Specialization Slicing

MIN AUNG, SUSAN HORWITZ, and RICH JOINER, University of Wisconsin
THOMAS REPS, University of Wisconsin and GrammaTech, Inc.

This paper defines a new variant of program slicing, called *specialization slicing*, and presents an algorithm for the specialization-slicing problem that creates an optimal output slice. An algorithm for specialization slicing is *polyvariant*: for a given procedure `p`, the algorithm may create multiple specialized copies of `p`. In creating specialized procedures, the algorithm must decide for which patterns of formal parameters a given procedure should be specialized, and which program elements should be included in each specialized procedure.

We formalize the specialization-slicing problem as a partitioning problem on the elements of the possibly *infinite* unrolled program. To manipulate possibly infinite sets of program elements, the algorithm makes use of automata-theoretic techniques originally developed in the model-checking community. The algorithm returns a *finite answer* that is optimal (with respect to a criterion defined in the paper). In particular, (i) each element replicated by the specialization-slicing algorithm provides information about specialized patterns of program behavior that are intrinsic to the program, and (ii) the answer is of minimal size (i.e., among all possible answers with property (i), there is no smaller one).

The specialization-slicing algorithm provides a new way to create executable slices. Moreover, by combining specialization slicing with forward slicing, we obtain a method for removing unwanted features from a program. While it was previously known how to solve the feature-removal problem for single-procedure programs, it was not known how to solve it for programs with procedure calls.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs and Features—*Control structures, procedures, functions, and subroutines, recursion*; F.1.1 [**Computation by Abstract Devices**]: Models of Computation—*Automata*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Partial evaluation, program analysis*

General Terms: Algorithms

Additional Key Words and Phrases: Program slicing, program specialization, executable slice, feature removal, program dependence graph, pushdown system, reverse-deterministic automaton

Authors' addresses: M. Aung, S. Horwitz, and R. Joiner, Computer Sciences Dept., Univ. of Wisconsin, 1210 W. Dayton St., Madison, WI 53703, {aung, horwitz, joiner}@cs.wisc.edu. T. Reps, Computer Sciences Dept., Univ. of Wisconsin, 1210 W. Dayton St., Madison, WI 53703, and GrammaTech, Inc., 531 Esty St., Ithaca, NY 14850; reps@cs.wisc.edu.

The work was supported in part by NSF under grants CCF-{0524051, 0540955, 0810053, 0904371}; by ONR under grants N00014-{01-1-0708, 01-1-0796, 09-1-0510, 09-1-0776, 10-M-0251, 11-C-0447}; by ARL under grant W911NF-09-1-0413; by AFRL under grants FA8750-05-C-0179, FA8750-06-C-0249, 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.

1. INTRODUCTION

Program slicing [Weiser 1984] provides a useful tool for many semantics-based program-manipulation operations. In particular, slicing produces semantically meaningful static decompositions of programs [Giacobazzi and Mastroeni 2003], consisting of program elements that are not textually contiguous.[1] Slicing is a fundamental operation that can aid in solving many software-engineering problems, including program understanding, maintenance, debugging, testing, differencing, specialization, and merging. (See §10 for references.) The term "program slicing" has been used to describe a number of different but related operations, but for purposes of introducing the specialization-slicing problem, we can start with the original definition due to Weiser [1984]: the *(static backward) slice of a program P from element q with respect to a set of variables V* is any (executable) 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* [Harman and Danicic 1997].

In this paper, we investigate the opportunities to be gained from broadening the definition of slicing and abandoning the restriction to syntax-preserving algorithms. A major inspiration for our work comes from the field of partial evaluation [Jones et al. 1993], in which a wide repertoire of techniques have been developed for specializing programs.

While slicing can also be harnessed for specializing programs [Reps and Turnidge 1996], due to the emphasis on syntax-preserving algorithms, the kind of specialization obtainable via slicing has heretofore been quite restricted, compared to the kind of specialization allowed in partial evaluation. In particular, a syntax-preserving slicing algorithm corresponds to what the partial-evaluation community calls 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 [Jones et al. 1993, p. 370].

In this paper, we investigate polyvariant program slicing—which we call *specialization slicing*—and present an algorithm that solves the specialization-slicing problem. For a given procedure `p`, a specialization-slicing algorithm may create multiple specialized copies of `p`. For example, consider the program shown in Fig. 1(a), and the specialization slice of the program with respect to the actual parameters

[1]Throughout the paper, we use the term "program elements" to mean items that one typically has in a relatively fine-grained intermediate representation, such as individual assignment statements, branch conditions, formal parameters, actual parameters, etc. We do *not* mean coarser-granularity program fragments, such as basic blocks or procedures.

(a) Backward closure slice w.r.t. printf's parameters on line (17)	(b) Specialized slice with two versions of p

(a)

```
 (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)

```
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);
}
```

Fig. 1. (a) Example program and (in boxes) the elements of the backward closure slice with respect to the actual parameters of the call to `printf` on line (17). (b) Specialization slice with respect to the same slicing criterion. Note that procedure `p` is specialized into two variants: `p_1` and `p_2`.

of the call to `printf` on line (17), shown in Fig. 1(b). Because `g2` is used on line (17) and `g1` is not, just after line (16) `g2` is relevant to the slice and `g1` is irrelevant. Therefore, the first actual parameter on line (16), which determines the value assigned to `g1`, is also irrelevant. Similarly, just after line (15) both `g1` and `g2` are relevant, and just after line (14) only `g2` is relevant. Because of these differences, the calls to `p` on lines (14) and (16) of Fig. 1(a) are converted in Fig. 1(b) into calls to the one-parameter procedure `p_1`, which assigns to `g2` but not `g1`, while the call to `p` on line (15) is converted into a call to the two-parameter procedure `p_2`, which assigns to both `g1` and `g2`.

The specialization-slicing algorithm still has the main characteristics of a slicing algorithm—that is, the elements of the output slice are all elements from the input program: no evaluation or simplification is performed. Stated another way, our work adopts just one feature from the partial-evaluation literature—polyvariance—and studies how that extension changes the slicing problem.

The specialization-slicing algorithm can be thought of as starting with information similar to what is obtained from what has been called "closure slicing," and then modifying the closure-slice result. Closure slicing and other relevant background material will be reviewed in §2. For the purposes of the discussion here, the relevant issue is that a closure slice can have multiple calls to the same procedure, with different subsets of actual parameters at different call-sites. A closure slice contains just a single version of each called procedure P. P's formal-parameter set is the union of the formal parameters that correspond to the actual parameters (at call-sites that call P) in the closure slice. Consequently, a closure slice can have mismatches between the actual parameters at a call-site and the procedure's formal parameters [Horwitz et al. 1990, §1]. For instance, in lines (14) and (16) of Fig. 1(a), the second actual parameter is in the closure slice, but the first actual parameter is not; in line (15), both actual parameters are in the closure slice; and in line (3), both formal parameters are in the closure slice. Thus, there are mismatches between lines (14) and (3), and between lines (16) and (3). In contrast, the specialization-slicing algorithm can create specialized versions of a called procedure—one for each different subset of the actual parameters occurring at different call-sites—so that each call-site calls a specialized version whose formal parameters match the call-site's actual parameters (cf. lines (14), (16), and (3) and lines (15) and (7) of Fig. 1(b)).

Specialization slicing represents a new point in the "design space" of slicing problems: (i) the specialization-slicing algorithm is not syntax-preserving because a slice can contain multiple versions of a procedure; however, (ii) the specialization-slicing algorithm *never* introduces program elements that are not also in the closure slice (albeit the specialization slice may contain multiple copies of such elements, such as the two copies of "`g2 = b`" in lines (4) and (9) of Fig. 1(b)).

In creating specialized procedures, a specialization-slicing algorithm must decide for which subsets of the formal parameters a given procedure should be specialized, and which program elements should be included in each specialized procedure. As we show in §3.2 and §5.3, 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 (§5.3).

If exponential explosion was the usual outcome, our algorithm would not be useful. However, 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 (90.8%) had just a single version (see §9). Moreover, the experiments showed that the overall size increase is also modest. Normalized to "|closure slice| = 100," on average (computed as the geometric mean) specialization slices are 105.5. That is, 5.5% worth of closure-slice elements are *replicated*. However, there is a sense in which our specialization-slicing algorithm returns an answer that is optimal: (i) the result is minimal in a sense defined precisely in §3.1, and (ii) each element replicated by the specialization-slicing algorithm is necessary for the slice to capture one of the program's specialized patterns of behavior.[2]

Fig. 1 is an example of specialization slicing applied to a non-recursive program. To see the outcome of specialization slicing on a recursive program, consider the program shown in Fig. 2(a) and the specialization slice with respect to line (28), shown in Fig. 2(b). This example shows what happens when some program elements are relevant in one recursive invocation, but not relevant in another recursive invocation of the same procedure. Note that just after line (27), `g1` is relevant to the slice, but `g2` is not. However, just after line (16), in some calling contexts `g1` is relevant, but not `g2`, while in other calling contexts `g2` is relevant, but not `g1`.

[2]It will become clearer in later sections that—as with much past work on program slicing—the informal notion of "patterns of behavior" will be formalized as "patterns of dependences."

	(a) Recursive program	(b) Specialization 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)	`}`	`}`

Fig. 2. (a) A program with recursive procedure `r`. All elements of the program are in the closure slice with respect to line (28). (b) The specialization slice of the program with respect to line (28).

These differences produce two kinds of effects:

(1) Procedure `s` is specialized into two versions (`s_1` with parameter `b` and `s_2` with parameter `a`).

(2) Procedure `r`, which has a single parameter in the original program and still has a single parameter in the output slice, is *also* specialized into two versions, which have two different patterns of behavior in their bodies:
—`r_1` makes the calls "`s_2(g1); r_2(k-1); s_1(g2);`"
—`r_2` makes the calls "`s_1(g2); r_1(k-1); s_2(g1);`."

As a consequence of item (2), the pattern of recursion in the specialization slice shown in Fig. 2(b) is different from the pattern of recursion in Fig. 2(a). The program in Fig. 2(a) uses *direct* recursion: `r` calls `r` calls ... In contrast, in Fig. 2(b), `r_1` and `r_2` are *mutually* recursive. That is, specialization slicing caused a use of direct recursion to be converted into mutual recursion. As we will show, the specialization-slicing algorithm that we give in §4 automatically identifies the correct procedure variant to call, even when specialization slicing introduces mutual recursion among specialized procedures.

1.1 Two Strawman Algorithms

Given the level of sophistication of the machinery that will be introduced in the paper, it is natural to wonder whether a simpler method suffices. At the beginning of our work on specialization slicing, we considered numerous candidate algorithms, including the two discussed in §1.1.1 and §1.1.2. We present these here because the first provides some insight on the correct *specification* of the specialization-slicing problem, while the second is representative of several plausible-sounding attempts that all turned out to be flawed.

The flaws in such attempts motivated us to back up and reconsider the problem from first principles. The ideas that we came up with are presented in §3, and used to define the goals of a specialization-slicing algorithm in terms of *soundness*, *completeness*, and *minimality* conditions. We then give an algorithm that is sound and complete, and returns a minimal specialization slice (§4).

1.1.1 *Procedure Cloning.* One approach to specialization slicing is based on exhaustive *procedure cloning* (performed in a manner similar to exhaustive in-line expansion):

(1) Starting with the call-sites in the main procedure, give each call-site that calls procedure p its own copy of p, which itself is recursively unfolded in the same manner.
(2) Perform closure slicing on the unfolded program.
(3) Repeatedly merge copies of procedures that, in the closure slice, have identical sets of formal parameters and identical sets of elements in the procedure body.

For instance, in Fig. 1(a), step (1) would create three copies of procedure `p`, corresponding to the calls on lines (14), (15), and (16), respectively. The closure slice would involve elements in all three copies of `p`. The slices of the copies from lines (14) and (16) would be identical, and would be folded together by step (3) to create the one-parameter procedure `p_1` given on lines (3)–(5) of Fig. 1(b).

The cloning-based algorithm partially satisfies our requirements; however, it has two drawbacks:

—Cloning step (1) would not terminate for a program that uses recursion.
—For a non-recursive program, step (1) can lead to a program that is exponentially larger than the original program.[3]

However, the cloning-based algorithm does provide intuition about what a specialization-slicing algorithm should achieve, and thus helps with understanding the *specification* of the specialization-slicing problem. For instance, if you imagine performing a closure slice of the infinite program that would result from applying the cloning step to Fig. 2(a), it is not hard to convince yourself that the slice has only two variants of procedure `s`, and two variants of procedure `r`—where the latter call each other mutually recursively.

[3]Consider a non-recursive program in which procedure p_k contains two calls to p_{k-1}; p_{k-1} contains two calls to p_{k-2}; etc. The program obtained via cloning step (1) will be exponentially larger than the original program.

The steps of the specialization-slicing algorithm that we present in §4 can be viewed as harnessing *symbolic techniques* to perform steps (1)–(3) of the cloning-based approach—but without explicit exhaustive procedure cloning, thereby sidestepping the two problems identified above.

An algorithm is said to be *output-sensitive* if "[its] running time depends on the size of the output, instead of or in addition to the size of the input" [Wikipedia: Output-Sensitive Algorithm 2014]. Our algorithm is an output-sensitive algorithm for specialization slicing. An algorithm that performs explicit cloning eagerly accepts an exponential cost up-front, whereas it may not actually be necessary to do that much work. Our algorithm shows that one can do better than cloning eagerly. For examples with the calling structure described in footnote 3, our algorithm exhibits exponential behavior *if each of the leaf procedure instances needs to be specialized differently.* (The family of examples discussed in the paper in §5.3 is exactly such a case—see Figs. 14 and 15.) For instance, the algorithm uses an exponential amount of time just to print out the answer. However, our algorithm does *not* exhibit exponential behavior if the leaf procedures do not all need to be specialized differently. Our algorithm is "lazy" in the sense that it ends up considering only the specialized procedures that are actually needed in the answer. While there are many lazy algorithms in computer science, the way laziness is achieved in our algorithm is somewhat unusual (see §4).

1.1.2 *Removing Extra Vertices.* A *forward* slice with respect to a slicing criterion C is the set of all program elements that might be affected by the computations performed at members of C [Horwitz et al. 1990, §4.5]. Another approach to specialization slicing might be to create the specialized version of a callee by removing vertices that are in the forward slice from "extra" formal parameters in the closure slice. That is, to specialize procedure `p` for call-sites like the ones on lines (14) and (16) of Fig. 1(a), where the second actual parameter is in the closure slice but the first actual parameter is not, the proposed algorithm would

—make a copy of the closure slice of `p`, and

—remove from the copy of `p` all vertices in the forward slice with respect to the first formal-parameter.

It would be necessary to iterate this process until there are no further parameter mismatches; however, tabulation can be used to avoid re-analyzing a method that has already been specialized.

Unfortunately, the (non-recursive) example shown in Fig. 3 shows that this approach can leave in unneeded program elements in some cases. In Fig. 3(b), the assignment `z = 3` in specialized procedure `p_2` is needed because variable `z` is used in the very next statement, `g2 = b + z`. In contrast, in specialized procedure `p_1`, the assignment `z = 3` is an *extra* statement; in compiler parlance, `z` is a dead variable there, and `z = 3` is a useless assignment. The statement `z = 3` is retained in `p_1` by the candidate algorithm because `z = 3` is in the closure slice, but is *not* in the forward slice from the unneeded formal-in `b` (which corresponds to the unneeded actual-in `2` in the call `p(g2,2)` in `main`).

(a) Program and a closure slice	(b) Candidate specialization slice
`int g1, g2;`	`int g1, g2;`
`void p(int a, int b) {`	`void p_1(int a) {`
`  g1 = a;`	`  g1 = a;`
`  int z = 3;`	`  int z = 3; // EXTRA!`
`  g2 = b + z;`	`}`
`}`	
	`void p_2(int b) {`
	`  int z = 3; // OK`
	`  g2 = b + z;`
	`}`
`int main() {`	`int main() {`
`  p(11,4); // g2 = 4 + 3;`	`  p_2(4);  // g2 = 4 + 3;`
`  p(g2,2); // g1 = g2;`	`  p_1(g2); // g1 = g2;`
`  printf("%d",g1); // slice pt.`	`  printf("%d",g1);`
`}`	`}`

Fig. 3. Non-recursive program that demonstrates that the specialization-slicing method discussed in §1.1.2 can leave in unneeded program elements. (a) Example program and (in boxes) the elements of the closure slice with respect to line (17). (b) The specialization slice returned by the proposed method.

1.2 Contributions

This paper defines specialization slicing, describes an elegant algorithm for solving the problem, and presents results from studying specialization slicing from a number of angles.

—We formalize the problem of specialization slicing as a partitioning problem on the elements of the (possibly infinite) unrolled program (§3.1). We give definitions of *soundness*, *completeness*, and *minimality* for specialization slicing (§3.1).

—To solve the partitioning problem, we bring to bear techniques that are quite different from what have been used in most other work on program slicing. In particular, to represent finitely the infinite sets of objects that we need to manipulate to solve the partitioning problem, we make use of symbolic techniques originally developed in the model-checking community [Bouajjani et al. 1997; Finkel et al. 1997]. Using this machinery, we give an algorithm in §4 that with just a few simple automata-theoretic operations identifies
  —the minimal set of specialized procedures that capture each of the different patterns of behavior for a given procedure, as well as
  —the minimal set of program elements required in each specialized procedure.

—We prove that our specialization-slicing algorithm is sound and complete, and returns a minimal specialization slice (§4.4 and Appendix A); consequently, the algorithm always creates an optimal output slice (§5.1).

—We characterize the running time and space used by the algorithm (§5).
  —We present a family of examples for which the running time and space of the specialization-slicing algorithm can be exponential in certain parameters of the input program (§5.3).

  - —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.
- —The specialization-slicing algorithm provides a new way to create executable slices—in particular, it creates polyvariant executable slices (§6).
- —We describe several extensions of the basic specialization-slicing algorithm:
  - —We describe how to extend the algorithm to handle programs that (i) make calls to library procedures, and (ii) use calls via pointers to procedures. (§7.1 and §7.2, respectively).
  - —We show that the algorithm possesses a kind of idempotence property (§7.3).
  - —We show how to speed up one of the key steps of the algorithm (§7.4).
- —We describe a method for removing unwanted program features (§8). The method uses specialization slicing in conjunction with forward slicing. While it was previously known how to solve the feature-removal problem for single-procedure programs, no algorithm was known for multi-procedure programs.
- —In §9, we present the results of experiments using C programs to evaluate (i) our specialization-slicing algorithm (for polyvariant executable slicing), and (ii) an algorithm for monovariant executable slicing [Binkley 1993]. To the best of our knowledge—confirmed by Binkley [2012]—these results represent the first published data on the performance of the monovariant algorithm for executable slicing.

§10 discusses related work. §11 concludes. Proofs are given in Appendix A.

2. BACKGROUND

2.1 System Dependence Graphs

In some slicing algorithms, a slicing criterion (q, V) is restricted so that V can only include variables used at program element q. Ottenstein and Ottenstein [1984] observed that with such slicing criteria, slicing can be performed efficiently using program dependence graphs. Our specialization-slicing algorithm adopts this restriction on slicing criteria. (In the remainder of the paper, we also assume that slices are with respect to *all* variables used at q. However, the slicing criteria that we consider can consist of *sets* of program elements, not just a single element q.)

Horwitz et al. [1990] presented a context-sensitive algorithm for interprocedural slicing that uses a data structure they defined, called a *system dependence graph* (SDG). An SDG 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., [Kuck et al. 1981; Ottenstein and Ottenstein 1984]), but represents collections of procedures rather than just monolithic programs.

Definition 2.1. *(System Dependence Graph (SDG)) A program's* ***system dependence graph (SDG)*** *is a collection of procedure dependence graphs (PDGs), one for each procedure. Each PDG has an* ***entry vertex****, plus vertices that represent the procedure's statements and conditions [Ottenstein and Ottenstein 1984].*

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 [Cooper and Kennedy 1988]. 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 a collection of* ***formal-in*** *and* ***formal-out vertices****.*

A PDG's edges represent the control and flow dependences among the procedure's statements and conditions.[4] *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 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).*[5] □

The size of the SDG is polynomial in various parameters that characterize the size of the original program (see [Horwitz et al. 1990, §5.1] for details).

Example 2.2. Fig. 4 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. 4 does not include the PDG for `printf`. We discuss how we handle library calls in §7.1. □

2.2 Closure Slicing

Horwitz et al. [1990] presented a context-sensitive algorithm for interprocedural slicing that produces more precise (smaller) slices than Weiser's algorithm. However, while Weiser's slices are *executable*, the algorithm given by Horwitz et al. produces a *closure* slice: the slice of a program with respect to criterion $(q, \{x\})$ consists of all statements and conditions of the program that might affect the value of x at program element q.

Example 2.3. The elements of Fig. 4 shown with bold font and darker borders, lines, and dashed lines are the ones identified by the context-sensitive, interprocedural closure-slice algorithm of Horwitz et al. [1990] when the SDG is sliced with respect to $\{m22, m23\}$. This example corresponds to slicing Fig. 1(a) with respect to the actual parameters of the call to `printf` on line (17). □

[4]As defined by Horwitz et al. [1990], 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.

[5]The SDGs defined by Horwitz et al. [1990] also include *summary edges*, which run from actual-in to actual-out vertices. Summary edges are not needed for defining the goal of specialization slicing, nor for presenting our initial specialization-slicing algorithm (Alg. 1). Summary edges are introduced in §7.4, where we give an improved algorithm for one of the key steps of specialization slicing (Alg. 2).

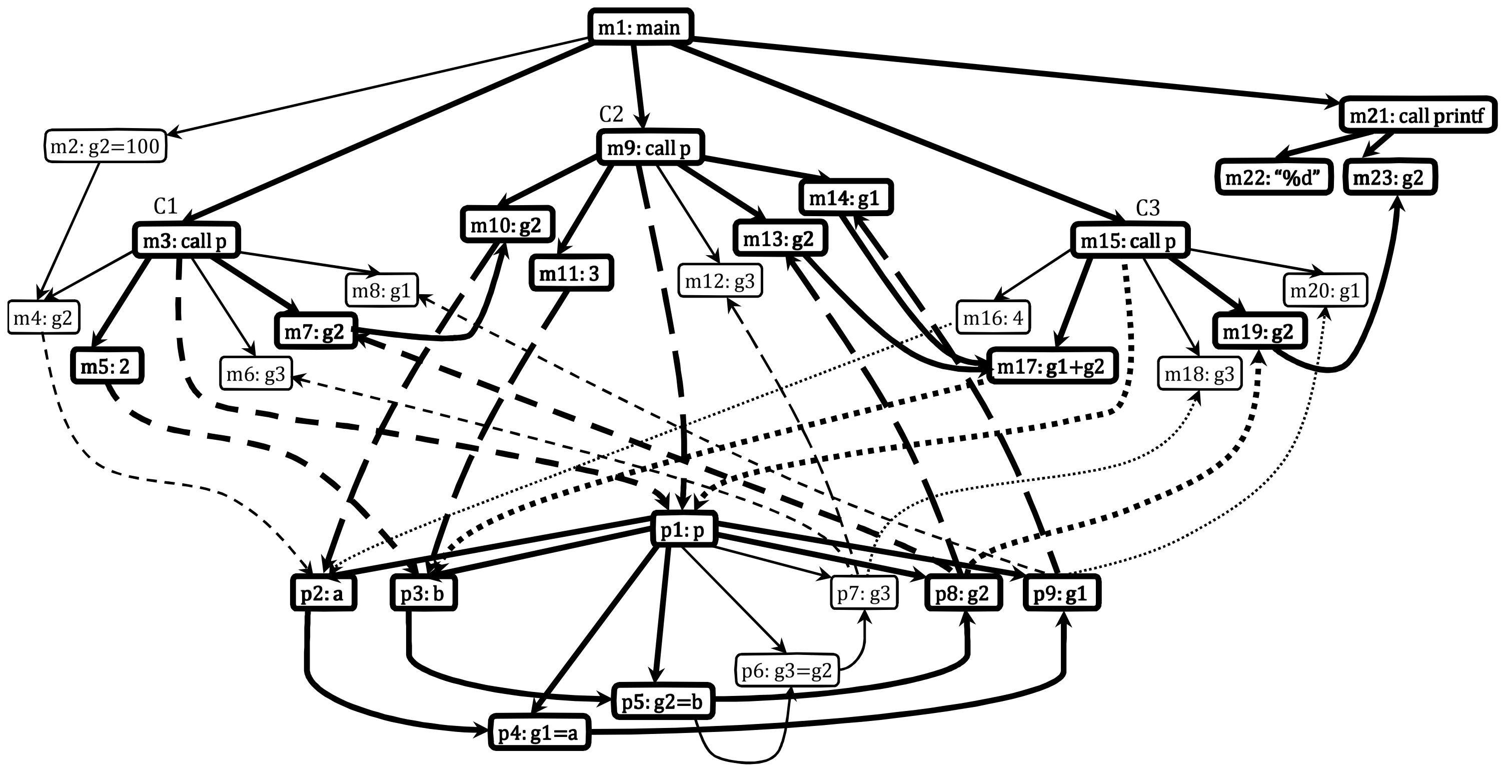


Fig. 4. System dependence graph (SDG) for the program shown in Fig. 1(a). The vertices with labels starting with m are part of the PDG for `main`; those whose labels start with p are part of the PDG for `p`. The two entry vertices are $m1$ and $p1$. The formal-in vertices of the PDG for `p` are $p2$ and $p3$; the formal-out vertices are $p7$, $p8$, and $p9$. The four call vertices are $m3$, $m9$, $m15$, and $m21$; the actual-in vertices associated with $m3$ are $m4$ and $m5$; the actual-out vertices associated with $m3$ are $m6$, $m7$, and $m8$. Solid lines represent control and flow dependences; dashed lines represent parameter-in, parameter-out, and call edges. The elements of the context-sensitive closure slice of the SDG with respect to $\{m22, m23\}$ are shown with bold font and darker borders, lines, and dashed lines.

A closure slice can have multiple calls to the same procedure, with different subsets of actual parameters at different call-sites. However, the slice contains the union of the corresponding formal-parameter sets, which causes a mismatch between the actual parameters at a call-site and the procedure's formal parameters [Horwitz et al. 1990, §1] (e.g., compare the boxes on lines (14) and (16) of Fig. 1(a) with the box on line (3)).

Example 2.4. The parameter-mismatch problem is illustrated in Fig. 4 by the mismatch between actual-ins $m4$ and $m16$ from call-sites $C1$ and $C3$, respectively, which are *not* in the slice, and formal-in $p2$, which *is* in the slice. □

For most programming languages, a program with a parameter mismatch would trigger a compile-time error, or at least a warning. (The main use of closure slicing *per se* is in tools for code understanding, such as CodeSurfer® [Anderson et al. 2003], which lets the user navigate along the dependence edges in the closure slice.) In contrast, slices produced by our specialization-slicing algorithm never exhibit such parameter mismatches, and thus specialization slices are executable.

We will have more to say about executable slicing in §6, and a different kind of parameter-mismatch problem in §7.2.

2.3 Pushdown Systems, SDGs, and Unrolled SDGs

In §3.1, we use a family of infinite graphs to define the specialization-slicing problem, and in §4 we use symbolic techniques for working with such infinite graphs. This section defines the infinite graphs that we use—namely, the transition relations of pushdown systems (PDSs) [Bouajjani et al. 1997; Finkel et al. 1997].

Definition 2.5. *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* $u \in \Gamma^*$.[6]

The rules in Δ *define 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 each* $u \in \Gamma^*$. *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. 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.

[6]We use ϵ to denote an empty sequence. We use the juxtaposition of symbols, separated by spaces, to denote a non-empty sequence. For emphasis, we sometimes enclose a sequence in parentheses. Following the conventions of the literature on pushdown systems, if the sequence $(A\ B\ C)$ denotes a stack or portion of a stack, the symbol A is the top-of-stack symbol—i.e., the top of the stack is at the left.

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

Fig. 5. A schema for encoding an SDG's edges using PDS rules.

Definition 2.6. *We* ***encode*** *an SDG as a PDS using the schema given in Fig. 5. 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 parameter-in edge is encoded with a push rule*

—*a parameter-out edge is encoded with a pop rule and an internal rule.*

A common control location p is used in all of the PDS rules Fig. 5, except in the rules that encode parameter-out edges.

The principle behind the rules that encode a parameter-out edge $fo \to ao$ at call-site C is as follows. Reading the rules in the forward direction,

—*the next-to-last rule of Fig. 5, 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 is available when call-site symbol C is exposed at the top of the stack.*

—*the last rule of Fig. 5, the internal rule $\langle p_{fo}, C\rangle \hookrightarrow \langle p, ao\rangle$, replaces C on the stack with the actual-out vertex ao that matches fo at call-site C.*

In other words, a parameter-out edge $fo \to ao$ at call-site C causes the PDS's transition relation to contain, for each $u \in \Gamma^*$, a path of length two of the form

$$(p, fo\, C\, u) \Rightarrow (p_{fo}, C\, u) \Rightarrow (p, ao\, u).$$

Given SDG G, the ***unrolling*** *of G is the transition relation of the PDS that encodes G via the schema given in Fig. 5.* □

Example 2.7. Consider again the SDG from Fig. 4 and program from Fig. 1(a). The three call-sites on procedure `p` are labeled with $C1$, $C2$, and $C3$.

Tab. I shows the PDS rules that encode the SDG from Fig. 4. Fig. 6 shows the unrolled SDG (i.e., the transition relation generated by Tab. I, which encodes Fig. 4).[7] Note that the unrolled SDG is quite similar to the SDG that would be obtained by repeatedly performing PDG cloning (similar to the procedure-cloning idea discussed in §1.1.1): starting with the call-sites in the PDG for `main`, give each call-site that calls PDG p its own copy of p, which itself is recursively unfolded in the same manner.

[7]Technically, Fig. 6 shows just the part of the transition relation that is reachable from $\mathcal{P}$-configuration $(p, m1)$.

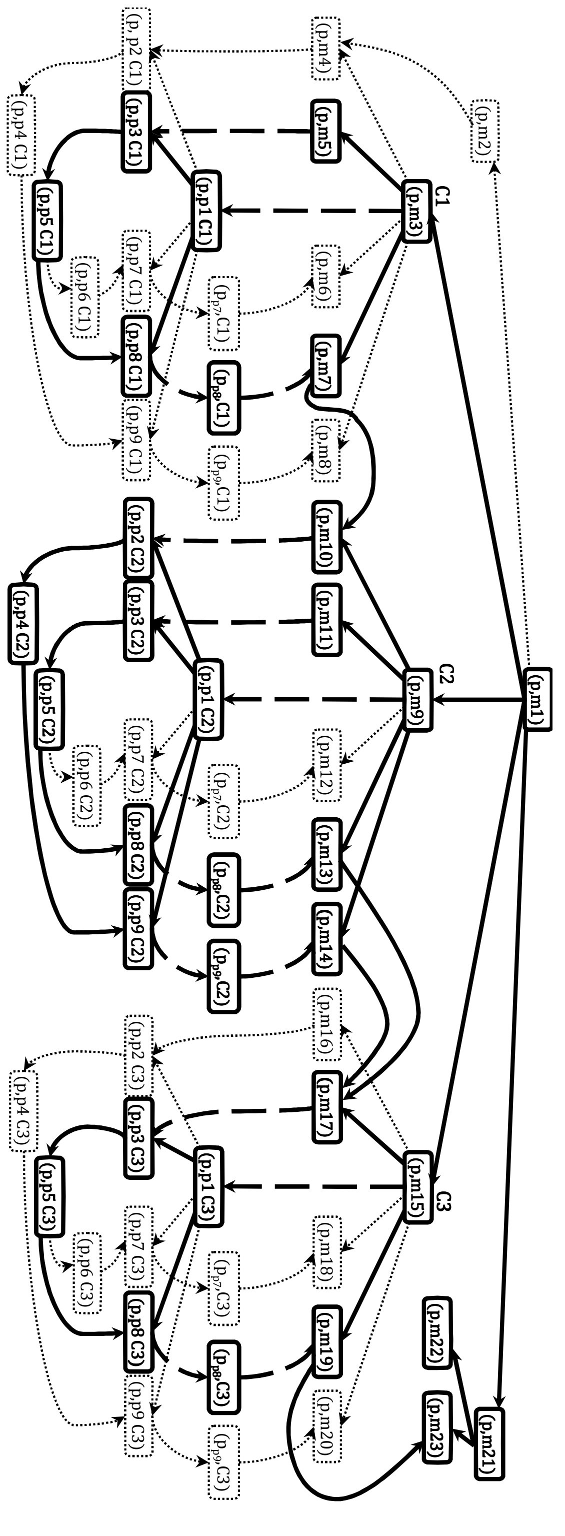


Fig. 6. Transition relation of the PDS encoding of the SDG shown in Fig. 4. Each vertex is labeled with a $\mathcal{P}$-configuration of the form $(p, \text{PDG-vertex call-stack})$. The closure slice with respect to the $\mathcal{P}$-configuration set $\{(p, m22), (p, m23)\}$ is shown with bold font and darker borders, lines, and dashed lines, and corresponds to the boxed elements in Fig. 1(a).

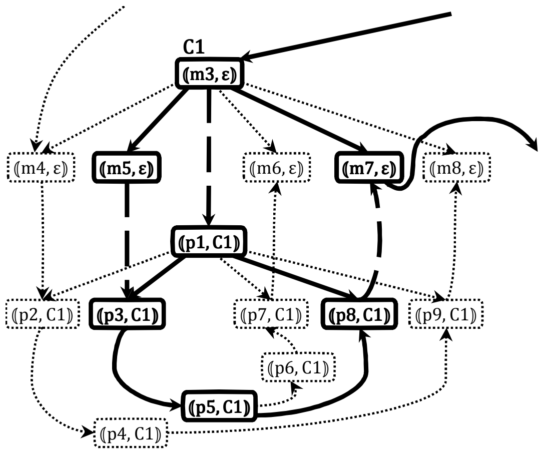
Control-dependence and flow-dependence edges in `main`	
1. $\langle p, m1\rangle \hookrightarrow \langle p, m2\rangle$	2. $\langle p, m1\rangle \hookrightarrow \langle p, m3\rangle$
3. $\langle p, m1\rangle \hookrightarrow \langle p, m9\rangle$	4. $\langle p, m1\rangle \hookrightarrow \langle p, m15\rangle$
5. $\langle p, m1\rangle \hookrightarrow \langle p, m21\rangle$	6. $\langle p, m2\rangle \hookrightarrow \langle p, m4\rangle$
... 18 rules omitted	...
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. $\langle p, p1\rangle \hookrightarrow \langle p, p5\rangle$
... 8 rules omitted	...
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 I. The PDS rules that encode the SDG shown in Fig. 4, using the schema given in Fig. 5.

The one place where there is a slight difference between the PDS's transition system and the result of exhaustive PDG cloning is in the treatment of parameter-out edges. For instance, under the schema given in Fig. 5, the three parameter-out edges $p9 \to m8$, $p9 \to m14$, and $p9 \to m20$ in Fig. 4, at call-sites $C1$, $C2$, and $C3$, respectively, are encoded in Tab. I by (i) rules 59 and 60, (ii) rules 59 and 61, and (iii) rules 59 and 62, respectively. Consequently, the unrolled SDG contains the following three paths of length two:

$$\begin{array}{l}(p, p9\,C1) \Rightarrow (p_{p9}, C1) \Rightarrow (p, m8)\\ (p, p9\,C2) \Rightarrow (p_{p9}, C2) \Rightarrow (p, m14)\\ (p, p9\,C3) \Rightarrow (p_{p9}, C3) \Rightarrow (p, m20).\end{array}$$

Henceforth, to reduce clutter in diagrams, we will not show intermediate $\mathcal{P}$-configurations like $(p_{p9}, C1)$ in Fig. 6; instead, we will just show a single edge $(p, p9\,C1) \Rightarrow (p, m8)$. Moreover, because each $\mathcal{P}$-configuration $(p, v\,u)$ in such diagrams has the same control-location p, we will typically drop the control location, and denote the remainder as $(\!|v, u|\!)$. (Note that v corresponds to an SDG vertex, and u is a sequence of call-sites in the SDG.) For instance, the leftmost PDG in Fig. 6 would be shown as follows:



□

Definition 2.8. *(Configuration and Stack Configuration) In an unrolled SDG, each call-site invokes a unique instance of the called PDG. Consequently, each vertex c in an unrolled SDG is associated with a unique sequence of call-sites in the SDG, and thus c can be labeled uniquely with a pair* $⦇v, u⦈$, *where v is a PDG vertex from the original SDG, and u is a sequence of call-sites that represents the calling context in which c arises. We say that* $⦇v, u⦈$ *is a* ***configuration*** *that has the* ***stack configuration*** *u. The sequence u represents the stack of calls that are pending in configuration* $⦇v, u⦈$*.*

Given a set of configurations C, Elems(C) denotes the set of PDG vertices

$$\mathit{Elems}(C) \stackrel{\text{def}}{=} \{v \mid ⦇v, u⦈ \in C\},$$

and Stacks(C) denotes the set of stack configurations

$$\mathit{Stacks}(C) \stackrel{\text{def}}{=} \{u \mid ⦇v, u⦈ \in C\}.$$

□

Note that with our nomenclature, a *configuration* such as $⦇p9, C1⦈$ has *stack configuration* $C1$, and corresponds to $\mathcal{P}$*-configuration* $(p, p9\,C1)$. Throughout, we are careful to use the qualifying terms “stack configuration” and “$\mathcal{P}$-configuration,” so there should be no confusion.

3. PROBLEM STATEMENT

As we explored the concept of specialization slicing, we considered numerous candidate algorithms, such as the one discussed in §1. The flaws in such attempts motivated us to find foundational principles that we could use to define the objective that specialization slicing should achieve. This section presents these principles, building up to a three-part definition of the specialization-slicing problem that we wish to solve (Eqn. (3), Defn. 3.5, and Defn. 3.6). In §3.1, we use a non-recursive program to motivate and illustrate these definitions. In §3.2, we double-check the problem definition by considering a specialization slice of a recursive program.

3.1 Insights and Definitions

Insight 1: Symbolic Techniques for Infinite Graphs. We formalize the specialization-slicing problem in terms of the (conceptual) unrolled SDG (Defn. 2.6), which for recursive programs has an infinite number of vertices and edges. To represent finitely the possibly infinite sets of objects that we will use to solve the specialization-slicing problem, we adopt symbolic techniques that were originally developed in the model-checking community. As explained in §2.3, we use a pushdown system (PDS) to encode the SDG. This approach immediately provides us with an *algorithm* to perform a generalized kind of program slicing: the pre^* operation [Bouajjani et al. 1997; Finkel et al. 1997] 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 3.1. Each vertex in Fig. 6 is labeled with a configuration of the form $\llparenthesis \text{PDG-vertex}, \text{call-stack} \rrparenthesis$. Because Fig. 1(a) is not recursive, the set of configurations in the unrolled SDG is a finite set, namely,

$$\left\{ \begin{array}{llllll}
\llparenthesis m1,\epsilon\rrparenthesis, & \llparenthesis m10,\epsilon\rrparenthesis, & \llparenthesis m19,\epsilon\rrparenthesis, & \llparenthesis p1,C1\rrparenthesis, & \llparenthesis p1,C2\rrparenthesis, & \llparenthesis p1,C3\rrparenthesis, \\
\llparenthesis m2,\epsilon\rrparenthesis, & \llparenthesis m11,\epsilon\rrparenthesis, & \llparenthesis m20,\epsilon\rrparenthesis, & \llparenthesis p2,C1\rrparenthesis, & \llparenthesis p2,C2\rrparenthesis, & \llparenthesis p2,C3\rrparenthesis, \\
\llparenthesis m3,\epsilon\rrparenthesis, & \llparenthesis m12,\epsilon\rrparenthesis, & \llparenthesis m21,\epsilon\rrparenthesis, & \llparenthesis p3,C1\rrparenthesis, & \llparenthesis p3,C2\rrparenthesis, & \llparenthesis p3,C3\rrparenthesis, \\
\llparenthesis m4,\epsilon\rrparenthesis, & \llparenthesis m13,\epsilon\rrparenthesis, & \llparenthesis m22,\epsilon\rrparenthesis, & \llparenthesis p4,C1\rrparenthesis, & \llparenthesis p4,C2\rrparenthesis, & \llparenthesis p4,C3\rrparenthesis, \\
\llparenthesis m5,\epsilon\rrparenthesis, & \llparenthesis m14,\epsilon\rrparenthesis, & \llparenthesis m23,\epsilon\rrparenthesis, & \llparenthesis p5,C1\rrparenthesis, & \llparenthesis p5,C2\rrparenthesis, & \llparenthesis p5,C3\rrparenthesis, \\
\llparenthesis m6,\epsilon\rrparenthesis, & \llparenthesis m15,\epsilon\rrparenthesis, & & \llparenthesis p6,C1\rrparenthesis, & \llparenthesis p6,C2\rrparenthesis, & \llparenthesis p6,C3\rrparenthesis, \\
\llparenthesis m7,\epsilon\rrparenthesis, & \llparenthesis m16,\epsilon\rrparenthesis, & & \llparenthesis p7,C1\rrparenthesis, & \llparenthesis p7,C2\rrparenthesis, & \llparenthesis p7,C3\rrparenthesis, \\
\llparenthesis m8,\epsilon\rrparenthesis, & \llparenthesis m17,\epsilon\rrparenthesis, & & \llparenthesis p8,C1\rrparenthesis, & \llparenthesis p8,C2\rrparenthesis, & \llparenthesis p8,C3\rrparenthesis, \\
\llparenthesis m9,\epsilon\rrparenthesis, & \llparenthesis m18,\epsilon\rrparenthesis, & & \llparenthesis p9,C1\rrparenthesis, & \llparenthesis p9,C2\rrparenthesis, & \llparenthesis p9,C3\rrparenthesis
\end{array} \right\} \tag{1}$$

For the program shown in Fig. 1(a), the stack-configuration slice from line (17) corresponds to the closure slice of the unrolled SDG shown in Fig. 6 from configuration set $\{\llparenthesis m22,\epsilon\rrparenthesis, \llparenthesis m23,\epsilon\rrparenthesis\}$. The items in the slice are shown in Fig. 6 with bold font and darker borders, lines, and dashed lines. In this case, the set of configurations in the slice is the finite set

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

□

Stack-configuration slicing finds all $\llparenthesis \text{PDG-vertex}, \text{call-stack} \rrparenthesis$ configurations on which a given language of $\llparenthesis \text{PDG-vertex}, \text{call-stack} \rrparenthesis$ 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.[8] (This material will be reviewed in §4.1.)

DEFINITION 3.2. *The unrolled SDG may contain many* ***instances*** *of a procedure P; each instance of procedure P is a set C_u of configurations of the form $(\!|v,u|\!)$, where all the u are the same—i.e., $C_u \overset{\text{def}}{=} \{(\!|v,u|\!) \in$ unrolled SDG $\mid v \in P\}$. If C_u is an instance of procedure P, and T is the set of configurations of a stack-configuration slice, then $P_u = C_u \cap T$ is the u-****variant*** *of P in T.*

A ***specialization*** *of P with respect to T is the set of PDG vertices Elems(P_u) (for some u-variant P_u in the slice). Because Elems(P_u) drops all stack-configuration components from u-variant P_u, multiple variants P_{u_1} and P_{u_2} can result in the same specialization, consisting of Elems(P_{u_1}) = Elems(P_{u_2}).* □

EXAMPLE 3.3. In Fig. 6 and Eqn. (1), there are three *instances* of procedure `p`, consisting of the configurations

(1) $C_{C1} = \{(\!|p1, C1|\!), (\!|p2, C1|\!), (\!|p3, C1|\!), \ldots, (\!|p8, C1|\!), (\!|p9, C1|\!)\}$
(2) $C_{C2} = \{(\!|p1, C2|\!), (\!|p2, C2|\!), (\!|p3, C2|\!), \ldots, (\!|p8, C2|\!), (\!|p9, C2|\!)\}$
(3) $C_{C3} = \{(\!|p1, C3|\!), (\!|p2, C3|\!), (\!|p3, C3|\!), \ldots, (\!|p8, C3|\!), (\!|p9, C3|\!)\}$

In the slice shown in Fig. 6, there are three *variants* of procedure `p`:

(1) $\mathtt{p}_{C1} = \{(\!|p1, C1|\!), (\!|p3, C1|\!), (\!|p5, C1|\!), (\!|p8, C1|\!)\}$
(2) $\mathtt{p}_{C2} = \left\{\begin{array}{l}(\!|p1, C2|\!), (\!|p2, C2|\!), (\!|p3, C2|\!), (\!|p4, C2|\!), (\!|p5, C2|\!), \\ (\!|p8, C2|\!), (\!|p9, C2|\!)\end{array}\right\}$
(3) $\mathtt{p}_{C3} = \{(\!|p1, C3|\!), (\!|p3, C3|\!), (\!|p5, C3|\!), (\!|p8, C3|\!)\}$

However, because the $C1$-variant and the $C3$-variant have the same Elems components, there are only two *specializations* of `p`—namely, those for vertex sets $\{p1, p3, p5, p8\}$ and $\{p1, p2, p3, p4, p5, p8, p9\}$. Consequently, Fig. 1(b) has two specialized versions of `p` (named `p_1` and `p_2`). □

Problem Statement (Part I): Definition of Specialization Slicing. With these concepts, we can formulate the problem of specialization slicing as follows. The core problem is to identify, for each procedure P, each of the different sets of program elements that make up the variants of P. These sets can be characterized as follows:

$$\begin{aligned}&\text{Specializations}(P) = \\ &\qquad \{\text{Elems}(V) \mid V \text{ a variant of } P \text{ in the stack-configuration slice}\}.\end{aligned} \tag{3}$$

For instance, in Fig. 6, Specializations(`p`) is

$$\{\{p1, p3, p5, p8\}, \{p1, p2, p3, p4, p5, p8, p9\}\}.$$

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

[8] For instance, consider the PDS that consists of the single pop rule $\langle p, A\rangle \hookrightarrow \langle p, \epsilon\rangle$. The PDS's transition relation is

$$\ldots \Rightarrow (p, A\,A\,A) \Rightarrow (p, A\,A) \Rightarrow (p, A) \Rightarrow (p, \epsilon)$$

The result of the query $Prestar((p, A))$ is the infinite set $\{(p, A), (p, A\,A), (p, A\,A\,A), \ldots\}$, which is the regular language of configurations $(p, A\,A^*)$. Such a language can also be represented by an FSA.

of different Elems(V) sets. Therefore, Specializations(P) is 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 3.4. Fig. 7 shows the specialized SDG that corresponds to the closure slice shown in Fig. 6. The two specialized PDGs for `p` in Fig. 7 correspond to the two specializations of `p` discussed in Ex. 3.3. The SDG in Fig. 7 corresponds to the program shown in Fig. 1(b). □

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. The one point that we will note here is how the issue of vertex ordering [Horwitz et al. 1989] is handled. Specialization slicing introduces new procedures, each of which corresponds to a unique procedure in the original program. When applied to a procedure q that is a specialization of original procedure Q, the pretty-printer makes use of the order in which q's program elements appear in Q.

Problem Statement (Part II): Soundness and Completeness. Suppose that for SDG S and slicing criterion C, a specialization slice of S with respect to C produces SDG R. (That is, R satisfies the conditions described in "Problem Statement (Part I).") Ideally, the unrolling of R should be identical to the closure slice with respect to C of the unrolling of S; that is, the unrolling of R should have two properties:

(1) It should contain *only* configurations that are in the closure slice with respect to C of the unrolling of S (*soundness*).

(2) It should contain *all* of the configurations in the closure slice with respect to C of the unrolling of S (*completeness*).

However, because the vertices and call-site labels in R and S are different, the above two properties do not hold (in the form stated above). The naming differences between R and S 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 be based on pure equality; instead, the comparison of configurations in the unrollings of R and S must account for the changes in the alphabet symbols.

Fortunately, each procedure introduced in R by specialization slicing corresponds to a unique procedure in the original program, so it is easy to identify each vertex or call-site in R as the specialization of some vertex or call-site in S. Thus, there is a mapping M_C that maps SDG vertices and call-sites in R one-to-one into the vertices and call-sites in S. M_C can be extended in the obvious way to map configurations in the unrolling of R to configurations in the unrolling of S. Using M_C, we can state the concepts of soundness and completeness formally as follows:

Definition 3.5. *Let A be a specialization-slicing algorithm, and consider an input pair consisting of SDG S and slicing criterion C, and the corresponding output SDG R and mapping M_C (as described above) computed by A. Let G_R be the unrolling of R.*

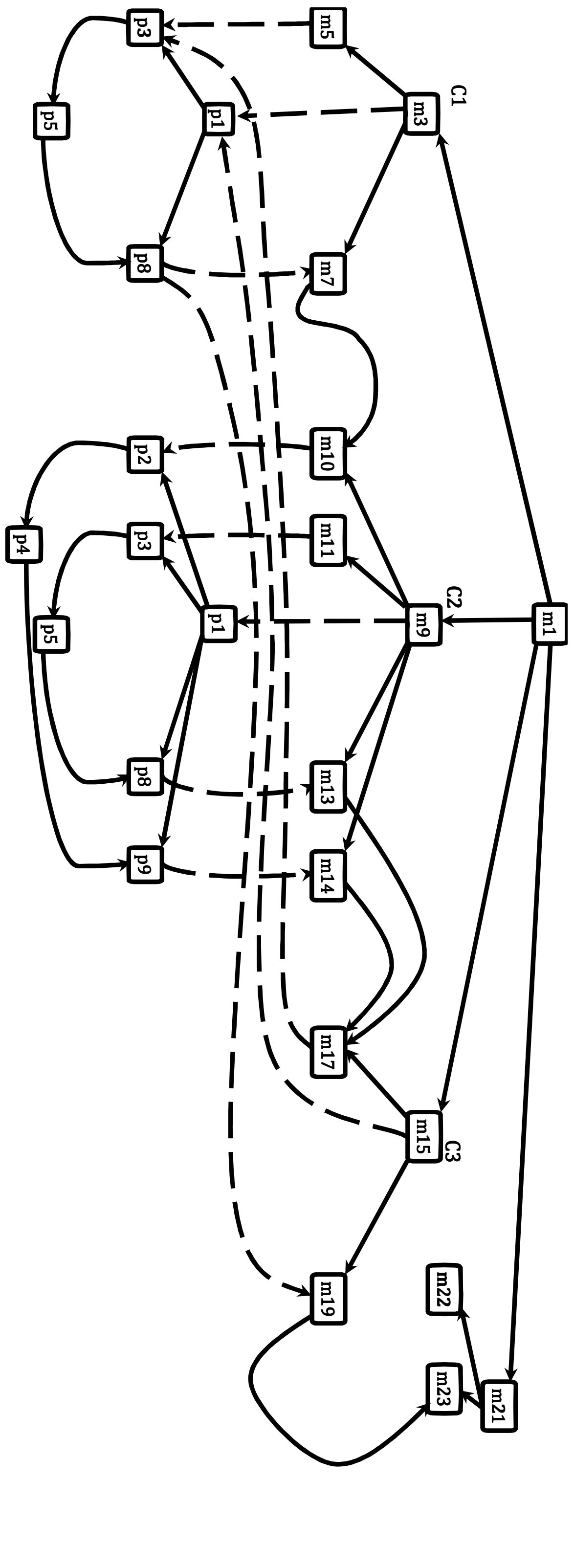


Fig. 7. The specialized SDG for the closure slice shown in Fig. 6.

(1) R is ***sound*** *if each $M_C(G_R)$ contains only configurations that are in the closure slice with respect to C of the unrolling of S.*

(2) R is ***complete*** *if each $M_C(G_R)$ contains all of the configurations in the closure slice with respect to C of the unrolling of S.*

Algorithm A is ***sound*** *(respectively,* ***complete****) if, for all slicing problems, A computes a sound (respectively, complete) slice.* □

The algorithm presented in this paper is both sound and complete (see Thm. 4.14). In contrast, both Weiser's algorithm [Weiser 1984] and the candidate algorithm discussed in §1 are complete, but can include extra program elements—and thus extra configurations in the unrolled slice—and hence are not sound. Binkley's algorithm for monovariant executable slicing [Binkley 1993], discussed in §6, is also complete but not sound.

Remark. Our terminology is inspired by the use of the terms in logic:

—A proof system S is *sound* when its set of inference rules prove only valid formulas with respect to its semantics. (However, S may not have a proof for every valid formula.)

—A proof system T is *complete* when every valid formula has a proof in T. (However, T's set of inference rules may also allow "proofs" to be given for formulas that are not valid.)

The analogy adopted in Defn. 3.5 is that (i) the specialization-slicing algorithm plays the role of the set of inference rules, and (ii) configurations that are in the closure slice with respect to C of the unrolling of S plays the role of the set of valid formulas.

Our terminology is also similar to the way "sound" and "complete" are used to describe the properties of bug-finding tools [Godefroid et al. 2005]: "sound" means that every bug reported by a tool is an actual bug, and "complete" means that every bug (possibly along with some false positives) will be reported by the tool.

However, the reader should be aware that such terminology is the opposite of another common usage of "sound" in the program-analysis community to mean "a conservative over-approximation." (With the latter definition of soundness, one would say that Weiser's and Binkley's algorithms generate a sound slice, but not necessarily a most-precise one.) □

Insight 3: Formulation as a Partitioning Problem. Because an unrolled SDG can be infinite—and even when not infinite, can be exponentially larger than the original SDG—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. 3.6. (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.)

The intuition behind Defn. 3.6 is as follows:

—For each configuration $(\!|v, u|\!)$ in the stack-configuration slice of the program, there needs to be some specialized procedure that can be called with the stack-configuration u. Each partition-element corresponds to a specialization of some procedure.

—The partition-elements should only include configurations that are in the stack-configuration slice.

—We can find the PDGs of the specialization slice given in Fig. 1(b) and Fig. 7 by partitioning the set given as Eqn. (2) as follows:

$$\left\{ \left\{ \begin{array}{l} (\!|m1,\epsilon|\!), (\!|m10,\epsilon|\!), (\!|m19,\epsilon|\!),\\ (\!|m3,\epsilon|\!), (\!|m11,\epsilon|\!), (\!|m21,\epsilon|\!),\\ (\!|m5,\epsilon|\!), (\!|m13,\epsilon|\!), (\!|m22,\epsilon|\!),\\ (\!|m7,\epsilon|\!), (\!|m14,\epsilon|\!), (\!|m23,\epsilon|\!)\\ (\!|m9,\epsilon|\!), (\!|m15,\epsilon|\!),\\ \qquad\quad (\!|m17,\epsilon|\!), \end{array} \right\}, \left\{ \begin{array}{l} (\!|p1,C1|\!),\\ (\!|p3,C1|\!),\\ (\!|p5,C1|\!),\\ (\!|p8,C1|\!),\\ (\!|p1,C3|\!),\\ (\!|p3,C3|\!),\\ (\!|p5,C3|\!),\\ (\!|p8,C3|\!) \end{array} \right\}, \left\{ \begin{array}{l} (\!|p1,C2|\!),\\ (\!|p2,C2|\!),\\ (\!|p3,C2|\!),\\ (\!|p4,C2|\!),\\ (\!|p5,C2|\!),\\ (\!|p8,C2|\!),\\ (\!|p9,C2|\!) \end{array} \right\} \right\} \qquad (4)$$

The configurations in the first partition-element correspond to the PDG for `main` in the specialized SDG. The configurations in the second partition-element consist of those from variants $\mathtt{p}_{C1}$ and $\mathtt{p}_{C3}$ of procedure `p`; they form a single partition-element that corresponds to the specialized version of `p` named `p_1` in Fig. 1(b). The configurations in the third partition-element consist of those in variant $\mathtt{p}_{C2}$, and correspond to the second specialized version of `p`, named `p_2` in Fig. 1(b).

—Configurations that are not in the stack-configuration slice, such as $(\!|m8,\epsilon|\!)$, are not part of any variant, and hence do not contribute vertices to the specialized SDG.

—Some elements of the original SDG, 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`.

Definition 3.6. *(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. □

Returning to Eqn. (4), the configurations in the second partition-element—consisting of those from variants $\mathtt{p}_{C1}$ and $\mathtt{p}_{C3}$—satisfy rules (1) and (3) of Defn. 3.6, and so form a single partition-element.

Defn. 3.6 implicitly defines a ***minimality condition*** for specialization slicing. As stated, because of the "iff" in item (3), Defn. 3.6 defines a *unique* partition. An alternative would have been to state item (3) as "... $B \subseteq E$ *implies* Elems(A) = Elems(B)." In that case, we would want the *coarsest* partition among all candidates, so that the specialized program consisted of as *few* specialized procedures as possible. By stating item (3) as "$\ldots B \subseteq E$ iff $\text{Elems}(A) = \text{Elems}(B)$," the unique partition specified by Defn. 3.6 is the coarsest of the (hypothetical) candidates.[9]

DEFINITION 3.7. *A specialization slice is **optimal** if it is sound, complete, and minimal. A specialization-slicing algorithm A is **optimal** if for all slicing problems, A computes an optimal slice.* □

Defn. 3.6 is a declarative specification of the partitioning problem, but does not provide a method to construct the desired finite partition. In §4, we show how to identify the desired partition by performing just a few simple automata-theoretic operations. 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. 4.15, the SDG obtained via our algorithm avoids the parameter-mismatch problem.

Discussion. In the partition given in Eqn. (4), the set of languages defined by $\{\text{Stacks}(E) \mid E \text{ a partition-element}\}$ partitions the set of stack-configurations as follows: $\{\{\epsilon\}, \{C1, C3\}, \{C2\}\}$. These correspond to a partition of the variants according to their stack-configurations, namely, $\{\{\mathtt{main}_\epsilon\}, \{\mathtt{p}_{C1}, \mathtt{p}_{C3}\}, \{\mathtt{p}_{C2}\}\}$. More generally, we have

OBSERVATION 3.8. *Each partition-element E in the partition defined in Defn. 3.6 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.* □

Thus, an alternative formulation of the search for the sets Specializations(P) could have been given as a (different) partitioning problem on the *variants* in the stack-configuration slice (or on their stack-configurations). However, the technique used in §4 to identify the partition manipulates descriptions of sets of *configurations*, not variants; consequently, Defn. 3.6 more closely matches the concepts needed to understand our presentation of the algorithm.

3.2 A Recursive Example

The partitioning problem in the example discussed in §3.1 is quite simple, because the program from Fig. 1(a) is non-recursive, and thus the unrolled SDG in Fig. 6 is finite. In contrast, for a program that uses recursion, the unrolled SDG is infinite. Moreover, for some slicing criteria of the unrolled SDG, the set of configurations that we need to partition can be of infinite cardinality. Fortunately, as illustrated by the example shown in Fig. 2, the partition that satisfies Defn. 3.6 is still finite.

As already mentioned in §1, one issue that can arise with recursion is that the pattern of recursion can change. For instance, as illustrated in Fig. 2, if the program uses *direct* recursion (`r` calls `r` calls ...), a specialization slice of the program

[9]Our notion of minimality should not be confused with Weiser's notion of a *statement-minimal* slice: a slice S of program P with respect to criterion C is statement-minimal if no other slice of P with respect to C has fewer statements than S. Weiser showed that the problem of finding statement-minimal slices is, in general, unsolvable [Weiser 1984, p. 353].

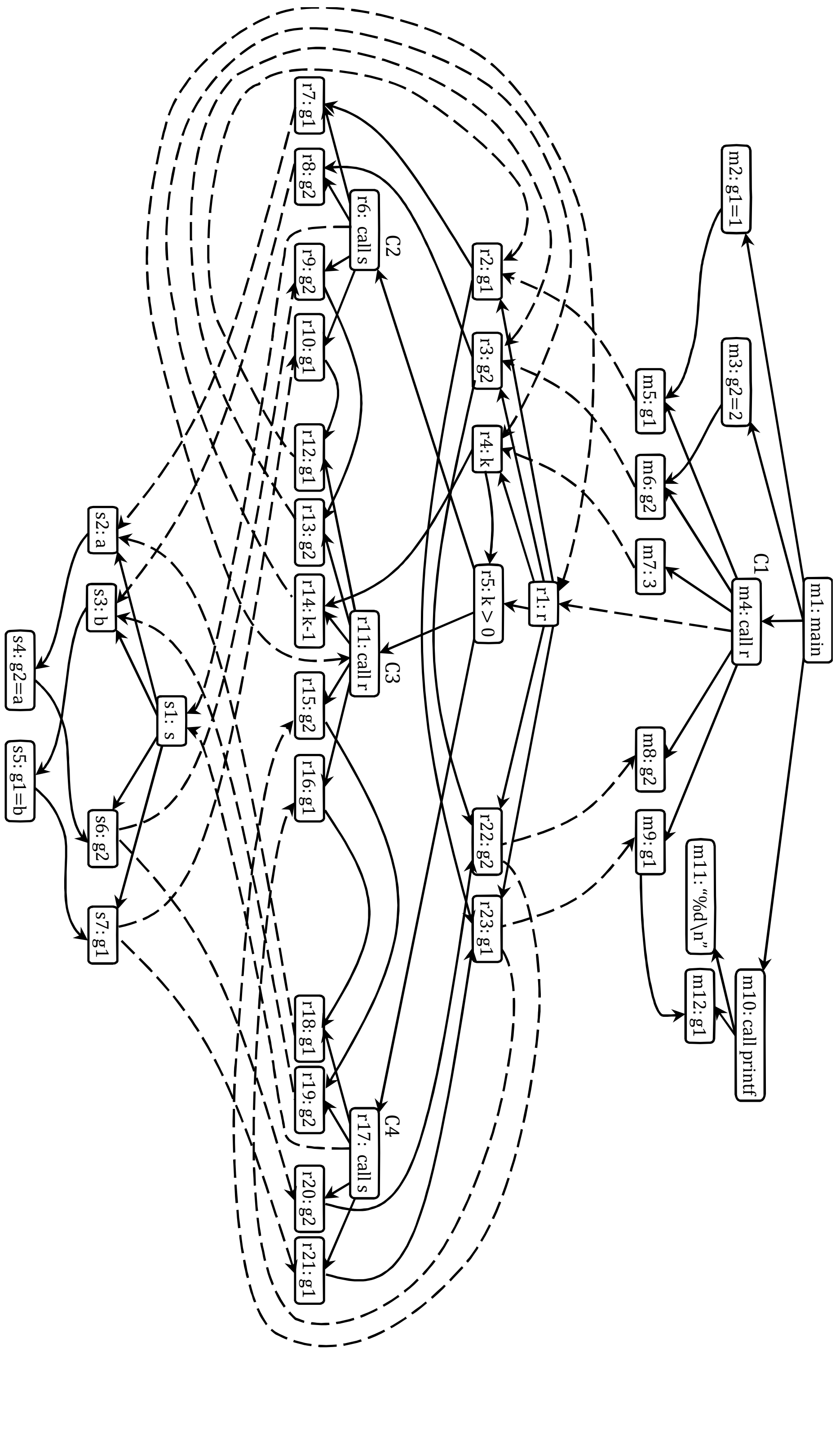


Fig. 8. The SDG of the recursive program shown in Fig. 2(a).

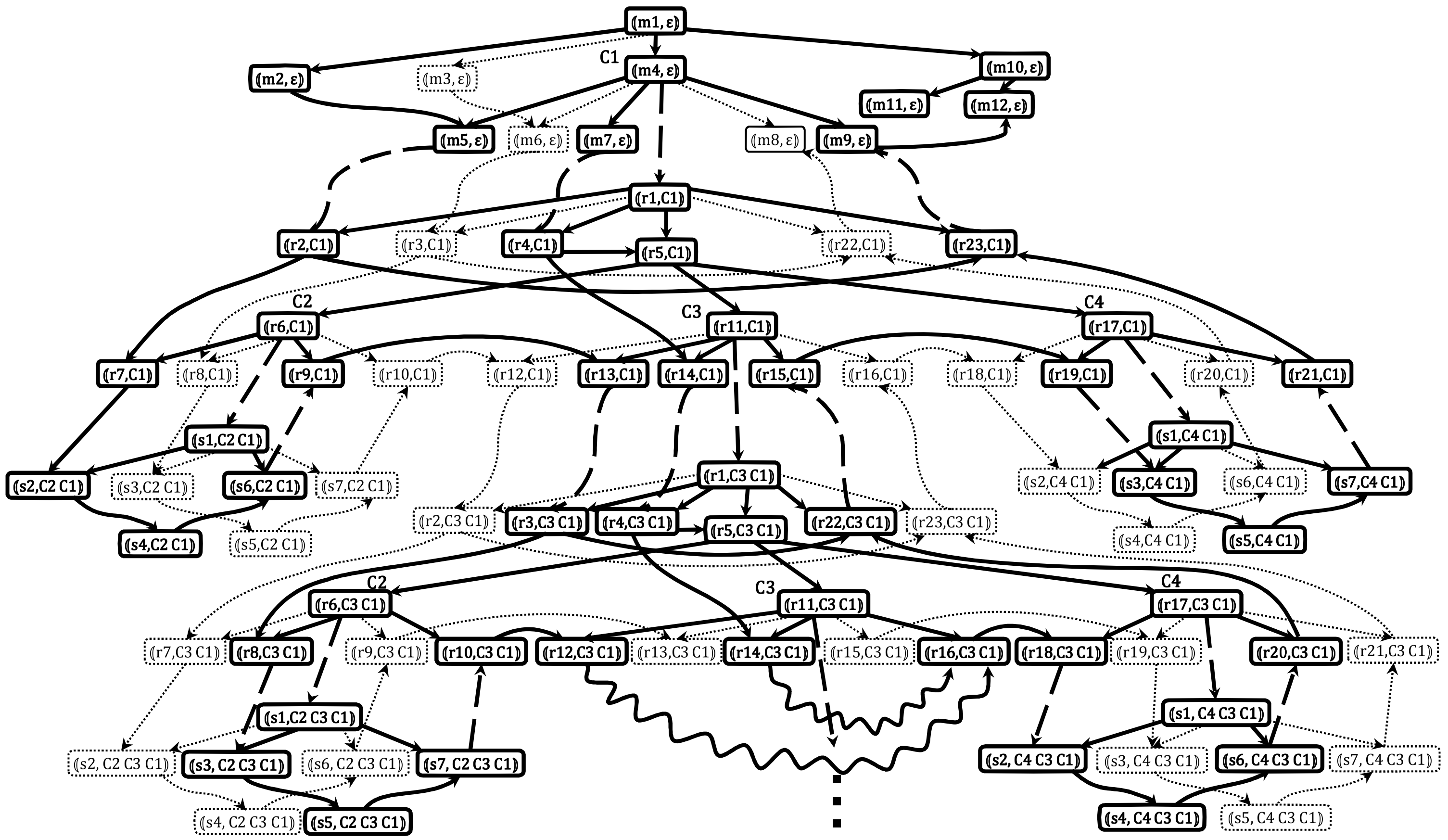


Fig. 9. The unrolled SDG of the recursive program shown in Fig. 2(a). The two edges $(\!|r12, C3\,C1|\!) \rightsquigarrow (\!|r16, C3\,C1|\!)$ and $(\!|r14, C3\,C1|\!) \rightsquigarrow (\!|r16, C3\,C1|\!)$ indicate transitive dependences in the unrolled SDG. The closure slice of the unrolled SDG with respect to configuration set $\{(\!|m10, \epsilon|\!), (\!|m11, \epsilon|\!), (\!|m12, \epsilon|\!)\}$ is shown with bold font and darker borders, lines, and dashed lines.

may need *mutual* recursion (`r_1` calls `r_2` calls `r_1` calls ...). Fortunately, the specialization-slicing algorithm that we give in §4 automatically identifies the correct procedure version to call, even when specialization slicing introduces mutual recursion among specialized procedures.

Consider the recursive program shown in Fig. 2(a) and the slice of the program with respect to line (28). The SDG of the program is shown in Fig. 8. The slicing criterion that corresponds to line (28) of Fig. 2(a) is the configuration set $\{(\!|m10, \epsilon|\!), (\!|m11, \epsilon|\!), (\!|m12, \epsilon|\!)\}$.

Fig. 9 shows the structure of the unrolled SDG for 2–3 levels of unrolling. As in Fig. 6, bold font and darker borders, lines, and dashed lines are used to indicate the vertices that are in the stack-configuration slice with respect to $\{(\!|m10, \epsilon|\!), (\!|m11, \epsilon|\!), (\!|m12, \epsilon|\!)\}$.

Fig. 9 includes two variants of procedure `r`, whose configurations have stack-configurations $(C1)$ and $(C3\,C1)$, respectively. For instance, the complete set of PDG vertices in the $(C1)$ variant 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 have stack-configurations 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. 2(b).

Similarly, all variants of `r` whose configurations have 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. 2(b). Note that specialized procedures `r_1` and `r_2` are mutually recursive.

The use of the regular languages $(C3\ C3)^*\ C1$ and $(C3\ C3)^*\ C3\ C1$ in the discussion above provides a clue as to how the specialization-slicing algorithm can represent finitely the infinite sets of configurations that it needs to manipulate to solve the partitioning problem. As described in §4, the algorithm actually makes use of automata, rather than regular expressions, to represent such languages.

4. AN AUTOMATON-BASED ALGORITHM FOR SPECIALIZATION SLICING

The specialization conditions given in Eqn. (3) and Defn. 3.6 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 take advantage of the fact that the model-checking community has developed powerful and concise machinery for working with infinite-state transition systems [Bouajjani et al. 1997; Finkel et al. 1997]. In particular, the encoding of an SDG as a PDS allows us to represent finitely the infinite sets of configurations that are part of Defn. 3.6. Moreover, with this powerful machinery at our disposal, we can obtain an algorithm for specialization slicing that involves just a few simple automata-theoretic operations.

The algorithm for specialization slicing involves the following five steps:

(1) encode the program's SDG as a PDS (§2.3),
(2) perform stack-configuration slicing by applying the PDS *Prestar* algorithm (§4.1),

(3) carry out several transformations of the result of *Prestar* to construct an automaton of a special form (§4.2),

(4) create the specialized SDG from the automaton created in the previous step (§4.3), and

(5) pretty-print the specialized SDG as source-code text.

This section presents the details of items (2), (3), and (4). As mentioned earlier, item (5) is straightforward, but lies outside the scope of the paper.

The theorems that demonstrate the correctness of the algorithm are stated in §4.4. The proofs are given in Appendix A. Bounds on the time and space used during the steps of the algorithm are presented in §5.

4.1 Symbolic Stack-Configuration Slicing

This section reviews the symbolic techniques for working with PDSs upon which our specialization-slicing algorithm is based.

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 §3.1. Moreover, the symbolic method 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. More precisely, one finite automaton will be used to specify the set of configurations that make up the slicing criterion; the *pre** algorithm returns another finite automaton whose language specifies the set of configurations in the stack-configuration slice.

Definition 4.1. *If $\mathcal{P} = (P, \Gamma, \Delta)$ is a PDS, a **$\mathcal{P}$-automaton** is a finite automaton $(Q, \Gamma, \rightarrow, P, F)$, where $Q \supseteq P$ is a finite set of states, $\rightarrow \subseteq Q \times \Gamma \times Q$ is the transition relation, P is the set of initial states, and F is the set of final states.*

*The $\rightarrow$ 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.* □

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 [Bouajjani et al. 1997; Finkel et al. 1997]. 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., $\mathcal{A}_{post^*}$ and $\mathcal{A}_{pre^*}$ are initially empty; transitions are added to them according to an appropriate augmentation rule until no more can be added.

The saturation procedure for *Prestar* can be stated as follows:

Definition 4.2. *(Algorithm Prestar) $\mathcal{A}_{pre^*}$ is constructed from query automa-*

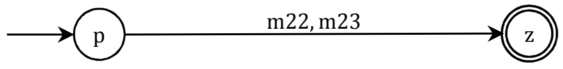


Fig. 10. The query automaton, which accepts the configurations $(p, m22)$ and $(p, m23)$.

ton $\mathcal{A}$ by augmenting $\mathcal{A}_{pre^*}$ according to the following rules, until $\mathcal{A}_{pre^*}$ is saturated:

$$\frac{p \xrightarrow{\gamma} q \in \mathcal{A}}{p \xrightarrow{\gamma} q \in \mathcal{A}_{pre^*}} \textsc{Pre1} \tag{5}$$

$$\frac{\langle p, \gamma\rangle \hookrightarrow \langle p', w\rangle \in \Delta \qquad\qquad p' \xrightarrow{w}{}^{*} q \in \mathcal{A}_{pre^*}}{p \xrightarrow{\gamma} q \in \mathcal{A}_{pre^*}} \textsc{Pre2} \tag{6}$$

Esparza et al. [2000] present an efficient implementation of Prestar, which uses $O(|Q|^2|\Delta|)$ time and $O(|Q|\,|\Delta| + |{\to_{\mathcal{A}}}|)$ space, where $\to_{\mathcal{A}}$ denotes the transition relation of $\mathcal{A}$. □

The saturation procedure for *Poststar* can be stated as follows:

Definition 4.3. *(Algorithm Poststar) $\mathcal{A}_{post^*}$ is constructed from query automaton $\mathcal{A}$ 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 $\overset{\gamma}{\rightsquigarrow}$ denotes the relation $(\xrightarrow{\epsilon})^\star \xrightarrow{\gamma} (\xrightarrow{\epsilon})^\star$.)*

$$\frac{p \xrightarrow{\gamma} q \in \mathcal{A}}{p \xrightarrow{\gamma} q \in \mathcal{A}_{post^*}} \textsc{Post1}$$

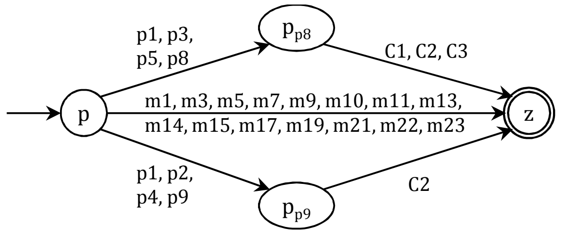
$$\frac{\langle p, \gamma\rangle \hookrightarrow \langle p', \epsilon\rangle \in \Delta \qquad\qquad p \overset{\gamma}{\rightsquigarrow} q \in \mathcal{A}_{post^*}}{p' \xrightarrow{\epsilon} q \in \mathcal{A}_{post^*}} \textsc{Post2}$$

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

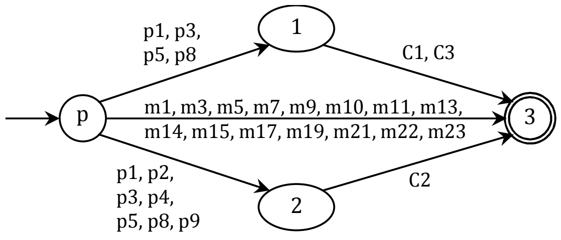
$$\frac{\langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\gamma''\rangle \in \Delta \qquad\qquad p \overset{\gamma}{\rightsquigarrow} q \in \mathcal{A}_{post^*}}{p' \xrightarrow{\gamma'} p'_{\gamma'} \in \mathcal{A}_{post^*} \qquad p'_{\gamma'} \xrightarrow{\gamma''} q \in \mathcal{A}_{post^*}} \textsc{Post4}$$

$\mathcal{A}_{post^*}$ can be constructed in time and space $O(n_P n_\Delta(n_1 + n_2) + n_P n_0)$, where $n_P = |P|$, $n_\Delta = |\Delta|$, $n_Q = |Q|$, $n_0 = |{\to_{\mathcal{A}}}|$, $n_1 = |Q - P|$, 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 [Schwoon 2002]. □

Example 4.4. Consider how the *Prestar* algorithm from Defn. 4.2 identifies the configurations in the stack-configuration slice shown in Fig. 6. The SDG from Fig. 4 is encoded as the PDS whose rules are given in Tab. I. The query automaton $\mathcal{A}$

(a)


p1, p3,
p5, p8
p1, p2,
p4, p9
m1, m3, m5, m7, m9, m10, m11, m13,
m14, m15, m17, m19, m21, m22, m23
C1, C2, C3
C2
p
p_{p8}
p_{p9}
z

(b)


p1, p3,
p5, p8
p1, p2,
p3, p4,
p5, p8, p9
m1, m3, m5, m7, m9, m10, m11, m13,
m14, m15, m17, m19, m21, m22, m23
C1, C3
C2
p
1
2
3

Fig. 11. (a) An automaton that accepts the configurations of the stack-configuration slice of Fig. 1(a) with respect to line (17). (b) A minimal reverse-deterministic (MRD) automaton for the same language. As discussed in Ex. 4.6, one can immediately read off the answer to the configuration-partitioning problem (Defn. 3.6)—and hence to the specialization-slicing problem—from the MRD automaton: the outgoing transitions from initial state p capture exactly the desired partition—see Eqn. (4).

that specifies the slicing criterion has transitions from initial state p to final state z on the symbols $m22$ and $m23$—see Fig. 10.

Technically, a $\mathcal{P}$-automaton has an initial state for each control location of PDS $\mathcal{P}$ (Defn. 4.1). 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. 10). (Also, because $\mathcal{P}$-automata have just the one initial state p, we will typically ignore p in the accepted words, and concentrate on the (SDG) configuration that a $\mathcal{P}$-configuration models. For instance, technically $\mathcal{A}$ in Fig. 10 accepts the language of $\mathcal{P}$-configurations $\{(p, m22), (p, m23)\}$, which model the following configurations of the unrolled SDG: $\{(\!|m22, \epsilon|\!), (\!|m23, \epsilon|\!)\}$.)

Prestar produces the automaton $\mathcal{A}'$ shown in Fig. 11(a). For instance, according to Eqn. (6), the transition $(p, m19, z)$ can be added to $\mathcal{A}'$ using PDS rule 25 and transition $(p, m23, z)$; the transition $(p_{p8}, C3, z)$ can be added using PDS rule 58 and transition $(p, m19, z)$; and so on. Each new state p_{fo} added to $\mathcal{A}'$ during *Prestar*—in this example, p_{p8} and p_{p9}—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.

The significance of last two rule-schemas in Fig. 5, $\langle p, fo\rangle \hookrightarrow \langle p_{fo}, \epsilon\rangle$ 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, q), then $\mathcal{A}'$ will have the transitions (p, fo, p_{fo}) and (p_{fo}, C, q). For instance, in our example, let

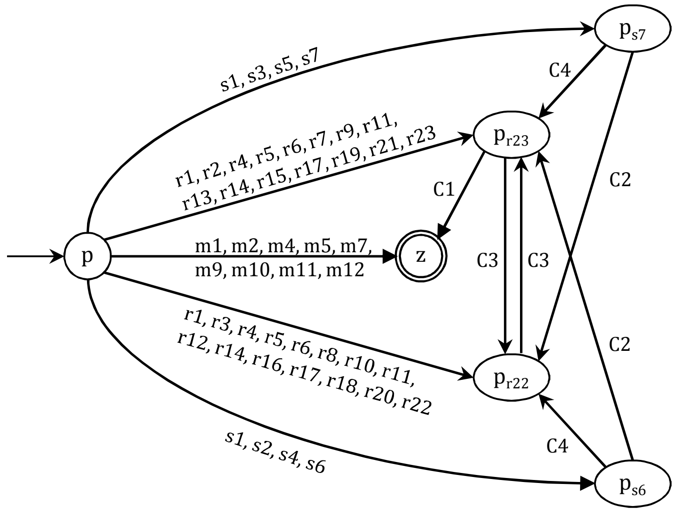

p
z
p_{s7}
p_{r23}
p_{r22}
p_{s6}
s1, s3, s5, s7
r1, r2, r4, r5, r6, r7, r9, r11,
r13, r14, r15, r17, r19, r21, r23
m1, m2, m4, m5, m7,
m9, m10, m11, m12
r1, r3, r4, r5, r6, r8, r10, r11,
r12, r14, r16, r17, r18, r20, r22
s1, s2, s4, s6
C4
C1
C3
C3
C2
C2
C4

Fig. 12. An automaton that accepts the configurations of the stack-configuration slice of Fig. 2(a) from line (28).

fo be $p8$, *ao* be $m19$, and q be z. Because (i) the SDG has a parameter-out edge $p8 \to m19$ at call-site $C3$, and (ii) Fig. 11(a) has a transition $(p, m19, z)$, Fig. 11(a) also has the transitions $(p, p8, p_{p8})$ and $(p_{p8}, C3, z)$.

After saturation has quiesced, 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 $\{(\!| m22, \epsilon |\!), (\!| m23, \epsilon |\!)\}$. □

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

Example 4.5. Consider the example discussed in §3.2, where we want to create a specialization slice with respect to line (28) of the program shown in Fig. 2. We first encode the program's SDG (Fig. 8) 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. 12.

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 from *Prestar* (Fig. 12) 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.

Note the order of the stack symbols in such words: call-site $C1$ of `main` appears *last* (i.e., as the rightmost symbol). □

4.2 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 on the saturated automaton $\mathcal{A}'$. Each such operation manipulates indirectly the possibly infinite sets of configurations that are part of Defn. 3.6.

EXAMPLE 4.6. As discussed in Ex. 4.4, given Tab. I and Fig. 10 as input, the *Prestar* saturation technique given in Defn. 4.2 constructs Fig. 11(a), which accepts exactly the configurations in the stack-configuration slice shown in Fig. 6. However, Fig. 11(a) does not immediately provide a solution to the configuration-partitioning problem (Defn. 3.6) in the sense that the three sets of the partition given in Eqn. (4) are not immediately apparent from Fig. 11(a).

Fortunately, Fig. 11(a) is not the only automaton that accepts the language of configurations in the stack-configuration slice. In particular, the automaton shown in Fig. 11(b) also accepts the language of configurations in the stack-configuration slice. Moreover, from Fig. 11(b) we can immediately read off the main aspects of the desired specialization-slice answer.

—The label sets on the three transitions emanating from the initial state p represent the three sets of the partition given in Eqn. (4), and thus correspond to the program elements of the specialized procedures `p_1`, `p_2`, and `main` of Fig. 1(b).

—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(b).

—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.)

□

We now seek

(1) a condition that characterizes the essential property of Fig. 11(b), and

(2) an algorithm that lets us construct Fig. 11(b) from Fig. 11(a).

OBSERVATION 4.7. *The property possessed by Fig. 11(b) is that it is* ***minimal reverse-deterministic*** *(MRD)—i.e., it is a minimal deterministic FSA when considered as an automaton that accepts reversed strings via a backward traversal along transitions, starting from the accepting state.* □

Because we are interested in partitioning the language of represented configurations, we will exploit the fact that each state of an FSA can be thought of as defining two languages. For instance, consider the FSA $\mathcal{A}$ depicted in Fig. 13(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.

To see why the MRD property is the one we seek, suppose that $\mathcal{A}$ in Fig. 13(a) is deterministic. In this case, the set of languages $\{P(q) \mid q \text{ a state of } \mathcal{A}\}$ *partition*

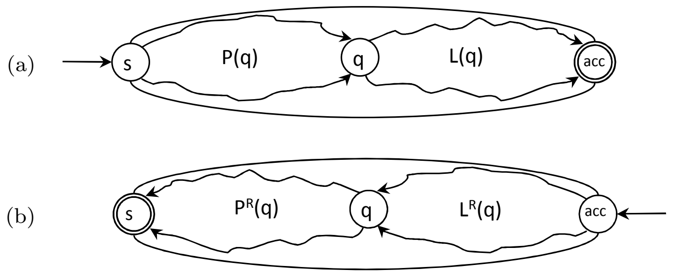


Fig. 13. (a) Depiction of the prefix and suffix languages associated with state q in automaton $\mathcal{A}$. (b) Reversal of (a) (denoted by $R(\mathcal{A})$). $P^R(q)$ and $L^R(q)$ denote the respective languages of reversed strings.

the prefix closure $L^{\leftarrow}(\mathcal{A})$ of $L(\mathcal{A})$.[10] 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. 3.8, each specialized procedure is associated with a partition-element of a partition of the *stack-configurations*. Recall that in a configuration $(\!| v, u |\!)$, u represents the stack of pending calls. If we determinized Fig. 11(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 u part of a configuration. Moreover, because Fig. 11(a) recognizes a stack-configuration from top-of-stack to bottom-of-stack (i.e., `main`), its $P(q)$ languages recognize $(\!|\text{PDG-vertex}, \text{partial-stack}|\!)$ 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. 13(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. 4.13, the automaton obtained in this manner solves the configuration-partitioning problem (Defn. 3.6).

Example 4.8. Consider again the recursive example discussed in §3.2. Fig. 12 shows an MRD automaton for the stack-configuration slice of Fig. 8 with respect to $\{(\!| m10, \epsilon |\!), (\!| m11, \epsilon |\!), (\!| m12, \epsilon |\!)\}$. A comparison of Fig. 12 with Fig. 2(b) shows that the sets that label the five transitions that emanate from initial state `p` correspond to the five procedures of Fig. 2(b), namely, `s_1`, `s_2`, `r_1`, `r_2`, and `main`.

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

In particular, procedure `s` is specialized into two versions (`s_1` with parameter `b` and `s_2` with parameter `a`). Procedure `r`, which has a single parameter in the original program and still has a single parameter in the output slice, is also specialized into two versions. These specializations are read off of the automaton shown in Fig. 12 as follows:

—transitions of the form $(p, *, p_{s7})$ correspond to elements of procedure `s_1`

—transitions of the form $(p, *, p_{s6})$ correspond to elements of procedure `s_2`

—transitions of the form $(p, *, p_{r23})$ correspond to elements of procedure `r_1`

—transitions of the form $(p, *, p_{r22})$ correspond to elements of procedure `r_2`

The transitions among states z, p_{r23}, p_{r22}, p_{s7}, and p_{s6} correspond to the call graph of Fig. 2(b). In particular, the languages $L(z)$, $L(p_{r23})$, $L(p_{r22})$, $L(p_{s7})$, and $L(p_{s6})$ in the automaton shown in Fig. 12 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. 3.6) for the stack-configuration slice of Fig. 8 with respect to $\{(\!| m10, \epsilon |\!), (\!| m11, \epsilon |\!), (\!| m12, \epsilon |\!)\}$.

Because of these properties, the specialization-slicing algorithm automatically identifies the correct procedure version to call, even when specialization slicing introduces mutual recursion among specialized procedures. □

4.3 The Algorithm for Specialization Slicing

The SDG for the specialization 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 (i.e., the configurations of the closure slice of the unrolled SDG). In lines 4–8, Alg. 1 applies automaton operations to $A1$—reverse, determinize, minimize, reverse, and removeEpsilonTransitions—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 transform* $A6$ *into the minimal reverse-deterministic FSA for the language* $L(A1)$.

Remark. The step "$A6 =$ removeEpsilonTransitions$(A5)$" in line 8 is only needed for certain implementations of reverse. 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)).

As shown in the proof of Thm. 4.12 (see Appendix A), the minimized automaton $A4$ has a single accepting state. Moreover, because $A4$ is deterministic, it has no ϵ-transitions. Consequently, the statement "$A5 =$ reverse$(A4)$" could 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, $A5$ would have no ϵ-

```
Input: SDG $S$ and slicing criterion $C$
Output: An SDG $R$ for the specialization 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. 2.6
2  $A0$ = a $\mathcal{P}_S$-automaton that accepts $C$
3  $A1 = Prestar[\mathcal{P}_S](A0)$
4  $A2$ = reverse($A1$)
5  $A3$ = determinize($A2$)
6  $A4$ = minimize($A3$)
7  $A5$ = reverse($A4$)
8  $A6$ = removeEpsilonTransitions($A5$)
// Read out SDG $R$ from $A6$ ----------------------------------------------
9  $R$ = the empty SDG
10 $q_0$ = InitialState($A6$)
11 $StateToPDGMap$ = empty map
// Identify PDGs ------------------------------------------------------------
12 foreach $q \in (States(A6) - \{q_0\})$ do
13     $V = \{v \mid (q_0, v, q) \in \text{Transitions}(A6)\}$
14     $G_{orig}$ = the PDG of $S$ that contains the vertices in $V$
15     $G_V$ = a new PDG consisting of copies of $V$, plus copies of the edges from $G_{orig}$ induced by $V$
16     Add PDG $G_V$ to SDG $R$
17     $StateToPDGMap = StateToPDGMap[q \mapsto G_V]$
18 end
// Connect PDGs -------------------------------------------------------------
19 foreach transition $(q_1, C, q_2) \in Transitions(A6)$ such that $C$ is a call-site label do
20     Let $C'$ be the call-site of PDG $StateToPDGMap(q_2)$ that corresponds to $C$
21     Add to $R$ a call edge from the call vertex at $C'$ to the entry vertex of $StateToPDGMap(q_1)$
22     Add to $R$ a parameter-in edge from each actual-in vertex at $C'$ to the corresponding formal-in
       vertex of $StateToPDGMap(q_1)$
23     Add to $R$ a parameter-out edge from each formal-out vertex of $StateToPDGMap(q_1)$ to the
       corresponding actual-out vertex at $C'$
24 end
25 return $R$
```

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

transitions, and thus removeEpsilonTransitions would have no effect (i.e., $A6 = A5$). Because $A4$ is a minimal deterministic FSA, $A5$ and $A6$ would both be MRD.

In our implementation, lines 4–8 are implemented with OpenFST FSAs [OpenFST 2012], and the reverse operation in OpenFST introduces a dummy initial state with an ϵ-transition. Alg. 1 follows our implementation, which creates MRD automaton $A6$ by calling removeEpsilonTransitions in line 8 to remove the single, initial ϵ-transition from $A5$. □

Constructing the Specialized SDG. 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.

Line 15 uses the edge-induction operation defined as follows:

DEFINITION 4.9. *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': $\{s \rightarrow t \in E \mid s, t \in V'\}$.* □

The read-out method to construct the specialized SDG relies on the special nature of automaton $A6$.

—Each word in $L(A6)$ has the form (vertex-symbol call-site*). A word $(v\, u)$ denotes the configuration $(\!|v, u|\!)$.

—$A6$ is a minimal reverse-deterministic (MRD) automaton.

—Each non-initial state q represents a set of variants of some PDG G_{orig} (for some procedure P). More precisely, q represents the set of all u-variants for which $u \in L(q)$. Consequently, each non-initial state q in $A6$ represents a specialized PDG, and the transitions between non-initial states of $A6$ tell us about the interprocedural edges in the answer SDG.

—Let V be the set of vertex symbols on transitions from the initial state of $A6$ to q. For each variant W associated with q, $\text{Elems}(W) = V$. Consequently, the transitions from the initial state to q say what vertices populate the specialized PDG for q.

Lines 12–18 construct the specialized PDGs as follows: 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 for each 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 4.10. Consider how Alg. 1 works when S is the SDG shown in Fig. 4 and $C = \{(\!|m22, \epsilon|\!), (\!|m23, \epsilon|\!)\}$. The rules of PDS $\mathcal{P}_S$ are given in Tab. I. Automaton $A0$, which accepts the language C, is shown in Fig. 10. Automaton $A1 = \mathit{Prestar}[\mathcal{P}_S](A0)$ is shown in Fig. 11(a). Automaton $A6$ is shown in Fig. 11(b); note that $A6$ is MRD.

The second half of Ex. 4.6 already hinted at how the SDG shown in Fig. 7 is constructed by lines 9–24 of Alg. 1. In Fig. 11(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 Fig. 1(b), the corresponding call site is the call to procedure `p_1` on line (14) of `main`.)

The SDG formed from the resulting specialized PDGs is the one shown in Fig. 7. □

Example 4.11. Consider the example in §3.2, where we want to create a specialization slice with respect to line (28) of the recursive program shown in Fig. 2. Automaton $A6$ is shown in Fig. 12.[11] 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 z, p_{r23}, p_{r22}, p_{s7}, and p_{s6}.

The program for the specialized SDG is shown in Fig. 2(b). Note how the transitions among states z, p_{r23}, p_{r22}, p_{s7}, and p_{s6} correspond to the call graph of Fig. 2(b). □

4.4 Correctness Issues

The correctness of the specialization-slicing algorithm is established via the propositions listed below. Their proofs can be found in Appendix A.

THEOREM 4.12. *Automaton $A6$ created in line 8 of Alg. 1 is a minimal reverse-deterministic automaton.* □

THEOREM 4.13. *A solution to the configuration-partitioning problem (Defn. 3.6) is encoded in the structure of automaton $A6$ created in line 8 of Alg. 1.* □

THEOREM 4.14. *Alg. 1 is a sound and complete algorithm for stack-configuration slicing.* □

COROLLARY 4.15. *Let R be the SDG created via Alg. 1 for SDG S and slicing criterion C. R has no parameter mismatches.* □

5. COST OF ALG. 1

In this section, we establish that output slices created by Alg. 1 are optimal, and characterize the running time and space used by the algorithm in terms of the size of the original SDG. (Recall that the size of the SDG is polynomial in various parameters that characterize the size of the original program [Horwitz et al. 1990, §5.1].)

5.1 Minimality and Optimality

As discussed in §3.1, the configuration-partitioning problem (Defn. 3.6) implicitly defines a notion of minimality for specialization slicing. By Thm. 4.13, a solution to

[11]For this example, Fig. 12 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. 12 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 $p8$ in Fig. 6. There are three occurrences of $p8$ in different variants: $(p8, C1)$, $(p8, C2)$, and $(p8, C3)$. 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 after applying *Prestar* to query automaton Fig. 10 is the automaton shown in Fig. 11(a), which has the transition $(p, p8, p_{p8})$. The MRD automaton $A6$ obtained after performing lines 4–8 of Alg. 1 is the different automaton shown in Fig. 11(b). It has two transitions $(p, p8, 1)$ and $(p, p8, 2)$, which correspond to the two occurrences of $p8$ in Fig. 7.

the configuration-partitioning problem is encoded in the structure of automaton $A6$ from Alg. 1, which, by Thm. 4.12, is a minimal reverse-deterministic automaton. 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 $A6$. Consequently, SDG R is a minimal specialization slice.

Corollary 5.1. *Alg. 1 is an optimal specialization-slicing algorithm in the sense of Defn. 3.7: i.e., for every specialization-slicing problem, Alg. 1 creates a minimal, sound and complete output slice.* □

Proof. By Thm. 4.14, Alg. 1 is a sound and complete algorithm for stack-configuration slicing. By the argument made above, the output slice created by Alg. 1 is minimal. □

5.2 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. 4.2, *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 $O(n \log n)$ [Hopcroft 1971]. 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 the output SDG R. Thus, the running time of Alg. 1 is output-sensitive [Wikipedia: Output-Sensitive Algorithm 2014].

Our experiments indicate that for the automata that arise from *Prestar*, the size of automaton $A3$ created by determinize is significantly *smaller*—by 4.4%–34%—than the size of $A2$, given as input to determinize.

5.3 Space

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

—As mentioned in Defn. 4.2, the space for *Prestar* is bounded by $O(|Q|\,|\Delta|+|\rightarrow_{\mathcal{A}}|)$, where $|\rightarrow_{\mathcal{A}}|$ is the size of the transition relation for the automaton $\mathcal{A}$ for the slicing criterion.

—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 $\mathcal{A}_C$.[12]

(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, exponential explosion is achievable, as demonstrated by the family of (non-recursive) programs whose k^{th} member is presented schematically in Fig. 14. The k^{th} member of the family consists of $k+1$ procedures: `P0`, `P1`, …, `Pk`. Each of the k procedures `P1`, …, `Pk` has an if-then-else statement with a procedure call in each branch—in particular, procedure `Pi` has two calls to `P{i-1}`. After the first call to `P{i-1}`, `Pi` has an assignment `gi = 0` (see line (25)). This assignment breaks the dependence between the actual-out for `gi` at the call to `P{i-1}` on line (24) and `Pi`'s formal-out for `gi`. In contrast, the SDG has a dependence edge from the actual-out for `gi` at the call to `P{i-1}` on line (27) to `Pi`'s formal-out for `gi`.

As we progress down from `Pk` to `P0`, the different dependence patterns at the pairs of call-sites at each level cause the specialization slice with respect to the expression "`g1 + ... + gk`" on line (43) to generate a slice of procedure `P0` for each subset of the formal-outs for `g1`, …, `gk`. Because `P0` "sliced" with respect to the empty set of formal-outs consists of the empty set of vertices, it does not contribute a variant of `P0` to the pretty-printed result; consequently, the total number of variants of `P0` created by specialization slicing is $2^k - 1$.

Fig. 15 shows how the slice of the unrolled SDG for the 2^{nd} representative of the family of programs from Fig. 14 has four different combinations of formal-outs in the four different instances of `P0`.

Explosion in Practice. As discussed in more detail in §9, 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 (90.8%) had just a single version (see Fig. 20). Moreover, worst-case exponential behavior of operations

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

```
unsigned int g1, ..., gk;

void P0() {
  unsigned int t1, ... tk;
  t1 = g1; ... tk = gk;  /* Use all globals */
  g1 = t1; ... gk = tk;  /* Kill all globals */
  return;
}

void P1() {
  int v;
  v = scanf("%d\n", &v);
  if (v > 0) {
    P0();
    g1 = 0;
  }
  else P0();
}
...
void Pi() {
  int v;
  v = scanf("%d\n", &v);
  if (v > 0) {
    P{i-1}();
    gi = 0;
  }
  else P{i-1}();
}
...
void Pk() {
  int v;
  v = scanf("%d\n", &v);
  if (v > 0) {
    P{k-1}();
    gk = 0;
  }
  else P{k-1}();
}

int main() {
  g1 = 1; ... gk = k;
  Pk();
  printf("%d\n", g1 + ... + gk); /* SLICE HERE */
  return 0;
}
```

Fig. 14. The k^{th} representative of a family of programs for which a specialization slice is exponentially larger than the program.

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 Alg. 1 are bounded by the sum of

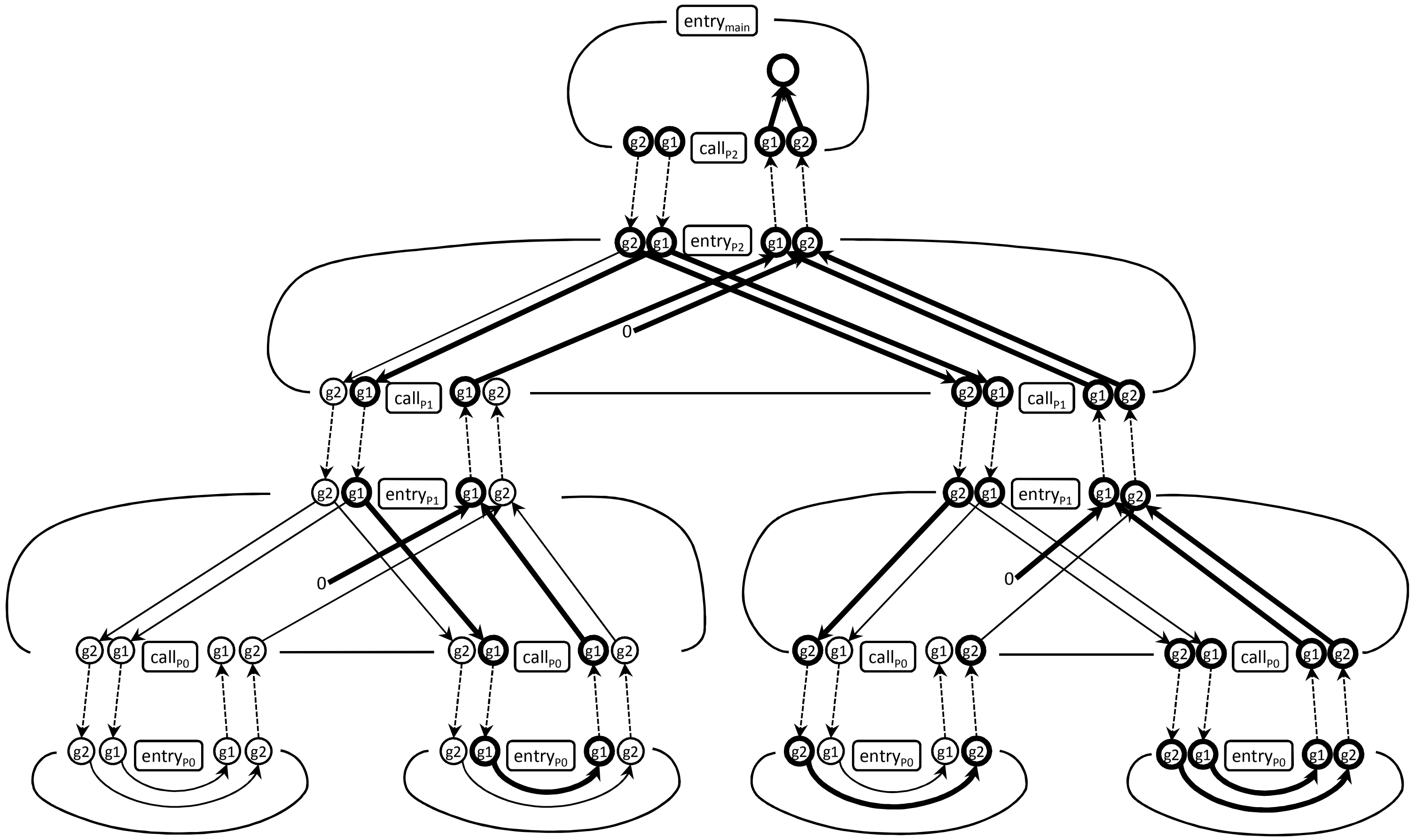


Fig. 15. Closure slice of the unrolled SDG of the 2^{nd} representative of the family of programs from Fig. 14, showing how the slice has four different combinations of formal-outs in procedure P0. (Only dependence edges that connect formal-ins, formal-outs, actual-ins, and actual-outs are shown.)

(a) $\boxed{\text{Backward closure slice}}$ w.r.t. `printf`'s parameters on line (17)		(b) Polyvariant slice with two versions of `p`	(c) Monovariant slice with matched actuals
(1)	$\boxed{\texttt{int g1, g2, g3;}}$	`int g1, g2;`	`int g1, g2;`
(2)			
(3)	$\boxed{\texttt{void p(int a, int b}}$`, int c`$\boxed{\texttt{) \{}}$	`void p_1(int b) {`	`void p(int a, int b) {`
(4)	$\boxed{\texttt{g1 = a;}}$	`g2 = b;`	`g1 = a;`
(5)	$\boxed{\texttt{g2 = b;}}$	`}`	`g2 = b;`
(6)	`g3 = c;`		
(7)	$\boxed{\texttt{\}}}$	`void p_2(int a, int b) {`	`}`
(8)		`g1 = a;`	
(9)		`g2 = b;`	
(10)		`}`	
(11)			
(12)	$\boxed{\texttt{int main() \{}}$	`int main() {`	`int main() {`
(13)	`g2 = 100;`		`g2 = 100;`
(14)	$\boxed{\texttt{p(}}$`g2, `$\boxed{\texttt{2}}$`, 4`$\boxed{\texttt{);}}$	`p_1(2);`	`p(g2, 2);`
(15)	$\boxed{\texttt{p(g2, 3}}$`, 5`$\boxed{\texttt{);}}$	`p_2(g2, 3);`	`p(g2, 3);`
(16)	$\boxed{\texttt{p(}}$`4, `$\boxed{\texttt{g1+g2}}$`, 6`$\boxed{\texttt{);}}$	`p_1(g1+g2);`	`p(4, g1+g2);`
(17)	$\boxed{\texttt{printf("\%d", g2);}}$	`printf("%d", g2);`	`printf("%d", g2);`
(18)	`}`	`}`	`}`

Fig. 16. (a) Variant of the program from Fig. 1, and (in boxes) the elements of the closure slice with respect to the actual parameters of the call to `printf` on line (17). (b) Polyvariant executable slice with respect to the same slicing criterion. (c) Monovariant executable slice created by Binkley's algorithm.

two terms: one is polynomial in the size of the input program; the other is linear in the size of the output slice.

6. EXECUTABLE SLICING

As mentioned in §2.2, the issue that prevents closure slices from being executable is the *parameter-mismatch problem* ([Horwitz et al. 1990, §1] and [Binkley 1993]). A closure slice can have multiple calls to the same procedure, with different subsets of actual parameters at different call-sites. However, the slice contains the union of the corresponding formal-parameter sets, which can cause a mismatch between the actual parameters at a call-site and the procedure's formal parameters (e.g., compare lines (14) and (16) of Fig. 16(a) with line (3)). For most programming languages, a program with such a mismatch would trigger a compile-time error.

In contrast, the output slices created via Alg. 1 never contain such parameter mismatches (Cor. 4.15), and thus Alg. 1 represents a new approach to *executable slicing*; namely, it provides an algorithm for *polyvariant executable slicing.*[13] The algorithm for executable slicing due to Binkley [1993] is an algorithm for *monovariant executable slicing.*

An example that illustrates the difference between a closure slice and the two kinds of executable slices is shown in Fig. 16. Fig. 16(a) shows the original program

[13]Throughout the remainder of the paper, we use the terms "specialization slicing" and "polyvariant executable slicing" interchangeably. We generally use the latter when we want to emphasize how the properties of Alg. 1 differ from those of the algorithm for monovariant executable slicing.

and its closure slice with respect to the actual parameters of the call to `printf` on line (17). Fig. 16(b) shows the polyvariant executable slice produced by the algorithm of §4. Fig. 16(c) shows the monovariant executable slice that would be created by Binkley's algorithm.

While a closure slice can be useful—e.g., for program understanding—there are some contexts in which an executable slice 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 versions of the word-count utility `wc` that count just characters, just words, or just lines), and (ii) creating versions of libraries specialized to an application. Such specialized components can run faster than the original program. In general, 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. For instance, an experiment with `wc` showed that on average—computed as the geometric mean—its executable slices with respect to its calls to `printf` took 32.5% of the time used by the original `wc`.

According to Binkley [1993, p. 32], 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." While this approach ensures that Weiser's slices are executable, they can include many irrelevant components. For instance, Weiser's algorithm applied to Fig. 16(a) would produce a program in which procedure `p` has all three formal parameters, `a`, `b`, and `c`, and all three actual parameters at each of the call-sites on lines (14)–(16). 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.

Binkley [1993] addressed the parameter-mismatch problem by expanding the closure slice of Horwitz et al. [1990] to include missing actual parameters, together with everything in the backward slice from the missing actuals. The latter slices can themselves introduce new parameter mismatches, so the process is repeated until there are no further parameter mismatches. For instance, as shown in Fig. 16(c), Binkley's algorithm would include the missing first parameters in the two calls to `p` on lines (14) and (16), as well as the assignment to `g2` on line (13). The latter is added back into the slice so that `g2` is initialized properly by the time it is used on line (14).

Binkley's algorithm is never worse than Weiser's, and can lead to smaller executable slices. However, both of their algorithms have the undesirable property that they can cause arbitrarily many program elements that were not in the closure slice to be included in the final answer. In the worst case, they could include *all* of the elements in the input program, even when the closure slice contains just a few program elements. In contrast, Alg. 1 *never* introduces program elements that are not already in the closure slice (albeit the output specialization slice may contain multiple copies of such elements). Thus, Alg. 1 represents a different point in the "design space" of slicing algorithms than previous work on executable slicing.

As discussed in §9, our experiments showed that, for both monovariant executable slicing and polyvariant executable slicing, the size increase is modest. Normalized to "|closure slice| = 100," on average (computed as the geometric mean) monovariant executable slices are 104.6 and polyvariant executable slices are 105.5. However, one should bear in mind that in the first case 104.6 means that 4.6% worth of *extraneous* elements are added (i.e., elements not in the closure slice), whereas in the second case 105.5 means that 5.5% worth of closure-slice elements are *replicated*. While the extraneous elements from Binkley's algorithm are "noise," the replicated elements from specialization slicing provide information about specialized patterns of dependences.

Why Not Just Color a Monovariant Executable Slice?. One might ask, "Why not just highlight with a different color the extraneous program elements (not in the closure slice) that are added back by Binkley's algorithm?" This feature would be desirable to have in the user interface of a system that supports monovariant executable slicing. However, it is not a full substitute for the information obtained via polyvariant executable slicing. In particular, the closure slice of an SDG folds together in callees information from different calling contexts, and thus different patterns of dependences through a procedure would not be highlighted by the coloring scheme. In contrast, these patterns are made explicit through the specialized procedures created via Alg. 1.

For instance, the boxes in Fig. 16(a) show the elements in the closure slice with respect to the actual parameters of the call to `printf` on line (17). The two boxed statements in the body of procedure `p` in the closure slice are the same as the two statements in the body of procedure `p` in the Binkley slice (Fig. 16(c)). Thus, even though the first actual parameters at the call sites on lines (14) and (16) of `main` would be colored, it would not be immediately apparent in procedure `p` itself that the slice really has two different patterns of dependences in the body of `p` (cf. the bodies of procedures `p_1` and `p_2` in Fig. 16(b)). Moreover, such different patterns of dependences can occur many levels down the call chain (cf. Fig. 2).

7. EXTENSIONS AND IMPROVEMENTS

7.1 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 7.1. In C, calls to `exit` cause program termination. This behavior 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

```
 (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)  }
```

Fig. 17. Example illustrating some of the problems that arise in the presence of procedure pointers and indirect calls.

back from a program element that follows a call to `exit` includes the call but not the actual. For the purposes of specialization slicing, adding a dependence edge from the actual to the call solves the problem. □

7.2 Pointers to Procedures and Indirect Calls

Pointers to procedures and calls via those pointers also require special handling. Consider slicing the program shown in Fig. 17 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 issue involves a new kind of parameter-mismatch problem that was not considered by Binkley [1993]. Like the original parameter-mismatch problem, this one can be solved either using a monovariant approach, à la Binkley, or using a polyvariant approach, à la §4. The monovariant 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 polyvariant approach involves adding a new procedure for each indirect call in the program, then performing specialization slicing 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. 17 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)`. After this transformation has been applied, the specialization-slicing algorithm will automatically 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`.

Note that 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 `f_1` and `g_1`. In essence, the original procedures are retained because their addresses define the space of values `V` that pointer `p` can hold. `p`'s value is used to dispatch to the appropriate specialized version of a procedure in `V`. Here is the specialization slice with respect to 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);
}
```

A Caveat. In our implementation, which is based on CodeSurfer/C, there is one limitation on the transformation of indirect calls to explicit PDGs. The transformation described above assumes that the points-to set of the procedure-pointer is exhaustive. However, the pointer-analysis algorithms supported by CodeSurfer/C are all variants of Andersen's analysis [Andersen 1993], 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 procedures, the indirect-call PDG will still have the form

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

and thus the transformed program would call g whenever p is uninitialized (and thus the specialized-slice would call g_1). Note, however, that a program that makes a call via an uninitialized pointer variable is not a "strictly conforming program" according to the ANSI C standard.[14]

7.3 Reslicing and Idempotence

Ordinary slicing algorithms are *idempotent*: let S/C denote the closure slice of SDG S with respect to slicing criterion C; then $(G/C)/C = G/C$. In this section, we show that specialization slicing also has such an idempotence property, modulo some name changes. Our implementation of specialization slicing (§9.1) supports the option to perform the "reslicing" check described below, which serves to demonstrate the near-idempotence property of specialization slicing. (It also provides a way to detect bugs in the implementation: if the reslicing check fails, the implementation has a bug.)

Suppose that for SDG S and slicing criterion C, the SDG that results from specialization slicing is R. Now suppose that we perform a specialization slice of R with respect to C to obtain R', and compare R' to R. If specialization slicing were idempotent, R' and R would be identical. However, similar to the issues raised in the discussion of soundness and completeness in §3.1, because the PDG vertices and call-site labels in S and R are named differently, R' and R will not be identical, in general.

The differences in the names of PDG vertices and call-site labels in S and R create a problem for reslicing because the PDG vertices and call-site labels are alphabet symbols in the FSAs used by the algorithm described in §4. The reslicing check must compensate in two places for 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 symbols.

As in §3.1, 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 finite-state 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.

[14]According to the ANSI C standard [ANSI C 2005, p. 7], "A strictly conforming program shall use only those features of the language and library specified in this International Standard. It shall not produce output dependent on any unspecified, undefined, or implementation-defined behavior, and shall not exceed any minimum implementation limit."

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. 2.6; the language of configurations of R can be obtained by the PDS query $\mathit{Poststar}[\mathcal{P}_R](\text{entry}_{\text{main}})$ [Bouajjani et al. 1997; Finkel et al. 1997].

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

$$L(A6_S) \overset{?}{=} L(T_C(A6_R)). \tag{7}$$

Note that $A6_S$ represents the configurations of the closure slice of the (possibly infinite) unrolling of SDG S. Similarly, $A6_R$ represents the configurations of the closure slice of the unrolling of R. Consequently, Eqn. (7) tests whether the two closure slices have identical configurations after T_C is used to map each R configuration back to a configuration in the alphabet of S. The comparison performed in Eqn. (7) is equivalent to testing whether SDGs R and R' are equal because R and R' are constructed merely by reading out information contained in the automata $A6_S$ and $A6_R$, respectively.

7.4 Speeding Up *Prestar*

The interprocedural-slicing algorithm of Horwitz et al. [1990, §4.1] performs context-sensitive closure slicing on an SDG in three phases:

(1) Construct *summary edges*, which represent transitive dependences due to procedure calls. Each summary edge connects an actual-in vertex *ai* to an actual-out vertex *ao* at the same call-site when the SDG contains a path that has matched calls and returns (see below).

(2) Find all vertices reachable from the slicing criterion in a version of the SDG (with summary edges), but without parameter-out edges.

(3) Find all vertices reachable from the result of step (2) in a version of the SDG (with summary edges), but without parameter-in edges.

In each of the last two phases, the presence of summary edges permits the graph traversal to move "across" a call without having to descend into it.

For context-sensitive, interprocedural closure slicing, the advantages of the three-phase approach can be phrased in terms of *language reachability* [Reps 1998, §4.2 and §5].

DEFINITION 7.2. *L-**reachability** [Yannakakis 1990] is the variant of graph reachability in which a path from vertex s to vertex t only counts as a valid s–t connection if the path's labels (in the sequence encountered along the path) form a word in a given formal language L. In particular,*

—*An L-reachability problem is a **context-free-language (CFL) reachability** problem if L is a context-free language.*

—*An L-reachability problem is a **regular-language reachability** problem if L is a regular language.*

□

Note that ordinary graph reachability is the regular-language reachability problem in which each edge is labeled with "e" and L is e^*. Regular-language reachability problems are less expensive to solve than CFL-reachability problems [Yannakakis 1990; Chaudhuri 2008].

In interprocedural slicing, CFL-reachability is used to create a context-sensitive algorithm by restricting attention to paths in which parameter-in, call-edges, and parameter-out edges along the path form a word in a certain language of (partially) balanced parentheses. Such paths respects the fact that a procedure always returns to the site of the most recent call.

Let each call vertex in the SDG be given a unique index from 1 to *CallSites*, where *CallSites* is the total number of call sites in the SDG. For each call site c_i, label each parameter-in edge and call edge with the symbol "$(_i$," and label each parameter-out edge with "$)_i$." Label all other edges of the SDG with the symbol e. Each path in the SDG defines a word, obtained by concatenating—in order—the labels of the edges on the path.

A path is a *slice path* iff the path's word is in the language $L(slice)$ generated by the context-free grammar shown below. A path is a *matched path* iff the path's word is in the language $L(matched)$ of balanced-parenthesis strings (interspersed with strings of zero or more e's) generated by the context-free grammar shown below on the right. The language $L(unbalR)$ is a language of *partially* balanced parentheses: every left parenthesis "$(_i$" is balanced by a later right parenthesis "$)_i$," but the converse need not hold. Similarly, in $L(unbalL)$ every right parenthesis "$)_i$" is balanced by a preceding left parenthesis "$(_i$," but the converse need not hold. (In each grammar, i ranges from 1 to *CallSites*.)

$$\begin{array}{rl} slice \rightarrow & unbalR \;\; unbalL \end{array} \qquad \begin{array}{rl} matched \rightarrow & matched \;\; matched \\ \mid & (_i \;\; matched \;\;)_i \\ \mid & e \\ \mid & \epsilon \end{array}$$

$$\begin{array}{rl} unbalR \rightarrow & unbalR \;\; matched \\ \mid & unbalR \;\;)_i \\ \mid & \epsilon \end{array} \qquad \begin{array}{rl} unbalL \rightarrow & matched \;\; unbalL \\ \mid & (_i \;\; unbalL \\ \mid & \epsilon \end{array}$$

```
Input: SDG $S$ and slicing criterion $C$
Output: An SDG $R$ for the specialization slice of $S$ with respect to $C$
1 Augment $S$ with summary edges, constructed via the algorithm of Reps et al. [1994]
  /* Phase 1 */
2 $\mathcal{P}_S^1$ = the PDS for $S$, encoded according to Defn. 2.6+, but with all pop rules removed
3 $\mathcal{P}_S^2$ = the PDS for $S$, encoded according to Defn. 2.6+, but with all push rules removed
4 $A0$ = a $\mathcal{P}_S$-automaton that accepts $C$
5 $A1' = Prestar[\mathcal{P}_S^1](A0)$                                   /* Phase 2 */
6 $A1 = Prestar[\mathcal{P}_S^2](A1')$                                   /* Phase 3 */
  // continue at line 4 of Alg. 1
```

Algorithm 2: A three-phase algorithm to create automaton $A1$, replacing lines 1–3 of Alg. 1.

The advantage of the three-phase approach for context-sensitive, interprocedural closure slicing is that only item (1) involves solving a true CFL-reachability problem (with respect to language $L(matched)$). The $L(matched)$-reachability results are used to add summary edges, labeled with "*summary*," to the SDG. Thereafter, items (2) and (3) are each *regular-language* reachability problems over the languages $L(unbalL')$ and $L(unbalR')$, respectively:

$$\begin{array}{rl} unbalR' \rightarrow & unbalR' \;\; summary \\ | & unbalR' \;\;)_i \\ | & \epsilon \end{array} \qquad\qquad \begin{array}{rl} unbalL' \rightarrow & summary \;\; unbalL' \\ | & (_i \;\; unbalL' \\ | & \epsilon \end{array}$$

(The grammar for $unbalR'$ is left-recursive; the grammar for $unbalL'$ is right-recursive; hence $L(unbalR')$ and $L(unbalR')$ are both regular languages.) In other words, the interprocedural-slicing algorithm of Horwitz et al. [1990] performs a staging transformation, using a pre-processing step involving CFL-reachability to compute summary edges, followed by two subsequent lower-cost steps that involve only regular-language reachability.

It turns out that a similar idea can be applied in specialization slicing. Alg. 2 shows a three-phase algorithm to replace lines 1–3 of Alg. 1, where the phrase "encoded according to Defn. 2.6+" in lines 2 and 3 means the PDS-encoding method of Defn. 2.6 extended with the following additional rule:

Rule	Dependence edge modeled
$\langle p, ai\rangle \hookrightarrow \langle p, ao\rangle$	Summary edge $ai \rightarrow ao$

(8)

Automaton $A1$ computed in line 6 of Alg. 2 is computed in three phases, rather than with a single call to *Prestar*, as in line 3 of Alg. 1. As argued below in Thm. 7.3, the $A1$ automata computed by the two algorithms are identical.

The advantage of computing $A1$ via the three-phase approach is somewhat similar to the advantage of the three-phase approach for context-sensitive, interprocedural closure slicing:

(1) Alg. 2 and the three-phase algorithm for context-sensitive, interprocedural closure slicing both involve a staging transformation, in which the first stage solves a CFL-reachability problem to compute summary edges.

(2) The second and third phases of each algorithm makes use of only a limited

version of the respective underlying formalism:

(a) In the three-phase algorithm for context-sensitive, interprocedural closure slicing, phases two and three involve regular-language reachability over $L(\mathit{unbalL}')$ and $L(\mathit{unbalR}')$, rather than full CFL-reachability.

(b) The PDSs $\mathcal{P}_S^1$ and $\mathcal{P}_S^2$ used in lines 5 and 6 of Alg. 2 have no pop rules and push rules, respectively. Now suppose that we put the label "$)_i$" on all transitions generated by a pop rule associated with call-site i, "$(_i$" on all transitions generated by a push rule associated with i, "*summary*" on all transitions generated by rule (8), and "e" on all other transitions. We then have

—Each (finite) path in the infinite transition relation of $\mathcal{P}_S^1$ that starts at a $\mathcal{P}$-configuration in $L(A0)$ generates a word in the regular language $L(\mathit{unbalL}')$.

—Each (finite) path in the infinite transition relation of $\mathcal{P}_S^2$ that starts at a $\mathcal{P}$-configuration in $L(A1')$ generates a word in the regular language $L(\mathit{unbalR}')$.

Theorem 7.3. *Automaton $A1$ computed in line 6 of Alg. 2 is identical to automaton $A1$ computed in line 3 of Alg. 1.* □

Proof. The proof follows from the observations made in item (2b) above.

Recall that automata $A1$ and $A6$ computed by Alg. 1 accept the same languages. By Thm. 4.14, it follows that $L(A1)$ computed by Alg. 1 consists of exactly the $\mathcal{P}$-configurations in the closure slice, with respect to $L(A0)$, of the infinite transition relation of $\mathcal{P}_S$. Those $\mathcal{P}$-configurations are reached along $L(\mathit{slice})$-paths, or equivalently, along $L(\mathit{unbalR}') \cdot L(\mathit{unbalL}')$ paths (where "$\cdot$" denotes language concatenation). However, the latter is exactly the set of $\mathcal{P}$-configurations accepted by automaton $A1$ computed by Alg. 2. □

In our experiments (§9), we found that using Alg. 2 in place of the one-phase method of computing $A1$ was 1.5–4.6 times faster (geometric mean: 2.9) on `gzip`, `flex`, and `go`—the only examples for which *Prestar* represented a non-negligible portion of the overall time for specialization slicing (cf. column 7/(column 2 + column 8) in Fig. 23).

8. FEATURE REMOVAL

If a program consists of only a single procedure, a forward closure slice of the procedure's PDG defines a kind of "feature" consisting of all elements possibly influenced by the slicing criterion. Moreover, it is possible to use a conventional PDG slicing algorithm to remove such a feature. The key property is the following:

Observation 8.1. *The complement of a (single-procedure) forward closure slice is a backward slice with respect to a different slicing criterion (namely, the vertices of the complement).* □

Thus, by removing the elements in the forward closure slice, one is left with a PDG without the feature identified by the forward closure slice. An executable program can be created from the latter PDG.

It is unclear how to implement feature removal for multi-procedure programs via standard SDG-based slicing techniques [Horwitz et al. 1990]. In particular, Obs. 8.1

(a) Program to compute sum and product

```
int add(int a,int b) {
  q: return a+b;
}

int mult(int a,int b) {
  int i = 0;
  int ans = 0;
  while(i < a) {
    c5: ans = add(ans,b);
    c6: i = add(i,1);
  }
  return ans;
}

void tally
(int& sum,int& prod,int N) {
  int i = 1;
  while(i <= N) {
    c2: sum = add(sum,i);
    c3: prod = mult(prod,i);
    c4: i = add(i,1);
  }
}

int main() {
  int sum = 0;
  int prod = 1;
  c1: tally(sum,prod,10);
  printf("%d ",sum);
  printf("%d ",prod);
}
```

(b) Summation program (before useless-code elimination)

```
int add(int a,int b) {
  q: return a+b;
}

int mult(      int b) {
  int i = 0;
  int ans = 0;






  return;
}

void tally
(int& sum,          int N) {
  int i = 1;
  while(i <= N) {
    c2: sum = add(sum,i);
    c3:        mult(     i);
    c4: i = add(i,1);
  }
}

int main() {
  int sum = 0;

  c1: tally(sum,     10);
  printf("%d ",sum);

}
```

Fig. 18. (a) A program to compute the sum and product of the integers from 1 to N. The boxed code indicates the forward closure slice with respect to "`prod = 1`" in `main`. (b) The program obtained by using specialization slicing to remove the forward slice with respect to configuration $(\!|$`prod = 1`$, \epsilon|\!)$ in `main`.

does not hold, in general, for the forward closure slice of an SDG. Consider the program shown in Fig. 18, which uses procedure `tally` to compute both the sum and product of the numbers from 1 to 10. Additions are performed using procedure `add`, which is also invoked repeatedly to perform multiplication. Suppose that we want to remove the computation of the product. That "feature" includes all of the code in the forward closure slice from "`prod = 1`" (i.e., the boxed code in Fig. 18(a)). However, we cannot just remove all elements of the forward closure slice: that approach would remove procedure `add`, but we must *keep* `add` because `add` is needed to compute the sum.

The cause of the problem is that the standard SDG closure-slicing algorithm merges different configurations of the unrolled SDG. When the standard SDG-based algorithm for context-sensitive closure slicing [Horwitz et al. 1990] is applied

```
Input: SDG $S$ and slicing criterion $C$
Output: A specialization-slice SDG $R$ for $S$ with the forward stack-configuration slice with
        respect to $C$ removed
1 $\mathcal{P}_S$ = the PDS for $S$, encoded according to Defn. 2.6
2 $A_E$ = a $\mathcal{P}_S$-automaton that accepts $\{(\!|enter_{main}, \epsilon|\!)\}$
3 $A_C$ = a $\mathcal{P}_S$-automaton that accepts $C$
4 $A0 = \mathit{Poststar}[\mathcal{P}_S](A_C)$
5 $A1 = \mathit{Poststar}[\mathcal{P}_S](A_E) \cap$ complement(determinize($A0$))
  // continue at line 4 of Alg. 1
```

Algorithm 3: An algorithm to remove a "feature" defined by the forward stack-configuration slice with respect to C.

to a multi-procedure program, the complement of the forward closure slice of an SDG is not necessarily a backward closure slice of the SDG, and thus it may not be possible to create an executable program from the complement.

We now explain how specialization slicing, in conjunction with *forward stack-configuration slicing*, can be employed to solve the feature-removal problem for multi-procedure programs. In particular, because the PDS-based machinery developed in §4 explicitly manipulates the possibly infinite set of configurations of the unrolled SDG, we regain the property stated in Obs. 8.1, and have the machinery needed to create a feature-removal algorithm.

In the PDS-based approach to feature removal (Alg. 3), the algorithm (i) subtracts the set of *configurations* of the forward stack-configuration slice with respect to criterion C from the set of *configurations* that are reachable from $(\!|enter_{main}, \epsilon|\!)$ (line 5), and (ii) creates an executable program from the result. For instance, for the program from Fig. 18(a), $\mathcal{P}_S$—defined in line 1 of Alg. 3—contains the following configurations for `q` in `add`: $\{(\!|q, c2\ c1|\!), (\!|q, c5\ c3\ c1|\!), (\!|q, c6\ c3\ c1|\!), (\!|q, c4\ c1|\!)\}$. When slicing criterion C is $\{(\!|\texttt{prod = 1}, \epsilon|\!)\}$, $A0$'s `q`-configurations are $\{(\!|q, c5\ c3\ c1|\!), (\!|q, c6\ c3\ c1|\!)\}$, and hence $A1$'s `q`-configurations are $\{(\!|q, c2\ c1|\!), (\!|q, c4\ c1|\!)\}$. By this means, Alg. 3 correctly keeps statement `q` (and all of procedure `add`) in the feature-removal result shown in Fig. 18(b).

The key property of automaton $A1$ in Alg. 3 is that it represents a set of configurations that is backwards-closed with respect to the transitions in $\mathcal{P}_S$'s possibly infinite transition relation.

Note how `tally` is specialized from three parameters in Fig. 18(a) to two in Fig. 18(b). This specialization happens automatically because the core of Alg. 3 is the specialization-slicing algorithm (Alg. 1). Fig. 18 also illustrates how the feature-removal algorithm operates at a fine-grained level: it leaves in all elements that are not in the forward stack-configuration slice from $(\!|\texttt{prod = 1}, \epsilon|\!)$, including some "extraneous" elements, namely, the specialization of `mult` and the specialized call to `mult` at `c3`. These elements are useless, and the program could be cleaned up by performing an interprocedural useless-code-elimination pass.

9. EXPERIMENTS

9.1 Structure of the Implementation

Our implementation was created using GrammaTech's CodeSurfer/C code-understanding tool [Anderson et al. 2003], which supports closure slicing of SDGs of C and C++ programs, and provides a scripting language to provide access to a program's SDG. The implementation supports specialization slicing (§4), a variant of the reslicing check (§7.3), and feature removal (§8).

CodeSurfer/C is used to build the input SDG, which is then translated into a PDS implemented using the Weighted Automaton Library (WALi) [Kidd et al. 2007]. WALi is used to perform the *Prestar* operation, and the resulting automaton is converted into an OpenFST "recognizer" (FSA) [OpenFST 2012]. OpenFST is used for the reverse, determinize, minimize, reverse, and removeEpsilonTransitions 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.

Calls to library procedures are handled by the technique described in §7.1. Dependences through calls to library procedures are specified by supplying *library models*: procedure stubs and variable declarations (in source-code form) that characterize the dependences through the user-facing procedures of the library. The library models used in the experiments are a variant of the standard library models distributed with CodeSurfer/C. I/O procedures are handled conservatively by the so-called "monolithic" strategy: a single object, `_INPUT_OUTPUT`, is used to represent the entire file system and all sockets [Codesurfer].

—Any flow of information from the file system or any socket is modeled by a read from `_INPUT_OUTPUT`.

—Any flow of information to the file system or a socket is modeled by a write to `_INPUT_OUTPUT`.

This conservative approach causes the SDG to contain flow dependences between separate calls to I/O procedures—such as a call on `fprintf()` and a subsequent call on `getchar()`, even if the two procedures would actually always operate on different file-system objects.

Memory-allocation procedures, such as `malloc()`, `free()`, etc., are handled by the so-called "discrete" strategy. This approach introduces no dependences between two calls to `malloc()`, for example, even though both would read and write variables associated with the runtime free-storage package. The rationale for using the discrete approach is that it is consistent with the idealized semantics that the free-storage package is attempting to implement—namely, "separate allocations are independent, drawn from an inexhaustible supply of memory."

All experiments were run on a 64-bit, 8-core Dell Latitude E6520 laptop with 8GB memory, running Ubuntu 12.04. (However, the specialization-slicing tool is a single-threaded application.)

9.2 Specialization-Slicing Experiments

Our experiments were designed to answer five questions:

Program	# Versions	Avg. # source lines	Avg. # of Procedures	Avg. # PDG vertices	Avg. # of Call sites	# Slices taken
tcas	37	564	9	478	38	37
schedule2	2	717	16	979	47	6
schedule	6	725	18	864	44	11
print_tokens	4	889	18	1261	89	4
replace	26	931	21	1330	65	58
print_tokens2	8	957	19	1080	84	42
tot_info	19	1414	7	675	37	23
wc	1	802	11	2048	170	13
gzip	4	5314	94	24826	556	26
space	20	7429	136	18799	1016	69
flex	5	10082	154	39816	1308	79
go	1	29246	372	102859	2084	10

Fig. 19. Information about the test programs. (Averages are rounded to the nearest whole number.)

(1) *Blow-up of polyvariant* (§9.2.1): Does Alg. 1 suffer from its potential worst-case, exponential blow-up in practice?

(2) *Polyvariant vs. closure* (§9.2.2): Are polyvariant executable slices created via Alg. 1 substantially larger than closure slices in practice?

(3) *Monovariant vs. closure* (§9.2.2): Are monovariant executable slices substantially larger than closure slices in practice?

(4) *Polyvariant vs. monovariant* (§9.2.2): How do polyvariant and monovariant executable slicing compare?

(5) *Specialization slicing of recursive programs in practice* (§9.2.3): What kinds of slices arise in practice when specialization slicing is applied to programs that use recursion?

Information about the test programs is given in Fig. 19. The first seven programs are the Siemens suite—a set of small programs originally collected by Hutchins et al. [1994]. The others are open-source applications.

—We picked the Siemens suite, `gzip`, `space`, and `flex`, because they were available from the Software-artifact Infrastructure Repository [Do et al. 2005; SIR], which provides multiple versions of each program, together with supporting artifacts such as test suites and fault data.

—We picked the word-count program `wc` because it is a nice example of how slicing can be used to create specialized versions of a program.[15]

—We picked `go` because it was three times larger than any of the other examples.

[15]In particular, slices taken from a certain output statement in `wc` with respect to the variables `total_ccount`, `total_wcount`, `total_lcount` specialize `wc` into programs that count just characters, just words, and just lines, respectively. We used such slices to perform an experiment to measure the execution speed-up of specialized components extracted from `wc` (see §6). For that experiment, we used one of the two standard sets of library models distributed with CodeSurfer/C—namely, the one that adopts a "discrete" strategy for both I/O procedures and memory-allocation procedures.

Moreover, for the Siemens suite, `gzip`, `space`, and `flex`, the debugging tools developed by Horwitz et al. [2010, §2.3 and §3] provided information about points of failure (i.e., unexpected output or crashes) and the call-stacks at those points. Our reasons for using such slicing criteria were two-fold:

—A slicing criterion of the form $(\!|\text{PDG-vertex}, \text{call-stack}|\!)$ exercises stack-configuration slicing at the granularity of an individual configuration.

—Slicing with respect to a configuration at which the symptom of a runtime error has been observed provides a rational method for generating slices that would be of interest to a programmer, at least in the context of program debugging.

For some program versions, the program's test suite has multiple test inputs that generate a runtime error; thus, some program versions had multiple slicing criteria. For each program, columns 2 and 7 of Fig. 19 show the number of program versions and the number of slicing criteria, respectively, that were available to us.

For `wc` and `go`, a slice was taken from each call-site on `printf` or `fprintf`, with respect to all calling contexts (by using an automaton that expressed this slicing criterion).

In the case of the Siemens suite, `gzip`, `space`, and `flex`, the algorithm for polyvariant executable slicing begins by performing a stack-configuration slice with respect to a singleton set of call-stack configurations. For these examples, by "monovariant executable slicing," we mean a generalization of Binkley's algorithm [1993] that (i) performs a calling-context slice [Binkley 1997; Krinke 2004] (i.e., a context-sensitive closure slice of the SDG with respect to a given call-stack configuration[16]), and then (ii) performs additional backward-slicing operations to address any parameter mismatches and to make the answer executable. (Call-stack information is not used during phase (ii).) The motivation for this choice is to be able to isolate the cost of finding *polyvariant* versus *monovariant* answers by, to the extent possible, controlling for the effect of using stack-configuration slicing in the algorithm for polyvariant executable slicing. Consequently, our implementations of both algorithms use call-stack-sensitive slicing methods when call-stack information is available.

For `wc` and `go`, "monovariant executable slicing" means Binkley's algorithm as originally stated.

In summary, the experiments were based on twelve different *program families.* In total, they involved 133 different program versions, on which we performed 378 different invocations of specialization slicing. (133 and 378 are the sums of the numbers in columns 2 and 7 of Fig. 19, respectively.) In the program versions used for the 378 slices, there were a total of 30,773 procedures, of which only 20,424 procedures appeared in the slice results. (30,773 is the sum of the products of columns 4 and 7 of Fig. 19; the number 20,424 is discussed below.)

9.2.1 *Blow-Up of Polyvariant.* Figs. 20–24 provide evidence that polyvariant executable slicing does *not* exhibit its worst-case exponential behavior in practice. The figures show that neither (i) the exponential blow-up that occurs for the family of examples discussed in §5.3, nor (ii) the worst-case exponential behavior of

[16]See §10 for more discussion of calling-context slicing.

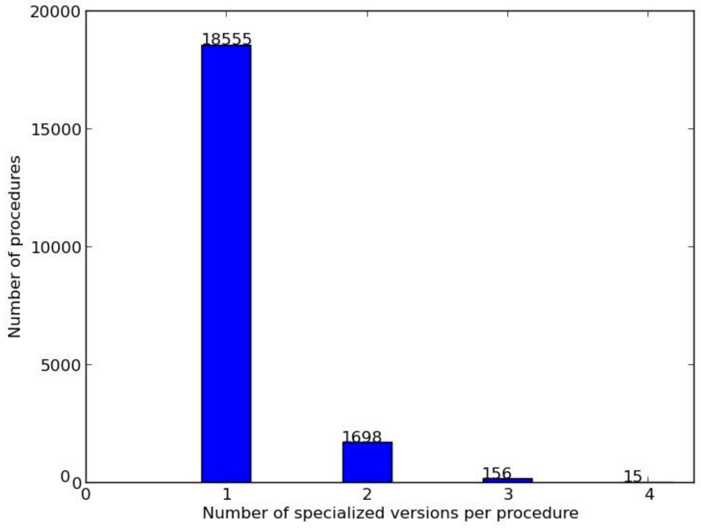


Fig. 20. Distribution of the number of specialized versions of procedures in executable slices created via polyvariant executable slicing.

operations like automaton determinization, arose in the programs and slices used in our experiments. (In fact, as mentioned in §5.2, for the automata that arise from *Prestar*, the result of determinize is significantly *smaller* than the input to determinize by 4.4%–34%.)

Fig. 20 shows the distribution of the number of specialized versions of procedures in all tested programs. From Fig. 20, we see that the closure slices from all tests consisted of 20,424 procedures (= 18,555 + 1,698 + 156 + 15). That is, 33% of the 30,773 procedures in all of our tests were eliminated by closure slicing. Out of the 20,424 procedures in the closure slices, multiple versions were created by Alg. 1 for 1,869 of the procedures (= 1,698 + 156 + 15); i.e., multiple versions were created for only 1,869/20,424 = 9.2% of the procedures in the closure slices.

Put another way, approximately 90.8% of the procedures in the closure slices have a single specialized version, and the count decreases rapidly as the number of specialized procedures increases. The largest number of specialized versions of a procedure that we saw in our experiments with specialization slicing is four. Fig. 20 shows that Alg. 1 produced a total of 22,479 procedures (= 18,555 + 2 $\times$ 1,698 + 3 $\times$ 156 + 4 $\times$ 15) of which 2,055 (= 1,698 + 2 $\times$ 156 + 3 $\times$ 15) were “extra” copies.

9.2.2 *Polyvariant vs. Monovariant vs. Closure.* Both Alg. 1 and the algorithm for monovariant executable slicing perform closure slices as their first step—although the monovariant slicing algorithm performs a closure slice of the SDG, whereas Alg. 1 performs a closure slice of the *unrolled* SDG. Nevertheless, if the stack configurations of the closure slice used in Alg. 1 are discarded, and we consider just the set of program elements involved, both algorithms identify the same set of program elements in their first step. Thus, we can normalize measurements

		Average % increase in #PDG vertices relative to closure slice			
	# Slices	Monovariant slicing		Polyvariant slicing	
Program	taken	% increase	Std. dev.	% increase	Std. dev.
tcas	37	5.3	0.1	0.2	0.0
schedule2	6	6.3	0.0	28.1	0.1
schedule	11	4.8	0.7	5.6	1.8
print_tokens	4	5.1	0.0	0.4	0.0
replace	58	7.1	0.7	5.6	5.4
print_tokens2	42	5.8	0.4	0.7	2.1
tot_info	23	2.8	0.1	0.0	0.0
wc	13	5.1	1.5	38.5	20.0
gzip	26	6.1	2.1	7.6	6.6
space	69	4.2	1.4	4.3	2.3
flex	79	2.1	0.1	6.7	2.4
go	10	4.4	7.6	7.9	3.9
geometric mean	N/A	4.6	N/A	5.5	N/A

Fig. 21. Comparison of the percentage of "extra" vertices, relative to the size of the closure slice, produced via monovariant executable slicing with those produced via polyvariant executable slicing.

relative to the size of the set of program elements in the closure slice of the SDG.

Figs. 21–24 provide four ways to compare polyvariant/monovariant executable slicing and closure slicing.

Blow-Up in Overall Slice Size. One way to compare polyvariant and monovariant executable slicing is to compare the overall sizes of the resulting slices. Columns 3 and 5 of Fig. 21 provide data about the percentage of "extra" vertices that are in the two kinds of executable slices, compared with the corresponding closure slices. Based on these results, it appears that in practice, blow-up in slice size is neither a problem for monovariant executable slicing nor for polyvariant executable slicing. Normalized to "|closure slice| = 100," on average (computed as the geometric mean) monovariant executable slices are 104.6 and polyvariant executable slices are 105.5. While polyvariant executable slicing had the largest average increases in size (e.g., 38.5% for `wc`), on the three largest programs size increases were very modest, and were similar for monovariant and polyvariant executable slicing. However, one should bear in mind that in the first case 104.6 means that 4.6% worth of *extraneous* elements are added (i.e., elements not in the closure slice), whereas in the second case 105.5 means that 5.5% worth of closure-slice elements are *replicated.* While the extraneous elements from monovariant executable slicing are "noise," the replicated elements from polyvariant executable slicing provide information about specialized patterns of dependences.

Sizes of Individual PDGs. Another way to compare polyvariant and monovariant executable slicing is to compare *procedure sizes*, as opposed to the *overall sizes* of slices, as we did above. As we saw from Fig. 21, the monovariant slicing algorithm introduces 4.6% worth of elements that are not in the closure slice. In contrast, Alg. 1 *never* introduces any program elements that are not in the closure slice.

Fig. 22 is a scatter-plot that compares the sizes of the PDGs produced via monovariant executable slicing with those produced via polyvariant executable slicing.

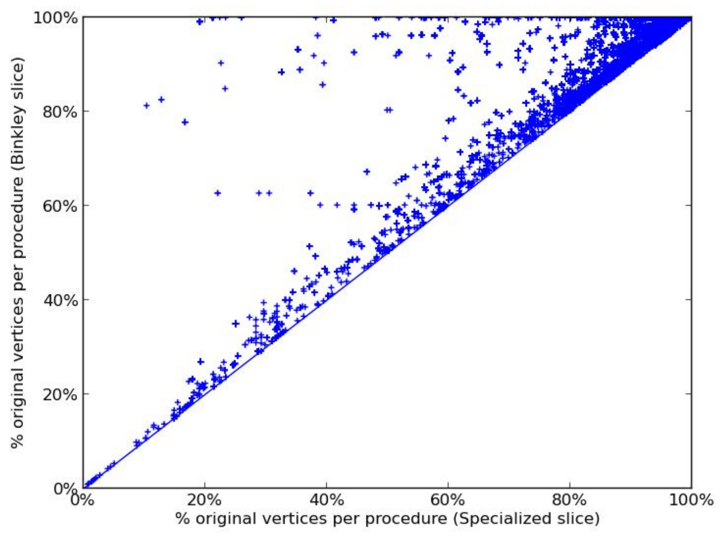


Fig. 22. Scatter-plot of procedure sizes. Each dot represents the relative size of a procedure produced via monovariant executable slicing against the size of a corresponding procedure produced via polyvariant executable slicing. Both axes are in terms of the percentage of vertices in the original procedure.

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 polyvariant slice; i.e., for each PDG `p_k` in each polyvariant 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 monovariant slice. If there are two specialized versions of PDG `p`, there will be two points in the plot, all with the same y coordinate.

Fig. 22 shows that many polyvariant-slice PDGs are close in size to the original PDGs (the points clustered in the upper right-hand corner); that monovariant-slice PDGs are often close in size to the corresponding polyvariant-slice PDGs (the points clustered along the 45-degree line); but that there are also cases where a monovariant-slice PDG is much larger than the corresponding polyvariant-slice PDG (the points along the top). For each point, including all "extra" procedures, we computed (%vertices in polyvariant PDG)/(%vertices in monovariant PDG); the geometric mean of these values is 92.9%.

Running Time. Fig. 23 provides information about slicing times when creating executable slices via monovariant and polyvariant executable slicing. Columns 5 and 10 of Fig. 23 report end-to-end slicing times—i.e., times for starting with the SDG, slicing via the respective algorithms, creating the output SDG, and generating the program text for the output slice. In the case of polyvariant executable slicing, column 9 of Fig. 23 reports the amount of time that was used for all FSA operations. (Column 9 times are included in column 10.)

	Average SDG construction time (secs.)		Average slicing time (secs.)							
			Monovariant slicing			Polyvariant slicing				
	Sum.	All	Slice	All	Std.	*Prestar*		Autom.	All	Std.
Program	edges	steps	time	steps	dev.	1-phase	3-phase	ops.	steps	dev.
tcas	0.001	0.16	0.02	0.58	0.014	0.000	0.004	0.004	0.75	0.011
schedule2	0.000	0.55	0.05	0.69	0.009	0.008	0.008	0.009	1.05	0.009
schedule	0.004	0.30	0.04	0.66	0.050	0.009	0.012	0.005	0.95	0.065
print_tokens	0.005	0.17	0.05	0.88	0.015	0.007	0.009	0.011	1.25	0.022
replace	0.014	0.59	0.06	0.95	0.102	0.009	0.009	0.010	1.37	0.153
print_tokens2	0.006	1.67	0.04	0.85	0.037	0.004	0.006	0.008	1.16	0.052
tot_info	0.003	0.25	0.04	0.71	0.014	0.004	0.006	0.005	0.93	0.016
wc	0.000	3.25	0.05	0.94	0.314	0.024	0.027	0.010	1.59	0.639
gzip	0.383	1.14	0.53	*0.90	0.398	17.826	4.806	0.280	*11.43	2.284
space	0.238	5.55	0.81	4.67	1.035	0.219	0.240	0.108	7.33	1.476
flex	4.739	10.24	4.62	14.31	4.974	35.690	18.978	2.428	61.93	21.292
go	7.700	34.80	2.06	26.70	11.306	115.118	17.587	5.654	107.17	34.260

Fig. 23. Average times (in seconds) to build SDGs and create monovariant and polyvariant executable slices. Column 2 reports the time to construct summary edges. Column 3 reports the overall time to construct the SDG. (Column 2 times are included in Column 3.) Column 4 reports the time to perform the original closure slice of the SDG, as well as the time for the additional slices needed to resolve parameter mismatches for monovariant executable slicing. Column 5 reports the time for all steps of monovariant executable slicing (including pretty-printing). Columns 7 and 8, respectively, show the times used for *Prestar* by the one-phase method, and for phases 2 and 3 of the three-phase method. (Column 2 corresponds to phase 1 of the three-phase method.) Column 9 reports the amount of time that was used for all FSA operations during polyvariant executable slicing. Column 10 reports the time for all steps of polyvariant executable slicing (including pretty-printing). (The times reported in columns 8 and 9—but not column 7—are included in column 10.) Columns 6 and 11 show the standard deviations in the overall times for the two methods. "*" indicates that the time for pretty-printing is not included (due to the lack of pretty-printing support for a feature used in the program).

Overall, our implementation of polyvariant executable slicing is about 1.4x slower than our implementation of monovariant executable slicing (geometric mean) on the seven examples above the line in Fig. 23, and 3.6x slower (geometric mean) on the five examples below the line in Fig. 23.

Remark. In considering the times reported in Fig. 23, one should bear in mind that our implementation of polyvariant executable slicing is an experimental implementation that makes use of WALi and OpenFST, in addition to CodeSurfer/C. There are several copies made of data structures that are $O(\text{size of program})$. The implementation of monovariant executable slicing makes use of the slicing algorithm used internally by the CodeSurfer/C product, as well as some fix-ups that we were able to implement entirely within CodeSurfer/C, using the scripting language that CodeSurfer/C supports.

For both implementations, the times should be taken with a grain of salt. Both implementations used two experimental additions to CodeSurfer/C: (i) operations for rewriting SDGs—in this case, removing vertices and edges—and (ii) a pretty-printer for generating program text from an SDG. Neither extension can be considered to be a first-class part of the API of CodeSurfer/C's scripting language. □

The differences between the times in the "All-steps" columns and the times for the individual steps shown in Fig. 23—i.e., column 5 vs. column 4, and column 10 vs. column 8 + column 9—are due to the costs of (i) serializing the SDG to and

	Average maximum memory usage (MB)					
	Monovariant slicing		Polyvariant slicing			
Program	CodeSurfer/C	Std. dev.	CodeSurfer/C	Std. dev.	Automaton operations	Std. dev.
tcas	150.40	72.89	129.02	0.49	21.49	0.05
schedule2	160.48	77.12	138.83	0.33	23.69	0.09
schedule	155.67	74.94	133.77	1.12	22.98	0.06
print_tokens	164.27	77.30	143.32	0.12	23.72	0.03
replace	163.27	76.11	142.61	2.54	23.81	0.01
print_tokens2	161.00	75.71	137.71	0.74	23.34	0.07
tot_info	161.63	77.32	137.40	0.39	23.99	0.02
wc	193.14	91.65	169.27	8.85	28.14	0.00
gzip	376.87	193.07	1087.35	86.71	362.45	28.90
space	361.70	155.52	358.41	79.85	54.51	0.19
flex	876.11	347.88	2124.43	427.11	708.14	142.37
go	1804.05	639.88	4025.09	65.02	1341.70	21.67

Fig. 24. The space (in MB) used when creating executable slices via monovariant and polyvariant executable slicing. Column 6 reports the amount of space that was used for all PDS and FSA operations during polyvariant executable slicing. (The space reported in column 6 is in addition to that reported in column 4 for the CodeSurfer/C process.)

from the C++ data structures used by WALi and OpenFST, (ii) rewriting SDGs, and (iii) pretty-printing.

We can also compare monovariant executable slicing and Alg. 2 based just on the operations performed up until SDG rewriting.

—Monovariant executable slicing performs SDG construction (column 3) and slicing (column 4).

—Alg. 2 performs SDG construction (column 3), phases 2 and 3 of the three-phase slicing method (column 8), and automaton operations (column 9). (Alg. 2 needs summary edges, but the computation of summary edges is included in SDG construction time (column 3).)

For the three examples for which *Prestar* represents a non-negligible portion of the overall time for specialization slicing—i.e., `gzip`, `flex`, and `go`, see column 8—the geometric mean of the polyvariant/monovariant slicing ratios, computed as (column 3 + column 4)/(column 3 + column 8 + column 9), is 2.32. For the Siemens suite, the geometric mean of the polyvariant/monovariant slicing ratios is 0.93 (i.e., polyvariant slicing is 7% faster).

As already mentioned in §7.4, we found that using Alg. 2 in place of the one-phase method of computing $A1$ was 1.5–4.6 times faster (geometric mean: 2.9) on `gzip`, `flex`, and `go` (cf. column 7/(column 2 + column 8)). One difference that could account for the improved running time is how summary edges are computed in Alg. 2. Our implementation of Alg. 2 uses CodeSurfer/C to compute summary edges (column 2). Although the computation of summary edges is similar to the computation of pre^* and $post^*$ for a PDS, CodeSurfer/C uses an implementation written in C, which has been extensively optimized, whereas WALi supports more straightforward implementations of pre^* and $post^*$, written in C++.

Space Used. Fig. 24 provides information about the amount of space used when creating executable slices via monovariant and polyvariant executable slicing. The

Program	Direct recursion in source?	Mutual recursion in source?	≥ 2 versions of a specialized recursive procedure?	Direct rec. transformed to mutual recursion?
Fig. 2	✓		✓	✓
print_tokens	✓			
replace	✓		✓	
tot_info	✓			
gzip		✓	✓	
flex	✓	✓	✓	
go	✓			

Fig. 25. Characteristics of specialization slicing applied to Fig. 2 and the six families of recursive programs in our suite of benchmarks. ✓ indicates that the indicated feature was observed in the original source code (columns 2 and 3), or in at least one program created via specialization slicing (columns 4 and 5).

space reported is the average, over all slices of a given example, of the maximum memory used during each slice (in MB). The two algorithms used roughly comparable amounts of space in their respective CodeSurfer/C processes.

In the case of polyvariant executable slicing, the additional amount of space used during PDS and FSA operations is reported separately from the space used by the CodeSurfer/C process (see column 6). In all cases, the peak memory usage for PDS and FSA operations occurred during *Prestar*; after *Prestar*, memory usage always dropped by 30–80%. These observations are consistent with the fact that slicing was performed with respect to slicing criteria expressed using non-trivial FSAs. That is, for the Siemens suite, `gzip`, `space`, and `flex`, the *Prestar* operations are with respect to FSAs that capture languages of the form $\{(\!|\text{PDG-vertex, call-stack}|\!)\}$, representing the configuration of a bug site. For `wc` and `go`, the slicing criterion for each call-site on `printf` or `fprintf` was expressed using an FSA that captured all of the call-site's calling contexts in the unrolled SDG. The significance of using a non-trivial FSA for the slicing criterion is that the number of states in the query FSA contributes multiplicatively to the space cost of *Prestar*. That is, using the algorithm of Esparza et al. [2000] (which is implemented in WALi) the space used during *Prestar* is bounded by $O(|Q_{\mathcal{A}}|\,|\Delta| + |{\rightarrow_{\mathcal{A}}}|)$, where $|Q_{\mathcal{A}}|$ is the number of states in the FSA $\mathcal{A}$ for the slicing criterion.

Given our earlier observations that (i) the sizes of the output slices created via Alg. 1 were only modestly larger than the corresponding closure slices, namely 5.5% (see column 5 of Fig. 21), and (ii) determinize never blew up, we believe it is fair to say that, for the *observed cost*, both the running time and space of Alg. 1 are bounded by the sum of two terms: one polynomial in the size of the input program, the other linear in the size of the output slice. (See the *non-exponential* costs of Alg. 1 discussed in §5.2 and §5.3.)

9.2.3 *Specialization Slicing of Recursive Programs in Practice.* This section presents our experience with applying specialization slicing to programs that use recursion. Fig. 25 summarizes our observations. Although the transformation of a directly recursive procedure into two mutually recursive procedures, as illustrated in Fig. 2, was not observed in the slices in our test suite, we did encounter several

(a) Program containing recursive procedure `r`	(b) Specialization of `r` resulting from slicing at line 23
(1) `int g = 0;`	`int g = 0;`
(2)	
(3) `int r(int k) {`	`int r_1(int k) {`
(4) `  int v = 0;`	`    int v = 0;`
(5) `  if (k > 0) {`	`    if (k > 0) {`
(6) `    v = r(k-1);`	`        v = r_1(k - 1);`
(7) `  }`	`    }`
(8) `  ++g;`	`    ++g;`
(9) `  return v;`	`    return v;`
(10) `}`	`}`
(11)	
(12)	`int r_2(int k) {`
(13)	`    int v = 0;`
(14)	`    if (k > 0) {`
(15)	`        v = r_2(k - 1);`
(16)	`    }`
(17)	`    return v;`
(18)	`}`
(19)	
(20) `int main() {`	`int main() {`
(21) `  int a = r(2);`	`  int a = r_1(2);`
(22) `  int b = r(g);`	`  int b = r_2(g);`
(23) `  printf("%d", a + b);`	`  printf("%d", a + b);`
(24) `}`	`}`

Fig. 26. (a) Example program with recursive procedure `r`. (b) Specialization slice with respect to the call to `printf` on line (23), which causes `r` to be specialized based on different patterns of formal-out vertices at different call-sites. Global variable `g` is not used after the second top-level call to `r` on line (22); thus, the increment of `g` during the second call does not contribute to any value used in the slicing criterion.

instances where a recursive procedure was specialized into two or more versions. In each case, the stack-configuration slice contained different patterns of formal-out vertices in different calling contexts.

Fig. 26 provides a simple example in which a *directly recursive* procedure `r` is specialized into two versions. Global variable `g` is not used after the second top-level call to `r` on line (22) of Fig. 26(a); thus, the increment of `g` during the second call does not contribute to any value used in the slicing criterion. Consequently, specialized procedure `r_1` retains the statement that increments `g` (see line (8) of Fig. 26(b)), whereas the increment of `g` is absent in procedure `r_2`. This example is representative of the majority of cases in which a specialization slice in our test suite caused a recursive procedure to be specialized into two or more versions.

We also observed cases in which a group of *mutually recursive* procedures was specialized into two or more groups of mutually recursive procedures. By a group of mutually recursive procedures, we mean a strongly-connected component of the program's call graph that contains more than one node. Such specializations were caused by different usage patterns of formal-out vertices in different calling contexts

(a) Program with two calling contexts for procedure `p`	(b) Specialized slice with two syntactically equal specializations of `p`
`(1) int g;`	`int g;`
`(2)`	
`(3) void p() {`	`void p_1() {`
`(4)   g++;`	`  g++;`
`(5)   printf("%d", g);`	`  printf("%d", g);`
`(6) }`	`}`
`(7)`	
`(8)`	`void p_2() {`
`(9)`	`  g++;`
`(10)`	`  printf("%d", g);`
`(11)`	`}`
`(12)`	
`(13) int main() {`	`int main() {`
`(14)   g = 0;`	`  g = 0;`
`(15)   p();`	`  p_1();`
`(16)   p();`	`  p_2();`
`(17) }`	`}`

Fig. 27. (a) Example program for which a specialization slice creates two specializations of procedure `p` that are syntactically equal. (b) Specialization slice with respect to the call to `printf` on line (5): procedure `p` is specialized into two syntactically identical procedures, `p_1` and `p_2`.

from which the mutually recursive group could be entered. The largest blow-up of source code incurred by this kind of example occurred in `flex` (which contains 147 procedures overall); it caused a mutually recursive group of three procedures in the original `flex` to become three mutually recursive groups of three procedures each in the specialization slice.

Thus, although small amounts of blow-up do occur when specialization slicing is performed on programs that use recursion, our experience to date supports the assertion that the potential exponential blow-up of specialization slicing is not observed in practice.

9.2.4 *Specialized Procedures can be Syntactically Equal.* In a few cases, we observed that a specialization slice contained two specializations of a given procedure, with identical pretty-printed source code. An example of this phenomenon is presented in Fig. 27, where the specialization-slicing algorithm creates two syntactically identical procedures, `p_1` and `p_2`, from the original procedure `p`. The stack-configuration slice of the program contains different variants of `p` that have different sets of formal-out vertices (where the formal-out vertices are the only differences in the two variants). In Fig. 27(a), both calls to procedure `p` are in the stack-configuration slice with respect to line (5). The two calling contexts of `p` differ in that the slice does not contain the formal-out vertex for `g` at the second call-site. (Thus, the underlying cause of the phenomenon is similar to reason behind the specializations discussed in §9.2.3.) However, no elements of `p` can be removed in either calling context because in each call the updated value of `g` affects the input to `printf`.

Considering all 1,869 (= 1,698 + 156 + 15) procedures that were specialized into two or more versions in our experiments, 273 of them (or 14.6%) had a pair of syntactically equal versions. Groups of three or more syntactically equal versions were not observed, although some procedures were specialized into two syntactically equal versions (with different sets of formal-out vertices), plus one other distinct version.

This phenomenon is not a counter-example to the minimality of our specialization-slicing algorithm (§5.1). In particular, the coarsest partition under Defn. 3.6 of the stack-configuration slice of Fig. 27(a)'s SDG places the two variants of procedure p in different partition-elements. Moreover, the transformation of a procedure into two or more specialized versions can be useful in the context of program optimization: an absent formal-out vertex represents a variable that is not live at the exit of the procedure variant, and thus the different sets of live variables at the different calling contexts for p present different opportunities to optimize the body of p. For instance, because g is not live at the exit of p_2, an optimizer might choose to optimize p_2 as follows:

```
void p_2() {
  printf("%d", g+1);
}
```

However, p_1 could not be modified in the same way because it needs to increment the value of g.

Forgács and Gyimóthy [1997] have proposed a version of SDGs in which a formal-out vertex for a variable v appears in the PDG for a given procedure only if v is live on exit from the procedure. With their SDG-construction method, even though p_1 and p_2 are textually identical, the PDGs created for p_1 and p_2 in the SDG reconstructed from Fig. 27(b) would have different sets of formal-out vertices. (CodeSurfer/C does not implement that construction method, and thus the PDGs created by CodeSurfer/C for p_1 and p_2 in the SDG reconstructed from Fig. 27(b) are identical.)

A final variation on this theme is presented in Fig. 28. The specialization-slice outcome shown in Fig. 28(b) is again due to different usage patterns of formal-out vertices in different calling contexts. However, in this example, p_1 and p_2 are syntactically different because the formal-out in question corresponds to p's return statement (line (5)). The first call to p is retained only for the side-effect of incrementing g, and thus p_1 has return type void (line (8)). The return value from the second call contributes to the value of b in line (15), and thus p_2 has a return statement and its return type is int (lines (3) and (5)).

Pairs of such nearly identical specialized versions—i.e., differing solely on return statements and the return type—occurred in a substantial number of cases in our test suite. Of the 1,869 (= 1,698 + 156 + 15) procedures that had two or more versions produced by specialization, 515 (27.6%) had versions that differed in this way.

10. RELATED WORK

Slicing has been applied to many software-engineering problems [Horwitz and Reps 1992], including program understanding [Jackson and Rollins 1994; Reps and Rosay

(a) Program with differing calling contexts for procedure `p`	(b) Specialization of `p` resulting from slicing at line 15
(1) `int g = 0;`	`int g = 0;`
(2)	
(3) `int p() {`	`int p_2() {`
(4) `  ++g;`	`  ++g;`
(5) `  return g;`	`  return g;`
(6) `}`	`}`
(7)	
(8)	`void p_1() {`
(9)	`  ++g;`
(10)	`}`
(11)	
(12) `int main() {`	`int main() {`
(13) `  int a = p();`	`  p_1();`
(14) `  int b = p();`	`  int b = p_2();`
(15) `  printf("b = %d", b);`	`  printf("b = %d", b);`
(16) `}`	`}`

Fig. 28. (a) Example program for which a procedure is specialized based on whether its return value can affect the variables in the slicing criterion. (b) Specialization slice with respect to the call to `printf` on line (15): the first call to `p` is retained only for the side-effect of incrementing `g`, and thus `p_1` has return type `void`; the return value from the second call contributes to the value of `b` in line (15), and thus `p_2` has a return statement and its return type is `int`.

1995; De Lucia et al. 1996; Field et al. 1995], maintenance [Gallagher and Lyle 1991; Canfora et al. 1994; Lakhotia and Deprez 1998], debugging [Lyle and Weiser 1986; 1987], testing [Binkley 1992; Bates and Horwitz 1993], differencing [Horwitz 1990; Horwitz and Reps 1991], specialization [Reps and Turnidge 1996], and merging [Horwitz et al. 1989]. The literature on program slicing is extensive; literature surveys include [Tip 1995; Binkley and Gallagher 1996; Mund and Mall 2007]. Some of the related work on slicing has already been summarized in §1 and §3.

Binkley et al. [2004] gave declarative semantic specifications for a number of different varieties of slices. Their work uses a formal framework of syntactic and semantic projections [Harman et al. 2003] to unify and relate eight different kinds of slicing criteria for static and dynamic slicing. Binkley et al. [2006] carry this approach further, using the same framework to relate slicing techniques to partial evaluation (for single-procedure programs). In contrast, our work is mainly algorithmic in nature; moreover, because our work shows how automata-theoretic techniques allow explicit manipulations of representations of unrolled SDGs and infinite sets of configurations to be performed, our work brings a new collection of algorithmic techniques to bear on slicing problems. While our experiments used C programs, the principles on which Alg. 1 is based should apply to interprocedural slicing for any language.

In the Weiser's original paper on program slicing [Weiser 1984], he shows that finding semantically minimal slices is, in general, undecidable. Consequently, research on slicing has studied different notions that side-step the undecidability problem. A common approach, which is used in this paper and goes back to the introduction of PDGs for program slicing [Ottenstein and Ottenstein 1984], is that no evaluation or simplification is performed, and the elements of the output slice are all elements from the input program. There has been more recent work by Snelting et al. [2006], Canfora et al. [1994], Fox et al. [2004], Danicic et al. [2005], and Jaffar et al. [2012] that combines slicing-like operations with simplification operations or symbolic execution. Some of that work still uses SDG-like structures.

Conditioned slicing [Canfora et al. 1994; Fox et al. 2004; Danicic et al. 2005] 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. Some of this work merely applies off-the-shelf slicing algorithms for the static-slicing component and hence could adopt the specialization-slicing algorithm presented in this paper: Danicic et al. [2005] use the static-slicing algorithm of Ouarbya et al. [2002]; Fox et al. [2004] use an implementation of Weiser's algorithm.

Harman and Danicic [1997] studied what they call *amorphous slicing*, which drops the requirement of syntax preservation. What distinguishes an amorphous slice is merely that the number of vertices in the slice's control-flow graph (CFG) is no greater than the number of vertices in the original program's CFG. We have no such restriction, and in fact a slice created by Alg. 1 could be larger overall than the original program. Consequently, a slice returned by Alg. 1 does not always qualify as an amorphous slice; however, our work is in somewhat the same spirit—especially the material in §7.2 on procedure pointers and indirect calls.

Binkley [1997] defined the notion of a *calling-context slice*: in addition to an SDG vertex v, a slicing criterion for a calling-context slice includes a calling context c, where c is a *single* stack configuration. A calling-context slice includes the vertices on which v depends in calling context c, but excludes vertices if they either have no influence on v or can only influence v in calling contexts other than c. Krinke [2004] identified a small degree of imprecision in Binkley's algorithm, and gave a more precise algorithm for calling-context slicing.

The PDS-based slicing algorithm that we use in this paper generalizes calling-context slicing in two ways:

(1) The use of a PDS allows slicing with respect to a slicing criterion that consists of a *regular language of configurations*, which can be an infinite set.

(2) The answer is reported in the form of an automaton that represents a regular language of configurations; the answer language can also be an infinite set.

Calling-context slicing addresses only the simpler case in which

—the language of stack configurations in each slicing criterion is restricted to be a singleton set $\{(\!|v, c|\!)\}$; and

—the answer is reported as merely a finite set of SDG vertices, rather than as a possibly infinite set of configurations.

Moreover, Alg. 1 relies on generalization (2) in a crucial way. The core task in specialization slicing is to find a finite partition of the potentially infinite set of configurations in a stack-configuration slice (Defn. 3.6). The latter set is captured exactly by the automaton $A1$ created in line 3 of Alg. 1. The fact that we can manipulate $A1$ explicitly allows us to convert $A1$ into automaton $A6$, from which we extract a solution to the configuration-partitioning problem (Defn. 3.6).

Our work is not the first to apply ideas and algorithms from the model-checking community to program slicing. Hong et al. [2005] introduced *abstract slicing*, which starts with a program model obtained via predicate abstraction [Graf and Saïdi 1997] (and hence is a more precise abstraction of a program than the program's control-flow graph). The dependence relations used for abstract slicing are generalizations of the standard notions of flow dependence and control dependence, suitable for use with their more refined model. They also find slices by using NuSMV—a model checker for the branching-time temporal logic CTL—to compute the least fixed point of a CTL formula that encodes the slicing problem. This approach enables them to avoid having to construct an explicit dependence graph, which could be very large as more predicates are used. Sun et al. [2010], [2011] use pushdown systems to perform information-flow-control analysis, a problem that is closely related to program slicing.

The history of the model-checking problem for PDSs is recounted by Bouajjani et al. [2000, §6], who credit Büchi with establishing the foundational result ([Büchi 1964] and [Büchi 1988, Ch. 5]), and point out that it was rediscovered several times (e.g., by Caucal [1992] and Book and Otto [1993]). By encoding SDGs as PDSs (Defn. 2.6 and Fig. 5), we are able to harness this powerful technology—and make use of an existing implementation—rather than having to "rediscover" and reimplement it for SDGs.

Minimal reverse-deterministic FSAs were used by Gupta [1994] 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 [2010] 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.

In §1, we discussed how our work was inspired by the notion of polyvariant specialization in partial evaluation [Jones et al. 1993, p. 370]. In addition to specialization slicing and feature removal, the automaton-based analysis developed in this paper also has an application in partial evaluation. So-called "off-line" partial evaluators perform a preliminary phase of *binding-time analysis* to identify, for each procedure, the different patterns of "static" (i.e., "supplied") and "dynamic" (i.e., "delayed") parameters that can arise during the second specialization phase [Jones et al. 1993, p. 84]. Such binding-time information can be obtained using the techniques developed in this paper. In particular, if one performs a forward stack-configuration slice of a program P (via *Poststar*) with respect to P's dynamic inputs, and then creates a minimal reverse-deterministic automaton A from the answer automaton, A contains explicit information about the set of possible patterns of "dynamic" bindings that can arise for different stack configurations—and hence provides implicit information about the complementary patterns of "static" bindings. That is, the method just sketched out is an algorithm for *polyvariant binding-time analysis*.

What is striking about the automaton-based algorithm is that it is quite different from a more standard analysis approach that builds up, for each procedure, a function from the set of input binding-time patterns to a return binding-time

[Bulyonkov 1993]. In essence, such an analysis propagates *tuples* of binding-times. In contrast, the automaton-based approach propagates binding-time facts *independently* in the potentially infinite unrolled SDG; however, each independent fact is indexed by a stack-configuration, which—via the construction of a minimal reverse-deterministic automaton—is used to coalesce related "independent" binding-time facts.

11. CONCLUSIONS

In this paper, we introduced a new variant of program slicing, called *specialization slicing*, and presented an algorithm for the specialization-slicing problem that creates an optimal output slice (Cor. 5.1). At an intuitive level, the claim that Alg. 1 produces an optimal answer follows from two properties: (i) the output slice created by Alg. 1 is minimal in the sense defined by Defn. 3.6, and (ii) each element replicated by Alg. 1 is necessary for the output slice to capture one of the input program's specialized patterns of dependences.

Given the level of sophistication of the pushdown-system machinery used in Alg. 1, it is natural to wonder whether some simple SDG-based algorithm solves the specialization-slicing problem. In working on the problem, we considered numerous such algorithms, one of which is discussed in §1.1.2. The flaws in such attempts motivated us to investigate the fundamental principles underlying specialization slicing. These principles are presented in §3, where we formalized specialization slicing in terms of a partitioning problem on the unrolled SDG (Eqn. (3) and Defn. 3.6) and formulated declarative conditions for soundness and completeness (Defn. 3.5).

Because the unrolled SDG is, in general, an infinite graph, in §4 we encoded the SDG as a pushdown system. This approach allowed us to represent finitely the infinite sets of objects that are needed to solve the partitioning problem. Moreover, it allowed us to bring to bear the repertoire of symbolic techniques that had already been developed for PDSs in the model-checking community [Bouajjani et al. 1997; Finkel et al. 1997]. With this powerful machinery at our disposal, we showed how to obtain the desired answer by performing just a few simple automata-theoretic operations (cf. lines 3–8 of Alg. 1).

There is a conceptual advantage to using PDS technology to work with the *unrolled* SDG rather than the SDG. Instead of working with vertices in a finite graph (i.e., the SDG) that are reachable along *context-free paths*, PDS technology allows one to work directly with $\mathcal{P}$*-configurations* that are reachable along *paths* in an infinite graph (i.e., the unrolled SDG). With the latter approach, there is a more direct correspondence between the concepts in the declarative specification of the desired result—Eqn. (3) and Defn. 3.6—and the operations performed by Alg. 1.

Although Alg. 1 can, in the worst case, exhibit exponential behavior, our experiments suggest that exponential behavior does not arise in practice: no procedure had more than four specialized versions, and the vast majority of procedures (90.8%) had just a single version (see Fig. 20). Moreover, worst-case exponential behavior of operations like automaton determinization also does not seem to arise in practice.

In §8, we showed that specialization slicing, in conjunction with forward stack-configuration slicing, provides a solution to the feature-removal problem for multi-procedure programs. While it was previously known how to solve the feature-removal problem for single-procedure programs, no algorithm was known for multi-procedure programs.

The paper by Horwitz et al. [1990] on closure slicing of SDGs is one of the "standard" papers cited about program slicing, and has served as the jumping-off point for much subsequent work on slicing in the programming-languages and software-engineering communities. In §7.4, we presented a PDS-based analogue of the three-phase algorithm of Horwitz et al., which was 2.9 times faster than the one-phase PDS-based algorithm on the larger examples. Many researchers have used SDG-like data structures, as well as algorithms similar to the one given by Horwitz et al., and thus it may be possible to carry over the principles used in Algs. 1 and 2 to such work.

Although parts of our implementation are specific to CodeSurfer/C, the principles can be used to perform specialization slicing on programs written in any language for which one has an SDG-like intermediate representation.

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. We thank the referees for their extensive comments on the submission, which have helped us to improve the presentation.

REFERENCES

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

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

ANSI C 2005. ISO/IEC 9899:TC2. "www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf".

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

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

Binkley, D. 1993. Precise executable interprocedural slices. *LOPLAS 2*, 31–45.

Binkley, D. 1997. Semantics guided regression test cost reduction. *TSE 23,* 8, 498–516.

Binkley, D. 2012. Personal communication.

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

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

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

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

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

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

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

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

Bulyonkov, M. 1993. Extracting polyvariant binding time analysis from polyvariant specializer. In *Part. Eval. and Semantics-Based Prog. Manip.*

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

Caucal, D. 1992. On the regular structure of prefix rewriting. *Theor. Comp. Sci. 106,* 1, 61–86.

Chaudhuri, S. 2008. Subcubic algorithms for recursive state machines. In *POPL.*

Codesurfer. CodeSurfer system. www.grammatech.com/products/codesurfer.

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

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

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

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

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

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

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

Forgács, I. and Gyimóthy, T. 1997. An efficient interprocedural slicing method for large programs. In *Int. Conf. on Softw. Eng. and Knowledge Eng.*

Fox, C., Danicic, S., Harman, M., and Hierons, R. 2004. ConSIT: A fully automated conditioned program slicer. *SPE 34,* 1.

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

Giacobazzi, R. and Mastroeni, I. 2003. Non-standard semantics for program slicing. *HOSC 16,* 4, 297–339.

Godefroid, P., Klarlund, N., and Sen, K. 2005. DART: Directed automated random testing. In *PLDI.*

Graf, S. and Saïdi, H. 1997. Construction of abstract state graphs with PVS. In *CAV.*

Gupta, A. 1994. Inductive boolean function manipulation: A hardware verification methodology for automatic induction. Ph.D. thesis, Carnegie Mellon Univ. Tech. Rep. CMU-CS-94-208.

Harman, M., Binkley, D., and Danicic, S. 2003. Amorphous program slicing. *J. Systems and Software 68,* 1, 45–64.

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

Hong, H., Lee, I., and Sokolsky, O. 2005. Abstract slicing: A new approach to program slicing based on abstract interpretation and model checking. In *SCAM.*

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

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

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

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

Horwitz, S. and Reps, T. 1991. Efficient comparison of program slices. *Acta Inf. 28,* 8, 713–732.

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

HORWITZ, S., REPS, T., AND BINKLEY, D. 1990. Interprocedural slicing using dependence graphs. *TOPLAS 12,* 1 (Jan.), 26–60.

HUTCHINS, M., FOSTER, H., GORADIA, T., AND OSTRAND, T. 1994. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Int. Conf. on Software Eng.* 191–200.

JACKSON, D. AND ROLLINS, E. 1994. A new model of program dependences for reverse engineering. In *FSE.*

JAFFAR, J., MURALI, V., NAVAS, J., AND SANTOSA, A. 2012. Path-sensitive backward slicing. In *SAS.*

JONES, N., GOMARD, C., AND SESTOFT, P. 1993. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall International.

KIDD, N., LAL, A., AND REPS, T. 2007. WALi: The Weighted Automaton Library. www.cs.wisc.edu/wpis/wpds/download.php.

KRINKE, J. 2004. Context-sensitivity matters, but context does not. In *SCAM.*

KUCK, D., KUHN, R., LEASURE, B., PADUA, D., AND WOLFE, M. 1981. Dependence graphs and compiler optimizations. In *POPL.* 207–218.

LAKHOTIA, A. AND DEPREZ, J. 1998. Restructuring programs by tucking statements into functions. *Inf. and Softw. Tech. 40,* 11–12, 677–690.

LYLE, J. AND WEISER, M. 1986. Experiments on slicing-based debugging tools. In *Conf. on Empirical Studies of Programming.*

LYLE, J. AND WEISER, M. 1987. Automatic program bug location by program slicing. In *Int. Conf. on Comp. and Applications.*

MUND, G. AND MALL, R. 2007. Program slicing. In *The Compiler Design Handbook*, 2nd. ed. CRC Press, Chapter 14.

OpenFST 2012. OpenFst library. www.openfst.org.

OTTENSTEIN, K. AND OTTENSTEIN, L. 1984. The program dependence graph in a software development environment. In *PSDE.*

OUARBYA, L., DANICIC, S., DAOUDI, M., HARMAN, M., AND FOX, C. 2002. A denotational interprocedural program slicer. In *WCRE.*

REPS, T. 1998. Program analysis via graph reachability. *Inf. and Softw. Tech. 40,* 11–12, 701–726.

REPS, T., HORWITZ, S., SAGIV, M., AND ROSAY, G. 1994. Speeding up slicing. In *FSE.* 11–20.

REPS, T. AND ROSAY, G. 1995. Precise interprocedural chopping. In *FSE.*

REPS, T. AND TURNIDGE, T. 1996. Program specialization via program slicing. In *Proc. of the Dagstuhl Seminar on Partial Evaluation.*

SCHWOON, S. 2002. Model-checking pushdown systems. Ph.D. thesis, Technical Univ. of Munich, Munich, Germany.

SEBEJ, J. 2010. Reversal of regular languages and state complexity. In *Conf. on Theory and Practice of Inf. Technologies.*

SIR. Software-artifact Infrastructure Repository. sir.unl.edu/portal/index.php.

SNELTING, G., ROBSCHINK, T., AND KRINKE, J. 2006. Efficient path conditions in dependence graphs for software safety analysis. *TOSEM 15,* 4, 410–457.

SUN, C., TANG, L., AND CHEN, Z. 2010. Enforcing reactive noninterference with reachability analysis. In *Int. Conf. on Quality Software.*

SUN, C., TANG, L., AND CHEN, Z. 2011. Enforcing reactive noninterference with reachability analysis. In *Int. Conf. on Inf. Tech. New Generations.*

TIP, F. 1995. A survey of program slicing techniques. *JPL 3,* 3.

WEISER, M. 1984. Program slicing. *TSE 10,* 4 (July), 352–357.

Wikipedia: Output-Sensitive Algorithm 2014. Output-sensitive algorithm. en.wikipedia.org/wiki/Output-sensitive_algorithm, Jan. 5, 2014.

YANNAKAKIS, M. 1990. Graph-theoretic methods in database theory. In *Symp. on Princ. of Database Syst.* 230–242.

A. CORRECTNESS OF ALG. 1

Theorem 4.12. *Automaton $A6$ created in line 8 of Alg. 1 is a minimal reverse-deterministic automaton.* □

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 a minimal deterministic automaton for the reversed language of configurations in the stack-configuration slice. That is, $A4$ is a minimal deterministic FSA 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 removeEpsilonTransitions 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.

As mentioned in the remark in §4.3, 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($A4$)" could 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, $A5$ would have no ϵ-transitions, and thus removeEpsilonTransitions would have no effect (i.e., $A6 = A5$). Because $A4$ is a minimal deterministic FSA, $A5$ and $A6$ would both be MRD.

In our implementation, lines 4–8 are implemented with OpenFST FSAs [OpenFST 2012]. The reverse operation in OpenFST introduces a dummy initial state with an ϵ-transition. Thus, the implementation calls removeEpsilonTransitions, which removes the single, initial ϵ-transition from $A5$ to create $A6$, which is MRD. □

Theorem 4.13. *A solution to the configuration-partitioning problem (Defn. 3.6) is encoded in the structure of automaton $A6$ created in line 8 of Alg. 1.* □

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 FSA, 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 $V_{q_k} \overset{\text{def}}{=} \{v \mid (q_k, v, acc) \in \text{Transitions}(A4)\}$. The set of configurations in the partition-element associated with q_k is, therefore,

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

The partition is defined by Part $\overset{\text{def}}{=} \{\text{PE}_{q_k} \mid q_k \in \text{States}(A4)\}$. (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. 3.6 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 4.14. *Alg. 1 is a sound and complete algorithm for 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 configurations in the closure slice with respect to C of the unrolling of S. Let $(\!| v, u |\!)$ 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 u, 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 $(\!| v', u' |\!)$ such that $M_C(u') = u$.

Thus, there is a configuration $(\!| v', u' |\!)$ in G_R such that $M_C((\!| v', u' |\!)) = (\!| v, u |\!)$.

Soundness: Let $(\!| v', u' |\!)$ 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 u' 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, u' must correspond (via M_C) to some word in $L(q)$. Consequently, $L(A6)$ has some configuration $(\!| v, u |\!)$ such that $M_C(u') = u$.

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

Corollary 4.15. *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. 4.14, $M_C(G_R)$ 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. $\square$