Executable Slicing via Procedure Specialization

Min Aung Susan Horwitz Rich Joiner
University of Wisconsin
{aung, horwitz, joiner}@cs.wisc.edu

Thomas Reps
University of Wisconsin and GrammaTech, Inc.
reps@cs.wisc.edu

Abstract

Although Weiser originally defined a program slice to be an *executable* projection of a program, much of the research on slicing has focused on *closure* slices, which consist of the set of statements and conditions of the program that might affect the value of a given variable at a given statement or condition of interest. While closure slices can be useful, there are some contexts in which executable slices are preferable. Closure slices are not generally executable because there can be mismatches in the slice between the sets of actual parameters at different call-sites to a procedure `p` and the formal parameters of `p`.

This paper presents a new approach to creating executable slices. Our algorithm addresses the parameter-mismatch problem by creating *specialized versions of procedures* that have different sets of formal parameters. Moreover, the slice returned by the algorithm is *minimal*: the slice consists of the set of specialized procedures that solves a certain *coarsest-partition problem*.

The paper presents solutions for some additional issues that arise with more realistic languages than considered in past work. It also presents the results of an experimental evaluation of the algorithm applied to C programs.

1. Introduction

Program slicing [54] provides a useful tool for many semantics-based program-manipulation operations. In particular, slicing allows one to find semantically meaningful static decompositions of programs [27], where the decompositions consist of elements—typically statements and conditions—that are not textually contiguous. It is a fundamental operation that can aid in solving many software-engineering problems [33] including program understanding [21, 23, 29, 38, 47], maintenance [16, 26, 42], debugging [43, 44], testing [5, 6], differencing [31, 32], specialization [48], and program merging [34]. The term "program slicing" has been used to describe a number of different but related operations, and to appreciate the results presented in this paper it is important to understand some of the different definitions.

Weiser [54] defined the *(static backward) slice of a program P from element q with respect to a set of variables V* to be any program P' such that

- P' can be obtained from P by deleting zero or more statements.
- Whenever P halts on input I, P' also halts on input I, and the two programs produce the same sequences of values for all variables in set V at element q if it is in the slice, and otherwise at the nearest successor to q that is in the slice.

The pair (q, V) is called the *slicing criterion.*

Most research on slicing adopts from Weiser the idea that slices should retain a close syntactic connection to the original program—roughly, algorithms for such approaches remove all program elements that cannot affect the slicing criterion. Such algorithms are called "syntax-preserving" [29]. In some slicing algorithms, the slicing criterion is restricted to involve only variables used at q, which allows slicing to be performed efficiently using program dependence graphs [35, 46].

Horwitz et al. [35] presented a context-sensitive algorithm for interprocedural slicing; however, they were careful to distinguish between two different types of slices:

Closure slices: The slice of a program from criterion (q, x) consists of all statements and conditions of the program that might affect the value of x at element q.

Executable slices: The slice of a program from criterion (q, x) consists of a reduced program that computes the same sequence of values for x at q.

(The terms "executable slice" and "closure slice" are due to Venkatesh [53].) The algorithm given by Horwitz et al. computes a closure slice.

This paper presents a new approach to creating executable slices. An example that illustrates the difference between executable and closure slices is shown in Fig. 1. Fig. 1(a) shows the original program; Figs. 1(b)–(d) show the slices produced by three different algorithms, all slicing with respect to `g2` at line (17).

While a closure slice (see Fig. 1(b)) can be useful—e.g., for program understanding—there are some contexts in which an executable slice (cf. Figs. 1(c) and (d)) is preferable. For example, in general, the smaller a program is, the easier it is to debug. Given a program that fails (throws an exception or prints a bad value) at an element q, if the executable slice from q is substantially smaller than the original program, then debugging the slice is likely to be easier than debugging the whole program. While it may be helpful for the programmer to examine the closure slice from q, being able to actually run the slice on different inputs may give the programmer much more leverage. Other applications of executable slicing include (i) extracting specialized components from application programs (e.g., creating a version of the word-count utility `wc` that counts only lines), and (ii) creating versions of libraries specialized to an application.

Because a slicing algorithm removes program elements when it establishes that those elements cannot affect the behavior at the slicing criterion, an executable slice can run faster than the original program. There is no *a priori* bound on the speed-up achievable: for some programs and some slicing criteria, a slice can be arbitrarily faster than the original program. An experiment with `wc` showed that on average—computed as the geometric mean—its executable slices took 32.5% of the time used by the original program (see §7).

(a) Example program

```
 (1)  int g1, g2, g3;
 (2)
 (3)  void p(int a, int b) {
 (4)    g1 = a;
 (5)    g2 = b;
 (6)    g3 = g2;
 (7)  }
 (8)
 (9)
(10)
(11)
(12)  int main() {
(13)    g2 = 100;
(14)    p(g2, 2);
(15)    p(g2, 3);
(16)    p(4, g1+g2);
(17)    printf("%d", g2);
(18)  }
```

(b) Closure slice with missing actuals

```
int g1, g2;

void p(int a, int b) {
  g1 = a;
  g2 = b;

}

int main() {

  p(   2);
  p(g2, 3);
  p(   g1+g2);
  printf("%d", g2);
}
```

(c) Binkley slice with matched actuals

```
int g1, g2;

void p(int a, int b) {
  g1 = a;
  g2 = b;

}

int main() {
  g2 = 100;
  p(g2, 2);
  p(g2, 3);
  p(4, g1+g2);
  printf("%d", g2);
}
```

(d) Specialized slice with two versions of *p*

```
int g1, g2;

void p_1(int b) {
  g2 = b;
}

void p_2(int a, int b) {
  g1 = a;
  g2 = b;
}

int main() {

  p_1(2);
  p_2(g2, 3);
  p_1(g1+g2);
  printf("%d", g2);
}
```

Figure 1. Example program and the results of three different algorithms for backwards slicing with respect to g2 at line (17).

One of the problems we encounter in creating executable slices is the *parameter-mismatch problem* ([35, §1] and [7]). A closure slice can have multiple calls to the same procedure, with different subsets of actual parameters at different call-sites. However, the slice has the union of the corresponding formal-parameter sets, which causes a mismatch between the actual parameters passed from a call-site and the procedure's formal parameters (e.g., lines (3) and (14)–(16) of Fig. 1(b)). For most programming languages, a program with such a mismatch would trigger a compile-time error.

According to Binkley, Weiser's slicing algorithm avoids the parameter-mismatch problem because "call sites are treated as indivisible components: if a slice includes one parameter, it must include all parameters" [7, p. 32]. While this approach ensures that Weiser's slices are executable, they can include many irrelevant components.[1]

Binkley [7] addressed the parameter-mismatch problem by expanding the closure slice of Horwitz et al. to include missing actual parameters, together with everything in the backward slice from the missing actuals. For instance, as shown in Fig. 1(c), Binkley's algorithm would include the missing first parameters at the two calls to p on lines (14) and (16), as well as the assignment to g2 at line (13). Although this approach leads to smaller slices than Weiser's approach, both Binkley's and Weiser's approaches have the undesirable property that they can cause arbitrarily many additional program elements to be included in the final slice.

In this paper, we present a new algorithm for finding executable slices. We take a different approach than Weiser and Binkley, and allow a looser form of slicing in which multiple versions of a procedure can be included in a slice.[2] The field of partial evaluation [39] was one of the inspirations for our work. Partial evaluation is a form of program specialization. Backward slicing can also be harnessed for program specialization [48], although algorithms for slicing and partial evaluation are much different in character. Heretofore, specialization via slicing has also been more restricted than the kind of specialization allowed in partial evaluation. A syntax-preserving slicing algorithm corresponds to what the partial-evaluation community would call a *monovariant* algorithm: each program element of the original program generates *at most one element* in the answer. In contrast, partial-evaluation algorithms can be *polyvariant*, i.e., one program element in the original program may correspond to more than one element in the specialized program [39, p. 370].

The slicing algorithm presented in this paper is polyvariant. In particular, when a closure slice has parameter mismatches for some procedure p, our algorithm creates multiple specialized copies of p. For example, as shown in Fig. 1(d), the calls to p at lines (14) and (16) are converted into calls to the one-parameter procedure p_1, while the call to p at line (15) is converted to a call to p_2.

In creating specialized procedures, we need to know for which formals the procedures should be specialized, and which program elements should be in each specialized procedure. As we show in §2.4 and App. A.2.3, there can be cascade effects: when a specialized copy of p is created, it may be necessary to create specialized copies of procedures called by p and so on. The process cannot go on forever, because there are only a finite number of combinations of actuals that are possible; however, the cascade effect can be exponential in the worst case (App. A.2.3).

Exponential explosion is not desirable, of course. Thus, the goal of our work was two-fold:

1. We wished to understand the fundamental underpinnings of the procedure-specialization approach. In particular, Defn. 2.8 provides a minimality criterion for specialized slices, and Thm. A.2 establishes that our slicing algorithm satisfies that criterion.
2. We also wished to understand whether the technique is efficient in practice, given the theoretical possibility of the cascade effect. Our experiments showed that no procedure had more than four specialized versions, and the vast majority of procedures had just a single version (see Fig. 16).

The Key Insight. We formalize the executable-slicing problem as a partitioning problem on a graph that, in general, can be infinite (§2.2). To represent finitely the infinite sets of objects that we use to solve the partitioning problem, we make use of symbolic techniques originally developed in the model-checking community

[1] Another disadvantage of Weiser's algorithm is that it is context-insensitive. That is, if a slice includes one call-site on procedure p, then it includes all call-sites on p, which is clearly undesirable.

[2] Harman and Danicic [29] studied what they call *amorphous slicing*, which drops the requirement of syntax preservation. What distinguishes an amorphous slice is merely that the number of nodes in the slice's control-flow graph (CFG) is no greater than the number of nodes in the original program's CFG. We have no such restriction, and in fact a slice created by our algorithm could be larger than the original program. Consequently, a slice returned by our algorithm does not always qualify as an amorphous slice; however, our work is in somewhat the same spirit—especially the material in §5.2 on function pointers and indirect calls.

[12, 24]. We show how to obtain the desired answer by performing just a few simple automata-theoretic operations.

Contributions. The contributions of the paper include

- The formalization of (a new version of) the executable-slicing problem as a partitioning problem on an infinite graph (§2.2).
- Definitions of *minimality*, *soundness*, and *completeness* for specialized slices (Defns. 2.8 and 2.7), and proofs that our algorithm—unlike Weiser's algorithm and Binkley's algorithm—satisfies these properties (Thms. A.2 and A.3).
- An algorithm that uses automata-theoretic operations to identify the minimal set of specialized procedures, as well as the minimal set of program elements required in each specialized procedure (§2.5 and §4).
- Characterizations of the running time and space used by our algorithm (App. A).
  - We present a family of examples for which the running time and space of the specialized-slicing algorithm can be exponential in certain parameters of the input program.
  - Our experience to date has been that neither such examples, nor the worst-case exponential behavior of operations like automaton determinization, arise in practice (see below). Hence, we believe it is fair to say that, for the *observed cost*, both the running time and space of the algorithm are bounded by the sum of two terms: one is polynomial in the size of the input program; the other is linear in the size of the output slice.
- Solutions for some additional issues that arise with more realistic languages than considered in past work on executable slicing (§5).
- Results of an experimental evaluation of the algorithm applied to C programs (§7). Our experiments were designed to
  - compare our approach to finding executable slices with Binkley's approach
  - determine whether the worst-case exponential cost of our algorithm arises in practice
  - determine how the execution time of an executable slice—which, in principle, can be arbitrarily better than that of the original program—compares in practice with the execution time of the original program.

  The experiments used slices taken with respect to slicing criteria obtained from run-time states at which symptoms of bugs were observed. The experiments showed that
  - Exponential explosion does not arise in practice: no procedure had more than four specialized versions, and the vast majority of procedures had just a single version (see Fig. 16).
  - Some executable slices run significantly faster than the original program.

Organization. §2 provides an overview of the approach that we use to create executable slices. §3 defines the infinite graphs that we use, and reviews the representation techniques upon which our slicing algorithm is based. §4 presents the core partitioning algorithm that solves our version of the executable-slicing problem. §5 describes how to extend our approach to handle programs that (i) make calls to library procedures, and (ii) use calls via pointers to procedures. §6 discusses limitations of the approach. §7 presents our experimental results. §8 discusses related work. Proofs are given in App. A.

2. Overview and Problem Statement

2.1 System Dependence Graphs

A *system dependence graph* (SDG) [35] is a graph used to represent multi-procedure programs ("systems"). The system dependence graph is similar to other dependence-graph representations of programs (e.g., [41, 46]), but represents collections of procedures rather than just monolithic programs.

We will not give a detailed definition here; however, the important ideas should be clear from the examples. A program's SDG is a collection of procedure dependence graphs (PDGs) [46], one for each procedure. The vertices of a PDG represent the individual statements and conditions of the procedure. A call statement is represented by a call vertex and a collection of actual-in and actual-out vertices: There is an actual-in vertex for each actual parameter as well as for the non-local locations—e.g., global variables and locations accessible via pointers—in the procedure's MayRef and (MayMod – MustMod) sets [18]. There is an actual-out vertex for the return value and for each non-local location in the procedure's MayMod set. Similarly, in the PDG of a called procedure, the parameter-passing actions in the procedure's prologue and epilogue are represented by an entry vertex and a collection of formal-in and formal-out vertices. The edges of a PDG represent the control and flow dependences among the procedure's statements and conditions.[3] The PDGs are connected together to form the SDG by *call* edges (which represent procedure calls, and run from a call vertex to an entry vertex) and by *parameter-in* and *parameter-out* edges (which represent parameter passing, and which run from an actual-in vertex to the corresponding formal-in vertex, and from a formal-out vertex to all corresponding actual-out vertices, respectively).[4]

Example 2.1. Fig. 2 shows the SDG for the program given in Fig. 1(a). Each PDG vertex in the SDG is labeled (e.g., $m1$), and each call-site has an additional label of the form $C1$, $C2$, etc. We later refer to the vertices and call-sites using those labels.

Because the call to `printf` is a library call, the SDG shown in Fig. 2 does not include the PDG for `printf`. We discuss how we handle library calls in §5.1.

We will return to this example several times to illustrate the steps that our algorithm performs to create an executable program for a backward slice from vertex set $\{m21, m22, m23\}$—which corresponds to the slice of Fig. 1(a) with respect to `g2` at line (17). □

2.2 Problem Statement and Key Insights

Insight 1: Symbolic Techniques for Infinite Graphs. We formalize the executable-slicing problem in terms of the (conceptual) unrolled SDG, which for recursive programs has an infinite number of vertices and edges. (Roughly, the unrolled SDG corresponds to the SDG of the program that is obtained by repeatedly—and possibly infinitely—performing in-line expansion in procedure `main`.) Each vertex c in the unrolled SDG can be labeled uniquely with a pair (v, w), where v is a PDG vertex, and w is a sequence of call-sites in the SDG. In a minor abuse of terminology, we do not distinguish between c and its label (v, w), and say that $c = (v, w)$ is a *configuration*. The calling context w represents the stack of calls that are pending in configuration c. Given a set of configurations C, Elems(C) denotes the set $\{v \mid (v, w) \in C\}$, and Stacks(C) denotes the set $\{w \mid (v, w) \in C\}$.

[3] As defined by Horwitz et al. [35], PDGs include four kinds of dependence edges: *control*, *loop-independent flow*, *loop-carried flow*, and *def-order*. However, for the purposes of this paper the distinction between loop-independent and loop-carried flow edges is irrelevant, and def-order edges are not used. Therefore, in this paper we assume that PDGs include only control-dependence edges and a single kind of flow-dependence edge.

[4] The SDGs defined by Horwitz et al. [35] also include *summary edges*, which run from actual-in to actual-out vertices. However, summary edges are not necessary for our executable-slicing algorithm.

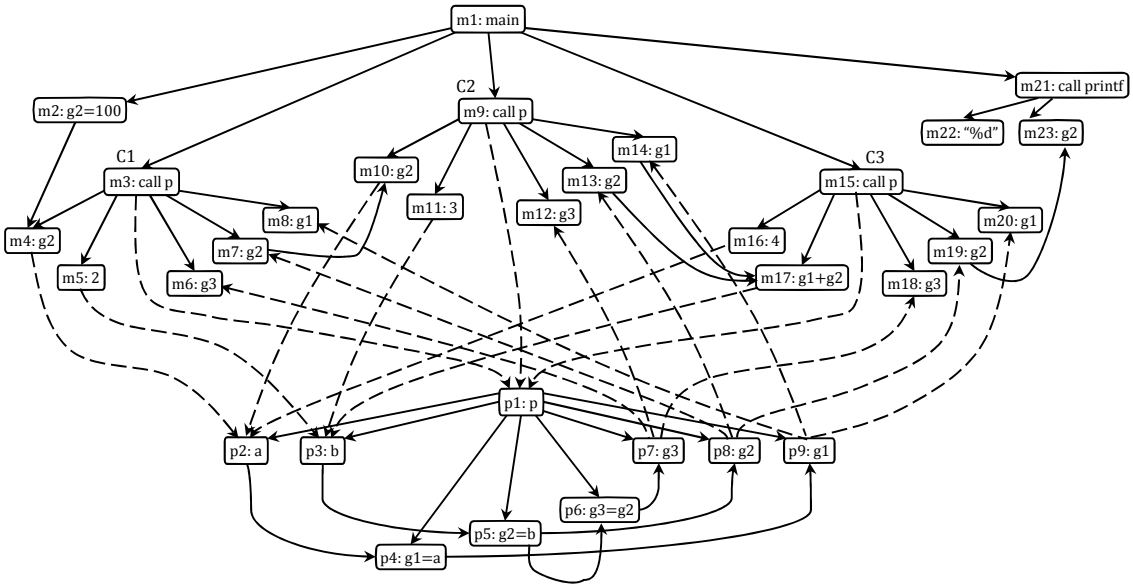

Figure 2. System dependence graph (SDG) for the program shown in Fig. 1(a).

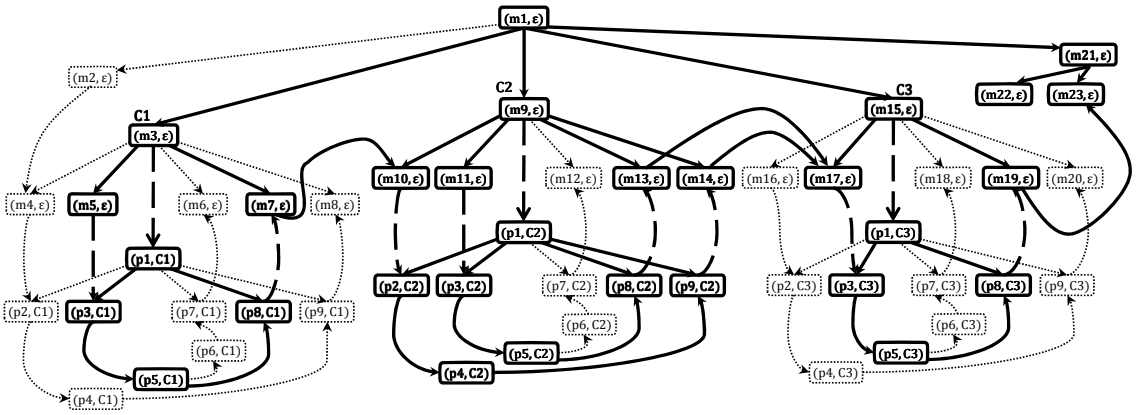

Figure 3. Unrolled SDG of the program shown in Fig. 1(a). Each vertex is labeled with a configuration of the form $(\text{PDG-vertex}, \text{stack-configuration})$. The closure slice of the unrolled SDG from configuration set $\{(m21, \epsilon), (m22, \epsilon), (m23, \epsilon)\}$ is shown with bold font and darker borders, lines, and dashed lines, and corresponds to the program in Fig. 1(d).

Example 2.2. Fig. 3 shows the unrolled SDG that corresponds to the program shown in Fig. 1(a). Each vertex in Fig. 3 is labeled with a configuration of the form $(\text{PDG-vertex}, \text{stack-configuration})$. □

To represent finitely the infinite sets of objects that we will use to solve the executable-slicing problem, we adopt symbolic techniques that were originally developed in the model-checking community. We use a pushdown system (PDS) [12, 24] to encode the unrolled SDG (see §3.1). This approach immediately provides us with an *algorithm* to perform a generalized kind of program slicing: the pre^* operation [12, 24] on a PDS performs a closure slice on the unrolled SDG. We call this operation ***stack-configuration slicing*** to distinguish it from the other kinds of slicing operations that were mentioned earlier.

Example 2.3. The stack-configuration slice of the program shown in Fig. 1(a) from line (17) corresponds to the closure slice of the unrolled SDG shown in Fig. 3 from configuration set $\{(m21, \epsilon), (m22, \epsilon), (m23, \epsilon)\}$. The elements of the slice are shown in Fig. 3 with bold font and darker borders, lines, and dashed lines. □

Stack-configuration slicing finds all (PDG-vertex, call-stack) configurations on which a given language of (PDG-vertex, call-stack) configurations depend. As is well-known in the literature on PDSs, for PDSs of recursive programs, the answer to such a query can be an *infinite* set. Nevertheless, an answer can be computed and stored in a *finite* amount of memory by using a finite-state automaton (FSA) to represent an answer set symbolically. (See §2.4 and §3.1.)

Definition 2.4. *The unrolled SDG may contain many **instances** of a procedure P; each instance of procedure P is a set C_w of configurations of the form (v, w), where all the w are the same—i.e., $C_w \stackrel{\text{def}}{=} \{(v, w) \in \text{unrolled SDG} \mid v \in P\}$. For a given stack-configuration slice, a **variant** of P in the slice is the subset $S_w \subseteq C_w$ of some instance C_w. A **specialization** of P is the set of PDG vertices $Elems(S_w)$ (for some variant S_w in the slice).* □

Example 2.5. In Fig. 3, there are three instances of procedure p: the vertices whose call-stack components are $C1$, $C2$, and $C3$. However, there are only two *specializations* of p because the variants of the $C1$ and $C3$ instances have the same Elems components. Consequently, Fig. 1(d) has two specialized versions of p. □

Problem Statement. With these concepts, we can formulate the problem of executable slicing via procedure specialization. The core problem is to identify, for each procedure P, all of the different sets of program elements among all variants of P:

$$\text{Specializations}(P) = \{\text{Elems}(V) \mid V \text{ a variant of } P \text{ in the stack-configuration slice}\}. \quad (1)$$

In general, in a closure slice of the unrolled SDG for a recursive program, the number of variants V of a procedure P can be infinite. However, because stack-configuration components are ignored in Elems(V), there are only a finite number of Elems(V) sets. Therefore, Specializations(P) is a *finite* set (i.e., a finite set, each of whose members is a finite set of program elements).

To create the SDG for the answer program, a specialized version of procedure P's PDG is instantiated for each different Elems(V) set in Specializations(P). The specialized PDGs are connected together to form the specialized SDG.

Example 2.6. Fig. 4 shows the specialized SDG that corresponds to the closure slice shown in Fig. 3. Each of the three specialized PDGs in Fig. 4 corresponds to one of the three Elems(V) sets in Fig. 3. The SDG in Fig. 4 corresponds to the program shown in Fig. 1(d). □

The final step is to pretty-print source-code text from the specialized SDG. (This step is straightforward, but lies outside the scope of the paper.)

Insight 2: Soundness and Completeness with respect to Stack–Configuration Slicing. Two properties that an executable-slicing method may or may not possess are what we will call *soundness* and *completeness* with respect to stack-configuration slicing. To understand what we mean by these terms, consider the following situation: Suppose that for SDG S and slicing criterion C, the resulting SDG is R. Ideally, the unrolling of R should be identical to the closure slice from C of the unrolling of S; that is, the unrolling of R should have two properties:

1. It should contain *all* of the elements in the closure slice from C of the unrolling of S (*completeness*).
2. It should contain *only* elements that are in the closure slice from C of the unrolling of S (*soundness*).

However, because the vertices and call-site labels in S and R are different, the properties as stated above are not achievable. The naming differences create a problem because the vertices and call-site labels are alphabet symbols in the configurations of the unrollings of R and S. Consequently, the notions of soundness and completeness cannot based on pure equality; instead, they must account for the changes in the alphabet symbols. Fortunately, it is easy to identify each vertex or call-site in R as the specialization of some vertex or call-site in S. This information can be used to define a mapping M_C that maps SDG vertices and call-sites in R one-to-one into the alphabet of S. Using M_C, we can restate the concepts of soundness and completeness as follows:

Definition 2.7. *Let A be a slicing algorithm, and consider all S, C, R, and M_C (as described above) created by A. Let G_R be the unrolling of R.*

1. *A is **complete** if each $M_C(G_R)$ contains all of the elements in the closure slice from C of the unrolling of S.*
2. *A is **sound** if each $M_C(G_R)$ contains only elements that are in the closure slice from C of the unrolling of S.*

□

The algorithm presented in this paper is both sound and complete (see Thm. A.3). Weiser's algorithm [7] and Binkley's algorithm [7] are both complete, but can include extra program elements, and hence are not sound.

Given the level of sophistication of the machinery that we use in §3 and §4, it is natural to wonder whether a simpler algorithm is possible. In particular, one approach to specializing a PDG would be to remove vertices that are in the *forward* slice [35, §4.5] from *unmatched* formal-in vertices. We developed such an algorithm, and found that the forward-slicing approach can leave in unneeded vertices in some cases. That is, such an algorithm is complete, but not sound. (Space limitations preclude presenting an example.)

The failure of the forward-slicing approach motivated us to investigate the fundamental principles underlying specialization slicing, and to formulate the declarative conditions given as Defns. 2.7 and 2.8 (see below).

Insight 3: Formulation as a Partitioning Problem. Because the unrolled SDG for the program being sliced can, in general, be infinite, the search for the sets Specializations(P), which make up the different PDGs of the answer SDG, must be carried out symbolically. We formulate this search as the partitioning problem defined below in Defn. 2.8. (To avoid ambiguity, we use the term "partition" for a collection of non-overlapping sets that subdivide a given set, and use "partition-element" for an individual set that is part of the partition.)

Definition 2.8 (Configuration-Partitioning Problem)**.** *Given a stack-configuration slice, the **configuration-partitioning problem** is to find a finite partition of the slice's configurations such that*

1. *For each variant V of some procedure P, all configurations in V are in the same partition-element.*
2. *For each pair of procedures P and Q, $P \neq Q$, no partition-element contains configurations from a variant of P and a variant of Q.*
3. *Let P be a procedure and A and B be a pair of different variants of P. Let E be the partition-element that contains the configurations in A (i.e., $A \subseteq E$). Then $B \subseteq E$ iff Elems(A) = Elems(B).*

Note that the partition is finite, although each partition-element may consist of configurations from an infinite number of variants. □

Defn. 2.8 is a declarative specification of the partitioning problem, but does not provide a method to construct the desired partition. The intuition behind Defn. 2.8 is as follows:

- For each configuration (v, w) in the stack-configuration slice of the program, there needs to be some specialized procedure that can be called with the stack-configuration w. Each partition-element corresponds to a specialization of some procedure.
- The partition-elements should only include configurations that are in the stack-configuration slice.

Because of the "iff" in item 3, Defn. 2.8 defines a unique partition. Suppose that item 3 were written as "$\ldots B \subseteq E$ implies Elems(A) = Elems(B)." Because we want the specialized program to consist of as few specialized procedures as possible, we would want the coarsest partition.

Observation 2.9. *Each partition-element E in the partition defined in Defn. 2.8 is associated with a language of stack-configurations: Stacks(E). **The collection of such languages is pairwise disjoint**—i.e., the set of languages $\{$Stacks$(E)\}$ partitions the set of stack-configurations in the stack-configuration slice.* □

Alternatively, the problem could have been formulated as a (different) partitioning problem on the *variants* in the stack-configuration slice. However, the technique used in §4 to identify the partition manipulates descriptions of sets of *configurations*, not variants; consequently, Defn. 2.8 more closely matches the concepts needed to understand our presentation of the algorithm.

In §2.5 and §4, we show how to identify the desired finite partition by performing just a few simple automata-theoretic operations.

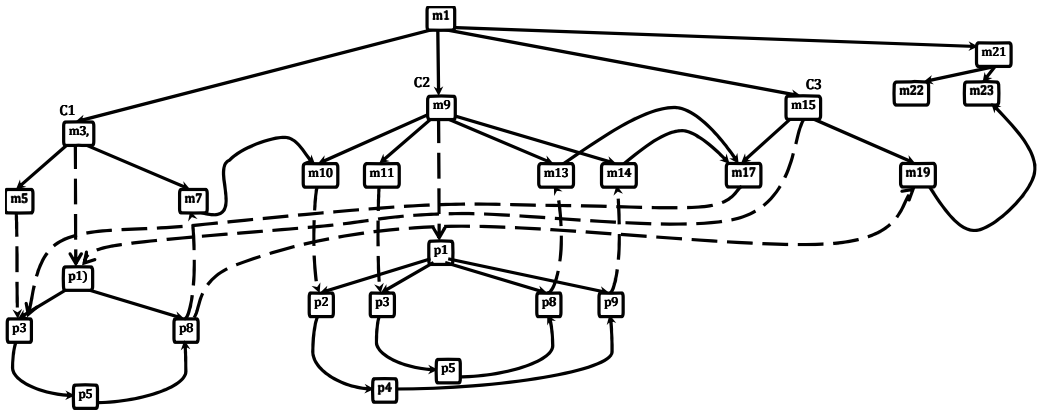

Figure 4. The specialized SDG for the closure slice shown in Fig. 3.

After these steps, the answer is available in the form of an automaton, from which we are able to read off the SDG of the desired specialized program. As shown in Cor. A.4, the SDG obtained via our algorithm avoids the parameter-mismatch problem.

2.3 A Non-Recursive Example

This section and §2.4 present two examples that illustrate how a solution to Defn. 2.8 identifies the program elements that make up each of the specialized procedures of an executable slice. This section discusses the slice of a non-recursive program; §2.4 discusses the slice of a recursive program. In this section, we consider the program shown in Fig. 1(a) sliced from with respect to g2 at (17), and explain how Defn. 2.8 identifies the program elements in each of the specialized procedures shown in Fig. 1(d).

The SDG for the program in Fig. 1(a) is shown in Fig. 2, and its unrolled SDG is shown in Fig. 3. In an unrolled SDG, each call-site invokes a unique instance of the called PDG. Consequently, each vertex of the unrolled SDG is associated with a unique stack-configuration. In Fig. 3, each vertex is labeled with its stack-configuration. A vertex in `main` has no calling context, and thus its stack-configuration is ϵ. If we consider all vertices of an unrolled SDG with a label of the form (v, w), the set $\{w \mid (v, w)\}$ is an over-approximation of the set of calling contexts that can arise for program element v when the program executes.

Fig. 3 uses bold font and darker borders, lines, and dashed lines to indicate the closure slice of the unrolled SDG from $\{(m21, \epsilon), (m22, \epsilon), (m23, \epsilon)\}$. The set of configurations in the slice is

$$\left\{\begin{array}{cccc} (m1,\epsilon) & (m13,\epsilon) & (p1,C1) & (p1,C2) \\ (m3,\epsilon) & (m14,\epsilon) & (p3,C1) & (p2,C2) \\ (m5,\epsilon) & (m15,\epsilon) & (p5,C1) & (p3,C2) \\ (m7,\epsilon) & (m17,\epsilon) & (p8,C1) & (p4,C2) \\ (m9,\epsilon) & (m19,\epsilon) & (p1,C3) & (p5,C2) \\ (m10,\epsilon) & (m21,\epsilon) & (p3,C3) & (p8,C2) \\ (m11,\epsilon) & (m22,\epsilon) & (p5,C3) & (p9,C2) \\ & (m23,\epsilon) & (p8,C3) & \end{array}\right\} \quad (2)$$

According to Defn. 2.8, we can find the PDGs of the executable slice by partitioning the preceding set into

$$\left\{\left\{\begin{array}{cc} (m1,\epsilon) & (m13,\epsilon) \\ (m3,\epsilon) & (m14,\epsilon) \\ (m5,\epsilon) & (m15,\epsilon) \\ (m7,\epsilon) & (m17,\epsilon) \\ (m9,\epsilon) & (m19,\epsilon) \\ (m10,\epsilon) & (m21,\epsilon) \\ (m11,\epsilon) & (m22,\epsilon) \\ & (m23,\epsilon) \end{array}\right\}, \left\{\begin{array}{c} (p1,C1) \\ (p3,C1) \\ (p5,C1) \\ (p8,C1) \\ (p1,C3) \\ (p3,C3) \\ (p5,C3) \\ (p8,C3) \end{array}\right\}, \left\{\begin{array}{c} (p1,C2) \\ (p2,C2) \\ (p3,C2) \\ (p4,C2) \\ (p5,C2) \\ (p8,C2) \\ (p9,C2) \end{array}\right\}\right\} \quad (3)$$

The configurations in the first partition-element correspond to the main-procedure PDG in the specialized SDG. The configurations

(a) Recursive program	(b) Executable slice
(1) `int g1, g2;`	`int g1, g2;`
(2)	
(3) `void s(int a,`	`void s_1(int b) {`
(4) `int b){`	`g1 = b;`
(5)	`}`
(6) `g1 = b;`	`void s_2(int a) {`
(7) `g2 = a;`	`g2 = a;`
(8) `}`	`}`
(9)	
(10)	`void r_1(int k) {`
(11)	`if (k > 0) {`
(12) `int r(int k) {`	`s_2(g1);`
(13)	`r_2(k-1);`
(14) `if (k > 0) {`	`s_1(g2);`
(15) `s(g1, g2);`	`}`
(16) `r(k-1);`	`}`
(17) `s(g1, g2);`	`void r_2(int k) {`
(18) `}`	`if (k > 0) {`
(19)	`s_1(g2);`
(20) `}`	`r_1(k-1);`
(21)	`s_2(g1);`
(22)	`}`
(23)	`}`
(24) `int main() {`	
(25) `g1 = 1;`	`int main() {`
(26) `g2 = 2;`	`g1 = 1;`
(27) `r(3);`	`r_1(3);`
(28) `printf("%d\n", g1);`	`printf("%d\n", g1);`
(29) `}`	`}`

Figure 5. An example program with recursive procedure `r` and the executable slice of the program with respect to `g1` at line (28).

in the second partition-element form two variants V_1 and V_2 of procedure `p`. They satisfy rules (1) and (3) of Defn. 2.8, and so form a single partition-element that corresponds to the specialized version of `p` named `p_1` in Fig. 1(d). The configurations in the third partition-element correspond to the second specialized version of `p`, named `p_2` in Fig. 1(d). (Configurations that are not in the closure slice, such as $(m8, \epsilon)$, are not part of any variant, and hence do not contribute vertices to the specialized SDG.)

Note that some elements of the original program, e.g., $p5$, occur in more than one partition-element. Hence $p5$ is specialized into two program elements, one in procedure `p_1` and one in `p_2`.

2.4 A Recursive Example

The partitioning problem in the example discussed in §2.3 is quite simple, in part because the unrolled SDG in Fig. 3 is finite. In contrast, as illustrated by the example discussed in this section, for a program that uses recursion, the unrolled SDG is infinite. More-

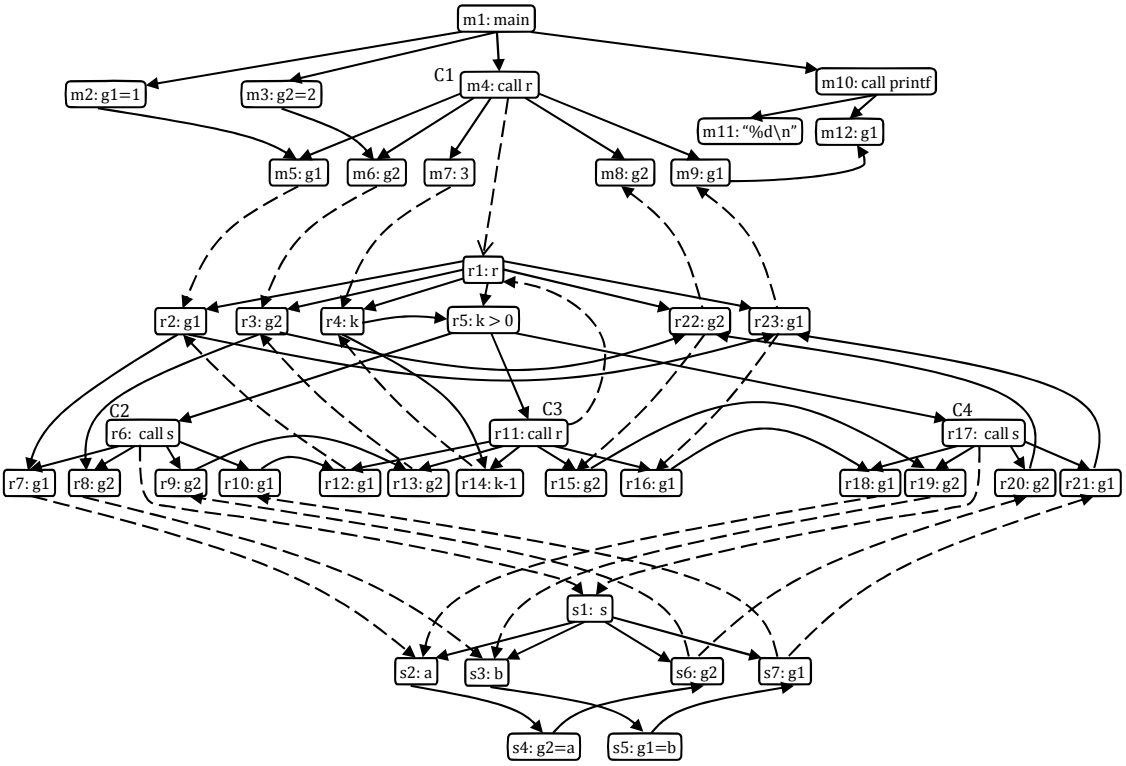

Figure 6. The SDG of the recursive program shown in Fig. 5(a).

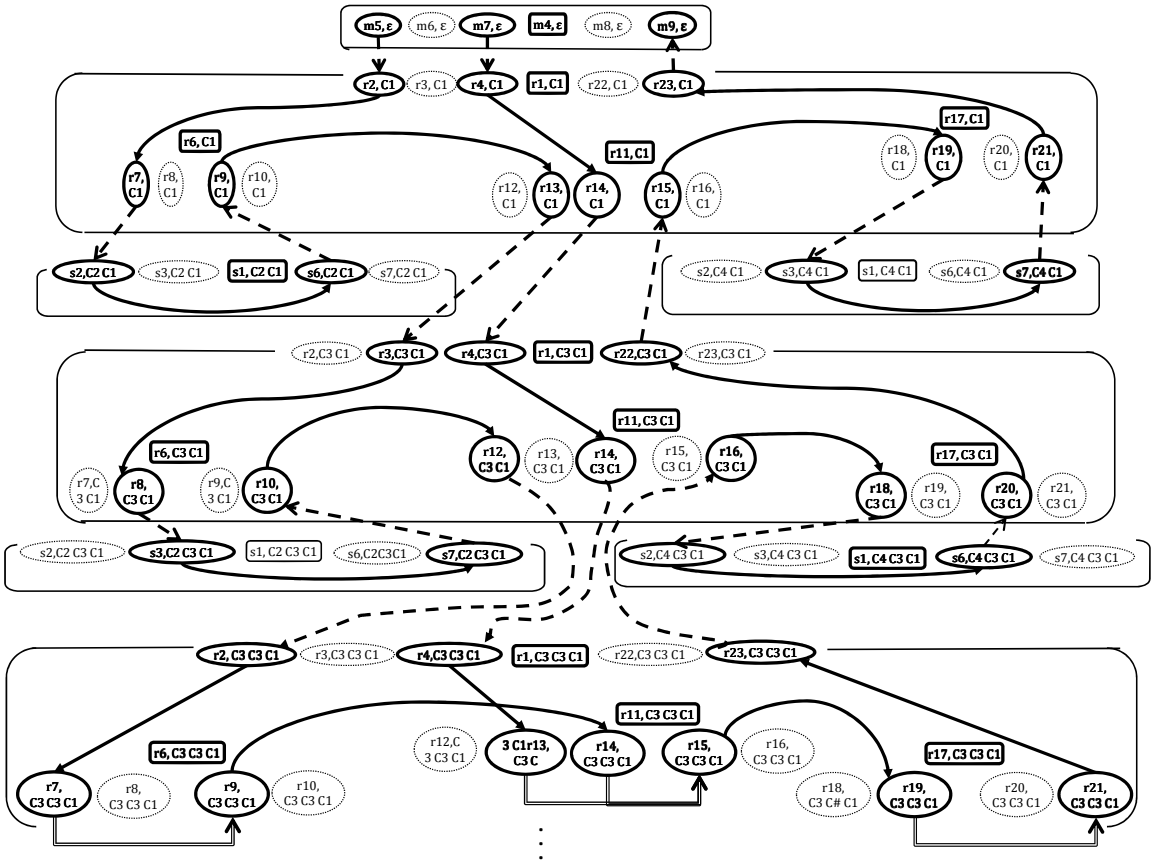

Figure 7. The structure of the unrolled SDG of the recursive program shown in Fig. 5(a). Only some of the edges in the slice are shown.

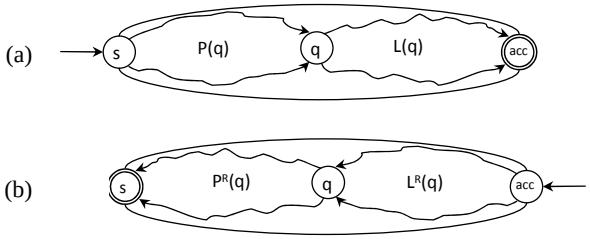

Figure 8. (a) Depiction of the prefix and suffix languages associated with state q in automaton $\mathcal{A}$. (b) Reversal of (a) (denoted by $R(\mathcal{A})$). $P^R(q)$ and $L^R(q)$ denote the respective languages of reversed strings.

over, for some slicing criteria of the unrolled SDG, the set of configurations that we need to partition can be of infinite cardinality. Fortunately, the partition that satisfies Defn. 2.8 is still finite.

Consider the recursive program shown in Fig. 5(a) and a slice of the program with respect to `g1` at line (28). The SDG of the program is shown in Fig. 6, and the slicing criterion that corresponds to Fig. 5(a) line (28) is $\{m10, m11, m12\}$.

Fig. 7 shows the structure of the unrolled SDG with vertices only for procedure-entry, call-sites, actual parameters, and formal parameters. Entry and call-site vertices are enclosed in rectangles, while parameter vertices are enclosed in ovals. As in Fig. 3, bold font and darker borders, lines, and dashed lines are used to indicate the vertices that are in the stack-configuration slice; however, to reduce clutter, only some of the edges in the slice are shown.

Fig. 7 includes three variants of procedure `r`, whose configurations have stack-configurations ($C1$), ($C3$ $C1$), and ($C3$ $C3$ $C1$), respectively. The first and third variants of `r` correspond to the same specialized version of `r`, because they have the same Elems components in the stack-configuration slice. (The complete set of PDG vertices in the ($C1$) and ($C3$ $C3$ $C1$) variants, not all of which are shown in Fig. 7, is $\{r1, r2, r4, r5, r6, r7, r9, r11, r13, r14, r15, r17, r19, r21, r23\}$.) In fact, the infinite set of variants of `r` whose configurations include call-stacks of the form $(C3\ C3)^*\ C1$—i.e., an even number of recursive calls via call-site $C3$—all correspond to that same specialized version. Therefore, those configurations make up one partition-element, which gives rise to one specialized version of `r`, namely `r_1` in Fig. 5(b).

Similarly, all variants of `r` whose configurations include call-stacks of the form $(C3\ C3)^*\ C3\ C1$—i.e., an odd number of recursive calls on $C3$—make up another partition-element, and give rise to a second specialized version of `r`, namely `r_2` in Fig. 5(b). Procedures `r_1` and `r_2` are mutually recursive.

2.5 An Automaton-Based Solution to Partitioning

We now describe how to identify the desired finite partitioning by performing just a few simple automata-theoretic operations. Each such operation manipulates indirectly the possibly infinite sets of configurations that are part of Defn. 2.8. Because configuration sets can be infinite, we represent each configuration set C symbolically, in terms of an automaton that accepts exactly the configurations that are members of C. (A configuration (v, w) is accepted by an automaton $\mathcal{A}$ iff the string vw is in $L(\mathcal{A})$.)

Moreover, because we are interested in partitioning the language of represented configurations, we will exploit the fact that each state of an automaton can be thought of as defining two languages. For instance, consider the FSA depicted in Fig. 8(a). Each state q defines (i) the prefix language $P(q)$ of strings accepted by considering q as the (only) final state, and (ii) the suffix language $L(q)$ of strings accepted by considering q as the initial state.

Example 2.10. Let us return to the example discussed in §2.3—i.e., Fig. 1(a) sliced with respect to `g2` at line (17), or equivalently,

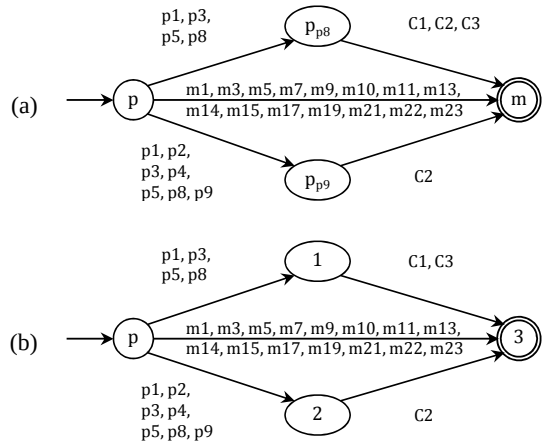

Figure 9. (a) An automaton that accepts the configurations of the stack-configuration slice of Fig. 1(a) from line (17). (b) A minimal-reverse-deterministic (MRD) automaton for the same language. As discussed in Ex. 2.10, one can immediately read off the answer from the MRD automaton.

the closure slice of Fig. 3 from $\{(m21, \epsilon), (m22, \epsilon), (m23, \epsilon)\}$. Fig. 9(a) shows an automaton that accepts the language of configurations in the closure slice. (The configurations were given explicitly in Eqn. (2).) For instance, the fact that $(p2, C2)$ is accepted by the automaton in Fig. 9(a) means that $(p2, C2)$ is in the closure slice of Fig. 3 from $\{(m21, \epsilon), (m22, \epsilon), (m23, \epsilon)\}$.

Fig. 9(a) is not the only automaton that accepts the configurations in the closure slice. However, we will make use of a technique (discussed in §3.1) that, for this example, constructs Fig. 9(a) to represent the closure slice. Fig. 9(a) does not immediately provide a solution to the configuration-partitioning problem (Defn. 2.8) in the sense that the three sets of the partition given in Eqn. (3) are not immediately apparent from Fig. 9(a).

Instead, we would rather have the automaton shown in Fig. 9(b). We can immediately read off the answer from Fig. 9(b):

- The label sets on the three transitions emanating from the initial state p represent the three sets of the partition given in Eqn. (3), and thus correspond to the program elements of the specialized procedures `p_1`, `p_2`, and `main` of Fig. 1(d).
- The non-initial states (1, 2, and 3) that are the targets of the three transitions emanating from state p represent, respectively, procedures `p_1`, `p_2`, and `main` of Fig. 1(d).
- The transitions $(1, C1, 3)$, $(1, C3, 3)$ and $(2, C2, 3)$ represent the two calls on `p_1` and the call on `p_2`, respectively, in the specialized version of `main`. (That is, these edges correspond to the call multi-graph of the specialized program.)

□

Because the program from Fig. 1(a) is such a simple, non-recursive program, the words accepted by Fig. 9(a) and (b) have at most two symbols. Suppose, however, that procedure `p` called an additional procedure `q` at call-site $C4$, and that `q` called itself recursively at $C5$. Suppose that $q5$ is one of the vertices in the PDG for `q`. In this case, the automaton returned by *Prestar* could have a cycle, allowing it to accept an infinite set of configurations, such as those of the form $(q5, C5^*\ C4\ C2)$ (as well as others). Note the order of symbols in such words: the call-site in `main` comes *last*.

We now seek (i) a condition that characterizes the essential property of Fig. 9(b), and (ii) an algorithm that lets us construct Fig. 9(b) from Fig. 9(a). The property possessed by Fig. 9(b) is that it is *minimal reverse-deterministic* (MRD)—i.e., it is a minimal deterministic FSA when considered as an automaton that accepts

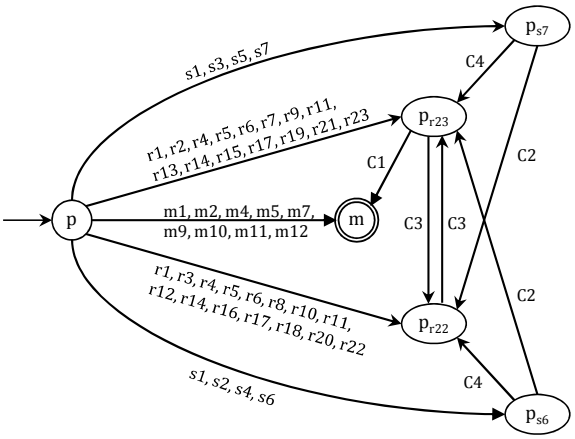

Figure 10. An automaton that accepts the configurations of the stack-configuration slice of Fig. 5(a) from line (28).

reversed strings via a backwards traversal along transitions, starting from the accepting state. To see why the MRD property is the one we seek, consider again the prefix and suffix languages, $P(q)$ and $L(q)$, respectively, discussed for automaton $\mathcal{A}$ from Fig. 8(a). Now suppose that $\mathcal{A}$ is deterministic. In this case, the set of languages $\{P(q) \mid q \text{ a state of } \mathcal{A}\}$ *partition* the prefix closure $L^{\leftarrow}(\mathcal{A})$ of $L(\mathcal{A})$.[5] That is, for each string $s \in L^{\leftarrow}(\mathcal{A})$, there is exactly one state q for which there is an s-path from the initial state to q.

As pointed out in Obs. 2.9, each specialized procedure is associated with a partition-element of a partition of the *stack-configurations*. Recall that in a configuration (v, w), w represents the stack of pending calls. If we determinized Fig. 9(a), the resulting $P(q)$ languages, where q is a state, would not be satisfactory: each word in one of the $P(q)$ languages starts with the symbol v for a PDG *vertex*, whereas the partition needed to identify specialized procedures should be based on the w part of a configuration. Moreover, because Fig. 9(a) recognizes a stack-configuration from top-of-stack to bottom-of-stack (i.e., `main`), its $P(q)$ languages recognize (PDG-vertex, partial-stack) pairs, where a "partial-stack" runs from top-of-stack to *middle*-of-stack. Such partial-stacks do not correspond to the languages of calling contexts for specialized procedures.

We are able to use determinization as a partitioning tool by observing that when we reverse an automaton $\mathcal{A}$—see Fig. 8(b)—a prefix-language $P_{R(\mathcal{A})}(q)$ in the reversed automaton $R(\mathcal{A})$ is the reversal of the suffix-language $L(q)$ of the original automaton; i.e., $P_{R(\mathcal{A})}(q) = L^R(q)$. Consequently, by determinizing the *reversed* automaton, the prefix languages identify a partition on stack-configurations. Moreover, the $P_{R(\mathcal{A})}(q)$ languages recognize a different kind of partial-stack: for a reversed automaton, a partial-stack runs from *bottom*-of-stack (`main`) to *middle*-of-stack. Such partial-stacks capture the languages of calling contexts for specialized procedures.

By minimizing the determinized reversed automaton (and then reversing the automaton that results), we find the desired MRD automaton. As shown in Thm. A.2, the automaton obtained in this manner solves the configuration-partitioning problem (Defn. 2.8).

Example 2.11. Consider again the recursive example discussed in §2.4. Fig. 10 shows an MRD automaton for the stack-configuration slice of Fig. 6 from $\{m10, m11, m12\}$. A comparison of Fig. 10 with Fig. 5(b) shows that the sets that label the five transitions that emanate from initial state `p` correspond to the five procedures of Fig. 5(b), namely, `s_1`, `s_2`, `r_1`, `r_2`, and `main`.

[5] If L is a language, the *prefix closure* $L^{\leftarrow}$ is $\{a \mid \exists b \text{ such that } ab \in L\}$.

Rule	Dependence edge modeled
$\langle p, u\rangle \hookrightarrow \langle p, v\rangle$	Flow- or control-dependence edge $u \to v$
$\langle p, c\rangle \hookrightarrow \langle p, e\ C\rangle$	Call edge from call vertex c to entry vertex e at call-site C
$\langle p, ai\rangle \hookrightarrow \langle p, fi\ C\rangle$	Parameter-in edge from actual-in vertex *ai* to formal-in vertex *fi* at call-site C
$\langle p, fo\rangle \hookrightarrow \langle p_{fo}, \epsilon\rangle$ $\langle p_{fo}, C\rangle \hookrightarrow \langle p, ao\rangle$	Parameter-out edge from formal-out vertex *fo* to actual-out vertex *ao* at call-site C

Figure 11. A schema for encoding an SDG's edges using PDS rules.

The transitions among states m, p_{r23}, p_{r22}, p_{s7}, and p_{s6} correspond to the call graph of Fig. 5(b). In particular, the languages $L(m)$, $L(p_{r23})$, $L(p_{r22})$, $L(p_{s7})$, and $L(p_{s6})$ in the automaton shown in Fig. 10 are exactly the stack-configuration languages for the different configuration partitions; i.e., each language equals Stacks(E), where E is one of the five partitions in the solution to the configuration-partitioning problem (Defn. 2.8) for the stack-configuration slice of Fig. 6 from $\{m10, m11, m12\}$. □

3. Stack-Configuration Slicing

This section defines the infinite graphs that we use—the transition relations of pushdown systems (PDSs) [12, 24]—and reviews the symbolic techniques for working with PDSs upon which our executable-slicing algorithm is based.

3.1 Pushdown Systems, SDGs, and Unrolled SDGs

Definition 3.1. *A **pushdown system** (PDS) is a triple $\mathcal{P} = (P, \Gamma, \Delta)$, where P is a finite set of **control locations**; Γ is a finite set of stack symbols; and $\Delta \subseteq P \times \Gamma \times P \times \Gamma^*$ is a finite set of rules. A $\mathcal{P}$-**configuration** is a pair (p, u) where $p \in P$ and $u \in \Gamma^*$. A rule $r \in \Delta$ is written as $\langle p, \gamma\rangle \hookrightarrow \langle p', u\rangle$, where $p, p' \in P, \gamma \in \Gamma$, and $u \in \Gamma^*$. The set of rules defines a **transition relation** $\Rightarrow$ on $\mathcal{P}$-configurations as follows: If $r = \langle p, \gamma\rangle \hookrightarrow \langle p', u'\rangle, then (p, \gamma u) \Rightarrow (p', u' u) for\ all\ u \in \Gamma^*$.*

*The reflexive transitive closure of $\Rightarrow$ is denoted by $\Rightarrow^*$. For a set of $\mathcal{P}$-configurations C, we define **pre*(C)** $= \{c' \mid \exists c \in C : c' \Rightarrow^* c\}$ and **post*(C)** $= \{c' \mid \exists c \in C : c \Rightarrow^* c'\}$, which are just backward and forward reachability under transition relation $\Rightarrow$.* □

Without loss of generality, we restrict a PDS's rules to have at most two stack symbols on the right-hand side [49]. A rule $r = \langle p, \gamma\rangle \hookrightarrow \langle p', u\rangle$, $u \in \Gamma^*$, is called a *pop* rule if $|u| = 0$, an *internal* rule if $|u| = 1$, and a *push* rule if $|u| = 2$.

Because the size of the stack component of a $\mathcal{P}$-configuration is not bounded, in general, the number of $\mathcal{P}$-configurations of a PDS—and hence its transition relation—is infinite.

Definition 3.2. *We **encode** an SDG as a PDS using the schema given in Fig. 11. The five kinds of edges that occur in SDGs are each encoded using one or two PDS rules:*

- *a flow-dependence or control-dependence edge is encoded with an internal rule*
- *a call edge or a parameter-in edge is encoded with a push rule*
- *a parameter-out edge is encoded with a pop rule and an internal rule.*

□

Example 3.3. Consider again the SDG from Fig. 2 and program from Fig. 1(a). The three call-sites on procedure `p` are labeled with $C1$, $C2$, and $C3$. Tab. 1 shows the PDS rules that encode the SDG from Fig. 2. □

Using the schema given in Fig. 11, a common control location p is used in all of the PDS rules, except in the rules that encode parameter-out edges. Reading the rules in the forward direction,

Control-dependence and flow-dependence edges in `main`	
1. $\langle p, m1\rangle \hookrightarrow \langle p, m2\rangle$	2. $\langle p, m1\rangle \hookrightarrow \langle p, m3\rangle$
3. $\langle p, m1\rangle \hookrightarrow \langle p, m9\rangle$	4. $\langle p, m1\rangle \hookrightarrow \langle p, m15\rangle$
5. $\langle p, m1\rangle \hookrightarrow \langle p, m21\rangle$	6. $\langle p, m2\rangle \hookrightarrow \langle p, m4\rangle$
7. $\langle p, m3\rangle \hookrightarrow \langle p, m4\rangle$	8. $\langle p, m3\rangle \hookrightarrow \langle p, m5\rangle$
9. $\langle p, m3\rangle \hookrightarrow \langle p, m6\rangle$	10. $\langle p, m3\rangle \hookrightarrow \langle p, m7\rangle$
11. $\langle p, m3\rangle \hookrightarrow \langle p, m8\rangle$	12. $\langle p, m7\rangle \hookrightarrow \langle p, m10\rangle$
13. $\langle p, m9\rangle \hookrightarrow \langle p, m10\rangle$	14. $\langle p, m9\rangle \hookrightarrow \langle p, m11\rangle$
15. $\langle p, m9\rangle \hookrightarrow \langle p, m12\rangle$	16. $\langle p, m9\rangle \hookrightarrow \langle p, m13\rangle$
17. $\langle p, m9\rangle \hookrightarrow \langle p, m14\rangle$	18. $\langle p, m13\rangle \hookrightarrow \langle p, m17\rangle$
19. $\langle p, m14\rangle \hookrightarrow \langle p, m17\rangle$	20. $\langle p, m15\rangle \hookrightarrow \langle p, m16\rangle$
21. $\langle p, m15\rangle \hookrightarrow \langle p, m17\rangle$	22. $\langle p, m15\rangle \hookrightarrow \langle p, m18\rangle$
23. $\langle p, m15\rangle \hookrightarrow \langle p, m19\rangle$	24. $\langle p, m15\rangle \hookrightarrow \langle p, m20\rangle$
25. $\langle p, m19\rangle \hookrightarrow \langle p, m23\rangle$	26. $\langle p, m21\rangle \hookrightarrow \langle p, m22\rangle$
27. $\langle p, m21\rangle \hookrightarrow \langle p, m23\rangle$	
Control-dependence and flow-dependence edges in `p`	
28. $\langle p, p1\rangle \hookrightarrow \langle p, p2\rangle$	29. $\langle p, p1\rangle \hookrightarrow \langle p, p3\rangle$
30. $\langle p, p1\rangle \hookrightarrow \langle p, p4\rangle$	31. $\langle p, p1\rangle \hookrightarrow \langle p, p5\rangle$
32. $\langle p, p1\rangle \hookrightarrow \langle p, p6\rangle$	33. $\langle p, p1\rangle \hookrightarrow \langle p, p7\rangle$
34. $\langle p, p1\rangle \hookrightarrow \langle p, p8\rangle$	35. $\langle p, p1\rangle \hookrightarrow \langle p, p9\rangle$
36. $\langle p, p2\rangle \hookrightarrow \langle p, p4\rangle$	37. $\langle p, p3\rangle \hookrightarrow \langle p, p5\rangle$
38. $\langle p, p4\rangle \hookrightarrow \langle p, p9\rangle$	39. $\langle p, p5\rangle \hookrightarrow \langle p, p6\rangle$
40. $\langle p, p5\rangle \hookrightarrow \langle p, p8\rangle$	41. $\langle p, p6\rangle \hookrightarrow \langle p, p7\rangle$
Call edges	
42. $\langle p, m3\rangle \hookrightarrow \langle p, p1\ C1\rangle$	43. $\langle p, m9\rangle \hookrightarrow \langle p, p1\ C2\rangle$
44. $\langle p, m15\rangle \hookrightarrow \langle p, p1\ C3\rangle$	
Parameter-in edges	
45. $\langle p, m4\rangle \hookrightarrow \langle p, p2\ C1\rangle$	46. $\langle p, m5\rangle \hookrightarrow \langle p, p3\ C1\rangle$
47. $\langle p, m10\rangle \hookrightarrow \langle p, p2\ C2\rangle$	48. $\langle p, m11\rangle \hookrightarrow \langle p, p3\ C2\rangle$
49. $\langle p, m16\rangle \hookrightarrow \langle p, p2\ C3\rangle$	50. $\langle p, m17\rangle \hookrightarrow \langle p, p3\ C3\rangle$
Parameter-out edges	
51. $\langle p, p7\rangle \hookrightarrow \langle p_{p7}, \epsilon\rangle$	52. $\langle p_{p7}, C1\rangle \hookrightarrow \langle p, m6\rangle$
53. $\langle p_{p7}, C2\rangle \hookrightarrow \langle p, m12\rangle$	54. $\langle p_{p7}, C3\rangle \hookrightarrow \langle p, m18\rangle$
55. $\langle p, p8\rangle \hookrightarrow \langle p_{p8}, \epsilon\rangle$	56. $\langle p_{p8}, C1\rangle \hookrightarrow \langle p, m7\rangle$
57. $\langle p_{p8}, C2\rangle \hookrightarrow \langle p, m13\rangle$	58. $\langle p_{p8}, C3\rangle \hookrightarrow \langle p, m19\rangle$
59. $\langle p, p9\rangle \hookrightarrow \langle p_{p9}, \epsilon\rangle$	60. $\langle p_{p9}, C1\rangle \hookrightarrow \langle p, m8\rangle$
61. $\langle p_{p9}, C2\rangle \hookrightarrow \langle p, m14\rangle$	62. $\langle p_{p9}, C3\rangle \hookrightarrow \langle p, m20\rangle$

Table 1. The encoding of the SDG shown in Fig. 2 as a PDS using the schema given in Fig. 11.

- the next-to-last rule of Fig. 11, the pop rule $\langle p, fo\rangle \hookrightarrow \langle p_{fo}, \epsilon\rangle$, uses control location p_{fo} to record that formal-out vertex *fo* was popped from the stack, so that *fo* can be paired with the call-site symbol C exposed at the top of the stack.
- the last rule of Fig. 11, the internal rule $\langle p_{fo}, C\rangle \hookrightarrow \langle p, ao\rangle$, replaces C on the stack with the actual-out vertex that matches *fo* at call-site C.

In Tab. 1, rules 59 and 60 encode parameter-out edge $p9 \to m8$ of call-site $C1$, whereas rules 59 and 61 encode parameter-out edge $p9 \to m14$ of call-site $C2$, and rules 59 and 62 encode parameter-out edge $p9 \to m20$ of call-site $C3$.

Definition 3.4. *Given SDG G, the **unrolling** of G is the transition relation of the PDS that encodes G via the schema given in Fig. 11.* □

3.2 Symbolic Stack-Configuration Slicing

When an SDG G is encoded as a PDS $\mathcal{P}$, a pre^* operation on $\mathcal{P}$ is, by definition, equivalent to performing a closure slice on the unrolling of G—what we called stack-configuration slicing in §2.2. Moreover, the symbolic method [12, 24] for finding the answer to a pre^* query immediately provides an *algorithm* for stack-configuration slicing. Because each control location can be associated with an infinite number of stack-components, the symbolic method is based on using finite automata to describe regular sets of configurations [12, 24].

Definition 3.5. *If $\mathcal{P} = (P, \Gamma, \Delta)$ is a PDS, a $\mathcal{P}$-**automaton** is a finite automaton $(Q, \Gamma, \to, P, F)$, where $Q \supseteq P$ is a finite set of states, $\to \subseteq Q \times \Gamma \times Q$ is the transition relation, P is the set of initial states, and F is the set of final states.*

*The $\to$ relation is extended to a word $u \in \Gamma^*$ in the natural way, denoted by $\xrightarrow{u}{}^*$. (I.e., $\xrightarrow{u}{}^* \subseteq Q \times \Gamma^* \times Q$.) A $\mathcal{P}$-configuration (p, u) is **accepted** by a $\mathcal{P}$-automaton if the automaton can accept u when it is started in the state p (i.e., $p \xrightarrow{u}{}^* q$, where $q \in F$). A set of $\mathcal{P}$-configurations is **regular** if it is accepted by some $\mathcal{P}$-automaton.* □

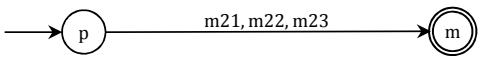

Figure 12. The query automaton, which accepts the configurations $(p, m21)$, $(p, m22)$, and $(p, m23)$.

An important result is that for a regular set of $\mathcal{P}$-configurations C, both $post^*(C)$ and $pre^*(C)$ (the forward and backward reachable sets of configurations, respectively) are also regular sets of $\mathcal{P}$-configurations [12, 24]. The algorithms for computing $post^*$ and pre^*, called *Poststar* and *Prestar*, respectively, take a $\mathcal{P}$-automaton $\mathcal{A}$ as input, and if C is the set of $\mathcal{P}$-configurations accepted by $\mathcal{A}$, they produce $\mathcal{P}$-automata $\mathcal{A}_{post^*}$ and $\mathcal{A}_{pre^*}$ that accept the sets of $\mathcal{P}$-configurations $post^*(C)$ and $pre^*(C)$, respectively. Both *Poststar* and *Prestar* can be implemented as *saturation procedures*; i.e., transitions are added to $\mathcal{A}$ according to an augmentation rule until no more can be added.

Definition 3.6 (Algorithm *Prestar*)**.** *$\mathcal{A}_{pre^*}$ is constructed by augmenting $\mathcal{A}$ according to the following rule, until $\mathcal{A}$ is saturated:*

$$\frac{\langle p, \gamma\rangle \hookrightarrow \langle p', w\rangle \in \Delta \qquad p' \xrightarrow{w}{}^* q \in \mathcal{A}}{p \xrightarrow{\gamma} q \in \mathcal{A}}\ \textsc{Pre} \tag{4}$$

Esparza et al. [22] present an efficient implementation of Prestar, which uses $O(|Q|^2|\Delta|)$ time and $O(|Q|\,|\Delta| + |\to_C|)$ space. □

Definition 3.7 (Algorithm *Poststar*)**.** *$\mathcal{A}_{post^*}$ is constructed from $\mathcal{A}$ by performing Phase I, and then saturating via the rules given in Phase II:*

- *Phase I. For each pair (p', γ') such that $\mathcal{P}$ contains at least one rule of the form $\langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\gamma''\rangle$, add a new state $p'_{\gamma'}$.*
- *Phase II (saturation phase). (The symbol $\xrightarrow{\gamma}$ denotes the relation $(\xrightarrow{\epsilon})^\star \xrightarrow{\gamma} (\xrightarrow{\epsilon})^\star$.)*

$$\frac{\langle p, \gamma\rangle \hookrightarrow \langle p', \epsilon\rangle \in \Delta \qquad p \xrightarrow{\gamma} q \in \mathcal{A}}{p' \xrightarrow{\epsilon} q \in \mathcal{A}}\ \textsc{Post1}$$

$$\frac{\langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\rangle \in \Delta \qquad p \xrightarrow{\gamma} q \in \mathcal{A}}{p' \xrightarrow{\gamma'} q \in \mathcal{A}}\ \textsc{Post2}$$

$$\frac{\langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\gamma''\rangle \in \Delta \qquad p \xrightarrow{\gamma} q \in \mathcal{A}}{p' \xrightarrow{\gamma'} p'_{\gamma'} \qquad p'_{\gamma'} \xrightarrow{\gamma''} q \in \mathcal{A}}\ \textsc{Post3}$$

$\mathcal{A}_{post^*}$ can be constructed in time and space $O(n_P n_\Delta (n_1 + n_2) + n_P n_0)$, where $n_P = |P|$, $n_\Delta = |\Delta|$, $n_Q = |Q|$, $n_0 = |\to_0|$, $n_1 = |Q - P|$, and n_2 is the number of different pairs (p', γ') such that there is a rule of the form $\langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\gamma''\rangle$ in Δ [49]. □

Example 3.8. This example illustrates how the *Prestar* algorithm from Defn. 3.6 performs the stack-configuration slice for the example discussed in §2.3. The SDG from Fig. 2 is encoded as the PDS whose rules are given in Tab. 1. The query automaton $\mathcal{A}$ that specifies the slicing criterion has transitions from initial state p to final state m on the symbols $m21$, $m22$, and $m23$—see Fig. 12.

Technically, a $\mathcal{P}$-automaton has an initial state for each control location of PDS $\mathcal{P}$ (Defn. 3.5). In our application, the control locations of the form p_{fo} are introduced solely for technical reasons. For stack-configuration slicing, we are only interested in

$\mathcal{P}$-configurations in which the control location is p, and hence the query automaton that we use has just one initial state, labeled p (see Fig. 12). Because our $\mathcal{P}$-automata have just the one initial state p, we shall henceforth ignore p in the accepted words, and concentrate on the (SDG) configuration that a $\mathcal{P}$-configuration models. For instance, technically $\mathcal{A}$ accepts the language of $\mathcal{P}$-configurations $\{(p, m21), (p, m22), (p, m23)\}$, which model the following configurations of the unrolled SDG: $\{(m21, \epsilon), (m22, \epsilon), (m23, \epsilon)\}$.

Prestar produces the automaton $\mathcal{A}'$ shown in Fig. 9(a). The fact that, e.g., configuration $(p5, C1)$ is accepted by $\mathcal{A}'$ means that $(p5, C1)$ is in the stack-configuration slice with respect to the set of configurations $\{(m21, \epsilon), (m22, \epsilon), (m23, \epsilon)\}$.

According to Eqn. (4), the transition $(p, m19, m)$ can be added to $\mathcal{A}$ using PDS rule 25; the transition $(p_{p8}, C3, m)$ can be added using PDS rule 58; and so on. Each new state p_{fo} added to $\mathcal{A}$ during *Prestar* corresponds to a formal-out vertex *fo* of some variant V that is in the slice. Transitions from the initial state p to p_{fo} are labeled by program elements in V in the backward slice from *fo*. $\mathcal{A}'$ can have transitions labeled with call-site symbols from p_{fo} to states other than the initial state.

The significance of last two rule-schemas in Fig. 11, $\langle p, fo\rangle \hookrightarrow \langle p_{fo}, \epsilon\rangle$ and $\langle p_{fo}, C\rangle \hookrightarrow \langle p, ao\rangle$, is that if (i) the SDG has a parameter-out edge from *fo* to actual-out vertex *ao* at call-site C, and (ii) $\mathcal{A}'$ has a transition (p, ao, γ), then $\mathcal{A}'$ will have the transitions (p, fo, p_{fo}) and (p_{fo}, C, γ). For instance, in our example, let *fo* be $p8$, *ao* be $m19$, and γ be m. Because (i) the SDG has a parameter-out edge $p8 \rightarrow m19$ at call-site $C3$, and (ii) Fig. 9(a) has a transition $(p, m19, m)$, Fig. 9(a) also has the transitions $(p, p8, p_{p8})$ and $(p_{p8}, C3, m)$. □

Let us now consider stack-configuration slicing for a program with a recursive procedure.

Example 3.9. Consider the example discussed in §2.4, where we want to create an executable slice with respect to line (28) of the program shown in Fig. 5. We first encode the program's SDG (Fig. 6) as a PDS similar to the previous example. We then construct a query automaton $\mathcal{A}_r$ that accepts the language of configurations $\{(m10, \epsilon), (m11, \epsilon), (m12, \epsilon)\}$. $\mathcal{A}_r$ is provided as an input to the *Prestar* algorithm. The automaton created by the *Prestar* algorithm is shown in Fig. 10.

The transitions $(p_{r22}, C3, p_{r23})$ and $(p_{r23}, C3, p_{r22})$ cover the recursive nature of the procedure call at call-site $C3$. Because of these transitions, the output automaton $\mathcal{A}'_r$ (Fig. 10) from *Prestar* accepts configurations for program element $r23$ that have the form of $(r23, (C3\ C3)^*\ C1)$. This language defines an infinite language of configurations in which the stack has an even number of $C3$ symbols, followed by a single $C1$ at the bottom. □

4. An Automaton-Based Algorithm for Finding Executable Slices via Procedure Specialization

In this section, we describe our technique for finding executable slices via procedure specialization. The technique involves the following five steps: (i) encode the program's SDG as a PDS, (ii) perform stack-configuration slicing by applying the *Prestar* algorithm, (iii) carry out several transformations of the result of *Prestar* to construct an automaton that is minimal reverse-deterministic (MRD), (iv) create the specialized SDG from the MRD automaton, and (v) pretty-print the specialized SDG as source-code text. This section concentrates on items (iii) and (iv).

The specialization conditions given in Eqn. (1) and Defn. 2.8 are not constructive; they provide a specification of the desired sets of SDG configurations and the desired SDG, but cannot be used directly as an algorithm. In this section, we present an algorithm that creates the specialized SDG. A proof of the algorithm's correctness and a discussion of time and space bounds can be found in Apps. A.1 and A.2, respectively.

Input: SDG S and slicing criterion C
Output: An SDG R for the specialized slice of S with respect to C

```
// Create A6, a minimal reverse-deterministic
// automaton for the stack-configuration slice
// of S with respect to C
```
1 $\mathcal{P}_S$ = the PDS for S, encoded according to Defn. 3.2
2 $A0$ = a $\mathcal{P}_S$-automaton that accepts C
3 $A1 = \mathit{Prestar}[\mathcal{P}_S](A0)$
4 $A2 = \text{reverse}(A1)$
5 $A3 = \text{determinize}(A2)$
6 $A4 = \text{minimize}(A3)$
7 $A5 = \text{reverse}(A4)$
8 $A6 = \text{removeEpilonTransitions}(A5)$

```
// Read out SDG R from A6
```
9 R = the empty SDG
10 $q_0 = \text{InitialState}(A6)$
11 *StateToPDGMap* = empty map
12 **foreach** $q \in (\mathit{States}(A6) - \{q_0\})$ **do** `// Identify PDGs`
13 $\quad V = \{v \mid (q_0, v, q) \in \text{Transitions}(A6)\}$
14 $\quad G_{orig}$ = the PDG of S that contains the vertices in V
15 $\quad G_V$ = a new PDG consisting of copies of V, plus copies of the edges from G_{orig} induced by V
16 $\quad$ Add PDG G_V to SDG R
17 $\quad \mathit{StateToPDGMap} = \mathit{StateToPDGMap}[q \mapsto G_V]$
18 **end**
19 **foreach** *transition* $(q_1, C, q_2) \in \mathit{Transitions}(A6)$ *such that* C *is a call-site label* **do** `// Connect PDGs`
20 $\quad$ Let C' be the call-site of PDG $\mathit{StateToPDGMap}(q_2)$ that corresponds to C
21 $\quad$ Add to R a call edge from the call vertex at C' to the entry vertex of $\mathit{StateToPDGMap}(q_1)$
22 $\quad$ Add to R a parameter-in edge from each actual-in vertex at C' to the corresponding formal-in vertex of $\mathit{StateToPDGMap}(q_1)$
23 $\quad$ Add to R a parameter-out edge from each formal-out vertex of $\mathit{StateToPDGMap}(q_1)$ to the corresponding actual-out vertex at C'
24 **end**
25 **return** R

Algorithm 1: The algorithm to create an SDG R for the specialized slice of S with respect to C.

The Algorithm to Construct the Specialized SDG. The SDG for the specialized slice is created by the method given in Alg. 1. Automaton $A1$ created in line 3 accepts the language of configurations of the stack-configuration slice. In lines 4–8, Alg. 1 applies automaton operations to $A1$—reverse, determinize, minimize, reverse, and removeEpilonTransitions—to obtain $A6$, from which the algorithm reads out the required specialized procedures and their program elements (lines 9–24).

Note that from the perspective of the *languages accepted* by $A1$ and $A6$, the five operations have *no net effect*: the operations determinize and minimize do not change the language that an automaton accepts, and thus the two calls to reverse in lines 4 and 7 cancel. That is, $L(A6) = L(A1)$, and so $A6$ accepts exactly the language of configurations of the stack-configuration slice. *The sole purpose of the five operations is to set $A6$ to the minimal reverse-deterministic FSA for the language.* The read-out method (lines 9–24) relies on the special nature of $A6$:

- Words in $L(A6)$ all have the form (vertex-symbol call-site*).
- $A6$ is MRD.
- Each non-initial state q represents a set of variants of some PDG G_{orig} (for some procedure P). Consequently, transitions

between non-initial states tell us about interprocedural edges in the answer SDG.

- Let V be the set of vertex symbols on transitions from the initial state to q. For each variant W associated with q, $\text{Elems}(W) = V$. Consequently, the transitions from the initial state to q tell us what vertices populate the specialized PDG for q.

Definition 4.1. *Given a graph $G = (V, E)$ and a vertex set $V' \subseteq V$, the set of **edges induced by** V' is the set of edges of the subgraph of G with vertices restricted to V':*

$$\{e \in E \mid e = s \to t \wedge s \in V' \wedge t \in V'\}.$$

□

Lines 9–24 read out SDG R from automaton $A6$. The basic idea is to find an appropriate set of vertices, and then include the edges induced by the vertex set. For instance, lines 12–18 construct the specialized PDGs in that manner: in line 13, the set V—a set of vertex symbols on transitions from the initial state to a given non-initial state—identifies the set of vertices in a specialized PDG; the edges in the specialized PDG are copies of the edges induced by V in the original PDG (line 15). Lines 12–18 create one PDG per (non-initial) state q. *StateToPDGMap* records, for each such q, which specialized PDG corresponds to q.

Lines 19–24 introduce call, parameter-in, and parameter-out edges to connect the specialized PDGs together. In essence, these steps put in induced interprocedural edges between the specialized PDGs for non-initial state q_1 and non-initial state q_2. Note that the alphabet symbol C on each transition (q_1, C, q_2) is a call-site label. Sequences of such transitions in $A6$ spell out, from top-of-stack to bottom-of-stack, the stack-configurations that are in the stack-configuration slice. Because each stack-configuration word is recognized from top-of-stack to bottom-of-stack, in line 19 q_2 represents the caller and q_1 represents the callee.

Example 4.2. Returning to the example from §2.3, consider how Alg. 1 creates the specialized SDG (Fig. 4) when S is the SDG shown in Fig. 2 and C equals $\{(m21, \epsilon), (m22, \epsilon), (m23, \epsilon)\}$. The rules of PDS $\mathcal{P}_S$ are given in Tab. 1. Automaton $A0$, which accepts the language C, is shown in Fig. 12. Automaton $A1 =$ *Prestar*$[\mathcal{P}_S](A0)$ is shown in Fig. 9(a). Automaton $A6$ is shown in Fig. 9(b); note that $A6$ is MRD.

In Fig. 9(b), each of the non-initial states $1, 2$, and 3 represents a specialized PDG. For instance, the vertices of the specialized PDG for procedure `p` that corresponds to state 1 are the labels on transitions from p to 1: $\{v \mid (p, v, 1) \in \text{Transitions}(A6)\} = \{p1, p3, p5, p8\}$. The edges of the specialized PDG are those induced by $\{p1, p3, p5, p8\}$ in the original PDG for procedure `p`.

$A6$ has a transition $(1, C1, 3)$, where states 1 and 3 correspond to procedures `p_1` and `main`, respectively. The stack symbol $C1$ corresponds to the call-site on `p` at line (14) of Fig. 1(a). Hence, lines 20–23 of Alg. 1 introduce a call edge, plus parameter-in and parameter-out edges to connect the specialized PDG for `main` to the specialized PDG for `p_1`. (In the pretty-printed program, the call at line (14) of Fig. 1(d) is a call on procedure `p_1`.)

The SDG formed from the resulting specialized PDGs is the one shown in Fig. 4. □

Example 4.3. Consider the example in §2.4, where we want to create an executable slice with respect to line (28) of the program shown in Fig. 5. Automaton $A6$ is shown in Fig. 10.[6] The PDG vertices in the five specialized PDGs are the respective label sets on the five transitions emanating from initial-state p. The edges of the five specialized PDGs are the edges induced from the original PDGs by the specialized sets of PDG vertices.

To complete the specialized SDG, Alg. 1 introduces call edges, parameter-in edges, and parameter-out edges among the specialized PDGs according to the transitions among states m, p_{r23}, p_{r22}, p_{s7}, and p_{s6}.

The program for the specialized SDG is shown in Fig. 5(b). Note how the transitions among states m, p_{r23}, p_{r22}, p_{s7}, and p_{s6} correspond to the call graph of Fig. 5(b). □

Correctness Issues. A proof of correctness of Alg. 1 is presented in App. A. The key properties established there are as follows:

- Automaton $A6$ created in line 8 of Alg. 1 is a minimal reverse-deterministic automaton (Thm. A.1).
- A solution to the configuration-partitioning problem (Defn. 2.8) is encoded in the structure of automaton $A6$ (Thm. A.2).
- Alg. 1 is sound and complete with respect to stack-configuration slicing (Thm. A.3).
- The output SDG R has no parameter mismatches (Cor. A.4).

5. Extensions

5.1 Calls to Library Procedures

Assuming that source code for library procedures is not available, and therefore those procedures cannot be specialized, we need to ensure that their signatures do not change: i.e., whenever a library procedure is included in a slice, all of its actual parameters must be included as well. We accomplish this by adding extra dependence edges to the SDG for library-procedure calls: for every vertex v that represents a library-procedure call, for every vertex a that represents an actual parameter associated with that call, we add an edge $a \to v$.

Example 5.1. In C, calls to `exit` cause program termination. This is modeled in the SDG by making all vertices that represent program elements that follow a call to `exit` control-dependent on that call. However, the value of `exit`'s argument does not affect its behavior, and so there are no dependence edges out of the vertex that represents the actual parameter. Consequently, the closure-slice back from a program element that follows a call to `exit` includes the call but not the actual. For the purposes of executable slicing, adding a dependence edge from the actual to the call solves the problem. □

5.2 Procedure Pointers and Indirect Calls

Procedure pointers and calls via those pointers also require special handling. Consider slicing the program shown in Fig. 13 with respect to `x` at line (16). The slice must include the call via procedure-pointer `p` on line (15), and the code in procedures `f` and `g` (the procedures that `p` may point to) that sets the return values of those procedures. For procedure `f`, that means the whole procedure (including both of its formals), while for procedure `g` that means just its first formal. However, the resulting code would not compile: the declared type of procedure-pointer `p` needs to match the types of both `f` and `g`, so those types must themselves match.

This is a new kind of parameter-mismatch problem that was not considered by Binkley. Like the original parameter-mismatch prob-

[6] For this example, Fig. 10 represents both $A1$ and $A6$. That is, the net result of applying the five automaton operations in lines 4–8 of Alg. 1 to Fig. 10 is that we get back an automaton identical to $A1$. The reason why $A6$ is the same as $A1$ is that, in this example, the stack-configuration slice does not have a procedure that has multiple variants that both (i) consist of different sets of PDG vertices, and (ii) include the same formal-out vertex.

In contrast, consider formal-out vertex $p9$ in Fig. 3. There are three occurrences of $p9$ in different variants: $(p9, C1)$, $(p9, C2)$, and $(p9, C3)$. The variants with stack-configurations $C1$ and $C3$ consist of the same set of PDG vertices, but the variant with stack-configuration $C2$ has a different set of PDG vertices. In this example, the output automaton $A1$ obtained from applying *Prestar* to the query automaton is the one shown in Fig. 9(a). The MRD automaton $A6$ obtained after performing lines 4–8 of Alg. 1 is the different automaton shown in Fig. 9(b).

```
(1)  int f(int a, int b) {
(2)    return a+b;
(3)  }
(4)
(5)  int g(int a, int b) {
(6)    return a;
(7)  }
(8)
(9)  int main() {
(10)   int (*p)(int, int);
(11)   int x;
(12)
(13)   if (...) p = f;
(14)   else p = g;
(15)   x = p(1,2);
(16)   printf("%d", x);
(17) }
(18)
```

Figure 13. Example illustrating some of the problems that arise in the presence of procedure pointers and indirect calls.

lem, this one can be solved either using a Binkley-style approach or using a specialization-style approach. The Binkley-style approach involves computing the closure slice, then finding all procedures in each procedure-pointer's points-to set, and adding back any mismatched formals (those in the closure slice for some but not all procedures in the points-to set).

The specialization-style approach involves adding a new procedure for each indirect call in the program, then applying specialization as usual. The new procedure makes explicit the fact that the choice of which procedure to call depends on which procedure the procedure-pointer currently points to. For example, the new procedure added to the program in Fig. 13 is shown below.

```
int indirect(int (*p)(int, int), int a, int b) {
  if (p == f) f(a, b);
  else g(a, b);
}
```

The indirect call in the original program (`x = p(1, 2)` in our example) is changed to call the new procedure, passing the procedure pointer in addition to the original actuals: `x = indirect(p, 1, 2)`

Once this is done, our executable-slicing algorithm will create the appropriate specialized versions of all of the procedures in the slice from line (15), including a specialized version of procedure `indirect` named `indirect_1`. The if-condition in `indirect_1` still tests whether `p` points to (the original) procedures `f` or `g`, but the calls themselves are changed to target the specialized versions of those procedures. Here is the executable slice from line (15):

```
int f(int a, int b) {
}

int f_1(int a, int b) {
  return a+b;
}

int g(int a, int b) {
}

int g_1(int a) {
  return a;
}

int indirect_1(int (*p)(int, int), int a, int b) {
  if (p == f) f_1(a, b);
  else g_1(a);
}

int main() {
  int (*p)(int, int);
  int x;

  if (...) p = f;
  else p = g;
  x = indirect_1(p,1,2);
  printf("%d", x);
}
```

5.3 Reslicing Check

Ordinary slicing algorithms are *idempotent*: let G/C denote the slice of PDG G with respect to slicing criterion C; then $(G/C)/C = G/C$. Thus, as an additional test to validate that the result computed by our implementation is correct, the implementation performs an additional "reslicing" check.

Suppose that for SDG S and slicing criterion C, the resulting SDG is R. Roughly, we slice R with respect to C to obtain R', and compare R' to R. The expectation is that they should be identical: if they fail to be identical, the implementation has a bug.

As in the initial discussion of soundness and completeness in §2.2, because the PDG vertices and call-site labels in S and R are different, the paragraph above is slightly inaccurate. Such naming differences create a problem for reslicing because the PDG vertices and call-site labels are alphabet symbols in the automata used by the algorithm described in §4. Consequently, the reslicing check must compensate in two places for such changes in the alphabet symbols.

1. The slice of R must be taken with respect to a suitably adjusted slicing criterion C', rather than with respect to C itself.
2. The comparison of R' to R is not a pure equality test; the comparison must account for the changes in the alphabet.

As in §2.2, it is possible to identify each vertex or call-site in R as the specialization of some vertex or call-site in S, and this information can be used to define a mapping that maps vertices and call-sites in R back to the original alphabet of S. In particular, to account for the changes in the alphabet from S to R, we create a transducer T_C, which maps a vertex or label of an R call-site to the corresponding vertex or label of an S call-site.

To implement the reslicing check, in addition to T_C, six other data objects come into play.

- two SDGs, S and R
- two automata for slicing criteria, C and C', and
- two automata that hold slice results, $A6_S$ and $A6_R$—i.e., the automata *A6* of Alg. 1 arising in the slices of S and R, respectively.

To reslice R, we need to map slicing criterion C to an appropriate *reslicing criterion* C' (item 1 above). Transducer T_C can be combined with automaton C to create an automaton for the inverse transduction $T_C^{-1}(C)$. However, because transduction T_C is many-to-one, the inverse transduction T_C^{-1} is, in general, one-to-many. In such cases, the language of $T_C^{-1}(C)$ contains strings that do not correspond to any configuration of R. Fortunately, we can rectify this problem by intersecting $T_C^{-1}(C)$ with an automaton for the language of all possible configurations of R. Let $\mathcal{P}_R$ be the PDS for R, encoded according to Defn. 3.2; the language of configurations of R can be obtained by the PDS query *Poststar*$[\mathcal{P}_R](\text{entry}_{\text{main}})$ [12, 24].

Program	# Versions	Avg. # source lines	# Procedures	Avg. # PDG vertices	# Call sites	# Slices taken	Avg. % increase in PDG vertices over closure slice			
							Spec. slices	Standard deviation	Binkley slices	Standard deviation
tcas	37	564	9	480	38	37	0.2	0.0	5.3	0.1
schedule2	2	717	16	980	47	6	28.4	0.1	6.3	0.0
schedule	6	725	18	873	44	11	5.4	1.7	4.8	0.7
print_tokens	4	889	18	1298	89	3	0.4	0.0	4.9	0.0
replace	26	931	21	1331	65	58	5.6	5.4	7.1	0.7
print_tokens2	8	957	19	1127	84	28	1.0	2.7	5.5	0.4
tot_info	19	1414	7	676	37	23	0.0	0.0	2.8	0.1
wc (v. 8.13)	1	802	11	1899	170	10	26.0	11.2	10.5	5.6
space	20	7429	136	18822	1016	35	4.4	2.2	4.4	1.4
go	1	29246	372	102455	2084	10	7.8	3.9	6.3	11.1

Figure 14. Information about the test programs, and comparison of the sizes of closure slices versus the sizes of executable slices created via specialized slicing and via Binkley's approach.

To recap, to handle item 1 above, the automaton C' for the reslicing criterion is created by performing the following PDS/-transducer/automaton computation:

$$C' = T_C^{-1}(C) \cap \mathit{Poststar}[\mathcal{P}_R](\text{entry}_{\text{main}}).$$

To handle item 2, it is convenient to compare the automata $A6_S$ and $A6_R$, rather than to compare the SDGs R and R'. In particular, we perform the language-equality check

$$L(A6_S) \stackrel{?}{=} L(T_C(A6_R)). \tag{5}$$

Note that $A6_S$ represents the vertices (configurations) of the closure slice of the (possibly infinite) unrolling of SDG S. Similarly, $A6_R$ represents the vertices of the closure slice of the unrolling of R. Consequently, Eqn. (5) tests whether the two closure slices have identical vertices after T_C is used to map each R configuration back to a configuration in the alphabet of S. The comparison performed in Eqn. (5) is an implicit test of SDGs R and R' because R and R' are constructed merely by reading out information contained in the automata $A6_S$ and $A6_R$, respectively.

6. Limitations

Hoisting of Indirect-Call PDGs. The translation of indirect function calls to explicit PDGs assumes that the points-to set of the function pointer is exhaustive. However, the pointer-analysis algorithms supported by CodeSurfer/C are a variant of Andersen's analysis [2], which does not account for uninitialized pointer variables. For example, suppose that there are three paths to an indirect-call site through `p`, and that `p` is initialized to `f` on one path, `g` on another path, and not initialized on the third path. The indirect-call procedure will dispatch just to `f` and `g`. Assuming that `f` and `g` are two-parameter functions, the indirect-call PDG will still have the following form:

```
int indirect(int (*p)(int, int), int a, int b) {
  if (p == f) f(a, b);
  else g(a, b);
}
```

7. Experiments

Our implementation of executable slicing was built using GrammaTech's CodeSurfer® code-understanding tool [3], which supports slicing of C and C++ programs. (We also have an experimental implementation using CodeSurfer/x86, which supports slicing of Intel x86 machine code [4].)

CodeSurfer is used to build the input SDG, which is then translated into a PDS implemented using the Weighted Automaton Library (WALi) [40]. WALi is used to perform the *Prestar* operation, and the resulting automaton is converted into an OpenFST "recognizer" (FSA) [1]. OpenFST is used for the reverse, determinize, minimize, reverse, and removeEpilonTransitions sequence to create an MRD automaton. The output SDG R is created from the MRD automaton, and the source text for the specialized program is pretty-printed from R.

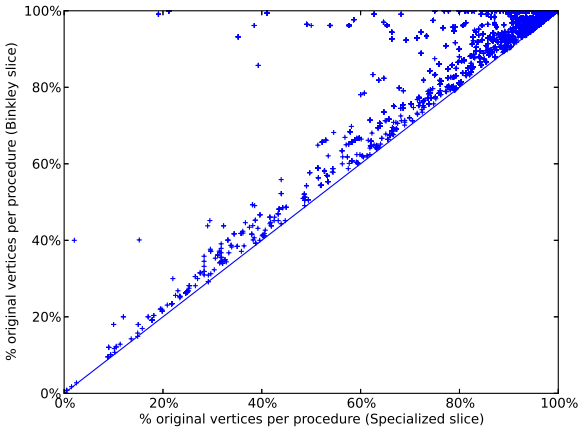

Figure 15. Comparison of the sizes of the procedures produced via Binkley's approach with those produced via specialized slicing.

The experiments we conducted were designed to

- compare our approach to finding executable slices with Binkley's approach
- determine whether the worst-case exponential cost of our algorithm arises in practice
- determine how the execution time of an executable slice—which, in principle, can be arbitrarily better than that of the original program—compares in practice with the execution time of the original program.

Information about the test programs is given in columns 1–7 of Fig. 14. The first seven programs are the Siemens suite: a set of small, toy programs collected by Hutchins et al. [37]. The other three are open-source applications gathered from the Internet. For the Siemens suite and `space`, the experiments use slices taken from (PDG-vertex, pending-call-stack) pairs at which symptoms of bugs (unexpected output or crashes) were observed. Those configurations were obtained using the debugging tools described in [36]. For `wc` and `go`, slices were taken from all calls to `printf`.

Figs. 14 and 15 provide two different ways to compare specialized slicing with Binkley's approach; Additionally, Figs. 14, 16, and 17 provide evidence that specialized slicing does *not* suffer from its potential worst-case blow-up in practice. Fig. 18 shows that using executable slicing to extract specialized components of an application can significantly improve runtimes.

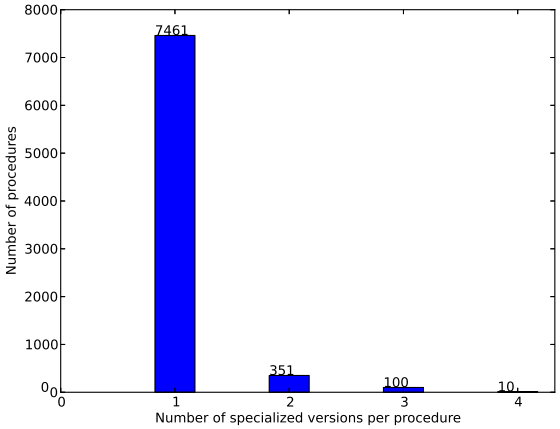

Figure 16. Distribution of the number of specialized versions of procedures in executable slices created via specialized slicing.

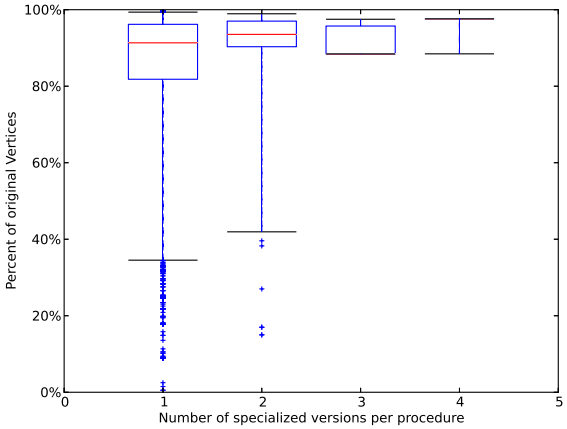

Figure 17. The percentage of PDG vertices in PDGs of the original SDG that are retained in their respective specialized PDGs. The boxes indicate the 25^{th} percentile, the median, and the 75^{th} percentile. The whiskers indicate the 2^{nd} and 98^{th} percentile. The additional points in the first and third columns indicate outliers beyond the whiskers.

Neither (i) the exponential blow-up that occurs for the family of examples discussed in App. A.2.3, nor (ii) the worst-case exponential behavior of operations like automaton determinization, arose in the programs and slices used in our experiments. As discussed in Apps. A.2.2 and A.2.3, we believe it is fair to say that, for the *observed cost*, both the running time and space of the algorithm are bounded by the sum of two terms: one is polynomial in the size of the input program; the other is linear in the size of the output slice.

Columns 8–11 of Fig. 14 provide data about how many "extra" vertices are in executable slices compared with the corresponding closure slices. Based on these results, it appears that in practice, blow-up in slice size is a problem neither for Binkley's approach nor for specialized slicing. While specialized slicing had the largest average increases in size (26.0% for `wc` and 28.4% for `schedule2`), on the three largest programs, size increases were very modest, and were similar for Binkley slicing and specialized slicing.

Fig. 15 is a scatter plot that compares the sizes of the PDGs produced via Binkley's approach with those produced via specialized slicing. PDG sizes are reported as the percentage of a PDG's vertices in the original program that occur in a PDG of an executable slice. Each point in the plot represents one PDG in one specialized slice; i.e., for each PDG `p_k` in each specialized slice, there is one point in the plot at position (x, y), where x is the percentage of vertices of the original PDG that are in `p_k` and y is the percentage that are in PDG `p` in the Binkley slice. If there are three specialized versions of PDG `p`, there will be three points in the plot, all with the same y coordinate.

Fig. 15 shows that many specialized-slice PDGs are close in size to the original PDGs (the points clustered in the upper right-hand corner); that Binkley-slice PDGs are often close in size to the corresponding specialized-slice PDGs (the points clustered along the 45-degree line); but that there are also many cases where a Binkley-slice PDG is much larger than the corresponding specialized-slice PDG (the points along the top). For each point, we computed $(\%\text{vertices in specialized PDG})/(\%\text{vertices in Binkley PDG})$; the geometric mean of these values is 93%.

Fig. 16 shows the distribution of the number of specialized versions of procedures in all tested programs, providing further evidence that the potential exponential explosion of specialized procedures (described in App. A.2.3) is unlikely to occur in practice. Approximately 93% of procedures have a single specialized version, and the count decreases rapidly as the number of specialized procedures increases. The largest number of specialized versions for a procedure we have seen in our experiments is four.

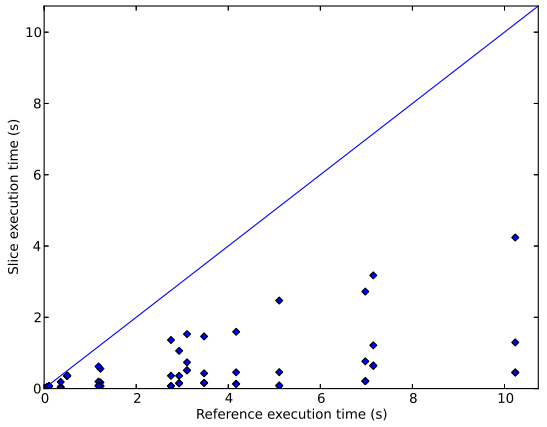

Figure 18. Average execution time of the original, unsliced `wc` application versus execution time of slice-derived executables on the same inputs.

Fig. 17 provides information about the distribution of the sizes of specialzed PDGs, broken down according to whether there were 1, 2, 3, or 4 specialized versions of a PDG. In each of the four cases, Fig. 17 shows five different percentiles for the percentage of PDG vertices from the PDG of the original SDG that are retained in the respective specialized PDGs.

To investigate the potential for speeding up runtimes via application specialization, we used executable slicing to specialize `wc`, the Linux word-count tool. We created four specialized slices and four Binkley slices: one from each of the four calls to `printf` in `wc.c` that print the number of lines, characters, bytes, and words in the input. We ran the original program and each slice on 16 inputs, recording total CPU time using the bash `time` utility. Fig. 18 plots the original-vs.-specialized-slice runtimes; each point is the average of 10 runs. (The performance improvements for specialized slicing and Binkley-slicing were almost identical: each point in Fig. 18 and the corresponding point for the Binkley-slice essentially coincide.) On average—computed as the geometric mean—

the executable slices took 32.5% of the time used by the original program.

8. Related Work

The literature on program slicing is extensive. (Literature surveys include [8, 9, 20, 33, 45, 51, 52].) Some of the related work on executable slicing has already been summarized in §1.

Binkley et al. [10] give declarative semantic specifications for a number of different varieties of slices. The work unifies and relates eight different kinds of slicing criteria for static and dynamic slicing. Our work is mainly algorithmic in nature, but our use of unrolled SDGs and stack-configurations provides an abstraction of the runtime stack. While our experiments used C programs, the principles on which our algorithm is based should apply to interprocedural slicing for any language.

Conditioned slicing [16, 19, 25] combines static slicing and program simplification to produce executable program slices. The simplification phase propagates information forward to remove statements that cannot be executed when a given constraint holds on the initial state. At least some of this work [19, 25] merely applies off-the-shelf slicing algorithms in the static-slicing component, and hence could adopt the algorithm presented in this paper.

In our implementation, an SDG is encoded as a PDS using the Weighted Automaton Library (WALi) [40], and WALi is used to perform *Prestar*. The algorithm for computing pre^* with respect to a regular set of configurations in a PDS is due to Bouajjani et al. [12] and Finkel et al. [24]. The history of the model-checking problem for PDSs is recounted by Bouajjani et al. [13, §6], who credit Büchi with establishing the foundational result ([14] and [15, Ch. 5]), and point out that it was rediscovered several times [11, 17].

Minimal reverse-deterministic FSAs were used by Gupta [28] as a symbolic representation for a class of inductive Boolean functions. The *state complexity* of a regular language L is the number of states in the minimal deterministic FSA for L. Sebej [50] studied the change in state complexity between a regular language L and its reversal L^R. He showed that if L has state complexity n, the state complexity of L^R is between $\log n$ and 2^n.

Acknowledgments

We are grateful for the help with CodeSurfer/C, and timely improvements to it, that was provided by Thomas Johnson, Jason Dickens, and Chi-Hua Chen of GrammaTech, Inc. We thank Ben Liblit, Evan Driscoll, Prathmesh Prabhu, Tushar Sharma, and Junghee Lim for their comments on a draft of this paper.

The work was supported, in part, by private funding on the part of GrammaTech, Inc., by NSF under grants CCF-{0701957, 0810053, 0904371}, by ONR under grants N00014-{09-1-0510, 10-M-0251}, by ARL under grant W911NF-09-1-0413, by AFRL under grants FA9550-09-1-0279 and FA8650-10-C-7088, and by DARPA under cooperative agreement HR0011-12-2-0012. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the sponsoring agencies.

T. Reps has an ownership interest in GrammaTech, Inc., which has licensed elements of the technology reported in this publication.

References

[1] C. Allauzen, M. Riley, J. Schalkwyk, W. Skut, and M. Mohri. OpenFST: A general and efficient weighted finite-state transducer library. In *Int. Conf. on Implementation and Application of Automata*, 2007.

[2] L. O. Andersen. Binding-time analysis and the taming of C pointers. In *Part. Eval. and Semantics-Based Prog. Manip.*, pages 47–58, 1993.

[3] P. Anderson, T. Reps, and T. Teitelbaum. Design and implementation of a fine-grained software inspection tool. *TSE*, 29(8), 2003.

[4] G. Balakrishnan and T. Reps. WYSINWYX: What You See Is Not What You eXecute. *TOPLAS*, 32(6), 2010.

[5] S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. In *POPL*, pages 384–396, 1993.

[6] D. Binkley. Using semantic differencing to reduce the cost of regression testing. In *ICSM*, pages 41–50, 1992.

[7] D. Binkley. Precise executable interprocedural slices. *LOPLAS*, 2: 31–45, 1993.

[8] D. Binkley and K. Gallagher. Program slicing. In *Advances in Computers, Vol. 43*. Academic Press, 1996.

[9] D. Binkley and M. Harman. A survey of empirical results on program slicing. In *Advances in Computers, Vol. 62*. Academic Press, 2004.

[10] D. Binkley, S. Danicic, T. Gyimóthy, M. Harman, A. Kiss, and L. Ouarbya. Formalizing executable dynamic and forward slicing. In *SCAM*, 2004.

[11] R. Book and F. Otto. *String-Rewriting Systems*. Springer-Verlag, 1993.

[12] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model checking. In *CONCUR*, 1997.

[13] A. Bouajjani, J. Esparza, A. Finkel, O. Maler, P. Rossmanith, B. Willems, and P. Wolper. An efficient automata approach to some problems on context-free grammars. *IPL*, 74(5–6):221–227, 2000.

[14] J. Büchi. Regular canonical systems and finite automata. *Arch. Math. Logik Grundlagenforschung*, 6:91–111, 1964.

[15] J. Büchi. *Finite Automata, their Algebras and Grammars*. Springer-Verlag, 1988. D. Siefkes (ed.).

[16] G. Canfora, A. Cimitile, A. De Lucia, and G. Di Lucca. Software salvaging based on conditions. In *ICSM*, 1994.

[17] D. Caucal. On the regular structure of prefix rewriting. *Theor. Comp. Sci.*, 106(1):61–86, 1992.

[18] K. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. In *PLDI*, pages 57–66, 1988.

[19] S. Danicic, M. Daoudi, C. Fox, M. Harman, R. Hierons, J. Howroyd, L. Ouarbya, and M. Ward. ConSUS: A light-weight program conditioner. *J. Syst. and Software*, 77(3), 2005.

[20] A. De Lucia. Program slicing: Methods and applications. In *SCAM*, 2001.

[21] A. De Lucia, A. Fasolino, and M. Munro. Understanding function behaviors through program slicing. In *WPC*, 1996.

[22] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *CAV*, July 2000.

[23] J. Field, G. Ramalingam, and F. Tip. Parametric program slicing. In *POPL*, 1995.

[24] A. Finkel, B.Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. *ENTCS*, 9, 1997.

[25] C. Fox, S. Danicic, M. Harman, and R. Hierons. ConSIT: A fully automated conditioned program slicer. *Software: Practice and Experience*, 34(1), 2004.

[26] K. Gallagher and J. Lyle. Using program slicing in software maintenance. *TSE*, 17(8):751–761, Aug. 1991.

[27] R. Giacobazzi and I. Mastroeni. Non-standard semantics for program slicing. *HOSC*, 16(4):297–339, 2003.

[28] A. Gupta. *Inductive Boolean Function Manipulation: A Hardware Verification Methodology for Automatic Induction*. PhD thesis, Carnegie Mellon Univ., 1994. Tech. Rep. CMU-CS-94-208.

[29] M. Harman and S. Danicic. Amorphous program slicing. In *WPC*, 1997.

[30] J. Hopcroft. An $n \log n$ algorithm for minimizing the states in a finite automaton. In *The Theory of Machines and Computations*. Acad. Press, Inc., 1971.

[31] S. Horwitz. Identifying the semantic and textual differences between two versions of a program. In *PLDI*, pages 234–245, 1990.

[32] S. Horwitz and T. Reps. Efficient comparison of program slices. *Acta Inf.*, 28(8):713–732, 1991.

[33] S. Horwitz and T. Reps. The use of program dependence graphs in software engineering. In *ICSE*, pages 392–411, May 1992.

[34] S. Horwitz, J. Prins, and T. Reps. Integrating non-interfering versions of programs. *TOPLAS*, 11(3):345–387, July 1989.

[35] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *TOPLAS*, 12(1):26–60, Jan. 1990.

[36] S. Horwitz, B. Liblit, and M. Polishchuck. Better debugging via output tracing and callstack-sensitive slicing. *TSE*, 36(1), Jan. 2010.

[37] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Int. Conf. on Software Eng.*, pages 191–200, May 1994.

[38] D. Jackson and E. Rollins. A new model of program dependences for reverse engineering. In *FSE*, 1994.

[39] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International, 1993.

[40] N. Kidd, A. Lal, and T. Reps. WALi: The Weighted Automaton Library, 2007. www.cs.wisc.edu/wpis/wpds/download.php.

[41] D. Kuck, R. Kuhn, B. Leasure, D. Padua, and M. Wolfe. Dependence graphs and compiler optimizations. In *POPL*, pages 207–218, 1981.

[42] A. Lakhotia and J. Deprez. Restructuring programs by tucking statements into functions. *Inf. and Softw. Tech.*, 40(11–12):677–690, 1998.

[43] J. Lyle and M. Weiser. Experiments on slicing-based debugging tools. In *Conf. on Empirical Studies of Programming*, June 1986.

[44] J. Lyle and M. Weiser. Automatic program bug location by program slicing. In *Int. Conf. on Comp. and Applications*, 1987.

[45] G. Mund and R. Mall. Program slicing. In Y. Srikant and P. Shankar, editors, *The Compiler Design Handbook: Optimizations and Machine Code Generation*, chapter 14. CRC Press, 2nd. edition, 2007.

[46] K. Ottenstein and L. Ottenstein. The program dependence graph in a software development environment. In *PSDE*, 1984.

[47] T. Reps and G. Rosay. Precise interprocedural chopping. In *FSE*, 1995.

[48] T. Reps and T. Turnidge. Program specialization via program slicing. In *Proc. of the Dagstuhl Seminar on Partial Evaluation*, 1996.

[49] S. Schwoon. *Model-Checking Pushdown Systems*. PhD thesis, Technical Univ. of Munich, Munich, Germany, July 2002.

[50] J. Sebej. Reversal of regular languages and state complexity. In *Conf. on Theory and Practice of Inf. Technologies*, 2010.

[51] J. Silva. A vocabulary of program slicing-based techniques. *ACM Computing Surveys*, 2012. Available at users.dsic.upv.es/$\sim$ jsilva/papers/Vocabulary.pdf.

[52] F. Tip. A survey of program slicing techniques. *J. Prog. Lang.*, 3(3), 1995.

[53] G. Venkatesh. The semantic approach to program slicing. In *PLDI*, 1991.

[54] M. Weiser. Program slicing. *TSE*, 10(4):352–357, July 1984.

A. Correctness and Cost Issues

A.1 Correctness of Alg. 1

Theorem A.1. *Automaton $A6$ created in line 8 of Alg. 1 is MRD.* □

Proof. Because the operations determinize and minimize do not change the language that an automaton accepts, the two calls on reverse in lines 4 and 7 cancel, and hence $L(A6) = L(A1)$.

Automaton $A4$ created in line 6 of Alg. 1 is the minimal deterministic automaton for the reversed language of configurations in the stack-configuration slice. That is, $A4$ is minimal deterministic and $L(A4) = L^R(A1) = L^R(A6)$. Consequently, we just have to argue that the call on reverse in line 7, followed by removeEpilonTransitions in line 8 causes $A6$ to be MRD.

Words in $L(A1)$ ($L(A6)$) all have the form (vertex-symbol call-site*); moreover, the call-site symbols are disjoint from the vertex symbols. Consequently, there cannot be any loops that allow repetitions of the vertex symbols. Thus, at the beginning (and end, for the reversed languages), there are no loops or self-loops. What this means is that the minimized automaton ($A4$) must have a single accepting state. Moreover, because $A4$ is deterministic, it has no ϵ-transitions.

In any call on reverse(A), the only condition that necessitates the introduction of an ϵ-transition is if A has multiple accepting states (because one needs to have a single start state in $A' =$ reverse(A)). Consequently, the statement "$A5 =$ reverse($A4$)" can be implemented by making $A5$'s initial state be the (unique) final state of $A4$, and $A5$'s final state be the initial state of $A4$. Because $A4$ has no ϵ-transitions, $A5$ has no ϵ-transitions, and thus removeEpilonTransitions has no effect (i.e., $A6 = A5$). Because $A4$ is minimal deterministic, $A5$ and $A6$ are MRD.

In our implementation, lines 4–8 are implemented with OpenFST [1] FSAs. The reverse operation in OpenFST introduces a dummy initial state with an ϵ-transition. Thus, the implementation does call removeEpilonTransitions, which removes the single, initial ϵ-transition from $A5$ to create $A6$, which is MRD. □

Theorem A.2. *Automaton $A6$ created in line 8 of Alg. 1 solves the configuration-partitioning problem (Defn. 2.8).* □

Proof. $A6$ is the unique MRD automaton for the language $L(A1)$ = *Prestar*$[\mathcal{P}_S](C)$—i.e., the stack-configuration slice of S with respect to C.

Instead of $A6$, consider how a word is accepted by $A4$ (which is nearly identical to $A6$ except for the direction of transitions). Each $A4$ word has the form (call-site* vertex-symbol). Because the call-site symbols are disjoint from the vertex symbols, and because $A4$ is a minimal deterministic machine, two properties must hold: (i) $A4$ must have a unique final state *acc*, and (ii) it can have no loops that involve the final state.

Processing starts in $A4$'s initial state ($A6$'s final state), and follows transitions of the form (q_i, C, q_j), where C is a call-site label. Because $A4$ is deterministic, the call-site* prefix of the word follows a unique path—in effect, transitioning from one calling-context partition-element to another at each step. Finally, there is a transition of the form (q_k, v, acc) to $A4$'s unique final state *acc*.

State q_k represents the set of variants (of some procedure Q) associated with the calling-contexts given by the languages $P_{A4}(q_k)$. The program-elements in each of the variants is the set of PDG vertices $V_{q_k} \stackrel{\text{def}}{=} \{v \mid (q_k, v, \mathit{acc}) \in \text{Transitions}(A4)\}$. The set of configurations in the partition-element associated with q_k is, therefore,

$$\text{PE}_{q_k} \stackrel{\text{def}}{=} V_{q_k} \times (P_{A4}(q_k))^R.$$

The partition is defined by Part $\stackrel{\text{def}}{=} \{\text{PE}_{q_k} \mid q_k \in \text{States}(A4)\}$. (Note that Part is truly a partition because $\bigcup_{q_k} \text{PE}_{q_k} = L^R(A4) = L(A1) = L(A6)$.)

The three conditions of Defn. 2.8 follow from the observations that (i) $A4$ is a minimal deterministic FSA, and hence a set V_{q_k} cannot be associated with any $(P_{A4}(q_m))^R$, for $k \neq m$; (ii) PE_{q_k} is defined as a cross-product; and (iii) in an unrolled SDG, different procedure instances always have different stack-configurations. □

Theorem A.3. *Alg. 1 is sound and complete with respect to stack-configuration slicing.* □

Proof. Let M_C be the mapping that maps PDG vertices and call-sites in R back to the original alphabet of S. Let G_R be the unrolling of R. The proof divides into two cases.

Completeness: $L(A6)$ consists of all elements in the closure slice with respect to C of the unrolling of S. Let (v, w) be an arbitrary configuration in $L(A6)$. Completeness holds because of the steps of the read-out process in lines 9–24:

- Lines 12–18 create, for the procedure variant that has stack-configuration w, an SDG in R that has a copy v' of v. Hence, $M_C(v') = v$.
- Lines 19–24 preserve the language of stack-configurations of $L(A6)$ in the calling structure of R, which controls the stack-configurations of G_R. Consequently, G_R has some configuration (v', w') such that $M_C(w') = w$.

Thus, there is a configuration (v', w') in G_R such that $M_C((v', w')) = (v, w)$.

Soundness: Let (v', w') be an arbitrary configuration in G_R. Soundness holds because of the steps of the read-out process in lines 9–24:

- The PDG in R that has vertex v' was created in lines 12–18 from some transition (q_0, v, q) in $A6$. Hence, $M_C(v') = v$.
- Moreover, because w' is a stack-configuration of G_R, and lines 19–24 preserve the language of stack-configurations of $L(A6)$ in the calling structure of R, w' must correspond (via M_C) to some word in $L(q)$. Consequently, $L(A6)$ has some configuration (v, w) such that $M_C(w') = w$.

Thus, there is a configuration (v, w) in $L(A6)$ such that $M_C((v', w')) = (v, w)$. □

Corollary A.4. *Let R be the SDG created via Alg. 1 for SDG S and slicing criterion C. R has no parameter mismatches.* □

Proof. Let M_C be the mapping that maps PDG vertices and call-sites in R back to the original alphabet of S. Let G_S be the unrolling of S and G_R be the unrolling of R. Because each procedure instance is called at a unique call-site in an unrolled graph, the closure slice in G_S has no parameter mismatches. By Thm. A.3, $M_C(G_R)$ is isomorphic to the closure slice of S with respect to C, and hence has no parameter mismatches. M_C is purely a name-change operation, and thus does not change the calling structure of G_R. The way the SDG is read out of automaton $A6$ preserves the calling structure of G_R in R, and thus R has no parameter mismatches. □

A.2 Cost of Alg. 1

A.2.1 Minimality

The configuration-partitioning problem (Defn. 2.8) provides one notion of minimality for specialized slicing. By Thm. A.2, a solution to the configuration-partitioning problem is encoded in the structure of automaton $A6$ from Alg. 1. Moreover, from lines 12–18 of Alg. 1, it is clear that the number of vertices in the answer

SDG R is proportional to the size of $A6$. In this sense, SDG R is a minimum-sized slice.

A.2.2 Running Time

Alg. 1 has four operations that can be expensive:

1. $A1 = \mathit{Prestar}[\mathcal{P}_S](A0)$ (line 3),
2. $A3 = \text{determinize}(A2)$ (line 5),
3. $A4 = \text{minimize}(A3)$ (line 6), and
4. reading out SDG R from automaton $A6$ (lines 9–24).

- As mentioned in Defn. 3.6, *Prestar*'s worst-case running time is $O(|Q|^2|\Delta|)$. Here $|Q|$ is $1 +$ #actual-out vertices, and Δ is the number of dependence edges in input SDG S. Consequently, the running time of item 1 is bounded by a polynomial in the size of the input program.
- Determinization is performed by the subset construction. In the worst case, item 2 can be exponential in the number of states of $A2$.
- Minimization can be performed in time $O(n \log n)$ [30]. In the worst case, item 3 can be exponential in the number of states of $A2$.
- The number of vertices in the answer SDG R is proportional to the size of $A6$. In addition, the read-out code adds copies of dependence edges from the original PDGs. Such work in any PDG is bounded by the square of the number of vertices, but is usually much lower. In any case, the time for item 4 is linear in the size of output SDG R.

Our experiments indicate that for the automata that arise from *Prestar*, the result of determinize is significantly *smaller* (4.4%–34%) than the input to determinize.

A.2.3 Space

The space needs of Alg. 1 are dominated by the same four operations discussed under "Running Time".

- As mentioned in Defn. 3.6, the space for *Prestar* is bounded by $O(|Q|\,|\Delta| + |{\rightarrow_C}|)$, where $|{\rightarrow_C}|$ is the size of the transition relation for the automaton for slicing criterion C.
- Determinize and minimize are standard automaton operations, with space cost similar to their time cost.
- The space for the read-out task is linear in the size of the result SDG R.

Number of Specialized PDGs. The number of specialized PDGs in R depends on both the query automaton $\mathcal{A}_C$, which specifies slicing criterion C, and the input SDG S. The specialized PDGs that appear in R can be placed in two categories:

1. PDGs associated with calling contexts in the suffix-closure of the stack-configurations defined by $\mathcal{A}_C$[7]
2. PDGs associated with calling contexts other than those in item 1.

- The number of specialized PDGs in R from item 1 is bounded by the number of states in the query automaton.
- The number of specialized PDGs in R from item 2 is bounded by a function of the number of actual-outs in each PDG of SDG S. In particular, for a PDG p that has n_p actual-outs, the maximum number of specialized PDGs possible for p is 2^{n_p}. Consequently, the number of specialized PDGs in R from item 2 is bounded by $\Sigma_{p \in \text{PDGs}(S)} 2^{n_p}$.

Achieving Exponential Explosion. The bound $\Sigma_{p \in \text{PDGs}(S)} 2^{n_p}$ give above is exponential in the number of actual-outs. In the worst case, this bound is achievable, as demonstrated by the family of programs presented in Fig. 19, which shows the k^{th} representative, `Pk`. `Pk` has k recursive call-sites that call `Pk`; after the call at each call-site, a different pattern of assignments to the temporary variables `t1` ... `tk` is performed. In particular, just after the call at the i^{th} call-site, variable `ti` is zeroed out; each of the rest of the temporaries receives the value of the corresponding global variable `g1` ... `gk`.

[7] If L is a language, the *suffix closure* $L^{\rightarrow}$ is $\{b \mid \exists a \text{ such that } ab \in L\}$.

```
(1) unsigned int g1, . . ., gk;
(2)
(3) void Pk(int m) {
(4)   int v;
(5)   unsigned int t1, . . ., tk;
(6)
(7)   if (m == 0) return;
(8)   v = scanf("%d\n", &v);
(9)   if (v == 1) {
(10)    Pk(m-1);
(11)    t1 = 0; t2 = g2; . . . tk = gk;
(12)  }
(13)  else if (v == 2) {
(14)    Pk(m-1);
(15)    t1 = g1; t2 = 0; . . . tk = gk;
(16)  }
(17)  . . .
(18)  else {
(19)    Pk(m-1);
(20)    t1 = g1; t2 = g2; . . . tk = 0;
(21)  }
(22)  g1 = t1; g2 = t2; . . . gk = tk;
(23) }
(24)
(25) int main() {
(26)   g1 = 1; . . . gk = k;
(27)   Pk(k);
(28)   printf("%d\n", g1 + . . . + gk); /* SLICE HERE */
(29)   return 0;
(30) }
```

Figure 19. K^{th} representative of a family of programs for which a specialized slice is exponentially larger than the program.

In essence, the assignment `ti = 0` breaks the dependence between the preceding call-site and formal-out `gi`. That is, the call-site does not need an actual-out to carry the value of `gi`.

Because `Pk` calls itself recursively, the broken-dependence patterns at different instances of `Pk` in the unrolled SDG interact, and the slice with respect to the indicated point in Fig. 19 generates specialized procedures with different sets of actual-outs for all elements of the power set of global variables, $\mathcal{P}(\{\texttt{g1} \ldots \texttt{gk}\})$. The number of such procedures is thus 2^k.

Explosion in Practice. Our experiments indicate that exponential explosion does not arise in practice: no procedure had more than four specialized versions, and the vast majority of procedures had just a single version (see Fig. 16). Moreover, worst-case exponential behavior of operations like automaton determinization also does not seem to arise in practice. Thus, based on our experience to date, we believe it is fair to say that, for the *observed cost*,

> Both the running time and space of the algorithm are bounded by the sum of two terms: one is polynomial in the size of the input program; the other is linear in the size of the output slice.