>
int main(int argc, char **argv) {
  struct lconv *cur_locale = localeconv();
  {
    printf("%s\n", cur_locale->decimal_point);
  }
}
```

glue.py:

```
import commands
import sys
use_locale = True
currency = "?"
if use_locale:
  decimal = commands.getoutput('./reader 1')
cmd = ('java checker ' + currency
      + sys.argv[1] + decimal + sys.argv[2])
print commands.getoutput(cmd)
```

Figure 2: Sliced Example from Fig. 1

2. SLICING DEFINITIONS

Program slicing is classified as either static or dynamic: A static slice considers all possible executions while a dynamic slice considers specific executions. We will show how to derive observation-based slicing from the traditional forms of dynamic slicing.

2.1 Traditional Slicing

Static slicing [41] computes a subset of a program such that executing the subset will have the same behaviour for a specified variable at a specified location (the slicing criterion) as for the original program *for all possible inputs*.

Dynamic slicing [20] uses a *specific input* and only preserves the behaviour for that input. Most work on dynamic slicing (e.g., the work of Agrawal and Horgan [1]) offers only a description rather than a definition of the term. Thus there exist many different formulations of dynamic slicing, relating to the particular technique being reported to compute the slices, rather than to a general definition. We use a generalised definition of dynamic slicing that involves a state trajectory and a projection function, *PROJ* [41]. Informally each state in a trajectory gives the value of each of the program's variables, while the projection function extracts those values relevant to a slicing criterion. The generalised definition of a dynamic slice is based on Weiser's definition of a static slice [41] (additions are shown in italics). This definition is similar to Korel and Laski's [20] definition:

Dynamic Slice: A *dynamic* slice S of a program P on a slicing criterion C *and for 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 $PROJ_C(T) = PROJ_C(T')$, where $PROJ_C$ is the projection function associated with criterion C.

The projection function and the criterion define the type of dynamic slicing. Usually, the criterion for a dynamic slice includes the inputs $\mathcal{I}$ and is given as $(v_i, l, \mathcal{I})$ denoting variable v at location l for the ith occurrence in the trajectory. However, this can also be specified as $(v, l, \mathcal{I})$ denoting variable v at location l for all occurrences in the trajectory.

Naturally, one is interested in the smallest slice possible. A slice is considered to be *minimal* if no further statements can be removed from it. There may be more than one minimal slice for a given program and slicing criterion [40].

2.2 Observation-Based Slicing

The definition (rather than the usual approach to the *computation*) of a dynamic slice holds the key to solving the challenges of multi-language slicing; slicing by actually *deleting* statements in a file of interest and *executing* the program to *observe* if the projected trajectory changes. Slicing operations need only take place in the source file(s) of interest: the remainder of the system (i.e., binary components or other source files) can remain untouched.

For the slice taken with respect to a given slicing criterion (i.e., a variable at a specified location for a specific input), the observation-based slicing tool

- must capture the state trajectory for the slicing criterion (all other elements of the trajectory are removed),
- must be able to delete statements from the components of interest, and
- may leave intact other components, including binary components and other source files.

Using the observation-based slicing approach one can slice any system where statements can be deleted from the components of interest, the component containing the criterion can be instrumented to capture the (projected) trajectory for the criterion, and the system can be built and executed with the modified components. In one sense, observation-based slicing can be seen as drawing on the kind of ad-hoc behaviours of developers when debugging: through formalisation we gain reliability and repeatability.

Observation-based slicing is based on preserving the relevant part of the state trajectory from the execution of an original program P.

Observation-Based Slice: An *observation-based* slice S of a program P on a 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. 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.
2. S can be obtained from P by deleting zero or more statements from P.
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. $\forall_{I \in \mathcal{I}} V(P, I, v, l) = V(S, I, v, l)$.

We call the sequences V *trajectories* for the criterion $(v, l, \mathcal{I})$. An implementation of observation-based slicing produces $V(P, I, v, l)$ by injecting, just before line l, a statement that captures the value of v and writes it to a file. The comparison of the trajectories restricts the observable variables somewhat: Their values must be comparable between different runs (requiring serialisation for objects, for example).

Algorithm 1: ORBS

ORBSlice$(P, v, l, \mathcal{I}, \delta)$
Input: Source program, $P = \{p_1, \ldots, p_n\}$, slicing criterion, $(v, l, \mathcal{I})$, and maximum deletion window size, δ
Output: A slice, S, of P for $(v, l, \mathcal{I})$

```
(1)  O ← Setup(P, v, l)
(2)  V ← Execute(Build(O), I)
(3)  S ← Reverse(O)
(4)  repeat
(5)      deleted ← False
(6)      i ← 1
(7)      while i ≤ length(S)
(8)          builds ← False
(9)          for j = 1 to δ
(10)             S' ← S − {s_i, ..., s_min(length(S),i+j−1)}
(11)             B' ← Build(Reverse(S'))
(12)             if B' built successfully
(13)                 builds ← True
(14)                 break
(15)         if builds
(16)             V' ← Execute(B', I)
(17)             if V = V'
(18)                 S ← S'
(19)                 deleted ← True
(20)         else
(21)             i ← i + 1
(22) until ¬deleted
(23) return Reverse(S)
```

We are interested in minimal subsets, thus we delete as many statements from P as possible such that the subset is still an observation-based slice, i.e. it is not possible to delete another statement. However, we are not aiming at finding the globally smallest possible subset (global minimum) as this search becomes computationally intractable.

We need our concept of a 'statement' to be language independent. Therefore, we delete lines from a source file and assume that the source files are formatted in such a way that there is no more than one statement on a single line. When this assumption is violated, the ORBS slicer will still produce slices, but may produce larger slices since the granularity of deletable objects will be coarser (see Section 8 for a more detailed discussion). Usually, a source code beautifier or formatter can be used to split lines that contain multiple statements.

3. ORBS

Our algorithm for observation-based slicing, ORBS, works by iteratively deleting longer sequences of lines (as long as the result is an observation-based slice) until no more lines can be deleted. A single iteration of ORBS attempts to delete from the source code and validates the deletion by compiling and executing the candidate, and comparing the trajectory against the trajectory for the original program. If a deletion produces compilation errors, the deletion cannot produce a correct executable slice. Similarly, if a deletion leads to a slice that produces a different trajectory from the original, the slice is not correct. A deletion is accepted as a part of a valid *slicing action* if it passes both checks.

3.1 Algorithm and Implementation

Algorithm 1 presents ORBS. It starts by setting up the program to capture the resulting trajectory for slicing criterion (v, l) and then executing it for all inputs $\mathcal{I}$, storing the trajectory. The Setup step simply inserts a line (without side effects) just before line l

```
1 if (x < 0) {
2   print x;
3 }
4 y = 42;
5 // Slice on y
```

Figure 3: Example Code for Deletion Window

that captures the value of variable v. The main loop tries to delete lines, as long as the deletion still results in a slice, until no more lines can be deleted. It does this using a moving deletion window (MDW) implemented by the **for** loop on Line 10, which tries to find the minimal sequence of lines that can be deleted such that the deletion results in a compilable program S'. If S' builds, it is executed (at Line 17) and the trajectory is captured. If this trajectory is the same as the original trajectory, the algorithm accepts S' as a slice (i.e., accepts the deletion) and continues looking for deletion opportunities in S'. Otherwise, it rejects the deletion and continues to the next line of the file being sliced.

For example, Figure 2 shows the slice generated by ORBS for the program shown in Figure 1. ORBS has removed the code in `checker.java` responsible for counting digits and printing the results, in `reader.c` the code for the currency symbol has been deleted, and in `glue.py` one of the invocations of `reader` has been removed.

The deletion window size parameter δ places an upper bound on the number of lines that can be deleted together in one deletion operation. Higher values offer potentially more precise slices but at the cost of increased slicing time. For example, consider the code segment shown in Figure 3. 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 them results in a syntax error. ORBS avoids this issue by increasing the deletion window until the result compiles. Assume that the maximum deletion window size is 2. In the initial pass, MDW only supports the deletion of Line 2: other lines cannot be deleted either because of syntax errors (from deleting Line 1 alone or Lines 1 and 2), or trajectory comparison failures (from deleting Line 4 when the slice's trajectory won't match the original trajectory). After the deletion of Line 2, in the next pass, the original Lines 1 and 3 are adjacent and can be deleted together, at which point we achieve the desired slice. Through experimentation we have found three to be a good maximum deletion window size.

ORBS does not know anything about the programming language, not even how comments are represented. A deletion of a blank line or a line that is part of a comment will not change the behaviour of the program. Our implementation, rather than undertake the expense of re-testing in such cases, caches intermediate results in BUILD and EXECUTE. If a subsequent build produces a cache hit then there is no need to run the test cases as the cached result can be used. Avoiding unnecessary builds and/or executions is particularly beneficial when a deletion leads to a non-terminating program (detected using a timeout). Avoiding re-executions that timeout saves a significant amount of time. This approach is used in the experiments reported in Section 6 where the number of test executions needed is essentially halved as a result.

3.2 ORBS Variants

The ORBS algorithm considers the program from its last line to its first. Otherwise an additional iteration of the algorithm's main loop is often required because variable definitions are typically lexically earlier in code than their uses. Consequently, the definition can only be removed after all uses (causing the additional loop iteration). Reversing the code means that the deletion of the definition will be attempted last. *Forward* ORBS (F-ORBS) is defined by skipping the REVERSE Lines 3 and 23 of the ORBS algorithm, and building S' rather than REVERSE(S') in step 12.

Two other variations are considered in the experiments. These two are based on delta debugging [45]. A delta-debugging based ORBS replaces the deletion operations of the ORBS algorithm with *ddmin* [45] so that the program is split into large chunks on which the deletion is then attempted. If no deletion is possible, the size of the chunks is decreased until no more decrease is possible. The first variant considered, DD-ORBS, uses plain delta debugging while the second, MDW-DD-ORBS, uses the moving deletion window approach. Plain delta debugging continuously decreases the size of the deltas it deletes. When the delta has reached the size of a single line, delta debugging can no longer delete a line that can only be deleted together with one of more other lines. MDW-DD-ORBS therefore uses a deletion window as soon as the delta debugging has reached a delta size of one line. The characteristics of these four variants are explored in Section 5.

4. RESEARCH QUESTIONS

We will study observation-based slicing and our implementation ORBS using the following research questions.

RQ1: How does ORBS compare to the results of related approaches? The first research question addresses the features of ORBS slices by comparing the computed slices to those computed on the same programs by alternative approaches. The scope of the investigation is restricted to multi-language capable slicing techniques (thus ruling out most dynamic slicing approaches). Section 5 compares results from STRIPE [10], Critical Slicing [12], manually-identified minimal slices, and the four ORBS variants described above.

RQ2: How does observation-based slicing scale? Assuming the answers to the first research question indicate that ORBS performs as well or better than alternative approaches, this question will investigate how it behaves on a larger case study, the program `bash`. Results are described in Section 6.

RQ3: How can observation-based parallelised? ORBS is a sequential iterative algorithm which needs large numbers of compilations and executions. Section 7 will investigate a parallel version and compare the sequential with the parallel version in terms of runtime and slice sizes.

RQ4: What are the impacts of external factors on ORBS? ORBS depends only on the standard tools available to the programmer in his or her development environment. It does not require bespoke dependence analysis tools, hence its language independence. Since ORBS relies on these tools to undertake its slicing operations, the effect of these must also be studied. This research question investigates the effect of the environment (e.g., compiler and operating system), the effect of source code layout and the ordering of files. The results are described in Section 8.

5. COMPARATIVE STUDY

This section addresses RQ1, looking at the operation of ORBS on a range of programs. Most dynamic slicing approaches are simply inapplicable for observation-based slicing, by virtue of their use of the execute–observe–delete (as opposed to the delete–execute–observe) paradigm. Those that are amenable to a delete–execute–observe paradigm are Critical Slicing (since it emphasises observation) and approaches based on delta debugging (since they emphasize the deletion operation). We discuss some characteristics before undertaking a larger comparison.

```
  E S D C O DD
1 | | | | | | int main(int argc, char **argv) {
2 | |   |   |   int a = 0;
3 |     |       int z = 0;
4 | |   | | |   int x = 0;
5 | |   | | |   int j = 0;
6 | |             a = atoi(argv[1]);
7 | | |           x = 0;
8 | | | | | |   j = 5
9 | |   |         a = a - 10;
10| |   |   |     if (a > j) {
11|   |             x = x + 1;
12                } else {
13|                z = 0;
14      |   |     }
15| | | | | |     x = x + j;
16|               printf("%d\n", x); // slice on x
17|               return 0;
18      | | | }
```

Figure 4: Comparison of E: Executed, S: Static Slice, D: Dynamic Slice, C: Critical Slice, O: ORBS, DD: DD-ORBS ('|' shows that a statement is in the slice).

5.1 General Characteristics

Figure 4 shows a comparison of ORBS to the approaches considered by DeMillo et al. [12] when they introduced Critical Slicing, using a minor modification of their example. The first column ('E'), shows the executed lines as measured by *gcov*. The results of applying five techniques are then shown in the subsequent columns: Column 'S' shows the static slice, Column 'D' shows the dynamic slice, Column 'C' shows the critical slice (based on our own implementation of the algorithm of DeMillo et al. [12]), Column 'O' shows the observation-based slice computed by ORBS, and finally, Column 'DD' for DD-ORBS. Note that only the critical and the observation-based slices are actually executable.

5.1.1 Critical Slicing

Although DeMillo et al. [12] implemented Critical Slicing on top of a debugger, they suggested (but did not implement) a simpler approach that independently deletes each individual line from the source code and then tests if the resulting code has the same behaviour on a test set. The critical slice is the program without those lines for which the deletion produced the same behaviour. We implemented this approach using the same framework as ORBS so that we can compare Critical Slicing with ORBS in detail.

Critical Slicing does not guarantee to produce legal slices: Although two lines may individually be removed without changing the behaviour at the criterion, their joint removal may produce a program with changed behaviour (or even fail to compile). Figure 5 shows an example where either Line 3 or Line 4 can be removed without changing the outcome at Line 7. Therefore, Critical Slicing excludes these two lines as well as Line 8. However, an execution of the critical slice will fail at Line 5 due to the division by zero. Observation-based slices, on the other hand, are always correct by construction. ORBS will only remove Line 4 and Line 8 as it cannot remove Line 3 after it has already removed Line 4 (forward ORBS would remove Line 3 instead of Line 4).

Because Critical Slicing needs a fixed number of compilations (one per line) it may run faster than ORBS. To investigate this we retain Critical Slicing in our next set of experiments, even though it may produce invalid slices.

5.1.2 STRIPE

STRIPE [10] uses delta debugging to identify the smallest subset of events in an execution trace relevant in producing a failure. STRIPE first runs the program to obtain an execution trace and then

```
1 int main(int argc, char **argv) {
2   int a = 0, b = 0, x = -1, y = 0;
3   a = 1; // not in the critical slice
4   b = 1; // not in the critical slice
5   x = 5 / (a+b);
6   y = (x > 0)? 1 : 2;
7   printf("%d\n", y); // slice on y
8   return 0; // not in the critical slice
9 }
```

Figure 5: Example – A Critical Slice is not a valid Slice

```
  E ST C O
1 |  |     $sum = 0;
2 |        $mul = 1;
3 |  |  |  | print "a? "; $a = <>;
4 |     |  | print "b? "; $b = <>;
5 |     |  | while ($a <= $b) {
6 |  |  |  |   $sum = $sum + $a;
7 |  |  |  |   $mul = $mul * $a;
8 |  |  |  |   $a = $a + 1;
9 |     |  | }
10|  |  |  | print "sum = ", $sum, "\n";
11|  |  |  | print "mul = ", $mul , "\n";
```

Figure 6: STRIPE Test Case `sample.pl`

uses a debugger to skip statements in the trace. STRIPE works with Perl programs and only one example is given by Cleve and Zeller, shown in Figure 6, which is executed with the input "`0 5`" for the slicing criteria `sum` in Line 10 and `mul` in Line 11. Column 'E' shows the lines that are executed for this input (i.e., all of them). Column 'ST' shows the lines that are not skipped by STRIPE. STRIPE operates on full traces where individual statement execution instances in a trace can be omitted. It seems that STRIPE never includes control structures (e.g., Line 5 and 9) which leads to the skipping of statements that affect the predicates (e.g., Line 4). ORBS and Critical Slicing only remove the first two lines (their removal does not change the outcome for the test "`0 5`").

STRIPE exhibits four disadvantages. First, it computes a trace subset which is not directly mapped onto source code. Second, it does not include control structures and consequently statements affecting them, causing the result to be too small (and thus not a valid slice at all). The third is that it is prohibitively expensive, not only because the use of a debugging infrastructure brings significant overheads, but because the complexity of the delta debugging algorithm requires an order of magnitude more executions: For the above example, STRIPE needed 176 executions [10] while ORBS needed only 29 compilations and 17 executions (Critical Slicing only needed 12 compilations and 10 executions). The last disadvantage of STRIPE in this context is that it operates only on a single language. Consequently, we do not discuss it further in this evaluation but focus on our own delta-debugging based variants.

5.2 Empirical Comparison

We have now reported on our initial experiments that investigated the general characteristics of techniques that use the delete–execute–observe paradigm and that are candidates for observation-based slicing. We found that both STRIPE and Critical Slicing may produce invalid slices. STRIPE is also prohibitively expensive and so it is omitted from the empirical study reported on in this section. We now move on to report the results of an empirical study that compares ORBS with its forward and delta-debugging variants, and Critical Slicing. In all the experiments performed here ORBS, F-ORBS, and MDW-DD-ORBS use a deletion window size of three. Note that the variations may produce different minimal observation-based slices, depending on deletion order.

Table 1: Comparison of ORBS Variants (C: Compilations, E: Executions, D: Deleted Lines).

System	Criteria	LOC	ORBS			F-ORBS			DD-ORBS			MDW-DD-ORBS			Critical		
			C	E	D	C	E	D	C	E	D	C	E	D	C	E	D
Fig. 1	dots,checker.java:13	48	132	36	22	189	49	21	217	73	21	374	75	21	*48*	*23*	*17*‡
swig	g,runme.py:10x201x		308	60	187	465	94	187	909	117	166	1 129	128	187	*201*	*65*	*113*‡
swig	g,example.c:15	201	416	86	167	569	99	167	1 741	272	145	2 371	309	167	201	65	104
swig	count,example.c:37	201	459	99	182	547	114	177	1 785	142	158	2 174	171	182	201	65	106
hanoi/java	nDiscs,22	171	807	153	123	769	140	127	2 045	500	102	3 675	407	127	*171*	*74*	*66*‡
hanoi/c#	nDiscs,24	200	793	213	159	775	191	163	3 178	519	132	5 358	600	163	*200*	*118*	*92*†
calendar/js	year,152	344	855	395	299	911	432	295	4 060	1 518	261	4 666	1 986	295	344	270	234
calendar/js	year,156	344	824	377	303	894	417	299	3 992	1 370	263	4 584	1 860	299	344	270	236
calendar/js	year,167	344	833	380	302	890	413	300	3 952	1 344	264	4 556	1 834	300	344	270	237
calendar/js	days,156	344	824	377	303	890	413	300	3 937	1 311	264	4 526	1 795	300	*344*	*270*	*237*‡
calendar/js	days,167	344	833	380	302	886	409	301	3 898	1 286	265	4 498	1 769	301	*344*	*270*	*238*‡
calendar/perl	year,166	370	920	436	317	945	456	317	3 285	1 496	298	4 831	2 047	317	*370*	*304*	*272*†
calendar/perl	year,171	370	880	418	323	908	435	323	3 180	1 378	301	4 541	1 905	323	*370*	*304*	*275*†
calendar/perl	year,185	370	790	396	336	916	439	335	2 823	1 106	309	3 663	1 507	336	*370*	*304*	*277*†
calendar/perl	days,171	370	878	418	324	912	438	322	3 207	1 404	300	4 625	1 962	322	*370*	*304*	*274*†
calendar/perl	days,185	370	788	396	337	921	443	334	2 993	1 216	303	4 002	1 668	330	*370*	*304*	*276*†
calendar/php	year,156	333	761	363	299	780	368	301	2 230	767	276	2 805	1 103	301	333	269	249
calendar/php	year,159	333	729	347	303	755	354	305	2 066	664	278	2 374	870	305	333	269	251
calendar/php	year,171	333	706	337	307	731	341	309	2 126	599	279	2 547	854	309	333	269	252
calendar/php	days,159	333	716	347	305	759	357	304	2 171	711	277	2 636	1 011	304	333	269	250
calendar/php	days,171	333	693	337	309	735	344	308	2 142	614	278	2 558	872	308	333	269	251
calendar/python	year,151	310	484	338	284	742	460	284	1 638	973	277	1 958	1 052	284	*310*	*280*	*199*†
calendar/python	year,155	310	425	318	288	714	451	288	1 631	955	279	1 932	1 043	288	*310*	*280*	*200*†
calendar/python	year,166	310	419	314	291	702	445	291	1 701	988	282	1 975	1 068	291	*310*	*280*	*200*†
calendar/python	days,155	310	425	318	288	666	432	290	1 631	955	279	1 932	1 043	288	*310*	*280*	*201*†
calendar/python	days,166	310	419	314	291	702	445	291	1 701	988	282	1 975	1 068	291	*310*	*280*	*200*†
printtokens	token_ind,273	563	2 516	385	401	2 285	364	377	21 200	2 091	350	38 851	2 553	405	*563*	*163*	*292*†
printtokens	cu_state,311	563	2 180	350	386	2 701	488	386	23 939	3 324	333	41 959	3 386	372	*563*	*163*	*259*†
printtokens	token_ind,311	563	2 180	350	386	2 701	488	386	24 180	3 267	333	41 958	3 385	372	*563*	*163*	*283*†
printtokens	state,358	563	2 237	349	372	2 323	402	371	21 836	2 370	337	42 557	3 161	388	*563*	*163*	*288*†
printtokens	state,383	563	2 677	354	379	2 316	413	366	24 355	2 775	342	45 366	3 364	396	*563*	*163*	*277*†
printtokens	state,434	563	2 553	403	400	2 572	407	403	21 751	1 982	348	39 888	2 554	404	*563*	*163*	*293*†
printtokens	token_ind,434	563	2 134	329	400	2 563	406	403	21 753	1 942	348	40 182	2 504	404	*563*	*163*	*293*†
printtokens	state,457	563	2 853	475	353	2 393	463	358	24 061	3 322	318	45 846	3 994	358	*563*	*163*	*269*†
printtokens	token_ind,550	563	1 915	377	342	2 944	496	344	25 394	3 673	305	45 813	3 911	343	*563*	*163*	*267*†
notepad	all	1 481	7 844	555	1 162	9 681	599	1 193	109 376	2 204	956	204 930	5 846	1 205	1 481	272	670
concordance	all	1 490	6 408	560	1 191	6 596	657	1 210	81 672	3 139	1 047	147 360	3 145	1 211	*1 490*	*371*	*825*†

Table 1 shows key values for applying the four variants (ORBS, F-ORBS, DD-ORBS, and MDW-DD-ORBS) and our implementation of Critical Slicing on ten test programs and 37 criteria. The first column gives the test program, the second the variable and location of the criterion, and the third gives the number of lines. For the five algorithms, three values are reported: The 'C' column gives the number of compilations done, the 'E' column gives the number of executions done, and the 'D' column gives the number of deleted lines. The best performing algorithm for each aspect in each case is shaded in dark gray as a visual guide. In instances where a *valid* critical slice achieves the best results, the second best values are shaded in a lighter gray, as Critical Slicing often results in an invalid slice (highlighted in italics). The last column shows why a critical slice is invalid, i.e. either does not compile (annotated with '†') or produces a different trajectory (annotated with '‡'), i.e. produces different values at the slice criterion. It should also be noted that critical slicing never deletes more lines then ORBS.

The first program is the example from Figure 1, the next three entries are an example taken from the SWIG [36] distribution. The example consists of three files: a C source, a SWIG wrapper definition, and a Python script using the wrapped library. All three files are sliced and the table shows the results for three different criteria. The next two tests are two Towers of Hanoi implementations [34] in Java and C#. Four versions of a library generating an HTML calendar [38] (Javascript, Perl, PHP, Python) have been sliced for six criteria. We have also applied the algorithms to `printtokens` from SIR [13] where we used as the slicing criteria, every formal parameter of type `int` at the beginning of its function and all 4 140 tests for the input set. The second to last test is the Notepad example (Java) from Oracle's JDK where the criteria were the ten occurrences of a `String` parameter at the beginning of a method. The final test is `concordance` (C++, from SIR) where we used the three occurrences of the parameter `locus` at the start of a method as criterion. When the system came with automatic tests, we have used them as inputs. Otherwise, an arbitrary input has been chosen.

We draw the following primary conclusions from this study:

- The two delta-debugging based approaches DD-ORBS and MDW-DD-ORBS are an order of magnitude more expensive than ORBS and F-ORBS, making them less scalable for observation-based slicing.
- ORBS usually needs fewer compilations and executions and deletes more lines (i.e., is faster and more precise) than F-ORBS, making it attractive for observation-based slicing.
- In some cases, MDW-DD-ORBS can delete more lines than any other variant, making it worthy of further study, particularly where slice precision is paramount.
- Critical Slicing needs the fewest compilations and executions, but the critical slices are always considerably larger, up to more than six times (for `swig`) and in 9 out of 11 cases more than twice. Since only 11 out of 37 slices are valid (i.e. 26 incorrect slices), we conclude that Critical Slicing is a poor contender for observation-based slicing.

In summary, we conclude that ORBS and F-ORBS perform better than Critical Slicing or the various delta-debugging based approaches to observation-based slicing.

Table 2: Four files of bash which are to be sliced

File	Lines	SLOC	Executable	Executed
variables.c	4 793	3 509	1 590	607
parse.y	6 011	4 531	2 393	753
lib/glob/glob.c	1 100	789	416	0
subst.c	9 392	6 890	3 370	1 123

5.3 Summary

To answer RQ1, we compared ORBS to several techniques finding that it produces typically smaller slices that retain executability over the whole test set, and that it incurs less computational expense. Although delta debugging has been used successfully in other areas, it proved to be much more expensive than ORBS for observation-based slicing. Since ORBS has proved to be the most effective and efficient of the techniques compared here, we now investigate how it scales when applied to a larger system.

6. CASE STUDY

To address RQ2 and as a real-world case study, we consider an 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 slicer that is capable of slicing bash). These properties make bash an excellent case study to explore, in more detail, the characteristics of observation-based slicing using ORBS.

Another problem of dynamic or static slicers is that they either have to analyse the whole system including all libraries or have to know the effects of all invoked external functions. ORBS does not need this and we will demonstrate this by applying ORBS only to a small set of source files.

The bash package contains 1 153 files and a total of 118 167 source lines of code (SLOC) as computed by *sloccount* [43]. It is written in eight different languages. For this case study we define a scenario (exercising the arithmetic functions of bash) and four execution cases. In the first two cases we explicitly choose two source files to be included in the ORBS analysis. The files to be sliced are variables.c, as variables are used in the tests, and parse.y, as the grammar defines the input format. The slicing of grammars has not previously been considered in the literature. The third and the fourth cases each add an additional file to be sliced. Case 3 adds lib/glob/glob.c, which performs file-name pattern matching and Case 4 adds subst.c, which is the largest single source code file within bash.

Table 2 shows different line-based measures for the four files: The number of lines in the file, the number of source lines of code (SLOC), and the number of executed and executable lines as computed by *gcov* [32]. Note that nothing in lib/glob/glob.c is executed in the scenario being considered because the execution of arithmetic functions does not involve file-name pattern matching.

6.1 Slicing Criterion

The slicing criterion we chose for all four cases is the variable val in line 1 393 of file expr.c with the input given by a test file arith.tests. At line 1 393 the result of converting a string to an integer is returned to the caller of the function strlong. It is expected that this function is called frequently while processing the test cases of arith.tests, because these test cases test the arithmetic functions of bash. This expectation is confirmed by measuring the statement coverage with *gcov*: the function strlong is invoked 80 425 times causing 80 425 occurrences of the criterion in the trajectory. Note that the file containing the criterion is not sliced (it is not the target of the deletion) in this case study.

Table 3: Results for four cases of applying ORBS

	Full	Partial Trajectory		
	2 Files	2 Files	3 Files	4 Files
Lines	10 804	10 804	11 904	21 296
Deletions	9 417	9 927	11 021	19 758
Compilations	42 793	34 947	36 812	66 402
Cached ¨	2 264	1 524	1 528	1 603
Executions	5 370	4 362	4 872	9 009
Cached ¨	4 657	3 960	4 197	7 799
Time (user)	359m	287m	326m	847m
Lines in slice				
variables.c	578	449	449	449
parse.y	795	422	422	412
lib/glob.glob.c	–	–	6	6
subst.c	–	–	–	665
Slice size	13%	8%	7%	7%
Slice size (SLOC)	17%	11%	10%	10%

6.2 Executing ORBS

ORBS relies on three operations: SETUP, BUILD, and EXECUTE. For bash, SETUP not only instruments expr.c but also runs the configuration script, ./configure, which configures the bash installation process for the local environment. The compilation phase BUILD strongly relies on incremental builds so that only the components dependent on the sliced files are rebuilt. EXECUTE runs the built bash on the test suite arith.tests.

In the first two cases, ORBS is executed in two different modes. The first, referred to as *full trajectory* is described in Section 3 and has the criterion (val, expr.c:1393). The second, referred to as *partial trajectory*, considers only a prefix of the trajectory and is used to illustrate that we can restrict an ORBS slice to a subset of the variable execution instances. In this part of the experiment the first 100 (an arbitrary cut off) entries from the trajectory are considered. The criterion is then ($\texttt{val}_{1...100}$, expr.c:1393). Restricting the trajectory can be used to focus on a range of computations. We have chosen the restricted trajectory to demonstrate a difference to dynamic slicing: In dynamic slicing, one usually specifies one single instance in the trajectory to be observed. ORBS can simulate this by restricting the trajectory.

Case 3 considers file lib/glob/glob.c, part of a library included with bash. The included libraries, although they are available in source code, are used as binary components in the build. They are only compiled in the first build and all the following builds use the binary library. Case 4 includes a fourth file to be sliced. The file we chose to add is subst.c as it is the largest source code file within bash. Cases 3 and 4 use the partial trajectory.

6.3 Results and Discussion

Table 3 compares the results obtained from the four cases. It shows the number of lines that are considered and how many are deleted. It also shows the number of compilations and executions performed by ORBS, the number of compilations and executions that were not necessary due to reuse of cached results, and the total time taken. For the four different sliced files it shows the number of SLOC remaining.

In the first case where two files are sliced based on the full trajectory, 9 417 of 10 804 lines are deleted. Comparing SLOC, the numbers are lower: From `variables.c`, 2 931 SLOC have been removed, leaving 578 (16%). This is in line with the expectations: The criterion and the files were chosen to exercise arithmetic functions involving variables (less than half of the executable lines are actually executed for the test). Moreover, the criterion is located in `strlong` (converting strings to integers) which results in the removal of code that deals with variables not holding integer values.

From `parse.y` 3 736 SLOC are removed, leaving 795 (18%). From the 37 rules in the grammar, 8 have been completely removed and from others large parts have been removed: From 849 non-empty lines in the grammar part, only 88 are left (10%). Most of the removed lines were in the declaration part and in the auxiliary function part in `parse.y`: From 4 460 non-empty lines 3 749 lines are removed, leaving 711 (16%). The resulting slice is very small, much smaller than a static slice would be.

ORBS needed 42 793 compilations, 5 370 executions and took almost six hours on a standard PC. This slice construction time means that ORBS cannot currently be used for on-demand slice construction. However, not all applications of slicing require on-demand slices. Furthermore, we used only standard desktop equipment; more powerful equipment would reduce slice construction time and parallelisation might dramatically reduce it further. It should also be remembered that no existing (dynamic or static) approach to slicing could even handle a system like `bash`. Section 7 will present a parallel version of ORBS which reduces the runtime to less than a third.

The switch from the full to the partial trajectory has a small effect. Now 9 927 out of 10 804 lines are deleted (510 more), reducing the slice size down to 8%. In terms of SLOC, `variables.c` is 13% of the original size and `parse.y` is 9% of the original size. The higher number of deletions affects the number of compilations and executions which dropped, causing a lower runtime (down to less than five hours). This change can be explained by the observation that only part of the input is considered with the partial trajectory and this part tests a smaller subset of the arithmetic functions.

The addition of the small file `lib/glob/glob.c` in Case 3 does not impact the slice of the other two files at all. The file itself is deleted almost completely, only six lines are left (1%). These six lines consist of three variable definitions and a function definition. None of these six lines can be deleted because they are referenced elsewhere (although the referenced function is never executed). This observation is in line with the observation that nothing in this file is actually executed and cannot have any influence.

The fourth case adds file `subst.c`, adding 9 392 lines (6 890 SLOC) to be considered for deletion. Doing so almost doubles the number of lines to be sliced; thus the number of compilations and executions is also almost doubled (in line with the expected linear complexity of the algorithm). However, the actual runtime has almost tripled because many more executions time-out. ORBS deletes 93% of the lines in `subst.c` (90% of the SLOC), leaving only 665 lines.

6.4 Summary

The case study in this section demonstrated that ORBS can be used to compute slices of multi-language production systems and that the resulting slices are significantly smaller than the original files. ORBS also allows the engineer to focus on slicing a specific set of files of interest. The four cases illustrate the linear nature of the algorithm in terms of the number of lines to be sliced. Finally, considering full and partial trajectories shows how only specific criterion instances can be focused on.

Table 4: Results of applying parallel ORBS to `bash`

	Full	Partial Trajectory		
	2 Files	2 Files	3 Files	4 Files
Lines	10 804	10 804	11 904	21 296
Deletions				
serial ORBS	9 417	9 927	11 021	19 758
2-ORBS	9 178	9 601	10 668	19 295
3-ORBS	9 523	9 999	11 093	19 879
4-ORBS	9 540	10 011	11 105	19 908
Compilations				
serial ORBS	42 793	34 947	36 812	66 402
2-ORBS	39 718	33 315	35 541	59 277
3-ORBS	46 870	36 288	38 238	66 939
4-ORBS	44 568	33 411	36 282	63 394
Executions				
serial ORBS	5 370	4 362	4 872	9 009
2-ORBS	5 302	4 237	4 841	7 893
3-ORBS	6 764	4 902	5 533	10 231
4-ORBS	7 173	4 532	5 195	9 486
Time (user/real)				
serial ORBS	359/657	287/448	326/512	847/1350
2-ORBS	291/235	227/171	246/187	639/524
3-ORBS	331/219	222/126	245/140	547/322
4-ORBS	336/198	231/101	253/115	557/244

7. PARALLEL ORBS

ORBS is inherently serial so it cannot be split up in parallel portions. However, the idea of a deletion window can be used to create a parallel variant in which a number of deletion windows of different sizes are tried in parallel. The largest deletion window that succeeds, i.e. compiles and produces the same trajectory, is accepted for deletion. The other attempts are discarded. The algorithm proceeds to the next line where again a number of deletion windows are tried in parallel.

We have implemented parallel ORBS and applied it to the four experiments on `bash` plus four new programs. Table 4 shows the results for serial ORBS and parallel ORBS with window sizes 2, 3, and 4. As expected, parallel ORBS with a window size 2 cannot delete as many lines as the serial ORBS that uses a window size of 3. Parallel ORBS with a window size of 3 or 4 can delete more lines than serial ORBS: serial ORBS only increases the window size as long as the slice does not compile, parallel ORBS will use the largest window size that compiles and executes correctly.

The number of compilations and executions performed by parallel ORBS does not increase much and often parallel ORBS needs fewer compilations than serial ORBS. Therefore, the user time does also not increase – it actually decreases which leads to a dramatic drop in real time: parallel ORBS with a window size of 4 needs 70%-82% less time than serial ORBS.

For another comparison, we have sliced four more systems: `ed` is a line-oriented text editor, `byacc` is Berkeley Yacc, `bc` is an arbitrary precision numeric processing language, `indent` is a code beautifier. In all four system we picked a formal parameter to an often-called function as the slicing criterion, executed the included test suite, and sliced all C source files (8–13). Table 5 shows the results which are not as clear as for `bash`. Parallel ORBS with larger window sizes does not consistently perform better or worse than smaller window sizes or serial ORBS in terms of slice size, or number of compilations or executions. However, parallel ORBS with a maximal window size of four is consistently faster than smaller window sizes and always much faster than serial ORBS.

From the experiments above we can see that parallel ORBS is much faster than serial ORBS while producing similar results. Parallel ORBS with a deletion window size of four needed up to 82% less time than serial ORBS.

Table 5: Results of applying parallel ORBS to four systems

	ed	byacc	bc	indent
Files	8	13	8	13
Lines	2 836	7 320	7 618	10 960
Deletions				
serial ORBS	1 817	6 828	5 806	4 898
2-ORBS	1 555	6 262	5 834	4 556
3-ORBS	1 617	6 773	5 930	4 554
4-ORBS	1 782	6 835	5 680	4 575
Compilations				
serial ORBS	17 450	19 853	41 775	78 982
2-ORBS	15 383	25 307	30 249	46 484
3-ORBS	22 409	18 614	53 878	61 099
4-ORBS	23 334	20 134	43 344	80 027
Executions				
serial ORBS	3 403	5 057	7 565	3 940
2-ORBS	4 106	5 243	8 799	7 563
3-ORBS	4 877	5 341	13 881	8 304
4-ORBS	5 905	5 331	17 132	9 771
Time (user/real)				
serial ORBS	121/360	125/262	697/1 165	501/1 111
2-ORBS	50/154	89/128	559/464	220/248
3-ORBS	126/234	70/69	909/556	268/229
4-ORBS	110/231	71/58	725/431	337/229

8. EXTERNAL FACTORS

This section considers RQ4, discussing three external factors that impact ORBS. We have already seen in the previous sections how a small change, such as the direction of deletion, can make a difference for an ORBS slice.

8.1 File Order

We have seen in the discussion of the algorithm that it matters if ORBS starts at the beginning and deletes in a forward direction or if it starts at the end and operates in a backward direction. However, as ORBS operates on a list of files which is specified by the user, the order of the files in the list may impact the slice too. An additional experiment with the last `bash` case study confirms this. Reversing the order of the files led to ORBS requiring 64 382 compilations and 8 881 executions to delete 19 789 lines, leaving 1 507 (7%). Not only are these numbers slightly different, but the individual slices are too: While the slices for `lib/glob/glob.c` and `variables.c` have not changed, the slices for `subst.c` and `parse.y` are slightly different.

8.2 Environment

The sliced version created by ORBS is perfectly adapted to the specific configuration and environment but is fragile to deviations in the environment which can cause unexpected results. For example, even the change necessary to compute coverage information (different arguments to the compiler) may make the sliced program fail on the same input with a crash. If the configuration of the build process for `bash` is changed to generate coverage information *before* ORBS is applied, then the generated slice is different but no longer fails.

The same sensitivity to the build and test environment holds if a different operating system is used, a different compiler, or just different arguments for the compiler. As an example, we considered the impact of optimisation: When optimisation was enabled the results were significantly different. Other experiments with different operating systems (OSX instead of Linux) or different compilers (*llvm* instead of *gcc*) caused similar differences.

In addition to differences arising from different build environments and configurations, some are due to explicitly undefined behaviour in programming languages. A notorious example is C with its wide range of undefined behaviours. This can actually generate illegal, but compilable and executable programs, an experience shared with similar program modifying approaches [30] but which can be avoided by integrating validity checkers into the compilation phase.

```
1 if (x < 0)
2 {
3   print x;
4 }
5 y = 42;
6 // Slice on y
```

Figure 7: Code from Fig. 3 with different formatting

```
1 main () {
2  int x;
3  int j = 5;
4  x = j;
5 }
```

Figure 8: Token-level ORBS Slice for Code in Fig. 4

8.3 Source Code Layout

Clearly, the layout of the code to be sliced influences ORBS. Usually, source code is formatted according to some guideline, for example, the Java Coding Conventions [35] or the GNU Coding Standards [33]. There is a subtle difference between the two: In C, an open brace, '{', is placed on a separate line while in Java it is placed at the end of the line containing the predicate. Figure 7 shows the code from Figure 3, reformatted to place the opening brace on a separate line. As discussed earlier, ORBS deletes lines 1–3 of Figure 3 together. If there are more statements between the { and }, ORBS will delete them in a first iteration. The next iteration will delete the if statement, {, and } together. However, the code in Figure 7 has a different format and is processed differently: Line 1 can and will be removed independently of the following lines because the remaining statement block can be always executed without affecting the criterion.

A question naturally follows from considering source code layout: Suppose ORBS were to operate not at the line level, but at token level (i.e., deleting tokens from the program). Here a token might be defined as a string separated by white-space characters. Alternatively, the definition of the underlying language may be used. In an experiment, we computed an ORBS slice at the token level using the example in Figure 4 by placing each C language token on its own line. After 494 compilations and 41 executions, the slice in Figure 8 was produced. It is clearly correct for the original criterion and input. However, the token-based variant is much more expensive than the line-based variant.

8.4 Summary

We found that the results ORBS produced are strongly dependent on external factors. However, this is exactly what we want: observation-based slicing is to be based on the *observed reality* of the environment in which the code is to be built and executed. We also found that this close coupling to the 'execution reality' allowed ORBS to produce smaller slices adapted to the specific environment in which the program is built and executed. Nevertheless, ORBS guarantees that the generated slice has the same behaviour as the original program for the slicing criterion. This is a particularly attractive finding, given the disappointingly large size of slices produced by existing approaches to slicing.

9. RELATED WORK

ORBS computes observation-based slices. Although there are few other techniques that do so, the approach is, in general, similar to dynamic slicing. Dynamic slicing is a concept introduced by Korel and Laski [20, 21]. 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. [1] and Demillo et al. [12]). Although many research prototypes and approaches exist [2, 5, 6, 28, 37, 46, 47], all approaches are for a single specific programming language and use complex program analyses. To the authors' knowledge, no tool exists that can slice a system written in multiple languages.

In dynamic slicing terms, the closest work to observation-based slicing is Critical Slicing [12] where a statement is considered to be critical if its deletion results in a changed observed behaviour for the slicing criterion. A critical slice consists of all the critical statements. One limitation of this approach is that it considers statements to be critical although they may not be, and thus could be deleted after another statement is deleted. We have seen that critical slices are significantly larger than observation-based slices and are often incorrect slices (while observation-based slices are correct by construction).

The idea to delete parts of a program or test input is most prominent in applications of delta debugging [10, 44, 45]. As plain delta debugging can be very expensive, a few approaches have modified delta debugging so that it exploits syntax and semantics of programming languages. Hierarchical Delta Debugging [27] exploits tree structures in inputs for a tree-based delta debugging approach, while Delta [26] is using a separate tool to flatten tree structures found in programs before applying delta debugging. Regehr et al. [30] exploit the syntax and semantics of C for four delta-debugging based algorithms to minimize C programs that trigger compiler bugs. One could integrate such approaches to observation-based slicing. However, this would sacrifice the language independence of ORBS.

The part of Delta [26] that manipulates the program after flattening is very similar to DD-ORBS and therefore suffers from the same problem that it cannot delete lines that can only be deleted together with the following line(s).

Another closely related approach is STRIPE [10] which eliminates statements from an execution trace with the help of a conventional debugger. The approach uses delta debugging [44] to delete statements from the trace. However, STRIPE ignores control dependence and does not produce executable slices. Because of the use of a debugger, STRIPE does not need any program analysis but cannot be applied to multi-language systems.

Two other related dynamic slicing techniques are Union Slicing [4] and Simultaneous Dynamic Program Slicing [17]. Like ORBS, the Union Slicing algorithm of Beszédes et al. [4] aims to approximate the realizable slice for a set of test inputs. It does so by producing the union of the independently-computed dynamic slices for each test case. The algorithm uses static analysis to compute local dependency information, instruments the source code, and then executes it. Then it computes dynamic slices globally. It is thus similar to ORBS in requiring execution and instrumentation (although ORBS instrumentation is lighter-weight) but unlike ORBS, the resulting union slice is not guaranteed to be executable. Furthermore, a union slice is not necessarily well-behaved for the set of test cases being considered as a whole. Separate dynamic slices can interfere with each other when simply unioned together [17]. Finally, Union Slicing requires static dependence information, which means that it is language dependent.

Hall's Simultaneous Dynamic Slicing (SDS) approach [17] addresses the problem of interference by iteratively building up a slice from a set of starting program points (which may be initially empty) and a dynamic slice for each execution in the test case set. The set expands with each iteration (and thus forms a partially-complete slice that may be missing some dependence information) until it converges.

This is effectively a variation on the traditional union operator. In this case, rather than creating a union of program points after dependence computation, SDS unions slices in the light of their combined dependence information to ensure that existing dependencies are retained as the union gets bigger, and that required but missing dependencies are included until no increase in size is observed.

Like Union Slicing (and unlike ORBS), SDS requires the computation of dependence information (and is thus language dependent). It also requires a dynamic slicer meeting certain assumptions. Like ORBS, SDS relies on instrumentation and execution (for SDS, to generate full traces). It also frames the slicing operation similarly to ORBS: as a problem of retaining the relationship between inputs and outputs and removing non-influencing code.

There have been a few previous approaches to multi-language slicing. Riesco et al. [31] parameterise languages and slice on their semantics. However, their approach is currently restricted to WHILE languages where ORBS has no such restriction and requires no semantic modelling.

Finally, Pócza et al. present an approach to dynamic slicing across languages on the .NET platform [29]. Their approach uses the Common Language Runtime (CLR) debugging framework to provide traceability between instructions and source code. They compute a slice on the execution trace via the CLR debugger. However, their approach still needs a specific analysis for every single language that not only extracts variable definitions and uses, but also creates a Control Dependence Graph. They have only implemented their approach for a very limited subset of C#.

Not directly targeted at slicing, Mayer and Schroeder [24] advocate to explicitly specify and exploit semantic links (dependences) for cross-language code analysis and refactoring. They also built and evaluated an infrastructure to link artefacts between six Java-framework based languages [25]. They use a static analysis for the Java source code and link the identifier between different languages based on framework semantics with a hand-crafted linker.

ORBS is not restricted to any particular language group or underlying representation: the tools that build the system can be used to undertake the slicing.

10. CONCLUSION

ORBS is the first language-independent program slicer that can compute slices for systems written in multiple languages, including systems which may contain binary components or libraries which cannot be analysed otherwise. ORBS uses statement deletion as its primary operation and observation as its validation criteria. The approach leverages existing tool chains, making it better suited to execution reality than previous slicing approaches based on models of dependence and semantics.

Future work will use search-based approaches to look for smaller observation-based slices. To reduce runtimes and produce smaller slices, ORBS could be extended with language specific extensions that exploit the syntax and semantics of a language without sacrificing the ability to slice multiple languages. Also, other methods for making the approach faster, with fewer compilations and executions, will be investigated. Finally, empirical study of the time versus slice-size tradeoffs of different deletion window sizes will be considered.

11. REFERENCES

[1] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proc. of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation (PLDI)*, pages 246–256, 1990.

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

[3] J. Beck and D. Eichmann. Program and interface slicing for reverse engineering. In *Proc. of the 15th International Conference on Software Engineering (ICSE)*, pages 509–518, 1993.

[4] Á. Beszédes, C. Faragó, Z. M. Szabó, J. Csirik, and T. Gyimóthy. Union slices for program maintenance. In *Proc. of the 18th International Conference on Software Maintenance (ICSM)*, pages 12–21, 2002.

[5] A. Beszedes, T. Gergely, and T. Gyimóthy. Graph-less dynamic dependence-based dynamic slicing algorithms. In *Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*, pages 21–30, 2006.

[6] 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 Conference on Software Maintenance and Reengineering*, pages 105–113, 2001.

[7] D. Binkley. The application of program slicing to regression testing. *Information and Software Technology*, 40(11 and 12):583–594, 1998.

[8] C. Cifuentes and A. Fraboulet. Intraprocedural static slicing of binary executables. In *Proc. of the International Conference on Software Maintenance (ICSM)*, pages 188–195, 1997.

[9] A. Cimitile, A. De Lucia, and M. Munro. Identifying reusable functions using specification driven program slicing: a case study. In *Proc. of the International Conference on Software Maintenance (ICSM)*, pages 124–133, 1995.

[10] H. Cleve and A. Zeller. Finding failure causes through automated testing. In *International Workshop on Automated Debugging*, pages 254–259, 2000.

[11] A. De Lucia, A. R. Fasolino, and M. Munro. Understanding function behaviours through program slicing. In 4^{th} *International Workshop on Program Comprehension*, pages 9–18, 1996.

[12] R. A. DeMillo, H. Pan, and E. H. Spafford. Critical slicing for software fault localization. In *Proceedings of the 1996 International Symposium on Software Testing and Analysis (ISSTA)*, pages 121–134, 1996.

[13] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.

[14] R. Ettinger and M. Verbaere. Untangling: a slice extraction refactoring. In *Proc. of the 3rd International Conference on Aspect-Oriented Software Development (AOSD)*, pages 93–101, 2004.

[15] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Trans. Softw. Eng.*, 17(8):751–761, 1991.

[16] Á. Hajnal and I. Forgács. A demand-driven approach to slicing legacy COBOL systems. *Journal of Software: Evolution and Process*, 24(1):67–82, 2011.

[17] R. Hall. Automatic extraction of executable program subsets by simultaneous dynamic program slicing. *Automated Software Engineering*, 2(1):33–53, 1995.

[18] R. M. Hierons, M. Harman, C. Fox, L. Ouarbya, and M. Daoudi. Conditioned slicing supports partition testing. *Software Testing, Verification and Reliability*, 12:23–28, Mar. 2002.

[19] C. Jones. *Software Engineering Best Practices*. McGraw-Hill, 2010.

[20] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1988.

[21] B. Korel and J. Laski. Dynamic slicing in computer programs. *Journal of Systems and Software*, 13(3):187–195, 1990.

[22] B. Korel and J. Rilling. Dynamic program slicing in understanding of program execution. In *Proc. of the* 5^{th} *International Workshop on Program Comprehension (IWPC)*, pages 80–89, 1997.

[23] S. Kusumoto, A. Nishimatsu, K. Nishie, and K. Inoue. Experimental evaluation of program slicing for fault localization. *Empirical Software Engineering*, 7:49–76, 2002.

[24] P. Mayer and A. Schroeder. Cross-language code analysis and refactoring. In *12th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 94–103, 2012.

[25] P. Mayer and A. Schroeder. Automated multi-language artifact binding and rename refactoring between Java and DSLs used by Java frameworks. In *Proc. of the European Conference on Object-Oriented Programming (ECOOP)*, 2014.

[26] S. McPeak, D. S. Wilkerson, and S. Goldsmith. Heuristically minimizes interesting files. `delta.tigris.org`.

[27] G. Misherghi and Z. Su. HDD: hierarchical delta debugging. In *Proc. of the 28th International Conference on Software Engineering (ICSE)*, pages 142–151, 2006.

[28] G. Mund and R. Mall. An efficient interprocedural dynamic slicing method. *Journal of Systems and Software*, 79(6):791–806, 2006.

[29] K. Pócza, M. Biczó, and Z. Porkoláb. Cross-language program slicing in the .NET framework. In *Proc. of the 3rd .NET Technologies Conference*, pages 141–150, Plzen (Czech Republic), 2005.

[30] 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 Conference on Programming Language Design and Implementation (PLDI)*, pages 335–346, 2012.

[31] A. Riesco, I. M. Asăvoae, and M. Asăvoae. A generic program slicing technique based on language definitions. In *Recent Trends in Algebraic Development Techniques*, volume 7841 of *Lecture Notes in Computer Science*, pages 248–264. Springer, 2013.

[32] R. Stallman. *Using and Porting GNU CC*. Free Software Foundation, 1998.

[33] R. Stallman et al. *GNU Coding Standards*. Free Software Foundation, 2013.

[34] K. Subbiah. Tower of Hanoi algorithm source code. `www.softwareandfinance.com/Software.html`.

[35] Sun Microsystems. *Code Conventions for the Java Programming Language*, 1999.

[36] SWIG version 1.3.40. `www.swig.org`.

[37] A. Szegedi and T. Gyimóthy. Dynamic slicing of Java bytecode programs. In *Proc. of the 5th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*, pages 35–44, 2005.

[38] G. Tentler. HTML calendar. `www.gerd-tentler.de/tools/`.

[39] P. Tonella. Using a concept lattice of decomposition slices for program understanding and impact analysis. *IEEE Trans. Softw. Eng.*, 29(6):495–509, 2003.

[40] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449, 1981.

[41] M. Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, 1982.

[42] M. Weiser and J. Lyle. Experiments on slicing-based debugging aids. In *Empirical Studies of Programmers: First Workshop*, pages 187–197, 1985.

[43] D. A. Wheeler. SLOC count user's guide. `www.dwheeler.com/sloccount/sloccount.html`, 2004.

[44] A. Zeller. Yesterday, my program worked. today, it does not. Why? In *European Software Engineering Conference and Foundations of Software Engineering*, pages 253–267, 1999.

[45] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.

[46] X. Zhang, N. Gupta, and R. Gupta. A study of effectiveness of dynamic slicing in locating real faults. *Empirical Software Engineering*, 12(2), 2007.

[47] X. Zhang and R. Gupta. Cost effective dynamic program slicing. In *Proc. of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 94–106, 2004.