Research Note

RN/13/13

Observation-Based Slicing

June 20, 2013

David Binkley[1], *Nicolas Gold*[2], *Mark Harman*[2], *Jens Krinke*[2], *and Shin Yoo*[2],

Affiliation: Loyola University[1] University College London[2]
E-Mail: binkley@cs.loyola.edu, n.gold@ucl.ac.uk, mark.harman@ucl.ac.uk
j.krinke@cs.ucl.ac.uk, shin.yoo@ucl.ac.uk

Abstract

Program slicing reduces the size of source code by statement deletion, while preserving the behaviour of a program with respect to a criterion. Previous approaches use complex analyses to identify deletion candidates. This typically requires language-specific algorithms and thus limits tool applicability to systems written in a single language. We present Observation-Based Slicing (ORBS), a language independent slicing technique capable of slicing multi-language systems and also systems that contain (third party) binary components. A potential slice obtained through repeated statement deletion is validated by observing the behaviour of the program under the criterion: if the sliced and original programs behave the same, the deletion is accepted. We evaluate ORBS on programs of different sizes, showing its feasibility and comparing it to dynamic slicing approaches. 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.

1 Introduction

Since Weiser introduced program slicing [14] more than thirty years ago, hundreds of papers have appeared and countless research prototypes have been developed. Most of them can be classified as being either static or dynamic: A static slice considers all possible executions of the program to be sliced while a dynamic slice considers a specific execution.

Two long standing challenges in program slicing are to slice systems that (a) consist of components written in different programming languages and (b) contain binary components or libraries. Although one may be able to slice those components of the system written in a particular single language, the resulting slice has limited utility because one either has to ignore the effects of the other components or use worst-case assumptions for them. Doing so results in a slice that contains too many false positives or negatives, respectively.

However multi-language codes are the norm. For example, Jones suggests that the number of languages in a software system is often between two and fifteen and, motivating our approach, notes that the costs of development and maintenance rise with the number of languages [7, pp. 504-505]. The potential impact of a multi-language slicing tool is thus multiplied compared to one developed for a single language.

Consider, for example, the source code shown in Figure 1, which consists of three components in three languages: A Java and a C program connected by some Python glue. Assume that you want to know which statements can influence the value of the variable `dots` just after Line 13 of `checker.java`. A backward slice would capture the set of influencing statements (Figure 2 shows the slice as computed using our implementation), but the authors are not aware of any previous program analysis approach that can handle such systems without requiring complex abstract language models [11] or restrictions to particular groups of languages [10].

This paper presents an algorithm and implementation that can compute slices for such 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 analysed), instead the approach leverages the existing build tool-chain. ORBS thus provides a way to construct a slicer out of the build tools already being used rather than requiring the costly development of a new language-specific toolset.

Informally, Weiser defined a slice to be 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, almost no approach actually uses deletion. Instead most slicers compute slices by analysing dependences in the program 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. 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, our slices are similar to dynamic slices. However, as there are important differences (discussed later), we call them *observation-based slices*.

The contributions of this paper are

- an algorithm for computing observation-based slices,
- a prototype implementation of the algorithm,
- feasibility studies that demonstrate the application and operation of the approach, and
- a case study that explores characteristics of the new algorithm.

The rest of the paper is structured as follows: Section 2 presents the definitions of a slice used in this paper, Section 3 discusses observation-based slicing and presents the algorithm and its implementation. Section 4 checker.java:

```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);
  }
}
```

glue.py:

```python
# 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)
```

reader.c:

```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;
}
```

Figure 1: Example Multi-Language Application

discusses the research questions used to investigate ORBS. Sections 5 and 6 present basic results and a case study respectively. Section 7 discusses the influence of external factors. Finally, Section 8 presents related work and Section 9 concludes.

2 Slicing Definitions

Static slicing [15] 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 [8] 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 often checker.java:

```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;
      }
    }
  }
```

glue.py:

```python
import commands
import sys
use_locale = True
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)
```

reader.c:

```c
#include <locale.h>
int main(int argc, char **argv) {
  struct lconv *cur_locale = localeconv();
  if (atoi(argv[1]))
  {
   printf("%s\n", cur_locale->decimal_point);
  }
}
```

Figure 2: Multi-Language Sliced Example

to the particular technique being reported to compute the slices, rather than to an appropriate definition. We use a generalised definition of dynamic slicing that involves a state trajectory and a projection function, *PROJ* [15]. 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 [15] (additions are shown in italics). This definition is similar to Korel and Laski's [8] definition:

> 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 an input I *from* $\mathcal{I}$ with state trajectory T, then S also halts on input I with state trajectory T', and $\mathit{PROJ}_C(T) = \mathit{PROJ}_C(T')$, where PROJ_C is the projection function associated with criterion C.

Note that 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 finding the smallest slice possible. A slice is considered to be *minimal* if no further statement can be removed from it without causing it to be no longer a slice. There may be more than one minimal slice for a given program and slicing criterion [14].

3 Observation-Based Slicing

The definition of a dynamic slice holds the key to solving the challenges of multi-language and mixed source/binary slicing, slicing by actually *deleting* statements and *executing* the program to *observe* if the projected trajectory is the same.

The infrastructure for this approach is simple:

- Capturing the state trajectory is only required for the slicing criterion (i.e., a variable at a specified location in a single component). All other elements of the trajectory are removed by projection.
- Deletion of statements is only necessary for the components one is interested in.
- Other components can be used as they are.

Under these conditions, one can slice any software system where statements are deleted from the components of interest, the component containing the criterion is instrumented to capture the (projected) trajectory for the criterion, and the system can be generated and executed with the modified components. Implementation is correspondingly simple. Our ORBS implementation contains just 170 SLOC of python code and 10-20 lines of shell script depending on the project.

3.1 Algorithm and Implementation

Observation-based slicing is based on preserving the relevant part of the state trajectory from the execution of an original program P. For the slicing criterion $(v, l, \mathcal{I})$ composed of variable v, line l, and set of inputs $\mathcal{I}$, the execution of P for every input I in $\mathcal{I}$ produces a sequence of values $V(P, I, v, l)$ for variable v at line l. S, a subset of P, is an observation-based slice for criterion $(v, l, \mathcal{I})$ iff $\forall_{I \in \mathcal{I}} V(P, I, v, l) = V(S, I, v, l)$. We call the sequences V *trajectories* for the criterion $(v, l, \mathcal{I})$. The implementation produces $V(P, I, v, l)$ by injecting a statement just before line l that captures the value of v and writes it to a file.

We are interested in small subsets, thus we will try to delete as many statements from P as possible such that the subset is still an observation-based slice. However, we are not aiming at finding the smallest possible subset as this search can become computationally intractable.

The concept of a statement is language dependent. For example, a language like C has expressions with side effects and concatenation just like statements. As we want to be language independent, we delete lines from a source file and assume that the source files are formatted in a way such that there is no more than one statement on a single line. When this assumption is violated, the ORBS slicer will still work but may produce larger slices since the granularity of deletable objects will be larger (see Section 7 for further discussion on this).

Our algorithm for observation-based slicing, ORBS, works by iteratively deleting more and more 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 lines of code 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.

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 (non side-effecting) line just before line l 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 9, which tries to find the minimal sequence of lines that can be deleted together such that the deletion results in a compilable program. If X' compiles, it is executed (at Line 16) and the trajectory is captured. If this trajectory is the same as the original trajectory, the algorithm accepts X' as slice (i.e., accepts the deletion) and continues looking for deletion opportunities in X'. Otherwise, it rejects the deletion and continues at the next line.

Algorithm 1: ORBS

```
ORBSLICE($S, v, l, \mathcal{I}, \delta$)
Input: Source code, $S = \{s_1, \ldots, s_n\}$, slicing criterion, $(v, l, \mathcal{I})$, and maximum deletion window size, $\delta$
Output: A backward slice, $X$, of $S$ for $(v, l, \mathcal{I})$
(1)  $X \leftarrow$ SETUP($S, v, l$)
(2)  BUILD(X)
(3)  $V \leftarrow$ EXECUTE($X, \mathcal{I}$)
(4)  repeat
(5)      $deleted \leftarrow$ False
(6)      $i \leftarrow 1$
(7)      while $i < length(X)$
(8)          $builds \leftarrow$ False
(9)          for $j = 1$ to $\delta$
(10)             $X' \leftarrow X - \{x_i, \ldots, x_{min(length(X), i+j-1)}\}$
(11)             if BUILD($X'$)
(12)                 $compiles \leftarrow$ True
(13)                 break
(14)         if $compiles$
(15)             $V' \leftarrow$ EXECUTE($X', \mathcal{I}$)
(16)             if $V = V'$
(17)                 $X \leftarrow X'$
(18)                 $deleted \leftarrow True$
(19)             else
(20)                 $i \leftarrow i + 1$
(21) until $\neg deleted$
(22) return $X$
```

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` the variable initialisation 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 (since longer multi-line blocks of code can be deleted) 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 MDW until the result passes both the compilation check and testing. Let us consider the example in Figure 3 again. 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 (e.g., from deleting Line 1 alone or Line 1 and 2), or trajectory comparison failures (e.g., from deleting Line 4 because then the trajectory from the slice won't match the trajectory from the original). 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.

ORBS does not know anything about the programming language, not even how comments are represented. A deletion of a line that is part of a comment or the deletion of a blank line will not change the behaviour of the program. Our implementation, rather than undertake the expense of re-testing in such cases, caches compiled binaries. 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 test execution is particularly beneficial when a deletion leads to a non-terminating program (detected using a timeout). Avoiding the re-execution of a prospective slice that

```
if (x < 0) {
 print x;
}
y = 42;
//slice on y
```

Figure 3: Example Code for Deletion Window

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 4: Sliced Example with Backward Deletion

will time out for most tests saves significant time. This approach is used in Section 6 where the number of test executions needed is essentially halved.

A slight modification of the ORBS algorithm can cause a different observation-based slice. As defined, the algorithm attempts deletions in a forward direction (i.e., it tries to delete later lines only after earlier lines already have been deleted). If the traversal is inverted so that it starts with the deletion at the end of each file, ORBS may produce a different slice. Consider the example in Figure 1 again. In `glue.py` a 'backward' ORBS deletes the execution of `reader` with the parameter `0` (Line 10) and leaves the initialisation of `currency` instead (Line 6). Because `reader` is now always called with the parameter `1`, the check for the parameter in `reader.c` (Line 4) is no longer needed and is thus deleted. With the original algorithm, the initialisation of `currency` in Line 6 is deleted before the deletion of the invocation is attempted. However, in python a variable has to be defined before it can be used and thus the attempt to delete the second of the two definitions will fail as it will cause a use of the undefined variable `currency` in Line 13. Figure 4 shows the result with backward deletion.

4 Research Questions

RQ1: Is observation-based slicing feasible? The first research question addresses the fundamental questions of ORBS. For example, does it delete anything from a program, is it computationally tractable on even small programs, does it delete lines correctly? We answer this research question through a prototype implementation and the results are discussed in Section 5.

RQ2: How do observation-based slices compare to the results of other dynamic slicing approaches? The second research question addresses the features of observation-based slices by comparing the computed slices to those computed on the same programs by other slicing approaches and tools. Specifically, we consider Union Slicing [3] (using Whole Execution Trace (WET) slicing [18]), and Critical Slicing [5].

	E	S	SE	ES	D	C	O	
1	|	|	|	|	|	|	|	`int main(int argc, char **argv) {`
2	|	|	|			|		`  int a;`
3	|					|		`  int z;`
4	|	|	|	|		|	|	`  int x;`
5	|	|	|	|		|	|	`  int j;`
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\u", x);`
17	|							`  return 0;`
18						|	|	`}`

Figure 5: Comparison of E: Executed, S: Static Slice, SE: Executed Slice, ES: Static Slice for Executed, D: Dynamic Slice, C: Critical Slice, O: ORBS

We also compare ORBS slicing with manually identified minimal slices.

RQ3: How does observation-based slicing scale? Assuming observation-based slicing works with small programs, this research question will investigate how it behaves on a larger case study, the program `bash`. The results are described in Section 6.

RQ4: What are the impacts of external factors on ORBS? Since ORBS relies on other tools to undertake its slicing operations, the effect of these must be considered. This research question thus investigates the effect of source code layout, the ordering of files analysed, and the effect of the environment (e.g. compiler and operating system). The results are described in Section 7.

5 Experiments and Results

This section addresses RQ1 and RQ2, looking at the feasibility and operation of ORBS on a range of small programs, in comparison to other related techniques. There are three related techniques we focus on: DeMillo et al.'s Critical Slicing [5] where every statement is checked whether it is critical (i.e., the statement cannot be deleted without changing the observed behaviour for the slicing criterion), Union Slicing [3] where the dynamic slices from a set of test inputs are unioned, and STRIPE [4] that uses delta debugging to find the smallest set of statements in a trace that cannot be *skipped* during an execution.

5.1 General Comparison

Figure 5 shows a comparison of ORBS to the approaches considered by DeMillo et al. [5] 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 six techniques are then shown in the subsequent columns: Column 'S' shows the static slice (as computed by Codesurfer [2]), Column 'SE' shows the "executed slice" (intersection of 'S' and 'E' [5]), Column 'ES' shows the static slice computed for the executed part of the program, Column 'D' shows the dynamic slice (as computed by WET [18]), Column 'C' shows the critical slice (based on our own implementation of the algorithm of DeMillo et al. [5]), and finally, Column 'O' shows the observation-based slice computed by ORBS. Note that only the critical and the observation-based slice are actually executable.

```
1 int main(int argc, char **argv) {
2   int a = 0, b = 0, x = -1, y = 0;
3   a = 1;
4   b = 1;
5   x = 5 / (a+b);
6   y = (x > 0)? 1 : 2;
7   printf("%d\n", y); // slice here
8   return 0;
9 }
```

Figure 6: Example where the Critical Slice is not a Slice

5.2 Detailed Investigations

Having made a general comparison of ORBS to various techniques on a small example, we now investigate three techniques in further detail. Our investigation was based on three approaches: our own implementation of the Critical Slicing method of DeMillo et al. [5], the publicly-available implementation of WET[1] which we wrapped in a python script to implement Union Slicing [3], and hand-computation of Cleve and Zeller's STRIPE [4].

We applied these three techniques, along with ORBS, to various small test programs using the same input sets. The results are discussed below and various summary statistics are shown in Table 1. Overall, ORBS produces smaller slices (except in one case) than Critical Slicing, but larger slices than Union Slicing. However, the union slices are syntactically incorrect and/or non-executable (as is expected for Union Slicing). ORBS thus produces the most useful slices of all the techniques: they are both small and executable.

5.2.1 Critical Slicing

Although DeMillo et al. [5] implemented Critical Slicing on top of a debugger, they suggested a simpler approach that 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 where deletion produced the same behaviour. We implemented this approach in a similar way to ORBS so that we can compare Critical Slicing with ORBS in detail.

The first observation is that Critical Slicing is not guaranteed 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 6 shows an example. Either Line 3 or Line 4 can be removed without changing the outcome at Line 7. Therefore, Critical Slicing will not include the two lines in the critical slice (in addition to deleting Line 8). However, an execution of the critical slice will fail at Line 5 due to the division by zero. ORBS, on the other hand, will only remove Line 3 and Line 8 as it cannot remove Line 4 after it already removed Line 3 (backward ORBS would remove Line 4 instead of Line 3).

The same occurs in the initial example in Figure 1: A critical slice will identify lines 6 and 10 in `glue.py` as not being critical, as both can be removed individually. However, the resulting critical slice where both lines are removed produces an error because no initialisation of variable `currency` is left and thus the usage on Line 13 causes a runtime error. This problem is not uncommon, we experienced it with a few tests as shown in Table 1. In three cases the critical slice does not compile (annotated with "(C)") and in two of seven cases the observed behaviour for the slicing criterion is different (annotated with "(B)"). Only in two of seven cases is the critical slice a legal slice. This clearly illustrates that a critical slice does not capture the behaviour-preserving semantics required to be a slice.

[1]http://wet.cs.ucr.edu/

```
   E S 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 "ORBS sum = ", $sum, "\n";
11 | | | | print "ORBS mul = ", $mul , "\n";
```

Figure 7: STRIPE Test Case `sample.pl`

Table 1: Comparison of slices

Test Subject	Lines	SLOC	Executable	Executed	ORBS	Critical	Union
`demillo`	18	18	11	10	7	12	5
`interact`	28	25	15	15	20	19 (B)	13
`mbe_j_36`	88	63	27	25	24	48 (C)	15
`mbe_j_39`	”	”	”	”	24	48 (C)	15
`scam_x_32`	79	63	27	27	40	48 (C)	15
`wc_c_45`	60	48	23	23	15	30	6*
`wc_inword_29`	”	”	”	”	28	36 (B)	5*

5.2.2 *Union Slicing*

Beszédes et al. presented Union Slicing as a way to capture a summary of the source lines executed by a number of test inputs on the same program [3]. Their approach unions dynamic slices from a set of test inputs. The resulting slice is not necessarily executable. Since the aim of Union Slicing is similar to that of ORBS, we implemented a union slicer on top of WET, which profiles programs and gathers various types of information into a single unified structure [18]. One use of this structure is dynamic slicing. Since WET uses criteria at the trace level, our implementation wraps it to execute over all trace instances that correspond to the source criterion line, filtering the result for source lines, and forming the union of these lines. The sets of lines from each slice are then unioned over all test cases to yield the union slice. As expected, the resulting slices are typically not executable. In Table 1, the figures shown for Union Slicing are for slices computed over all criterion instances in the trace with the exception of the `wc` subjects (starred in Table 1) where the projected runtime was over 129 days. For these examples, only the final instance was used for slice computation.

5.2.3 *STRIPE*

Cleve and Zeller [4] present STRIPE which uses delta debugging to identify the smallest subset of events in an execution trace that are relevant in producing an observed failure. STRIPE first runs the program to obtain an execution trace and then uses a debugger to skip statements in the trace. STRIPE works with perl programs and only one example is given, shown in Figure 7, 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 (all of them). Column ‘S’ shows the lines that are not skipped by STRIPE. Note that STRIPE operates on full traces where individual statements are skipped. Therefore, Line 6 is not executed in the first instance (as `sum` and `a` are both 0 at that time), Line 7 is only executed in the first instance (as `mul` is 0 after that), and Line 8 is executed in all instances. It seems that STRIPE never includes control structures (e.g., Line 5 and 9) which in addition leads to the skipping of statements that affect the predicates (e.g.,

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

File	Lines	SLOC	Executable	Executed
`variables.c`	4793	3509	1590	607
`parse.y`	6011	4531	2393	753
`lib/glob/glob.c`	1100	789	416	0
`subst.c`	9392	6890	3370	1123

Line 4). Both ORBS and Critical Slicing only remove the first two lines (their removal does not change the outcome for the test "`0 5`").

STRIPE exhibits three disadvantages. The first is that it computes a trace subset which is not directly mapped onto source code. The second is that it does not include control structures and consequently statements affecting them – causing the result to be too small (and not a slice at all). The third is that it is prohibitively expensive: Not only because the use of a debugging infrastructure brings significant overhead, but the complexity of the delta debugging algorithm requires an order of magnitude more executions. For the above example, STRIPE needed 176 executions while ORBS and Critical Slicing only needed 10 (ORBS needed 36 compilations[2] and Critical Slicing needed 12).

5.3 Summary

In answer to RQ1, we find that ORBS is feasible, simple to implement, and slices successfully. It deletes appropriate lines and does not delete those it should not. To answer RQ2, we compared ORBS to several techniques finding that of these, it produces typically smaller slices that retain executability over the whole test set, and requires less computational expense.

6 Case Study

To address RQ3 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 OSX. The `bash` source package includes various tools and libraries that are needed to build it. The build is complex from a slicing perspective in that 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. Thus, `bash` is an excellent case study for ORBS.

The `bash` source package contains 1,153 files and a total of 118,167 source lines of code (SLOC) as computed by *sloccount* [16]. This source 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 (note that grammars have not been sliced before). 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 also 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* [12]. 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 1393 of file `expr.c` with the input given by a test file `arith.tests`. At line 1393 the result of converting a string to an integer

[2]Here, compilation means to let perl check the syntax of the program without fully executing it.

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	3,586	9,959	11,503	19,782
Compilations	61,267	48,085	50,826	92,691
Executions	4,108	5,027	5,602	11,255
Reuses	15,227	5,128	5,480	10,041
Time	1047m	515m	585m	1584m
`variables.c`	3,042	456	456	456
`parse.y`	4,176	389	389	375
`lib/glob.glob.c`	–	–	6	6
`subst.c`	–	–	–	677
Slice size	67%	8%	7%	7%
Slice size (SLOC)	90%	11%	10%	10%

is about to be returned to the caller of the function `strlong`. It is expected that this function is called very often 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.

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. 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. Restricting the trajectory can be used to focus on a specific part of the used input. Here, the first 100 instances are caused by the first 66 lines in the test file `arith.tests` and all other lines will therefore not be considered in this part of the experiment.

Case 3 considers the deletion of lines from the file `lib/glob/glob.c`. This file is 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 of `bash` and all the following compilations use the binary library. Including this file required fixing several missing dependencies in the Makefile. 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`. In Cases 3 and 4 the partial trajectory is used.

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 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 the two files are sliced based on the full trajectory, 3,586 of 10,804 lines are deleted.

Comparing SLOC, the numbers are lower: From `variables.c`, only 467 SLOC have been removed. This is in line with the expectations: The criterion and the sliced files were chosen to exercise arithmetic functions involving variables. From `parse.y` 1835 SLOC are removed. However, no line in the actual grammar part has been removed (except for comments and unnecessary lines containing a single ';'). Most of the removed lines were in the declaration part and in the auxiliary function part in `parse.y`. ORBS needed 61,267 compilations and 20,335 executions (from which 15,227 were cache hits and thus avoided) and took 17 hours and 27 minutes.

The switch from the full to the partial trajectory has a dramatic effect. Now 9,959 out of 10,804 lines are deleted (92%) and the two sliced files have only a few hundred lines left. In terms of SLOC, `variables.c` is 13% of the original size and `parse.y` is 8.6% of the original size. The higher number of deletions affects the number of compilations and executions which dropped significantly, causing a much lower runtime (down to less than nine hours). This change can be explained by the observation that only part of the test input is considered with the partial trajectory and this part only tests a small subset of the arithmetic functions.

The addition of the small file `lib/glob/glob.c` in Case 3 does not impact the slice size of the other two files. The file itself is almost deleted completely, only six lines are left. 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).

In the fourth case a large file `subst.c` is added, adding 9,392 lines (6,890 SLOC) to be considered for deletion. As the number of lines to be sliced has almost doubled, the number of needed compilations and executions is also almost doubled (in line with the expected almost linear complexity of the algorithm). However, the actual runtime has almost tripled because many more executions time-out. ORBS deleted 93% of the lines in `subst.c` (90% of the SLOC), leaving only 677 lines.

6.4 Summary

The case studies considered in this section demonstrate 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. It also shows that ORBS allows to focus on a specific set of files of interest that are to be sliced. The four cases illustrate the linear nature of the algorithm in terms of the number of lines to be sliced. Finally, considering both full and partial trajectories shows how only specific parts in the executions of a program can be focussed on. It suggests an interesting avenue for future work: considering the impact of slicing with respect to only certain instances in the output trajectory.

7 External Factors

This section considers RQ4, discussing various 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. In this section, we look at three external factors.

7.1 Source Code Layout

Clearly, the layout of the code to be sliced influences ORBS. Usually, source code is formatted according to some guideline. Two examples are the Java Coding Conventions [13] and the GNU conventions for C. There is a subtle difference between the two: In C, an open brace, '{', is usually placed on a separate line while in Java it is usually placed at the end of the line containing the predicate.

Figure 8 shows the code from Figure 3, reformatted to place the opening brace on a separate line. As discussed earlier, ORBS will delete lines 1–3 of Figure 3 together. If there are more statements between the { and }, ORBS will delete them in a first iteration, before the if statement, {, and } will be deleted together. However, the code in Figure 8 has a different format and is processed differently: Line 1 can and

```
if (x < 0)
{
  print x;
}
y = 42;
//slice on y
```

Figure 8: Example Code from Figure 3 with different formatting

```
main () {
 int x;
 int j = 5;
 x = j;
}
```

Figure 9: A token-level ORBS Slice for the Code in Figure 5

will be removed independent 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 the token level (i.e., deleting tokens from the program). Here a token might be defined as a string of non-white-space characters separated by white-space characters. Alternatively, the definition of the underlying language might be used (e.g., a C or a Java token). Depending on the definition of a *token*, this might require some language dependent infrastructure. The difference is seen with strings such as “a = i++”, which has three white-space-separated tokens, but four C language tokens. In an experiment we computed an ORBS slice at the token level using the example in Figure 5 by placing each C language token on its own line. After 494 compilations and 41 executions the slice in Figure 9 was produced. It is clearly correct for the original slicing criterion and test case.

7.2 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. This is clearly an implementation choice that is not necessarily an external factor. 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. An additional experiment with the last `bash` case study confirms this. Reversing the order of the files led to ORBS requiring 90,532 compilations and 9,923 executions to delete 19,736 lines (i.e., this trades just over a 2.3% decrease in compilations and a 11.8% decrease in executions for a 0.2% decrease in deleted lines). Not only are these numbers slightly different, but the individual slice sizes are too: `lib/glob/glob.c` and `variables.c` have not changed, but `subst.c` has 708 lines left (more than before), and `parse.y` has 390 lines left (more than before), indicating significant changes.

7.3 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) will 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 slice generated by ORBS is different.

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. ORBS may cause undefined behaviour through deletion where the actual observed

Original

```
1  #include <stdio.h>
2
3  int count;
4
5  int inc(int start) {
6    int c;
7    c = count; /* restore */
8    if (start) {
9      c = 0;
10   } else {
11     c += 1;
12   }
13   count = c; /* save */
14   return c;
15 }
16
17 int main(int argc, char **argv) {
18   int i, a[10];
19   inc(1); /* initialise counter */
20   for (i = 0; i < 10; ++i) {
21     a[i] = inc(0); /* count */
22   }
23   for (i = 0; i < 10; ++i) {
24     int x = a[i];
25     printf("%d\n", x);
26   }
27   return 0;
28 }
```

Slice

```
1  int inc(int start) {
2    int c;
3    if (start) {
4      c = 0;
5    } else {
6      c += 1;
7    }
8    return c;
9  }
10 int main(int argc, char **argv) {
11   int i, a[10];
12   inc(1); /* initialise counter */
13   for (i = 0; i < 10; ++i) {
14     a[i] = inc(0); /* count */
15   }
16   for (i = 0; i < 10; ++i) {
17     int x = a[i];
18   }
19 }
```

Figure 10: Example of Undefined Behaviour Impacting a Slice

behaviour does not change. Figure 10 shows an example. The function `inc` is used to increase a counter on every invocation. The current value of the counter is read from a global variable at the beginning of the function and it is saved in the global variable at the end of the function. The behaviour of the original program is defined. However, the slice that ORBS produces for variable `x` just before Line 18 no longer has the global variable. Clearly, every invocation of `inc(0)` now uses variable `c` without being initialised. Why does the slice still produce the same observed behaviour compared to the original although formally its behaviour is undefined? The reason is the way the generated code uses the stack: The first invocation of `inc(1)` defines the local variable which has a specific location on the stack. The following invocations use the same location on the stack and because no other function has been invoked, the old value of `c` has not been overwritten and the new instance of `c` contains the old value from the previous invocation.

7.4 Summary

In answer to RQ4 we find that the results ORBS produces are strongly dependent on external factors. However, this dependence allows 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.

8 Related Work

ORBS computes dynamic slices, a concept introduced by Korel and Laski [8, 9]. They gave a formal definition of a dynamic slice and 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.'s work [1] and Demillo et al.'s work [5]).

In dynamic slicing terms, the closest work is Critical Slicing [5] where a statement is considered to be critical if its deletion results in changes in 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. For example, a variable declaration can be deleted after all occurrences of the variable have been deleted. Thus, in most cases, critical slices are significantly larger than observation-based slices. The other closely related approach is STRIPE [4] which eliminates statements from an execution trace with the help of a conventional debugger. The approach uses delta debugging [17] to maximise the set of deleted statements, however, STRIPE ignores control dependence and does not produce executable slices. Moreover, no evaluation of STRIPE has been presented. Both approaches use a debugger as the underlying infrastructure and thus cannot be applied to multi-language systems (which ORBS has no problem with). They have been compared to ORBS in Section 5.

Two other related dynamic slicing techniques are Union Slicing [3] and Simultaneous Dynamic Program Slicing [6]. Like the ORBS approach, the Union Slicing algorithm of Beszédes et al. [3] 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 (thus requiring only one slicing pass for all slices).

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 (even though each of the constituent dynamic slices is for its respective test case). 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 [6]. Finally, Union Slicing requires static dependence information, which means that it is language dependent.

Hall's Simultaneous Dynamic Slicing (SDS) approach [6] 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 starting program point 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, Hall 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 (in Hall's terms, an iosequence) and removing non-influencing code.

There have been a few previous approaches to multi-language slicing. Riesco et al. [11] describe an approach to parameterising languages and slicing 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 [10]. 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. Since all Common Language Specification (CLS)-compliant languages (e.g., C# and Visual Basic.NET) are translated into the CLR framework, slicing can be carried out across the languages (or at least, the CLS-compliant parts of them). 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.

9 Conclusion

This paper presented ORBS, an approach to program slicing using statement deletion as the primary operation, and observation as the validation criteria. The approach is simple to implement and leverages existing tool chains. For the first time, it permits multiple languages (including binary code) to be sliced as a single system.

Future work will use search-based approaches to look for smaller observation-based slices. It will also consider the pros and cons of running the algorithm from the start of the file to the end versus starting at the end of the file and working toward the beginning. Finally, empirical study of the time versus slice-size tradeoffs of different deletion window sizes will be considered.

10 Acknowledgements

This work was partly funded by the Engineering and Physical Sciences Research Council [grant number EP/G060525/2] and NSF grant CCF 0916081. Data from the EPSRC-funded portions of the work may be available from Jens Krinke subject to licence and confidentiality restrictions.

References

[1] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 246–256, 1990.

[2] Paul Anderson and Tim Teitelbaum. Software inspection using CodeSurfer. In *Proc. of the 1st Workshop on Inspection in Software Engineering (WISE)*, pages 4–11, 2001.

[3] Árpád Beszédes, Csaba Faragó, Zsolt Mihály Szabó, János Csirik, and Tibor Gyimóthy. Union slices for program maintenance. In *ICSM*, pages 12–21, 2002.

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

[5] Richard A DeMillo, Hsin Pan, and Eugene H Spafford. Critical slicing for software fault localization. In *International Symposium on Software Testing and Analysis*, pages 121–134. ACM, 1996.

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

[7] Capers Jones. *Software Engineering Best Practices*. McGraw-Hill, 2010.

[8] Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, October 1988.

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

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

[11] Adrián Riesco, Irina Măriuca Asăvoae, and Mihail 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. 2013.

[12] Richard Stallman. *Using and Porting GNU CC*. Free Software Foundation, March 1998. Distributed with GNU CC.

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

[14] Mark Weiser. Program slicing. In *International Conference on Software Engineering*, pages 439–449. IEEE Press, 1981.

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

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

[17] Andreas Zeller. Yesterday, my program worked. today, it does not. why? In *ESEC/FSE*, pages 253–267. Springer, 1999.

[18] Xiangyu Zhang and Rajiv Gupta. Whole execution traces and their applications. *ACM Transactions on Architecture and Code Optimization*, 2(3), September 2005.