Program Slicing

Keith Gallagher
Computer Science Department
University of Durham
South Road
Durham DH1 3LE, UK
k.b.gallagher@durham.ac.uk

David Binkley*
King's College London
CREST Centre
Strand, London
WC2R 2LS, UK
binkley@cs.loyola.edu

Abstract

Program slicing is a decomposition technique that elides program components not relevant to a chosen computation, referred to as a slicing criterion. The remaining components form an executable program called a slice that computes a projection of the original program's semantics. Using examples coupled with fundamental principles, a tutorial introduction to program slicing is presented. Then applications of program slicing are surveyed, ranging from its first use as a debugging technique to current applications in property verification using finite state models. Finally, a summary of research challenges for the slicing community is discussed.

1 A Tribute to *The Little Lisper*[26]

Is Column 3 a slice of Column 1 with respect to Column 2?

Original Program	Slicing Criterion	Slice
x = 42	x	x = 42
d = 23	d	d = 23
x = 42 d = 23	d	x = 42 d = 23

Yes, Yes, Yes, regardless of the *slicing criterion* a program is always a slice of itself, albeit not always the best slice. A slice is computed with respect to a *slicing criterion* which consists of a *selected variable* and a *program location*. In the examples, the variable is listed in the middle column at the line that is the program location.

Are the following slices?

x = 42 d = 23	d	d = 23
x = 42 d = 23	x	x = 42

*On sabbatical leave from Loyola College in Maryland

Yes, Yes, at the *location* contained in the slicing criterion, the slice and the original program compute the same value for the selected variable (this variable is henceforth termed *of interest*). Statements not contributing to the computation of this variable can be elided. How about

a = 23 b = 42 c = b + 2	c	b = 42 c = b + 2
a = 12 x = 23 c = a + 2	c	a = 12 c = a + 2

Yes, Yes, to preserve semantics (value computed) of the chosen variable at the location in the slicing criterion, variables *used* to assign it a value are also *of interest* and assignments to them included in the slice. Next

a = 42 x = 2 b = 23 + a y = 3 c = b + 2	c	a = 42 b = 23 + a c = b + 2

Yes, variables used in assignments to variables *of interest* are *of interest* and assignments to them are included in the slice. Here, the assignment c = b + 2, causes variable b to be *of interest* and thus b = 23 + a is included in the slice. This makes a *of interest* and subsequently causes a = 42 to be included in the slice. And

a = 10 b = 42 a = 20 c = a + 2	c	a = 20 c = a + 2

Yes, only *reaching* assignments to a variable *of interest* are included in the slice. In this case the definition a = 20 kills the previous definition a = 10.

Principal 1 – Data Dependence. *A* Data Dependence *connects a definition of a variable to a use of that variable provided there exists a definition-free path in the program's control flow graph from the definition to the use [25]. In the presence of indirect memory references (*e.g.*, pointers) 'variable' becomes 'location' [2].*

Principals such as above formally capture key concepts used in program slicing, in contrast to the code fragments which are meant to be illustrative.

In the proceeding code there is a data dependence from a = 20 to c = a + 2, but not from a = 10 to c = a + 2 because the control-flow-graph path from a = 10 to c = a + 2 is not definition free: it includes the intervening definition a = 20. Next, consider (where $\mathbb{P}$ denotes some unspecified predicate)

a = 2		a = 2
while ($\mathbb{P}$)		
{		{
a = b − 2		a = b − 2
x = 10		
b = 64		b = 64
{		}
c = a + 2	c	c = a + 2

Data dependence can be *carried* from one iteration of a loop to another. Here the use of b in a = b − 2 is reached by the definition at b = 64 from the previous iteration. A (loop-carried) data dependence connects them and a slice including a = b − 2 will also include b = 64.

But, is this a slice? No, the while has errantly been elided. In the above, a = b − 2 is *control dependent* on while ($\mathbb{P}$). This motivates the second principal.

Principal 2 – Control Dependence. *A* Control Dependence *connects statement s to statement t when the predicate at s controls the execution of t [25]. For structured code, control dependences reflect the program's nesting structure. For unstructured code, s must have at least two control-flow graph successors, one of which can lead to the exit node without encountering t and the other must lead eventually to t [4, 35].*

In the preceding example there is a control dependence from the while to each of the three statements making up the body of the loop. Also note that depending on the value of predicate $\mathbb{P}$ the while statement may execute zero times and thus there are data dependences from both assignments of a to c = a + 2. Next consider

a = 20		a = 20
if ($\mathbb{P}$)		if ($\mathbb{P}$)
z = 30		
if ($\mathbb{Q}$)		if ($\mathbb{Q}$)
a = 21		a = 21
c = a + 2	c	c = a + 2

A statement that controls the execution of a statement in the slice is included in the slice. Slicing takes transitive control dependence into account. Here a = 21 is control dependent on if ($\mathbb{Q}$), which is control dependent on if ($\mathbb{P}$). Is this a slice?

x = 10		x = 10
y = 67		
a = 20		a = 20
if (x > 0)		if (x > 0)
a = 21		a = 21
c = a + 2	c	c = a + 2

Yes, slicing traverses intertwined sequences of control and data dependences. Encountered statements are included in the slice. The sequence from x = 10 to c = a + 2 includes the data dependence from x = 10 to if (x > 0), the control dependence from if (x > 0) to a = 21, and the data dependence from a = 21 to c = a + 2.

Is the following a slice (the program is the same as above)?

x = 10		x = 10
y = 67		
a = 20		
if (x > 0)	x	
a = 21		
c = a + 2		

Yes, a program has many different slicing criteria, which may, as in this case, lead to different slices. Note that x > 0 does not effect the value of x and is not included in this slice. Finally

sum = 0		
prod = 1		prod = 1
i = 1		i = 1
while (i < 11)		while (i < 11)
{		{
sum = sum + i		
prod = prod * i		prod = prod * i
i = i + 1		i = i + 1
}	prod	}

Yes, the computation of sum does not effect the computation of the prod; the slice taken with respect to prod at the end of the program produces a (smaller) program that only computes the product. This motivates the penultimate principal.

Principal 3 – A Slice is Semantically Meaningful. *A slice captures a semantically meaningful subcomputation from a program. Ignoring calling context [38], this means that the slice and the original program compute the same sequence of values for the selected variable at the chosen location [6, 21, 31, 52].*

1.1 Some Complications

```
while ( true )
  x = 1
y = 2                    y      y = 2
```

Programs that do not terminate still have (semantically) interesting slices because non-termination can be *sliced out*.

```
while (ℙ)                          while (ℙ)
{                                  {
  if (ℚ)                             if (ℚ)
    break                              break
  b = 12
  a = a + 1                          a = a + 1
}                          a       }
```

The execution of the break affects the number of times the loop execute; a = a + 1 is control dependent on the break and consequently the if statement.

```
      if (b)
        goto L2
L1:                                L1:
      y = 1                              y = 1
      goto L3                            goto L3
      z = 2
L2:
      x = 3
      goto L1
L3:                                L3:
      print(x)
      print(y)             y
```

Similar to a break statement, a goto affects the flow of control and induces control dependences.

1.2 One Final Principal

Principal 4 – Union of Slices. *The slice taken with respect to a set of criteria is the same as the union of the slices taken separately with respect to each of the criteria [59].*

Principal 4 is illustrated in Figure 1, which also illustrates Principal 3. Here slicing is used used to extract two of the three interesting subprograms from the word count program. (The three subprograms compute the number of characters, lines, and words in the program's input). The center two columns show slices taken with respect to chars and words. These two smaller programs compute only one of the three outputs. The final column shows the slice computed with respect to a criteria that contains both chars and words, which is the same as the union of the two individual slices.

2 Kinds of Slicing – Some Examples

The slices in Section 1 are formally *syntax preserving static backward slices* [19, 38, 59]. This section introduces several other variants of program slicing.

2.1 Forward Slices

Backward slices answer the question "what program components might effect a selected computation?" The dual of backward slicing, *forward slicing* [10, 38], answers the question "what program components might be effected by a selected computation?" A forward slice captures the *impact* of its slicing criteria [17]. Forward slicing gets it's name from the ability to compute a forward slice by traversing data and control dependence edges in the *forward* direction. In the following example the impact of prod (*i.e.*, those computations potentially influenced by the value assigned to prod at the point) is captured by a forward slice.

```
sum = 0
prod = 1                 prod     prod = 1
i = 1
while ( i < 11)
{
  sum = sum + i
  prod = prod * i                       prod = prod * i
  i = i + 1
}
```

This forward slice illustrates, among other things, that a change to the initialization of prod potentially affects (only) the computation of prod found in the loop. Because a forward slice is often not an executable program, one of the challenges posed by forward slicing is defining the semantics captured by a forward slice [10].

2.2 Dynamic Slices

A static backward slice preserve the meaning of the variable(s) in the slicing criterion for all possible inputs to the program. A dynamic slice does so only for a single input [1, 32, 60]. The slicing criterion is augmented to include a particular program input. In the following, using the input 42, the middle two statements can be elided.

Input < 42 >

```
read(a)                                read(a)
if (a < 0)
  a = -a
x = 1/a                  x       x = 1/a
```

2.3 Conditioned Slices

Conditioned slicing can be viewed as filling the *gap* between static and dynamic slicing. A conditioned slice preserves the semantics of the slicing criterion only for those inputs that satisfy a boolean condition [18, 20, 22]. In the following code when the input value for a is positive, the middle two statements can be elided.

```
read(a)                                read(a)
if (a < 0)
  a = -a
x = 1/a                  x       x = 1/a
```

Original Program	Slices taken at the end of the program with respect to chars	words	lines and words
inword = F	inword = F	inword = F	inword = F
chars = 0	chars = 0	chars = 0	chars = 0
lines = 0	lines = 0	lines = 0	lines = 0
words = 0	words = 0	words = 0	words = 0
read(c)	read(c)	read(c)	read(c)
while(c != EOF)	while(c != EOF)	while(c != EOF)	while(c != EOF)
{	{	{	{
chars++	chars++		chars++
if (c == '\n')			
lines++			
if (isletter(c)))		if (isletter(c)))	if (isletter(c)))
{		{	{
if (!inword)		if (!inword)	if (!inword)
{		{	{
words++		words++	words++
inword = T		inword = T	inword = T
}		}	}
}		}	}
else		else	else
inword = F		inword = F	inword = F
read(c)	read(c)	read(c)	read(c)
}	}	}	}

Figure 1. Three slices of the word count program.

Generally conditions can be placed at arbitrary program locations. In principal, such conditions can be expressed as restrictions on the allowed inputs.

2.4 Amorphous Slices

The aforementioned slicing techniques involve two requirements: a semantic requirement whereby the slice captures some projection of the original program's semantics and a syntactic requirement whereby the slice is constructed by deleting components from the original program. Amorphous slicing relaxes the syntactic requirement and thus allows additional transformations to be applied [34]. There are amorphous variants of static, dynamic, and conditioned slicing. To illustrate transformation's impact on a slice, the first example below applies constant propagation and then traditional slicing, which can then remove the first statement.

```
a = 42
b = a + 2                    b          b = 42+2
```

In the next example partial evaluation [11, 27, 50] like behavior is observed in the transformation of a statically determinable variable into a simple assignment.

```
sum = 0
i = 1
while ( i < 11)
{
  sum = sum + i
  i = i + 1
}                            sum        sum = 55
```

3 Uses of Slicing

This section considers six applications of slicing: four traditional applications and two recent applications. The first of the traditional applications is slicing's original application, debugging. This is followed by the application of slicing to testing and then to maintenance. The final traditional application is in the clustering of equivalent programs statements. The two recent applications are slicing in support of verifying state-based models and in determining the impact of a database schema change. Many of these application exploit relationships between slices.

3.1 Debugging

Program slicing was introduced by Mark Weiser as debugging aid [58, 59]. Consider a program that outputs a wrong answer. A programmer trying to ascertain what went wrong would have less code to consider

if they worked with the slice taken with respect to the errant output.

An extension of this idea further reduces the search space for the defect: consider the situation where a programmer knows that a program produced one correct and one errant output. Slices on these outputs can be used to narrow the search for the defect as it is unlikely that the statements that contribute to the correct output contain the defect [47]. In the code below, the slice on the left is taken with respect to chars, which procures a faulty output (it has a seeded initialization defect). The slice in the middle is taken with respect to lines, which produces the expected output. Looking at the statements from the first slice that are not in the second, provides candidate statements likely to contain the error. Of course, there is no guarantee that the fault lies in the difference; it might be an error of omission or one error might mask another in the slice with the correct output. This application, known as *program dicing* [47, 48], also shows that the study of the relationship between slices can be advantageous.

Slice on		Likely
chars (Incorrect Output)	lines (Correct Output)	Errant Statements
chars = 1		chars = 1
	lines = 0	
read(c)	read(c)	
while(c != EOF)	while(c != EOF)	
{	{	
chars++		chars++
	if (c == '\n')	
	lines++	
read(c)	read(c)	
}	}	

3.2 Regression Testing

The aim of regression testing is to ensure that a change to a software system does not introduce new errors in the unchanged part of the program [53]. Testing effort can be reduced if fewer tests cases are run on a simpler program. Program slicing can be used to partition tests cases into those that need to be re-run, as they may have been affected by a change, and those that can be ignored, as their behavior can be guaranteed to be unaffected by the change [7, 9, 53]. A partition is formed by finding all *affected* statements: those statements whose backward slice includes a new or edited statement. This set can be efficiently computed as the forward slice taken with respect to the new and edited statements. Only tests that execute an affected statement must be rerun.

Slicing can be used to further reduce the cost by reducing the size of the program that must be tested. This is done by applying *semantic differencing* [8] to Certified, the program that previous passed the test suite, and Modified, an updated version of Certified. In the example below Certified has an initialization error (line = 1) that is fixed in Modified. The program Differences captures (in an executable program) the computations of the changed code. Running Differences on selected tests reduces cost by running fewer tests on a simpler program.

Certified	Modified	Differences
inword = F	inword = F	
chars = 0	chars = 0	
lines = 1	lines = 0	lines = 0
words = 0	words = 0	
read(c)	read(c)	read(c)
while(c != EOF)	while(c != EOF)	while(c != EOF)
{	{	{
chars++		
if (c == '\n')		if (c == '\n')
lines++		lines++
if (isletter(c)))	if (isletter(c)))	
{	{	
if (!inword)	if (!inword)	
{	{	
words++	words++	
inword = T	inword = T	
}	}	
}	}	
else	else	
inword = F	inword = F	
read(c)	read(c)	read(c)
}	}	}

3.3 Software Maintenance

Maintaining a large software system is a challenging task. Most programs spend 70% or more of their life time in the software maintenance phase where they are corrected and enhanced. Slicing, in the form of *Decomposition Slicing* [30], has been applied to reducing the effort required to maintain software. The decomposition slice, taken with respect to variable v from function f, is the union of the slices taken with respect to v at each definition of v and at the end of f.

Decomposition slicing is another instance of a program slicing technique in which the relationships between slices is advantageously exploited. Returning to the slice shown in the third column of Figure 1, observe that the if statements inside the main while loop do not appear in the slices on chars or lines; they are solely contained within one slice. The same is true of the if statement in the slice on lines. Such statements are called *independent*. Furthermore, when *all* the assignments to a variable are solely within its decomposition slice then the variable is also called *independent*. The key insight behind the approach is that changes made

to independence statements and variables *cannot* impact computations in other decomposition slices. The ability to delimit changes also impacts the amount of regression testing needed [29, 30].

3.4 Clustering Equivalent Computations

Clustering groups a set into subsets where the elements of the subsets are similar to each other, but the subsets are dissimilar. A *dependence cluster* is a set of program points (statements), S, that mutually *depend* upon one another and for which there is no other mutually dependent set that contains S. This definition is parameterized by an underlying transitive *depends* relation.

One possible definition for *depends* is that two statements, s_1 and s_2, depend on each other if they have the same slice [15]. In practice, same size can be used to approximate size slice. Empirically this approximation is 99.5% accurate [15]. Using this definition, 30 of 45 programs contained a dependence cluster of at least 20% of the code and one of the programs included a cluster composed of 94% of the code.

Equivalent decomposition slices can also be used as the *depends* relation to construct clusters [28]. Variables that have equivalent decomposition slices form equivalence classes. An empirical study of 67 C programs found that the percentage of equivalent decomposition slices ranged from 50 to 60% ($p < 0.005$).

The negative side of large dependence can be seen in their impact on software maintenance. For example, consider trying to reuse a small component that turns out to be part of a large dependence cluster. On the other hand, having large clusters has a 'good' side: any test coverage method used for one variable in an equivalence class will apply to all the variables in that class. Furthermore, in support of program comprehension, knowing that a collection of variables belong to a cluster means that they capture the same abstraction. This observation can significantly reduces comprehension cost.

3.5 Model Reduction

Recently, finite-state models have been used to specify a number of non-trivial program properties (*e.g.*, the dead-lock freeness of a multi-threaded program). However, analyzing them is computationally expensive. Hatcliff, et al. [36] and later Dwyer, et al. [23] use static backward slicing to reduce the cost of property checking. The experiments make use of Indus, a library of Java program analysis and transformations tools [39].

In the earlier study, Hatcliff, et al. apply program slicing techniques to remove irrelevant code and reduce the size of the corresponding model. The study of slicing's impact considers two properties: Prop $I.i$ states

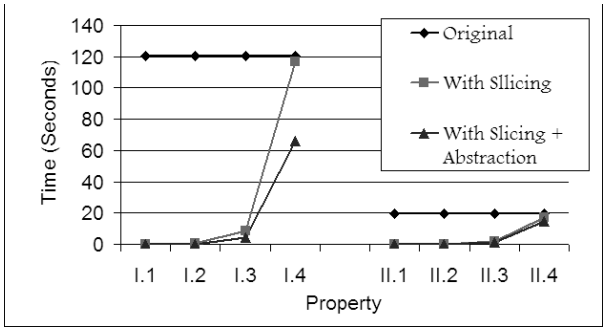


Figure 2. Slicing's impact on model checking.

that when the main thread sends the shutdown value, stage i will eventually shutdown. Prop $II.i$ states that stage i only shuts down after the shutdown signal has been sent from the main thread.

Figure 2 compares the running time for checking the two properties (I and II) using models generated first from the original program, then from the sliced program, and finally from the sliced program with remaining queue variables abstracted using a classic *signs* abstract interpretation over the abstract domain {pos, zero, neg, $\top$}. The cost savings from slicing comes primarily from slicing away of successive pipeline stages. Slicing based on Prop I.1 and Prop II.1 removes stages two, three and four. Generating these slice takes less than a second.

In later work, Dwyer, et al. [23] compare slicing with call-graph reduction and partial order reduction (POR), which exploit the independence of transitions to create equivalence classes of paths such that only a single path from each equivalence class need be explored. The motivation for applying slicing is that while existing static analysis attempts to remove unreachable/non-accessed program components, it does not eliminate code fragments that are reachable but where the intertwined code is irrelevant to the property begin checked – a task to which slicing is well suited. They observe that on average slicing provides a factor of four improvement for non-trivial model checks. With one exception slicing always yields a greater reduction than POR. Dwyer, et al. conclude that "slicing yields non-trivial additional reduction over partial-order reduction" and that "the reduction [due to slicing] is orthogonal to other reductions."

3.6 Database Schema Impact

In the second recent application, Maule, et al. study the impact on the source code of a relational database schema change [49]. An important compromise in the approach is the amount of calling-context information processed. Context-sensitivity is a measure of how precise the calling context of a procedure is represented

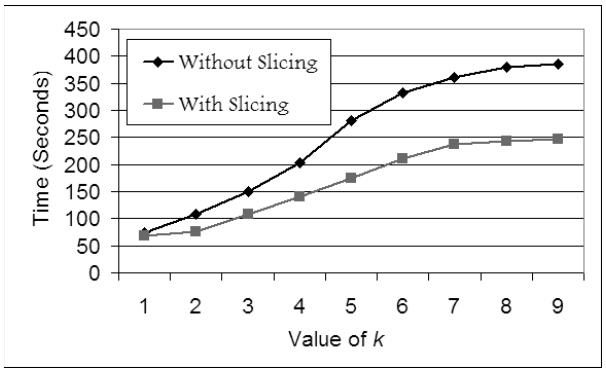


Figure 3. Slicing's influence on determining the impact of a database schema change.

during analysis. A common solution is kCFA analysis using a call-string approach to handle context where only the last (or first) k calls are tracked by the analysis. Maule, et al. record call strings that represent no more than the last k call-sites; *e.g.*, for $k = 2$, the most recent two call cites are included.

They employ program slicing to reduce the cost of identifying statements affected by a database scheme change. In a case study using the content management system itPublisch, which contains 78,133 lines of code with 417 query executions sites, program slicing reduced the part of the program that had to be considered by 63% (from 191,173 instructions to 70,050 instructions).

Figure 3 shows the execution times of their analysis for different values of k. It is clear that slicing provides a significant improvement. Furthermore, while the use of slicing always provides benefit, this benefit increases with k. statistically, it increases as k increases at a rate of 18 seconds per unit increase in k ($R^2 = 0.94$). This is because when $k = 1$ constant costs of the data-flow analysis that underlies slicing consumes 56% of the runtime. However, by $k = 9$, this has dropped to just 16%.

4 Current and Future Challenges

This section considers two current challenges for programs slicing and then speculates as to some future challenges. The two current challenges include the implementation of slicing tools and the size of the resulting slices. Each subsection provides first a broad overview and then considers one particular aspect in greater detail.

4.1 Implementation

The first slicers were data-flow based. They sliced by solving data flow equations [46, 59]. This approach was inefficient and gave way to an approach that involves first building a dependence graph to cache the dependence information implicitly considered in the data-flow equations [33, 38, 39, 42, 45, 51]. The latter approach is preferred when a large number of slices is to be computed [16, 43]. Having a dependence graph also has the advantage of supporting other dependence based algorithms [5]. Recently, improvements in data-flow analysis have led to more efficient data-flow based slicers [3]. Other approaches have also been considered. Ward implements slicing on top of the FermaT transformation tool which provides a rigorous mathematical foundation [56, 57]. Ward's approach is based on set theory and mathematical logic, and is thus independent of any representation technique.

As a representative example of this challenge, the problem of slicing parallel programs is considered in greater detail [37]. This requires considering three dependences beyond control and data dependence. First, an *interference dependence* can be viewed as the parallelized extension of data dependence. It connects a definition of a shared variable to a use of the shared variable in a different thread when the two may execute asynchronously. In the following the use of b in c = b + c makes this statement interference dependent on the two assignments to b from the other thread.

```
a = 10                                  a = 10
b = 11                                  b = 11
c = 12                                  c = 12
d = 13
cobegin                                 cobegin
{                                       {
  if (ℙ)                                  if (ℙ)
    b = a + b                               b = a + b
  else                                    else
    b = b * b                               b = b * b
}                                       }
{                                       {
   c = b + c                                c = b + c
   d = d + c
}                                       }
b = c + 1                    b          b = c + 1
```

These two interference dependences arise as b may hold one of three values at c = b + c (11, 21, or 121), depending on the value of ℙ, and the execution order of this assignment and the if statement.

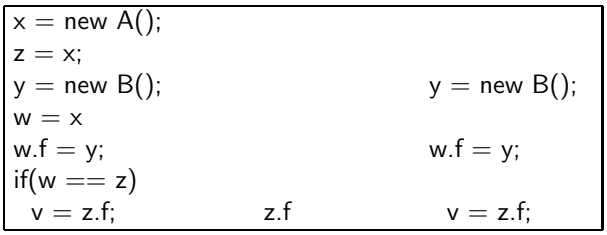
The remaining two dependence kinds are control dependences. The first, *parallel dependence*, connects the statement that initiates a task and the first statement of that task. The second, a *synchronization dependence*, connects statement s_1 to s_2 if the start or termination of s_1 depends on the start or termination of s_2 [37].

4.2 Size

The second current challenge facing program slicing is reducing the size of a slice. In almost all applications of program slicing, the smaller the slice the better. Empirical study places the size of the average backward slice at about one third of a program [13]. While this is a significant reduction, the remaining code can still be too large to comprehend as a unit. This observation led to the introduction of techniques such as dynamic slicing [60] and amorphous slicing [34].

One recent addition, *thin slicing*, reconsiders the fundamental dependences that slicing must capture [54]. A thin slice considers only *producer statements*, defined in terms of direct uses of variables and is thus always a subset of the traditional slices. A *direct use* of a location l is a use of l excluding those that require pointer dereferencing. The Java statement v = z.f, z is not directly used, but o.f is directly used, assuming that z points to o. Again, the crucial part of the definition is that it ignores *pointers*. A *producer statement* is then defined transitively as the variable of the criterion plus the other statements that write a value to a location used by a producer.

The following shows a thin slice, where statements in the 'full' slice but not the thin slice are called *explainer statements*. They capture the usual notions of control dependence and heap-based data dependence. In the example, there is a heap-based data dependence from w.f = y to v = z.f because z and w both point to the object created at x = new A(). When all of the explainers are transitively included, a traditional slice is obtained.



Thin slices have been shown useful in comprehension [54] where they reduce the number of statements that need to be examined in order to find an error by a factor of 3 (as compared to traditional slices), and reduce the number of statements that need to be examined for comprehension by a factor of 9 (as compared to traditional slices).

4.3 Future Research Challenges

This section concludes by considering future trends that provide research challenges for those working on and around program slicing.

Increasing Dynamic Nature of Languages
The increasingly dynamic nature of modern programming languages is an issue for all static analysis. This trend is clearly evident over the past 20 years in the transition from imperative languages to object-oriented languages to agent-based languages. In each case, it becomes less possible to predict (statically) which program elements will interact. Analysis of such programs will inevitably, become more specialized considering only certain classes of inputs or execution environments.

Slicing Will Become More Specialized
Weiser's initial study demonstrated that programmers intuitively sliced programs while debugging [58]. Early slicing tools attempted to mimic this behavior. To date, these tools have failed to capture the complete intuition that programmers appear to bring to the task. Until such time as they do, slicing will continue to be used more as a building block in analysis tools rather than a discipline in and of itself.

Beyond Slicing Programs
The first slicing algorithms applied to programs and produced slices that were *executable* programs. This helped defined the term *slice*. For better or for worse, what is slicable and what is a slice have broadened (*e.g.*, the expansion of 'program' slicing to finite state models was described in Section 3.5). Working in this domain, Korel, et al. report a significant reduction in the size of state-based models [41]. More recently, architecture descriptions written in the UML have been sliced. A Model Dependence Graph [44] supports slicing by representing UML use cases, classes, objects, and their interconnection similar to how programs are represented in Program Dependence Graph [24].

Fundamental Program Building Blocks
Fifty years ago programs were composed of assembler instructions. Modern programs of higher level syntactic entities such as statements and functions. Looking forward, perhaps future programs will be composed of semantic entities. Perhaps a library of slices could be selected by a programmers and 'woven' together by a compiler. This will provide programmers with more "bang for the buck" (assuming the US dollar recovers) or perhaps better stated as more "bang for the keystroke."

5 Summary

Program slicing has been applied to a range of maintenance tasks. This paper attempts to provide the intuition behind computing a program slice and considers

several representative applications. Several surveys on various types and applications of program slicing have been written and provide excellent sources for further information on program slicing [12, 14, 40, 55]. Finally, this paper lays out some challenges and future work for program slicing researchers.

6 Acknowledgements

Our special thanks to the following for their comments and suggestions Francoise Balmas, Andrea De Lucia, Mark Harman, Michael Hind, Jim Lyle, Rajib Mall, Thomas Reps, Frank Tip, Paolo Tonella, Neil Walkinshaw, and Martin Ward. Dave Binkley is supported by EPSRC grant GR/F010443 to the CREST Centre.

References

[1] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 246–256, June 1990.

[2] P. Anderson, D. Binkley, G. Rosay, and T. Teitelbaum. Flow insensitive points-to sets. In *Proceedings of the first IEEE Workshop on Source Code Analysis and Manipulation*, pages 79–89, Los Alamitos, California, USA, Nov. 2001. IEEE Computer Society Press.

[3] D. C. Atkinson and W. G. Griswold. Implementation techniques for efficient data-flow analysis of large programs. In *IEEE International Conference on Software Maintenance (ICSM'01)*, Los Alamitos, California, USA, Nov. 2001. IEEE Computer Society Press.

[4] T. Ball and S. Horwitz. Slicing programs with arbitrary control–flow. In P. Fritzson, editor, 1^{st} *Conference on Automated Algorithmic Debugging*, pages 206–222, Linköping, Sweden, 1993. Springer.

[5] F. Balmas. Using dependence graphs as a support to document programs. In 2^{st} *IEEE International Workshop on Source Code Analysis and Manipulation*, pages 145–154, Los Alamitos, California, USA, Oct. 2002. IEEE Computer Society Press.

[6] D. Binkley. Precise executable interprocedural slices. *ACM Letters on Programming Languages and Systems*, 3(1-4):31–45, 1993.

[7] D. Binkley. Semantics guided regression test cost reduction. *IEEE Transactions on Software Engineering*, 23(8):498–516, Aug. 1997.

[8] D. Binkley. Semantics guided regression test cost reduction. *IEEE Transactions on Software Engineering*, 23(8):498–516, August 1997.

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

[10] D. Binkley, S. Danicic, T. Gyimóthy, M. Harman, Ákos Kiss, and B. Korel. Theoretical foundations of dynamic program slicing. *Theoretical Computer Science*, 360(1):23–41, 2006.

[11] D. Binkley, S. Danicic, M. Harman, J. Howroyd, and L. Ouarbya. A formal relationship between program slicing and partial evaluation. *Formal Aspects of Computing*, 18(2):103–119, 2006.

[12] D. Binkley and K. B. Gallagher. Program slicing. In M. Zelkowitz, editor, *Advances in Computing, Volume 43*, pages 1–50. Academic Press, 1996.

[13] D. Binkley, N. Gold, and M. Harman. An empirical study of static program slice size. *ACM Transactions on Software Engineering and Methodology*, 16(2):1–32, 2007.

[14] D. Binkley and M. Harman. A survey of empirical results on program slicing. *Advances in Computers*, 62:105–178, 2004.

[15] D. Binkley and M. Harman. Locating dependence clusters and dependence pollution. In 21^{st} *IEEE International Conference on Software Maintenance*, pages 177–186, Los Alamitos, California, USA, 2005. IEEE Computer Society Press.

[16] D. Binkley, M. Harman, and J. Krinke. Empirical study of optimization techniques for massive slicing. *ACM Transactions on Programming Languages and Systems*, 2008. To appear.

[17] S. E. Black. Computing ripple effect for software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 13:263–279, 2001.

[18] G. Canfora, A. Cimitile, and A. De Lucia. Conditioned program slicing. *Information and Software Technology Special Issue on Program Slicing*, 40(11 and 12):595–607, 1998.

[19] S. Danicic, D. Binkley, T. Gyimóthy, M. Harman, Ákos Kiss, and B. Korel. A formalisation of the relationship between forms of program slicing. *Science of Computer Programming*, 62(3):228–252, 2006.

[20] S. Danicic, A. De Lucia, and M. Harman. Building executable union slices using conditioned slicing. In 12^{th} *International Workshop on Program Comprehension*, pages 89–97, Los Alamitos, California, USA, June 2004. IEEE Computer Society Press.

[21] S. Danicic, M. Harman, J. Howroyd, and L. Ouarbya. A lazy semantics for program slicing. In $1^{st.}$ *International Workshop on Programming Language Interference and Dependence*, Verona, Italy, Aug. 2004.

[22] M. Daoudi, S. Danicic, J. Howroyd, M. Harman, C. Fox, L. Ouarbya, and M. Ward. ConSUS: A scalable approach to conditioned slicing. In *IEEE Working Conference on Reverse Engineering (WCRE 2002)*. IEEE Computer Society Press, Oct. 2002.

[23] M. B. Dwyer, J. Hatcliff, M. Hoosier, V. P. Ranganath, Robby, and T. Wallentine. Evaluating the effectiveness of slicing for model reduction of concurrent object-oriented programs. In *TACAS*, 2006.

[24] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[25] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[26] D. P. Friedman and M. Felleisen. *The little LISPer (2nd ed.)*. SRA School Group, USA, 1986.

[27] Y. Futamura and K. Nogi. Generalized partial computation. In D. Bjrner, A. P. Ershov, and N. D. Jones, editors, *IFIP TC2 Workshop on Partial Evaluation and Mixed Computation*, pages 133–151. North–Holland, 1987.

[28] K. Gallagher and D. Binkley. An empirical study of computation equivalence as determined by decomposition slice equivalence. In 10^{th} *IEEE Working Conference on Reverse Engineering (WCRE 2003)*, Victoria, British Columbia, Canada, Nov. 2003. IEEE Computer Society Press.

[29] K. Gallagher, T. Hall, and S. Black. Reducing regression test size by exclusion. In *23rd International Conference on Software Maintenance*, pages 157 – 166, Paris, France, 2007. ISBN 1–4244–1256–0.

[30] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, Aug. 1991.

[31] R. Giacobazzi and I. Mastroeni. Non–standard semantics for program slicing. *Higher-Order and Symbolic Computation*, 16(4):297–339, 2003.

[32] R. Gopal. Dynamic program slicing based on dependence graphs. In *IEEE Conference on Software Maintenance*, pages 191–200, 1991.

[33] Grammatech Inc. The codesurfer slicing system, 2002.

[34] M. Harman, D. Binkley, and S. Danicic. Amorphous program slicing. *Journal of Systems and Software*, 68(1):45–64, Oct. 2003.

[35] M. Harman, A. Lakhotia, and D. Binkley. A framework for static slicers of unstructured programs. *Information and Software Technology*, 48(7):549–565, 2006.

[36] J. Hatcliff, M. B. Dwyer, and H. Zheng. Slicing software for model construction. *Higher-Order and Symbolic Computation*, 13(4):315–353, Dec. 2000.

[37] D. Hisley, M. J. Bridges, and L. L. Pollock. Static interprocedural slicing of shared memory parallel programs. In *PDPTA '02: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 658–664. CSREA Press, 2002.

[38] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–61, 1990.

[39] G. Jayaraman, V. P. Ranganath, and J. Hatcliff. Kaveri: Delivering the Indus java program slicer to Eclipse. In *LNCS*, volume 3442, pages 269–273. Springer-Verlag, April 2005.

[40] M. Kamkar, N. Shahmehri, and P. Fritzson. Interprocedural dynamic slicing. In *Proceedings of the* 4^{th} *Conference on Programming Language Implementation and Logic Programming*, pages 370–384, 1992.

[41] B. Korel, I. Singh, L. Tahat, and B. Vaysburg. Slicing of state based models. In *IEEE International Conference on Software Maintenance (ICSM'03)*, pages 34–43, Los Alamitos, California, USA, Sept. 2003. IEEE Computer Society Press.

[42] J. Krinke. Static slicing of threaded programs. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'98)*, pages 35–42, June 1998.

[43] J. Krinke. Evaluating context-sensitive slicing and chopping. In *IEEE International Conference on Software Maintenance*, pages 22–31, Los Alamitos, California, USA, Oct. 2002. IEEE Computer Society Press.

[44] J. Lallchandni and R. Mall. Personal communication, 2008.

[45] L. D. Larsen and M. J. Harrold. Slicing object-oriented software. In *Proceedings of the 18th International Conference on Software Engineering*, pages 495–505, Berlin, 1996.

[46] J. R. Lyle, D. R. Wallace, J. R. Graham, K. B. Gallagher, J. P. Poole, and D. Binkley. Unravel: A CASE tool to assist evaluation of high integrity software, Volume 1: Requirements and design. Technical Report NISTIR 5691, US Department of Commerce, Technology Administration, National Institute of Standards and Technology, Computer Systems Laboratory, Gaithersburg, MD 20899., 1995.

[47] J. R. Lyle and M. Weiser. Automatic program bug location by program slicing. In 2^{nd} *International Conference on Computers and Applications*, pages 877–882, Los Alamitos, California, USA, 1987. IEEE Computer Society Press.

[48] J. R. Lyle and M. D. Weiser. Experiments on slicing-based debugging aids. In E. Soloway and S. Iyengar, editors, *Empirical Studies of Programmers*. Ablex Publishing Corporation, Norwood, New Jersey, 1986.

[49] A. Maule, W. Emmerich, and D. Rosenblum. Impact analysis of database schema changes. In 20^{th} *IEEE International Conference and Software Engineering (ICSE 2008)*. IEEE Computer Society Press, May 2008.

[50] U. Meyer. Techniques for partial evaluation of imperative programs. In *Conference on Partial Evaluation and Semantics–Based Program Manipulation (PEPM)*. Association for Computer Machinery, 1991. Proceedings in SIGPlan Notices, 26(9), 1991.

[51] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in software development environments. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environmt, SIGPLAN Notices*, 19(5):177–184, 1984.

[52] T. Reps and W. Yang. The semantics of program slicing. Technical Report Technical Report 777, University of Wisconsin, 1988.

[53] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, Aug. 1996.

[54] M. Sridharan, S. J. Fink, and R. Bodik. Thin slicing. *SIGPLAN Not.*, 42(6):112–122, 2007.

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

[56] M. Ward. Program slicing via FermaT transformations. In 26^{th} *IEEE Annual Computer Software and Applications Conference (COMPSAC 2002)*, pages 357–362, Los Alamitos, California, USA, Aug. 2002. IEEE Computer Society Press.

[57] M. Ward and H. Zedan. Slicing as a program transformation. *ACM Transactions on Programming Languages and Systems*, 29(2), April 2007.

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

[59] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[60] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. In *IEEE/ACM International Conference on Software Engineering*, Portland, Oregon, May 2003.