INTERPROCEDURAL ANALYSIS AND THE VERIFICATION OF CONCURRENT PROGRAMS

by

Akash Lal

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences Department)

at the

UNIVERSITY OF WISCONSIN–MADISON

2009

To *mom* and *dad* ...

ACKNOWLEDGMENTS

First of all, I would like to thank my family for their love and constant support. They gave me the strength to complete my graduate studies. I am, as ever, indebted to my parents, Dr. Ashok Kumar and Dr. Vinod Lal. They encouraged me to pursue a doctorate degree. I would like to thank my brother Vikrant for teaching me the basics of computer science in the sixth grade and instilling a curiosity in me for the subject. These have brought me this far in the area. I would like to thank my wife Ira for making even the difficult times seem easy.

I am especially grateful to my advisor, Prof. Thomas Reps, for his guidance and support, without which this dissertation would not have been possible. He has always encouraged me to think creatively, aim higher and to believe in myself. He shared my disappointment when a paper was not accepted; he shared my excitement at a new research idea; and also my stress while facing a tough job market. His commitment and enthusiasm have constantly motivated me. He has patiently helped me develop my writing and speaking skills. I would also like to thank him for taking me along during his sabbatical in Paris. That was an amazing experience.

I would like to thank Prof. Susan Horwitz, Prof. Somesh Jha, Prof. Ben Liblit, and Prof. Parmesh Ramanathan for being on my final defense committee. I would specially like to thank Prof. Susan Horwitz and Prof. Tom Reps for their insightful comments on my dissertation, which have helped greatly to improve the quality of this dissertation.

I would also like to thank Dr. David Melski, my mentor during my internship at GrammaTech. He introduced me to Weighted Pushdown Systems, which I continued to use for most of my dissertation research. The experience I gained during that internship was

invaluable for my research. I would also like to thank my mentors at Microsoft Research, Dr. Sumit Gulwani and Dr. Ganesan Ramalingam, for teaching me about working in research labs.

I would especially like to thank Nick Kidd for sharing an office with me. Our numerous discussions and his constant feedback were instrumental in developing my research work. His friendship helped ease the stress of graduate school, not to mention the countless games of chess that we played in our office. I would also like to thank the PL group: Piramanayagam Arumuga, Gogul Balakrishnan, Evan Driscoll, Matt Elder, Denis Gopan, Junghee Lim, Alexey Loginov, Marina Polishchuk, Cindy Rubio González, and Aditya Thakur for always finding the time to attend my practice talks and give feedback.

In addition, I would like to thank Dr. Tayssir Touili for collaborating with me on a number of projects, and also for helping me obtain a French student visa for my stay in Paris.

I would also like to thank Prof. Ben Liblit for never being short on words of encouragement. Moreover, it was always a pleasure sitting in his class.

My dissertation research was supported by a UW CS Departmental Research Assistantship award, ONR grant N00014-01-1-0796, NSF grant CCF-0524051, NSF grant CCF-0540955, IBM Scholarship, and a Microsoft Graduate Fellowship.

Preface to the Technical Report Version

This technical report is a slightly revised version of my dissertation. It contains a small amount of extra material and minor corrections in Section 7.6. The material is added to fill in some details that were missing in the dissertation.

DISCARD THIS PAGE

TABLE OF CONTENTS

Page

Page

Page

DISCARD THIS PAGE

LIST OF TABLES

DISCARD THIS PAGE

LIST OF FIGURES

ABSTRACT

In the modern world, not only is software getting larger and more complex, it is also becoming pervasive in our daily lives. On the one hand, the advent of multi-core processors is pushing software towards becoming more concurrent, making it more complex. On the other hand, software is everywhere, inside nuclear reactors, space shuttles, cars, traffic signals, cell phones, etc. To meet this demand for software, we need to invest in automated *program-verification* techniques, which ensure that software will always behave as intended.

The problem of program verification is undecidable. A verification technique can only gain a limited amount of knowledge about a program's behavior by reasoning about certain aspects of the program. This dissertation addresses program verification by considering two important features of programs: (*i*) procedures (and procedure calls) and (*ii*) concurrency.

Interprocedural Analysis: An analysis that can precisely handle the procedural aspect of programs is called an *interprocedural analysis*. Procedures are an important feature of most programming languages because they allow for modular design of programs: each procedure is meant to perform a task, and they can be put together to implement more complex functionality. Because procedures serve as a natural abstraction mechanism for developers to organize their programs, an interprocedural analysis can leverage them to enable verification of a larger and more complex programs.

There is a long history of work on interprocedural analysis, including several frameworks that support a variety of different program abstractions, and provide algorithms for analyzing them. The advantage of having a framework is that any program abstraction that fits the

framework can make use of the algorithms for the framework. One such framework, called *Weighted Pushdown Systems* (WPDSs), was the subject of the research reported on in this dissertation.

The dissertation makes several contributions to interprocedural analyses that are based on WPDSs:

- We define the Extended WPDS (EWPDS) model, which removes a crucial limitation of WPDSs by providing a convenient abstraction mechanism for local variables of a procedure. Using EWPDSs, it is possible to model a program's behavior more precisely than with WPDSs. In our work, we used EWPDSs for checking properties of Boolean programs; computing affine relations in x86 programs; building debugging tools; computing alias pairs in programs with single-level pointers; and for checking properties of concurrent programs (where EWPDSs are used to model individual threads).

- We use graph-theoretic algorithms to speed up the analysis algorithms for WPDSs and EWPDSs. This results in immediate speedup in all of the applications based on these models without requiring any tuning for a particular application. The speedups ranged from 1.8× to 3.6×.

- We show how to answer more expressive queries on EWPDSs, such as computing the set of all error traces in the model, called an *error projection*. This enables faster verification.

Concurrency: The advent of multi-core processors is pushing software to become more concurrent. Concurrent programs are not only difficult to write, but are also difficult to analyze and verify. One reason is that the interprocedural analysis of concurrent programs is undecidable, even when all of the other aspects of a programs (like the program heap, non-scalar variables, pointers, etc.) are abstracted away. As a result, most verification tools do not mix interprocedural analysis with concurrency, i.e., tools that analyze concurrent programs give up on precise handling of procedures. This is unfortunate because precise handling of procedures has proven to be very useful for the analysis of sequential programs.

The contribution of our work is to give techniques for interprocedural analysis for concurrent programs. We show that one does not have to design new algorithms for concurrent programs; instead, it is possible to automatically extend most interprocedural analysis techniques for sequential programs to perform interprocedural analysis of concurrent programs.

As mentioned earlier, interprocedural analysis of concurrent programs is undecidable. We sidestep the undecidability by placing a bound on the number of *context switches*, i.e., we bound the number of times control is transferred from one thread to another. We call the analysis of concurrent programs under a bound on the number of context switches *context-bounded analysis* (CBA).

CBA is an interesting avenue of research that has attracted a lot of attention recently because empirical evidence suggests that many concurrency-related bugs can be found in a few context switches. Moreover, CBA was shown to be decidable for finite-data abstractions.

The dissertation makes two important contributions to interprocedural analysis of concurrent programs:

- We show that if each thread is modeled using a WPDS then CBA is decidable, and also give an algorithm for performing CBA. This represents the first step towards providing a general model for concurrent programs that can be used to perform interprocedural analysis.

- We show that, given a concurrent program P and a context bound K, one can create a sequential program P_K such that the analysis of P_K is sufficient for CBA of P under the bound K. This reduction is a source-to-source transformation, and requires no assumptions nor extra work on the part of the user, except for the identification of thread-local data. We implemented this technique to create the first known implementation of CBA. Using this tool, we conducted a study on concurrent Linux drivers to show that most bugs could not only be found in a few context switches, but, compared to previous approaches, they could be found must faster using our approach.

Chapter 1

Introduction

In the modern world, not only is software getting larger and more complex, it is also becoming pervasive in our daily lives. On the one hand, the advent of multi-core processors is pushing software towards becoming more concurrent, making it more complex. On the other hand, software is everywhere, inside nuclear reactors, space shuttles, cars, traffic signals, cell phones, etc. In meeting this demand for software, ensuring reliability is one of the major bottlenecks. Any approach for ensuring reliability that requires a substantial manual effort is not going to suffice in the future, and we need to invest in automated *program verification* techniques.

The goal of program verification is to inspect program behavior, and then conclude if some program execution can be faulty, or if there are no faulty executions. One key ingredient needed for program verification is a *property* that classifies if a program execution is faculty or not. The property of interest can, for example, state that there are no null-pointer dereferences or memory-safety violations, or be more functional and state that the result of executing a procedure on an array is that it sorts the array. Thus, in program verification, given a program P and a property A, one has to answer the question: "is there an execution of P that violates property A?". The answer can be in the form of the violating execution, or a proof that the property holds for all executions of P.

Program verification is undecidable, i.e., no single tool can always give an answer to the above question for given any program and any property. Consequently, program-verification research focuses on developing algorithms and tools that can only infer a few aspects of

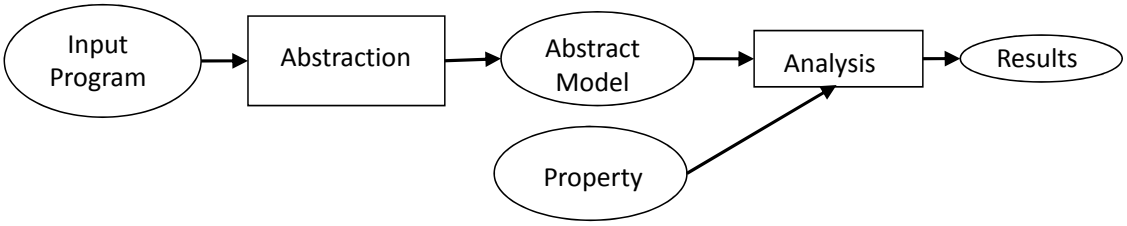


Figure 1.1 Typical design of verification tools based on abstraction.

a program's behavior. Such tools find answers for as many properties as possible under their limited knowledge of the program's behavior. Most tools are based on the notion of *abstraction*.

1.1 The Need for Abstraction

A common organization of a verification tool is shown in Fig. 1.1. It has two main phases. The first is an abstraction phase that creates an *abstract model* of a given program. The set of behaviors of this model is a superset of the set of behaviors of the original program. In other words, the abstract model over-approximates the original program. An example is discussed in Section 1.1.2.

The second phase is the analysis phase, which checks if the abstract model can violate the property of interest. This check, in essence, is to see if the set of behaviors of the abstract model is disjoint from the set of bad behaviors described by the property, as depicted in Fig. 1.2. If so, one can conclude that the original program has no bad behaviors. Otherwise, some of the behaviors in their intersection are reported (which may or may not be actual behaviors of the original programs — see Fig. 1.4).

The reason for the separation of the two phases is that the set of behaviors of a program, in general, is not computable, but is computable for the abstract model.[1] In the model checking community, the first phase is called *model extraction* and the second phase is called *model checking*.

[1]In some cases, it may not be computable even for the abstract model, in which case the analysis phase further over-approximates the set of behaviors of the abstract model.

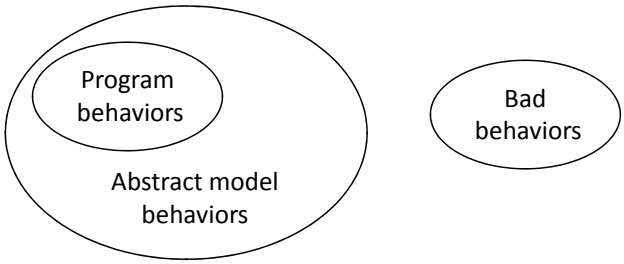


Figure 1.2 Proving the absence of bugs using abstraction.

Note that what we have described here is the common approach taken for program verification in which the set of program behaviors is over-approximated. The approach taken by some bug-finding tools is to under-approximate the set of program behaviors. Then any intersection between this set and the set of bad behaviors described by the property immediately indicates the presence of a bug. In this dissertation, we will mostly stick to the verification approach.

1.1.1 Abstraction Refinement

It is possible that the chosen abstraction is too coarse to prove that a property holds, i.e., the set of bad behaviors is not disjoint from the set of behaviors of the abstract model, but is disjoint from the set of behaviors of the program. In this case, *abstraction refinement* can be used: multiple abstractions with increasing precision are used until a bug is found or the property is proved to hold. In Fig. 1.4, abstractions A_1 and A_2 are not precise enough to prove that the property holds, but abstraction A_3 suffices.

The design of a verification tool based on abstraction refinement is shown in Fig. 1.3. The result of the analysis phase is used to refine the abstraction when necessary: if the current abstract model does not violate the property, then the analysis phase concludes that the program is correct; otherwise, it produces a behavior of the model, called a *counterexample*, that violates the property. If this is also a behavior of the original program, then a bug has been found. Otherwise, the abstraction is refined to produce a model that does not exhibit this behavior, and the process continues. Because verification is undecidable, it is possible

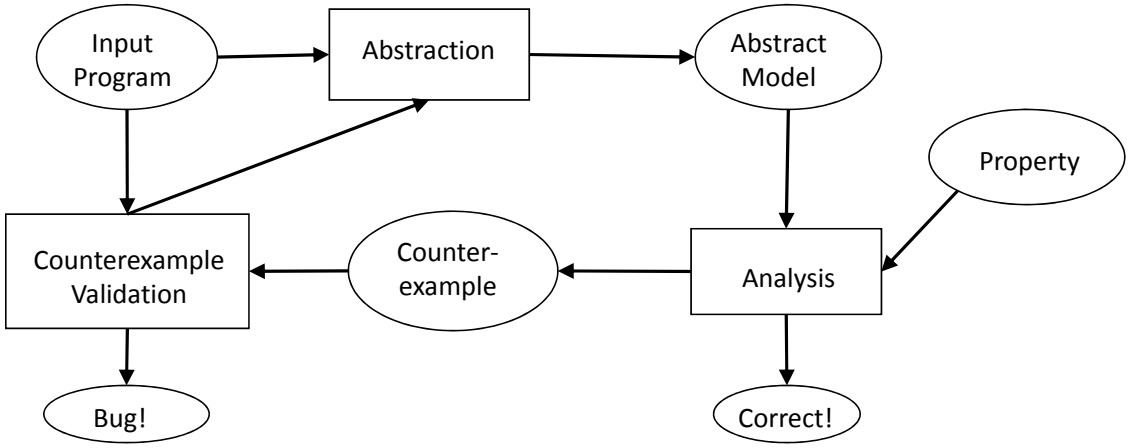


Figure 1.3 Typical design of verification tools based on abstraction refinement.

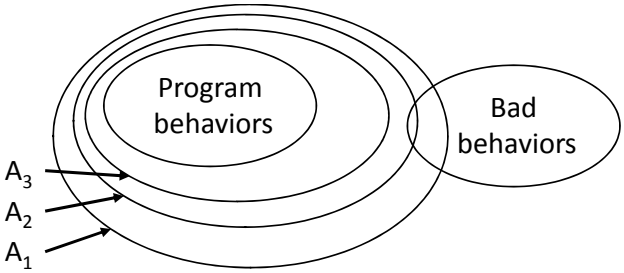


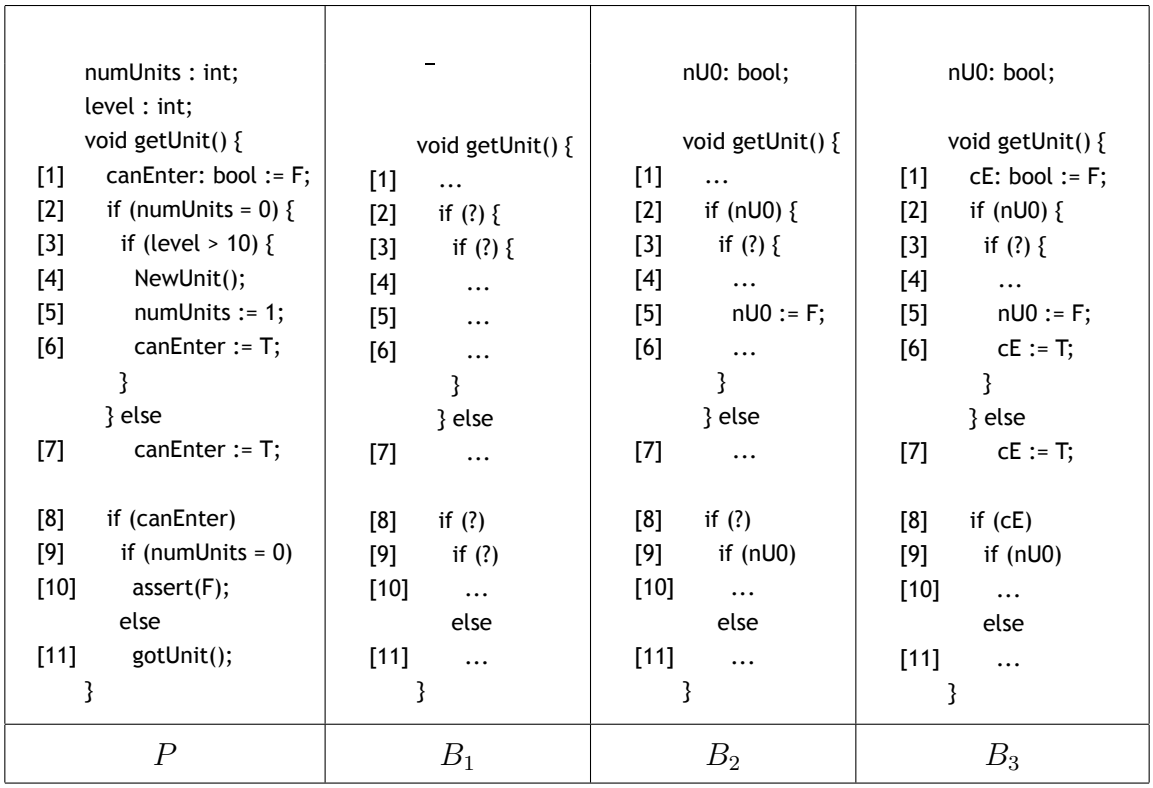
Figure 1.4 Proving the absence of bad behaviors using abstraction refinement.

that the refinement loop may fail to terminate; i.e., the abstraction is made more and more precise, but it always fails to show the absence of a bug, or to produce an actual bug.

1.1.2 Example: Predicate Abstraction and Boolean Programs

In this section, we describe how *predicate abstraction* is used to create abstract models of programs, called *Boolean programs*. We also illustrate how abstraction, as well as abstraction refinement, helps in verification. The examples used in this section are taken from [7], and will be used later in Chapter 5 to illustrate some of the contributions of this dissertation.

Consider the program P shown in the leftmost column of Fig. 1.5. We would like to verify that the assertion shown on line 10 can never fail. Because the assertion will always fail when executed, we essentially have to show that line 10 is never reached in any program execution.



```
numUnits : int;
level : int;
void getUnit() {
[1]   canEnter: bool := F;
[2]   if (numUnits = 0) {
[3]     if (level > 10) {
[4]       NewUnit();
[5]       numUnits := 1;
[6]       canEnter := T;
        }
      } else
[7]       canEnter := T;

[8]   if (canEnter)
[9]     if (numUnits = 0)
[10]      assert(F);
        else
[11]      gotUnit();
    }
```

P

```
void getUnit() {
[1]   ...
[2]   if (?) {
[3]     if (?) {
[4]       ...
[5]       ...
[6]       ...
        }
      } else
[7]       ...

[8]   if (?)
[9]     if (?)
[10]      ...
        else
[11]      ...
    }
```

B_1

```
nU0: bool;

void getUnit() {
[1]   ...
[2]   if (nU0) {
[3]     if (?) {
[4]       ...
[5]       nU0 := F;
[6]       ...
        }
      } else
[7]       ...

[8]   if (?)
[9]     if (nU0)
[10]      ...
        else
[11]      ...
    }
```

B_2

```
nU0: bool;

void getUnit() {
[1]   cE: bool := F;
[2]   if (nU0) {
[3]     if (?) {
[4]       ...
[5]       nU0 := F;
[6]       cE := T;
        }
      } else
[7]       cE := T;

[8]   if (cE)
[9]     if (nU0)
[10]      ...
        else
[11]      ...
    }
```

B_3

Figure 1.5 An example program P and its abstractions as Boolean programs. The "..." represents a "skip" or a no-op.

One simple abstraction of P is the program B_1 in Fig. 1.5. This program only retains the control-flow structure of P and all other data is abstracted away. Consequently, the branch conditions are non-deterministic, meaning that the branch may go either way in an execution. Other program statements are abstracted to a "skip" because the data being manipulated by those statements is not present in B_1. (The abstract model B_1 can also be thought of as the control-flow graph of P.) It is easy to see that B_1 over-approximates P. B_1 is very simple to analyze; however, it fails to show that line 10 is unreachable because there is an execution of B_1 that reaches line 10.

Program B_2 is a more precise model of P, as compared to B_1. It retains the control structure of P, but additionally, it keeps track of the value of the predicate {`numUnits = 0`} using the Boolean variable `nU0` as follows: if at some point in the execution of P, the

predicate {`numUnits = 0`} holds (does not hold), then the value of `nU0` in the corresponding execution of B_2 will be true (false). B_2 is still an over-approximation of P because some branch conditions cannot be decided using just the value of `nU0`. However, line 10 is reachable in B_2 as well.

Program B_3 is an even more precise more of P, as compared to B_1. It keeps track of two predicates: {`numUnits = 0`} (using variable `nU0`) and {`canEnter = T`} (using variable `cE`). B_3 is an over-approximation of P because the variable `level` is still abstracted away. Line 10 is not reachable in B_3, which proves that the assertion holds for P.

The process of iterating through the abstract models B_1, B_2, and B_3 is an example of how abstraction refinement is used.[2] Each of these three models are Boolean programs, which are defined as imperative programs, possibly with procedure calls, and only Boolean variables or fixed-size vectors of Boolean variables (and no heap). The process of creating Boolean programs from ordinary (executable) programs by keeping track of certain predicates is called *predicate abstraction*.

1.2 Challenges in Verification of Programs

The previous section gave an example for the abstraction phase of program verification. In this section, we discuss the analysis phase. The design of the analysis phase depends heavily on the kind of abstract model that is created. We discuss two features of programs that are important to retain in abstract models, but also pose challenges for the analysis phase. The first feature is procedures and procedure calls. An analysis that can precisely handle the procedural aspect of programs is called an *interprocedural analysis*. The second feature is concurrency.

1.2.1 Interprocedural Analysis

Procedures are an important feature of most programming languages because they allow for modular design of programs: each procedure is meant to perform a task, and they can

[2] See [7] for a description of how the SLAM tool systematically creates these models.

be put together to implement more complex functionality. Because procedures serve as a natural abstraction mechanism for developers to organize their programs, it is important that they be retained in the abstract model. However, this makes designing the analysis for the model more challenging.

The difficulty posed by interprocedural analysis is that it requires precise reasoning about the program's runtime stack, which can be unbounded in size because of recursion. This induces an infinite control-state space, even for Boolean programs (i.e., when there is no heap and all variables are Booleans or vectors of Booleans). Thus, straightforward techniques of enumerating the state space of the model do not work in the presence of recursion. Even in the absence of recursion, the state space of a model is exponential in the maximum call depth that can arise in an execution of the model.

We now describe some of the common ways of approximating analysis of programs with multiple procedures, and show how they fail to prove even very simple properties of programs. (For ease of discussion, we consider the analysis of C programs directly, instead of an abstract model.)

Consider the program shown in Fig. 1.6(a). It consists of a single recursive procedure `foo` that manipulates the array `arr`. This procedure is intended to operate in a multithreaded environment in which other threads may also be accessing `arr`. The programmer intends this procedure to be free of data races, i.e., no two threads should be allowed to access `arr` simultaneously. This is enforced in the program by having the same mutex protect accesses to the same array element (`m[i]` protects `arr[i]`). Proving this invariant requires an interprocedural analysis because one must reason about local variables stored on the stack to establish the invariant that in any execution, for all activation records in the runtime stack, $\texttt{l}_1 == \texttt{m[l}_2\texttt{]}$. One this is established, it is easy to conclude that $\texttt{arr[l}_2\texttt{]}$ is always protected by $\texttt{m[l}_2\texttt{]}$.

There are two common ways of approximating interprocedural analysis of programs. The program from Fig. 1.6(a) shows that neither is sufficient. The first option is to inline procedures (similar to a compiler's function-inlining optimization) until only one procedure

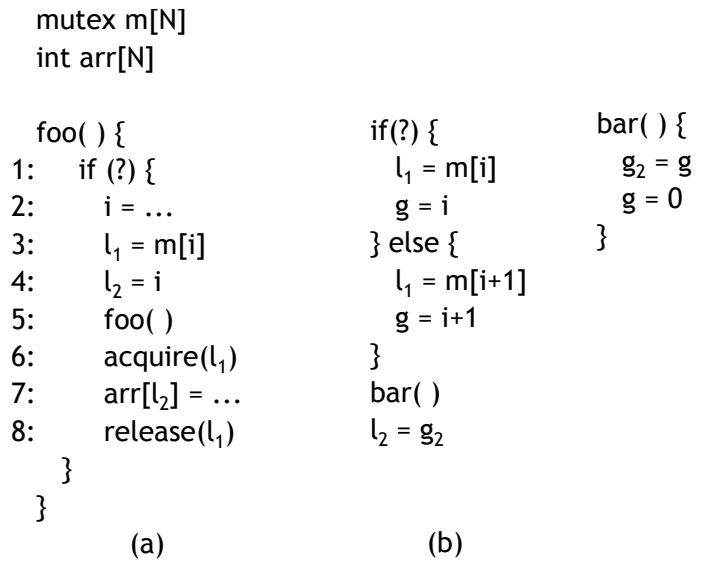


Figure 1.6 An example program. In (a) $\texttt{l}_1$and $\texttt{l}_2$are local variables; in (b) $\texttt{g}$ and $\texttt{g}_2$ are global variables.

remains in the program. Because `foo()` is recursive, inlining does not help here: one can never get to the point where only a single procedure remains.

The other option is to "short-circuit" an invariant on local variables across a procedure call, i.e., project out the local variables from any invariant I that holds before a procedure call to obtain an invariant I_l on only the local variables, and then assert that I_l holds after the procedure call. (For this discussion, we assume that local variables of a procedure cannot be accessed by any called procedure.) Such an approach would be sufficient for this example: it would establish that ($\texttt{l}_1\texttt{== m[l}_2\texttt{]}$) holds before the recursive call to `foo()`; hence, it must also hold after the call because $\texttt{l}_1$and $\texttt{l}_2$are local variables and `m` is not modified in `foo()`. This approach is only a heuristic and may lead to imprecision when the intermediate invariant involves global variables that can possibly be modified by a called procedure. Assume that lines 3 and 4 are replaced by the snippet of code in Fig. 1.6(b). Before the call to `bar()`, the invariant is $I_1 = (\texttt{l}_1\texttt{== m[g]})$, whereas after the call, it becomes $I_2 = (\texttt{l}_1\texttt{== m[g}_2\texttt{]})$. Short-circuiting the invariant I_1 across the call to `bar()` produces an invariant that says that $\texttt{l}_1$equals *some* entry in `m`. This information is insufficient to conclude that the program

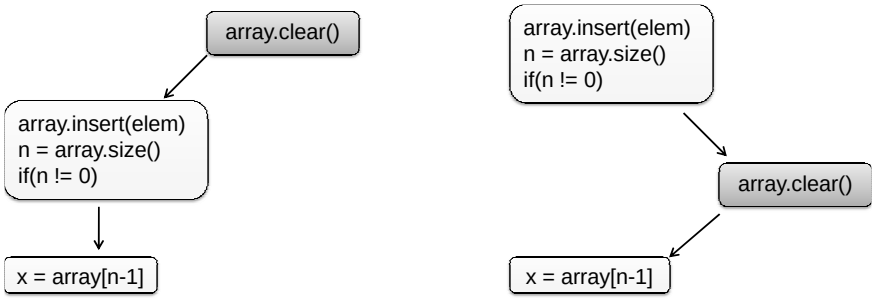


Figure 1.7 Two different interleavings of the same program trace. The first one runs correctly, while the second one may crash because the memory dereference "array$[n-1]$" is out of bounds.

behaves correctly. Establishing I_2 after the call to `bar()` requires an interprocedural analysis that tracks the invariant between $\texttt{l}_1$and `g` through `bar()`.

1.2.2 Analysis of Concurrent Programs

As mentioned previously, the advent of multi-core processors is pushing software to become more concurrent. Concurrent programs are not only difficult to write, but are also difficult to analyze and verify.

In industry, the most prevalent way of finding bugs in programs is testing, where programs are executed under fixed input for which the output is known a priori. If executing a program fails to produce the desired output, then there must be a bug in the program. Compared to verification, testing has the disadvantage of being incomplete: finding bugs depends crucially on choosing the right set of inputs for the program. Furthermore, the presence of concurrency greatly increases this incompleteness because it adds non-determinism to the program. Even under fixed input, a program can have a huge number of behaviors depending on the interleaving that occurs between different threads. For example, see Fig. 1.7. Because the interleavings are not in the control of the programmer, bugs that only arise on specific interleavings are especially hard to find. This motivates the need for verification tools that can not only prove properties for all program inputs, but also prove them for all interleavings that may happen during program execution.

One may wonder why the analysis of concurrent programs is considered harder than the analysis of sequential programs. The answer is obvious in the case of testing because sequential programs can exhibit only one behavior with a fixed input, but concurrent programs may exhibit several behaviors. However, in terms of verification, the answer is not that obvious because the analysis of both sequential and concurrent programs has to consider a possibly unbounded number of behaviors anyway. Also, in general, the analysis of both sequential and concurrent programs is undecidable.

The answer lies in the way we do verification. As mentioned before, verification has two main steps: abstraction to an abstract model, and then an analysis of the model. The complication introduced by concurrency is that even with very simple abstract models, the presence of concurrency makes their analysis computationally expensive.

For instance, the analysis of sequential Boolean programs can be carried out in time linear in the size of the program (but exponential in the number of variables). However, when, for a concurrent program with procedures, each thread is abstracted to a Boolean program, the analysis is undecidable, even for two threads. The situation is similar even in the absence of procedures: the analysis of concurrent Boolean programs without procedures is PSPACE-complete, i.e., expected to have a running time exponential in the size of the programs (as opposed to linear in the case of sequential Boolean programs).

This result shows that a finite abstraction of program data (into Boolean variables) is not sufficient to design effective verification algorithms for concurrent programs. There is hope if the program control is abstracted, i.e., if the procedures are abstracted so that the resulting abstract model only has a single procedure, then the abstract model can be analyzed precisely (albeit in exponential time). Because of this result, verification tools give up on precise handling of procedures while dealing with concurrent programs; i.e., they do not mix interprocedural analysis with concurrency. This is unfortunate because precise handling of procedures has proven to be very useful for analysis of sequential programs.

1.3 Contributions and Organization of the Dissertation

The dissertation makes several contributions in two main directions. First, it gives new algorithms and techniques for interprocedural analysis of sequential programs (Chapters 3, 4, and 5). Second, it shows how interprocedural analysis can be carried out in the presence of concurrency (Chapters 6 and 7). Background material that our work builds upon is covered in Chapter 2.

1.3.1 New Technology for Sequential Programs

It is important to choose an expressive abstract model so that it can retain enough of the important aspects of the program to be able to prove the desired property. Boolean programs can encode models with infinite state spaces, but the infiniteness can only come from the runtime stack. It is restricted to finite abstractions of program data. *Weighted pushdown systems* (WPDSs) are strictly more expressive models than Boolean programs. They can encode infinite-state abstractions of data as well.

WPDSs are based on *pushdown systems* (PDSs), which are essentially finite-state machines equipped with a stack. PDSs are expressive enough to encode the interprocedural control flow of a program by using the PDS stack to encode the runtime stack of the program. PDSs can also encode Boolean programs, but the encoding is not very efficient: the size of a PDS encoding a Boolean program B will be exponential in the number of variables of B.

WPDSs are a generalization of PDSs. WPDSs extend PDSs by adding a general "black-box" abstraction for expressing transformations of a program's data state (through *weights*). Thus, the common strategy of encoding a program abstraction as a WPDS is to encode the interprocedural control-flow graph (ICFG) of the program using a PDS and the data transformations induced by the program statements as weights. WPDSs generalize other frameworks for interprocedural analysis, such as the Sharir-Pnueli functional approach [88], as well as the Knoop-Steffen [52] and Sagiv-Reps-Horwitz summary-based approaches [84].

One advantage of using WPDSs is that one can make use of any of several algorithms that exist for analyzing them [83]. Thus, in order to design a new verification tool, one only has to encode the program as a WPDS and then the analysis part is available for free. In particular, there are algorithms that compute the set of all reachable states at a given program node (for checking assertions at that node), using backward or forward search. There are also algorithms for computing a set of *witnesses*—a set of paths that justify the computed set of reachable states. Such witnesses can be used for reporting errors or generating counterexamples for subsequent abstraction refinement. Moreover, because they are based on pushdown systems, WPDSs can answer a richer set of queries about the model than can be answered by classical interprocedural dataflow-analysis algorithms [88, 52, 84], which only provide the ability to compute the set of all reachable states at a given program node. There are algorithms for WPDSs that also compute the set of reachable states at a given program node and for a given calling context for that node, or for a regular set of calling contexts for the node. In our earlier work, we shoued that these queries, called *stack-qualified queries* can be useful in the interprocedural setting [56].

Three implementations of WPDSs are publicly available [49, 47, 86], and all three provide a convenient base for implementing different analyses. As a programming abstraction, these systems offer several benefits:

- An analyzer is created by means of a declarative specification: one specifies a weight domain, along with an encoding of the program's ICFG and a mapping of each ICFG edge to a weight.

- They permit the creation of libraries of reusable weight domains, which can also be used to create new weight domains by means of weight-domain construction operations (pairing, reduced product [26], tensor product [71], etc.)

- They allow for symbolic analysis. Encoding Boolean programs as WPDSs can be exponentially more succinct than encoding them as PDSs. This happens because the weights can be encoded symbolically, e.g., using BDDs. In this case, the analysis of the

WPDS using the standard WPDS reachability algorithms corresponds to a symbolic analysis of the Boolean program.

- Compared with other tools that support the creation of program analyzers from high-level specifications, (i) the WPDS implementations allow more sophisticated abstract domains to be used (such as the domains for affine-relation analysis [67, 68]), and (ii) they also permit a broader range of dataflow-analysis queries to be posed (in particular, stack-qualified queries) than is possible with tools such as Banshee [53] and BDDBDDB [94].

Several of the contributions of this dissertation are made using WPDSs as a starting point; these results all retain the benefits of WPDSs mentioned above. PDSs and WPDSs are discussed in more detail in Chapter 2. Readers familiar with PDSs and WPDSs may skip reading this chapter, and use it only as reference material.

The rest of this section describes the contributions made by this dissertation.

First, we generalized the WPDS model to *extended weighted pushdown systems* (EWPDSs). (This result is presented in Chapter 3.) WPDSs, while expressive, do not provide a way to model the local variables of a procedure. EWPDSs provide a way in which a weight only has to describe the transformation on the variables in scope. In addition to the weights, *merge functions* can be provided that take care of the change in scope across procedure boundaries.

With EWPDSs, it was possible to build many more applications than was possible with WPDSs. EWPDSs have been used for checking properties of Boolean programs (Chapter 4); computing affine relations in x86 programs (Section 3.5.2); computing aliasing in a program with single-level pointers (Section 3.5.3); and (as components of model checkers) for concurrent programs [48, 58] to model individual threads.

In our earlier work, we used EWPDSs to design a debugging application, called BTrace [56], which tries to find the erroneous run of a program, given certain data related to the error, like the stack trace dumped out at a program crash. BTrace benefitted from being

able to make use of the reachability algorithms for EWPDSs. For instance, it uses the ability of EWPDSs to answer stack-qualified queries to obtain information associated with a stack trace. It also uses the witness-tracing feature to reconstruct the failing path.

The EWPDS model and some of its applications are described in Chapter 3.

Second, we showed how to improve the fix-point computation of both WPDSs and EWPDSs using graph-theoretic techniques. We call the resulting algorithm FWPDS (the "F" stands for "Fast"). The previous algorithms for (E)WPDSs were based on chaotic iteration to compute the fixpoint, which is also typical of other program-analysis tools. We noticed that adding direction to the chaotic iteration could improve things drastically. Tarjan had earlier given an efficient iteration strategy for graphs, which applies to programs with a single procedure [91, 90]. We generalized his algorithm to (E)WPDS. As a result, FWPDS applies to programs with multiple procedures.

FWPDS applies to all applications that use (E)WPDSs. FWPDS resulted in median speedups of 1.8× for finding affine-relations in x86 programs, 3.6× for BTrace, 2.6× for checking properties of Boolean programs.

We also developed techniques for incremental analysis, as well as efficient counterexample generation. Both techniques are useful in program-verification tools. FWPDS is discussed in Chapter 4.

Third, we showed how to combine forwards and backwards interprocedural analysis to compute what we call *error projections* (Chapter 5). An error projection is the union of all error traces in a program.

Typically, when an analysis concludes that bad states are reachable in an abstract model, but the counterexample is infeasible in the original program, the model is refined and the search is restarted. The single counterexample is the only information that is carried forward to the next refined model. Error projections allow much more information to be carried forward. In particular, the part of the model outside the error projection is provably correct because any execution that travels outside the error projection cannot lead to an error. Thus, the error projection represents the smallest part of the model that needs to be refined (and

re-analyzed). Error projections can be used for speeding up abstraction refinement, as well as for reporting errors back to the user.

We showed how to efficiently and precisely compute error projections when the program model is an (E)WPDS. Our algorithm uses forward and backward reachability on the (E)WPDS, which can in turn be sped up using FWPDS. Error projections and algorithms for computing them are discussed in Chapter 5.

1.3.2 New Technology for Concurrent Programs

The dissertation also makes a significant contribution to the design and implementation of efficient and practical verification tools for concurrent programs. We target shared-memory concurrent programs, whose verification is considered a challenging problem because of the fine-grained interactions that can occur between threads.

Earlier in the chapter, we mentioned that combining interprocedural analysis and concurrency leads to undecidability, and as a consequence, most tools give up precise interprocedural reasoning. We make a trade-off in a different direction, which is to bound concurrency, but retain the interprocedural aspect.

We place a bound on the number of *context switches* that can happen in any execution of a program. A context switch is defined as the transfer of control from one thread to another thread. We build tools that perform verification under a given context bound K; i.e., they can determine if any bug exists in any program execution with K context switches or fewer. We call verification under a context bound *context-bounded analysis* (CBA).

A natural question to ask at this point is what abstraction is more useful: one that gives up precise handling of procedures but does not require any bound on the number of context switches, or one that can handle procedures but requires a bound on the number of context switches? While one approach may not be provably better than the other, there are advantages to exploring the latter approach:

- For finite-data programs, the analysis of concurrent programs without procedures is PSPACE-complete, whereas CBA of concurrent programs with procedures is only NP-complete (Chapter 6).

- Retaining procedures in the abstract model implies that the sequential part of the analysis remains precise. Consequently, if the global verification property requires strong "thread-local" invariants to be established in each executing thread, this technique will do well in proving the global property. A technique that does not handle procedures would not be able to get off the ground, even in the presence of a context bound. One such example is shown in Fig. 1.6.

- Many program bugs can be found in a few context switches [77, 78, 70, 59]. KISS [78] showed how a number of concurrency bugs could be found by exploring just two context switches and two threads. Furthermore, Musuvathi and Qadeer [70] used an explicit-state model checker on programs with a closed environment (i.e., with fixed input), to systematically explore all their interleavings; this approach uncovered numerous bugs. In our work (Chapter 7), we showed that most bugs can be found in a few context switches even for programs with an open environment (where the input is not fixed).

- The context bound can be iteratively increased to find more bugs. This has the added advantage of finding bugs in the smallest number of context switches needed to trigger them, which can help in understanding the bug. Thus, concurrency is added gradually by increasing the context bound.

- The number of context switches seems to be good measure of the "hardness" of a bug. A bug that requires more context switches before it is triggered can be regarded as being more complicated than a bug that can be triggered in fewer context switches. Hence, CBA is "demand-driven" in terms of concurrency. The context-switch bound can be iteratively increased to gain a greater degree of assurance about the correctness of the program or to find more bugs. The analysis will incur higher costs only for finding more complicated bugs.

The disadvantage of bounding the number of context switches is that it is *unsound* and cannot be used to verify the absence of bugs. It can only provide a correctness guarantee under a bound on the number of context switches, for example, by proving that the program does not have any bugs when fewer than 10 context switches occur.

There has been work on CBA prior to our work. Qadeer and Rehof [77] showed that when the number of context switches is bounded, the reachability problem for Boolean programs is decidable. They also give an algorithm for this problem, but its complexity is exponential in the number of context switches and has not been implemented. Subsequently, we showed that CBA of Boolean programs is NP-complete (Chapter 6), thus indicating that an algorithm with lower worst-case complexity may not be possible at all.

This dissertation makes two contributions towards realizing practical algorithms for CBA. The theme of each of these contributions is that one can take an existing analysis for sequential programs and automatically extend it to perform CBA.

Result 1 (Chapter 6): We show that if each thread is modeled using a WPDS then CBA is decidable, and also give an algorithm for performing CBA. This result requires one extra property on the weights: they must have a *tensor* operation. We also showed that this operation exists for a large class of abstractions A_{tensor}, which includes finite-state ones, such as the one required for encoding Boolean programs, as well as infinite-state abstractions, such as the one required for affine-relation analysis. Our result generalizes the work of Qadeer and Rehof to a larger class of abstractions (and does so using much different techniques).

The significance of our result is that one only needs to show two properties to automatically obtain an algorithm for CBA: (*i*) the abstraction satisfies (or approximates) the properties required by a WPDS; and (*ii*) the abstraction belongs to the class A_{tensor}, i.e., an appropriate tensor operation exists. Neither of these requires any concurrency-related reasoning.

We obtained this result in two steps. First, we showed that all behaviors of a WPDS can be captured using a *weighted transducer*. A transducer is like a finite-state machine, but has an output tape as well. A weighted transducer, additionally, produces a weight

for every string that it writes on the output tape. We showed that one can construct a weighted transducer τ for a thread T that summarizes the behavior of T (when T does not yield control to other threads) in the following sense: when a state s of T is written on the input tape of τ, it can write a state s' on the output tape with weight w if and only if the net effect of executing T from s to s' is w. This provides a strong characterization of the set of behaviors of thread T. Second, we used such thread-summarization transducers to devise a *compositional* approach to CBA: CBA reduces to composing thread-summarization transducers as many times as the number of context switches. We showed that this can be done provided the weights have a *tensor* operation.

This work provided theoretical insight into CBA. However, the construction of the transducers can be an expensive operation. We improved on this by giving a more direct way of performing CBA that avoids the transducer construction.

Result 2 (Chapter 7): We showed that, given a concurrent program P and a context bound K, one can create a sequential program P_K such that the analysis of P_K is sufficient for CBA of P under the bound K. This reduction is a source-to-source transformation, and requires no assumptions nor extra work on the part of the user, except for the identification of thread-local data. We implemented this technique for a language that is used to specify Boolean programs to create the first known implementation of CBA. It scales to programs with shared state space as large as 2^{24} states and 10 context switches.

The key insight behind this result is that in a program with two threads T_1 and T_2, execution proceeds with control alternating between the two threads: $T_1; T_2; T_1; \cdots$. Concurrent analysis is hard because during the analysis of T_2, one also has to keep track of the local state of T_1, so that it can be restored when T_1 resumes execution. Keeping track of multiple local states typically makes the verification task expensive (or even undecidable). We solved this by transforming the threads to T_1^s and T_2^s so that one only has to analyze $T_1^s; T_1^s; \cdots; T_2^s; T_2^s; \cdots$, which involves no thread interleaving. For the program to be able have this structure, the context switches have to be simulated. To simulate T_1 relinquishing control to T_2, T_1^s simply *guesses* the effect that T_2 will have on the shared state and resumes

execution. It does this K times, making a total of K guesses. Next, control is passed to T_2^s and it verifies whether the K guesses made by T_1^s were correct. If not, execution is aborted. This ensures that any execution that is not aborted is a valid execution of the concurrent program. We showed that this guess-and-check strategy can be implemented using a source-to-source transformation.

The program P_K is (i) nK-times larger than P, where n is the number of threads, and (ii) has K times the number of variables as P. The former shows another salient feature of our reduction: CBA scales linearly with the number of threads. The latter shows that there is no free lunch: the worst-case complexity of analyzing sequential programs typically grows exponentially with the number of variables. Thus, the analysis of P_K scales exponentially with K, in the worst case. This result was expected because we had earlier proved that CBA is NP-complete (when variables are Boolean-valued).

We present our conclusions in Chapter 8.

Chapter 2

Background: Abstract Models and Their Analysis

This chapter discusses some common abstract models and algorithms for their analysis. All of the models are for sequential programs with multiple procedures, and their analyses will be interprocedural.

In Section 2.1, we define the dataflow model, which has been commonly used in the compiler literature. It serves to lay down some of the useful definitions and concepts. Next, in Section 2.2 and Section 2.3, we define Boolean programs and pushdown systems, respectively, which are more popular in verification and model-checking communities. In Section 2.4, we define weighted pushdown systems (WPDSs), which are capable of encoding all of the previous models under certain conditions. WPDSs merge the concepts that are common to compilers and verification. We mostly present results for the analysis of PDSs and WPDSs. We will build on these results in later chapters.

While discussing the analysis of an abstract model, we focus attention only on assertion checking. There are two important class of properties that one would like to verify on programs: *safety* properties and *liveness* properties. Safety properties address finite (but possibly unbounded) behaviors of a program, e.g., memory safety, information flow, API usage rules, etc. Whereas liveness properties address infinite behaviors, e.g., non-termination, response time, etc. In this dissertation, we focus only on safety properties. For such properties, it is possible to reduce the problem of checking them on programs to assertion checking by inserting instrumentation in the program that keeps track of all the information relevant to checking the property (similar to inlined reference monitors [29]). For example, memory

safety can be checked by inserting assertions that check that a pointer is not null right before it is dereferenced. For array-bounds checking, one can keep track of the size of each allocated memory block and assert that any access to the array is within the bound. An instance of this technique for finite-state properties is given in Section 2.4.3.

Reducing the verification property to assertion checking simplifies the design of the analysis phase. To check an assertion at program node n, one only needs to find the set of reachable states at n, and then check if this set intersects with the asserted condition. This can further be simplified by converting an `assert`(ϕ) statement to "if(!ϕ) then goto `error`", and then simply check if node `error` is reachable or not.

To avoid overloading the term "state", we may refer to the instantaneous state of a program as a *memory configuration* when there is a possibility of confusion with other terms.

Notation. A binary relation on a set S is a subset of $S \times S$. If R_1 and R_2 are binary relations on S, then their relational composition, denoted by "$R_1; R_2$", is defined by $\{(s_1, s_3) \mid \exists s_2 \in S, (s_1, s_2) \in R_1, (s_2, s_3) \in R_2\}$. If R is a binary relation, R^i is the relational composition of R with itself i times, and R^0 is the identity relation on S. $R^* = \cup_{i=0}^{\infty} R^i$ is the reflexive-transitive closure of R.

2.1 The Dataflow Model

Dataflow analysis is used more broadly than just for program verification. It is a technique commonly used in compilers to enable compiler optimizations. Irrespective of whether a verification property is present or not, dataflow analysis is concerned with finding a *dataflow value* associated with each program node n that summarizes possible memory configurations whenever control reaches n. The dataflow value for n safely approximates (i.e., over approximates) the set of memory configurations reachable at node n.

The dataflow model of a program consists of the following elements:

- The interprocedural control-flow graph (ICFG) of the program.
- A join semilattice $(V, \sqcup)$ with least element $\perp$:

  - Elements of V are called *dataflow values*. A dataflow value represents a set of possible memory configurations.
  - The join operator $\sqcup$ is used for combining information obtained along different paths.

- A value $v_0 \in V$ that represents the set of possible memory configurations at the beginning of the program.
- An assignment M of dataflow *transfer functions* (of type $V \to V$) to the edges of the ICFG: $M(e) \in V \to V$.

A dataflow-analysis problem can be formulated as a *path-function problem*.

Definition 2.1.1. *A* **path** *of length j from node m to node n is a (possibly empty) sequence of j edges, denoted by $[e_1, e_2, \ldots, e_j]$, such that the source of e_1 is m, the target of e_j is n, and for all i, $1 \leq i \leq j-1$, the target of edge e_i is the source of edge e_{i+1}.*

The path function pf_q for path $q = [e_1, e_2, \ldots, e_j]$ is the composition, in reverse order, of q's transfer functions: $\mathrm{pf}_q = M(e_j) \circ \ldots \circ M(e_2) \circ M(e_1)$. The path function for an empty path is the identify function from V to V.

2.1.1 Join Over All Paths

In *intra*procedural dataflow analysis, the goal is to determine, for each node n, the "join-over-*all-paths*" (JOP) solution:

$$\mathrm{JOP}_n = \bigsqcup_{q \in \mathrm{Paths}(\mathrm{enter}, n)} \mathrm{pf}_q(v_0),$$

where $\mathrm{Paths}(\mathrm{enter}, n)$ denotes the set of paths in the ICFG from the enter node to n [51]. JOP_n represents a summary of the possible memory configurations that can arise at n: because $v_0 \in V$ represents the set of possible memory configurations at the beginning of the program, $\mathrm{pf}_q(v_0)$ represents the contribution of path q to the memory configurations summarized at n.

The soundness of the JOP_n solution with respect to the programming language's concrete semantics is established by the methodology of *abstract interpretation* [24]:

- A Galois connection (or Galois insertion) is established to define the relationship between sets of concrete states and elements of V.
- Each dataflow transfer function $M(e)$ is shown to overapproximate the transfer function for the concrete semantics of e.

In the discussion below, we assume that such correctness requirements have already been taken care of, and we concentrate only on algorithms for determining dataflow values once a dataflow model has been given.

2.1.2 Example: Copy-Constant Propagation

This section gives an example of a dataflow model that can be used to do *copy-constant propagation*, in which only statements of the form "`x = y`" and "`x = constant`" are interpreted, whereas the other statements are over-approximated. The goal of the analysis is to find if a variable always holds a constant value at some point in the program.

An example ICFG is shown in Fig. 2.1. The dashed and dotted arrows represent the two procedure calls to `f` and their return back to `main`. Let *Var* be the set of all variables in a program, and let $(\mathbb{Z}^\top, \sqsubseteq, \sqcup)$, where $\mathbb{Z}^\top = \mathbb{Z} \cup \{\top\}$, be the standard constant-propagation semilattice: for all $c \in \mathbb{Z}$, $\top \sqsupset c$; for all $c_1, c_2 \in \mathbb{Z}$ such that $c_1 \neq c_2$, c_1 and c_2 are incomparable; and $\sqcup$ is the least-upper-bound operation in this partial order. $\top$ stands for "not-a-constant". Let $D = (\mathit{Env} \to \mathit{Env})$ be the set of all environment transformers, where an environment is a mapping for all variables: $\mathit{Env} = (\mathit{Var} \to \mathbb{Z}^\top) \cup \{\bot\}$. We use $\bot$ to denote an infeasible environment. Furthermore, we restrict the set D to contain only $\bot$-strict transformers, i.e., for all $d \in D$, $d(\bot) = \bot$. We can extend the join operation to

```
int a,b,y;

void main() {
  n1: a = 5;
  n2: y = 1;
  n3,n4: f();
  n5: if(...) {
    n6: a = 2;
    n7,n8: f();
  }
  n9: ...;
}

void f() {
  n10: b = a;
  n11: if(...)
    n12: y = 2;
  else
    n13: y = b;
}
```

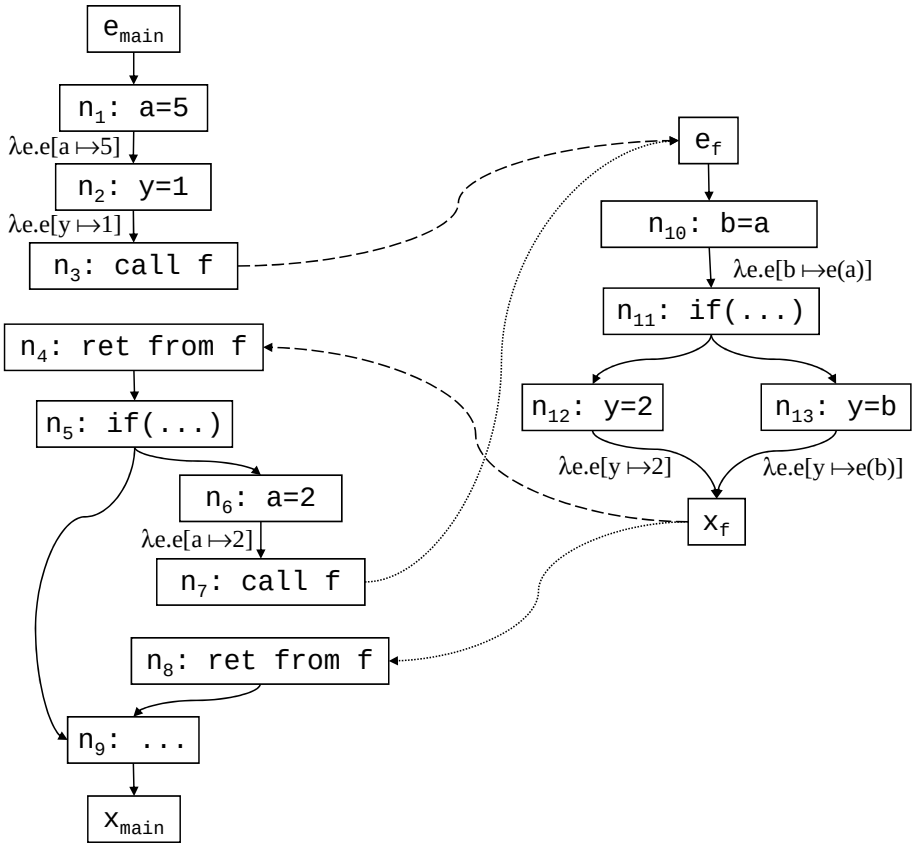


Figure 2.1 A program fragment and its ICFG. For all unlabeled edges, the environment transformer is $\lambda e.e$. The statements labeled "..." are assumed not to change any of the declared variables.

environments by taking join componentwise.

$$env_1 \sqcup env_2 = \begin{cases} env_1 & \text{if } env_2 = \bot \\ env_2 & \text{if } env_1 = \bot \\ \lambda v.(env_1(v) \sqcup env_2(v)) & \text{otherwise} \end{cases}$$

The dataflow transformers are shown as edge labels in Fig. 2.1. A transformer of the form $\lambda e.e[\mathtt{a} \mapsto 5]$ returns an environment that agrees with the argument e, except that `a` is bound to 5. The environment $\bot$ cannot be updated, and thus $(\lambda e.e[\mathtt{a} \mapsto 5])\bot$ equals $\bot$. The initial

dataflow value is the environment where all variables are uninitialized: $[\mathtt{y} \mapsto \top, \mathtt{a} \mapsto \top, \mathtt{b} \mapsto \top]$.

2.1.3 Interprocedural Join Over All Paths

The *inter*procedural dataflow analysis problem is similar to the intraprocedural one, except that the paths that are chosen in the ICFG must be *valid* interprocedural paths, i.e., they should have matching calls and returns. (The exact definition of valid paths can be found elsewhere [88, 84].)

For instance, in the ICFG shown in Fig. 2.1, the path $[e_{main}, n_1, n_2, n_3, e_f, n_{10}, n_{11}, x_f, n_4, n_5]$ has matching calls and returns, and hence it is a valid path; the path $[e_{main}, n_1, n_2, n_3, e_f, n_{10}, n_{11}, x_f, n_8]$ is not a valid path because the exit-to-return-site edge $x_f \rightarrow n_8$ does not correspond to the preceding call-to-enter edge $n_3 \rightarrow e_f$.

In interprocedural dataflow analysis, the goal shifts from finding the join-over-*all-paths* solution to the more precise "join-over-*all-valid-paths*" (JOVP), or "context-sensitive" solution. A context-sensitive interprocedural dataflow analysis is one in which the analysis of a called procedure is "sensitive" to the context in which it is called. A context-sensitive analysis captures the fact that the results propagated back to each return site r should depend only on the memory configurations that arise at the call site that corresponds to r. More precisely, the goal of a context-sensitive analysis is to find the JOVP value for nodes of the ICFG [88, 52, 84].

Definition 2.1.2. *The join-over-all-valid-paths (*JOVP*) value for an ICFG node n is defined as follows:*

$$\mathrm{JOVP}_n = \bigsqcup_{q \in VPaths(e_{main}, n)} pf_q(v_0),$$

where $VPaths(e_{main}, n)$ denotes the set of valid paths from the main procedure's enter node to n.

Although some valid paths may also be infeasible execution paths, none of the non-valid paths are feasible execution paths. By restricting attention to just the valid paths from e_{main}, we exclude some of the infeasible execution paths. In general, therefore, JOVP_n characterizes the memory configurations at n more precisely than JOP_n.

Example 2.1.3. *For the dataflow model described in Section 2.1.2,* $\text{JOVP}_{n_4} = [\mathtt{y} \mapsto \top, \mathtt{a} \mapsto 5, \mathtt{b} \mapsto 5]$ *and* $\text{JOVP}_{n_8} = [\mathtt{y} \mapsto 2, \mathtt{a} \mapsto 2, \mathtt{b} \mapsto 2]$. *This proves, for instance, that the variable* $\mathtt{y}$ *holds the value* 2 *whenever control reaches node* n_8*, irrespective of the path taken to reach* n_8.

2.1.4 Solving for JOP

In this section, we briefly sketch an algorithm for finding the JOP value. The JOP value cannot be computed directly by using its definition because it involves taking joins over an unbounded number of values. It is computed using a fixpoint iteration.

For each node n in the ICFG, let X_n be a variable ranging over V, the set of dataflow facts. Initialize all such variables to $\bot$. Next, repeat the following until the values of all of the variables stop changing: choose any edge $e = (n, m)$ in the ICFG; update X_m to $(X_m \sqcup M(e)(X_n))$. Once this iteration is finished, call the resulting value of X_n LFP_n (the least fixpoint value at n).

In the above algorithm, the aspect of choosing an edge randomly among all possible edges is an instance of the *chaotic iteration* strategy, where one chooses randomly from a set of possibilities to make progress. Chapter 4 discusses ways of improving over the chaotic iteration strategy.

An early result in dataflow analysis shows that $\text{LFP}_n = \text{JOP}_n$, provided that the functions $M(e)$ are *distributive* for all edges e [44]. A function $f : V \to V$ is distributive if $f(v_1 \sqcup v_2) = f(v_1) \sqcup f(v_2)$, for all $v_1, v_2 \in V$.[1] Moreover, the iteration in the above algorithm will terminate if V has no infinite ascending chains (in the partial order defined by $\sqcup$). Such

[1] We shall see that the distributivity property is an important requirement in later sections as well.

a result, which equates the ideal value (JOP) with one obtained from an iterative algorithm (LFP) is called a *coincidence* theorem.

A coincidence theorem for interprocedural dataflow analysis was given by Sharir and Pnueli [88]. The dataflow model was extended to better deal with local variables by Knoop and Steffen and they also gave a coincidence theorem for this extended model [52]. This theorem is covered in Section 3.3.

We do not give an algorithm for JOVP here (it can be found elsewhere [88, 84]), but weighted pushdown systems can encode dataflow models, and the algorithms for them also provide algorithms to solve for JOVP on dataflow models. As we shall see in later sections, whenever an interprocedural analysis is carried out (to compute a value similar to the JOVP value), the computation is always done over variables X_n that range over the function space $V \to V$, instead of variables that range over V.

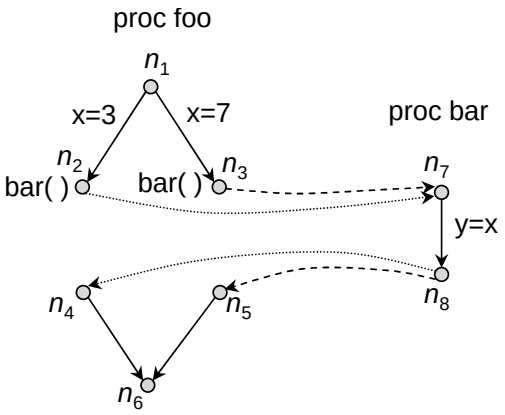
2.2 Boolean Programs

A Boolean program can be thought of as a C program with only the Boolean datatype. It does not have any pointers or heap-allocated storage. A Boolean program consists of a finite set of procedures. It has a finite set of global variables, and a finite set of local variables for each procedure. Each variable can only hold a value from a finite domain.[2] Boolean programs are very commonly used by model checkers [4, 6, 96]. They are often obtained as a result of predicate abstraction (Section 1.1.1).

To simplify the discussion, we assume that procedures do not have parameters (they can be passed through global variables). The variables in scope inside a procedure are the global variables and its set of local variables. Fig. 2.2(a) shows a Boolean program with two procedures and two global variables `x` and `y` over a finite domain $V = \{0, 1, \ldots, 7\}$.

Let G be the set of valuations of the global variables, and let Val_i be the set of valuations of the local variables of procedure i. The set of *global states* of a Boolean program is the set

[2] An assignment to a variable v that holds a value from a finite domain can be thought of a collection of assignments to a *vector* of Boolean-valued variables, namely, the collection of Boolean-valued variables that holds the encoding of v's value.



(a)

$$
\begin{aligned}
[\![x = 3]\!] &= \{((v_1, v_2), (3, v_2)) \mid v_1, v_2 \in V\} \\
[\![x = 7]\!] &= \{((v_1, v_2), (7, v_2)) \mid v_1, v_2 \in V\} \\
[\![y = x]\!] &= \{((v_1, v_2), (v_1, v_1)) \mid v_1, v_2 \in V\}
\end{aligned}
$$

(b)

Figure 2.2 (a) A Boolean program with two procedures and two global variables `x` and `y` over a finite domain $V = \{0, 1, \ldots, 7\}$. (b) The (non-identity) transformers used in the Boolean program. v_1 refers to a value of x and v_2 refers to a value of y.

G, and the set of *local states* L is defined as follows: a local state consists of the value of the program counter, a valuation of local variables from some Val_i, and the program stack (which, for each unfinished call to a procedure P, contains a return address and a valuation of the local variables at the time of the call to P). Let N_i be the set of all CFG nodes of procedure i. Then $L = (\cup_i (N_i \times \text{Val}_i))^+$, i.e., elements of L are a non-empty list of pairs from the set $(N_i \times \text{Val}_i)$ for some i. For convenience, we write the elements of L with an overbar, e.g. $\bar{l}$, and use juxtaposition to denote list concatenation.

The effect of executing a statement `st` of procedure i, denoted by $[\![\texttt{st}]\!]$, is a binary relation on $G \times \text{Val}_i$ that describes how values of variables in scope can change. Fig. 2.2(b) shows the (non-identity) transformers used in Fig. 2.2(a).

The operational semantics of Boolean programs is shown in Fig. 2.3. The instantaneous state of a Boolean program is an element of $G \times L$. The operational semantics define how the instantaneous state can change on the execution of a single statement in the program. Let entry(`f`) denote the entry node of procedure `f`, *proc*(n) denote the procedure that contains node n, ep(n) denote entry(*proc*(n)); let exitnode(n) denote a predicate on nodes that is true when n is the exit node of its procedure.

$$\frac{n \xrightarrow{\texttt{st}} m \quad (g_1, l_1, g_2, l_2) \in [\![\texttt{st}]\!]}{(g_1, (n, l_1)\ \bar{l}) \rightarrow (g_2, (m, l_2)\ \bar{l})}\ \textsc{Intra}$$

$$\frac{n \xrightarrow{\texttt{call f()}} m \quad e = \text{entry}(\texttt{f}) \quad l \in \text{Val}_{\texttt{f}}}{(g_1, (n, l_1)\ \bar{l}) \rightarrow (g_1, (e, l)\ (m, l_1)\ \bar{l})}\ \textsc{Call}$$

$$\frac{\text{exitnode}(n)}{(g_1, (n, l_1)\ \bar{l}) \rightarrow (g_1, \bar{l})}\ \textsc{Return}$$

Figure 2.3 Operational semantics of a Boolean program. In the rule CALL, node m is the return site for the procedure call at n.

A Boolean program with only global variables can be thought of as an instance of a dataflow model. In particular, it is one where a dataflow value is a subset of G, $\sqcup$ is defined as union, and the dataflow transformer associated with an edge (n, m) is $[\![\texttt{st}]\!]$, where `st` is the statement on node n. In this case, JOVP_n denotes the set of all values that the variables can hold at node n. Assertion checking at node n can be done using JOVP_n. To encode a Boolean program with local variables, an extended dataflow model is needed, such as the one used in [52], or pushdown systems (Section 2.3) and their extensions (Chapter 3). The analysis of Boolean programs can be carried out via its encoding to other models, which will be discussed in Chapter 3. A more direct way of analyzing Boolean programs is be discussed in Chapter 7.

The advantage of using Boolean programs is that one can encode branch conditions using assume statements. (An assume statement is one that states a condition but does not change the value of any variable.) For instance, the statement `x == y` in the program Fig. 2.2(a) would be associated with the transformer:

$$[\![\texttt{x == y}]\!] = \{((v, v), (v, v)) \mid v \in V\}$$

Such a statement would have the effect of only letting those states pass that satisfy the condition `x == y`. For example, composing the transformers of each statement along the

program path `x = 3; y = 7; x == y`, in order, leads to the empty relation, i.e., the path is infeasible.

2.3 Pushdown Systems

A pushdown system (PDS) is similar to a pushdown automaton but does not have an input tape. It is simply used to represent a transition system.

Definition 2.3.1. *A* **pushdown system** *is a triple* $\mathcal{P} = (P, \Gamma, \Delta)$ *where* P *is the set of states or control locations,* Γ *is the set of stack symbols and* $\Delta \subseteq P \times \Gamma \times P \times \Gamma^*$ *is the set of pushdown rules. A* **configuration** *of* $\mathcal{P}$ *is a pair* $\langle p, u \rangle$ *where* $p \in P$ *and* $u \in \Gamma^*$*. A rule* $r \in \Delta$ *is written as* $\langle p, \gamma \rangle \hookrightarrow \langle p', u' \rangle$ *where* $p, p' \in P$*,* $\gamma \in \Gamma$ *and* $u' \in \Gamma^*$*.*

The rules of $\mathcal{P}$ *define a transition relation* $\Rightarrow_{\mathcal{P}}$ *on the configurations of* $\mathcal{P}$ *as follows: If* $r = \langle p, \gamma \rangle \hookrightarrow \langle p', u' \rangle$ *then* $\langle p, \gamma u'' \rangle \Rightarrow_{\mathcal{P}} \langle p', u'u'' \rangle$ *for all* $u'' \in \Gamma^*$*. Moreover, if for two configurations* c *and* c'*,* $\sigma \in \Delta^*$ *is a rule sequence that transforms* c *to* c'*, we say* $c \Rightarrow_{\mathcal{P}}^{\sigma} c'$*. The set of all rule sequences that transform* c *to* c' *is denoted as* $paths(c, c')$*.*

The reflexive transitive closure of $\Rightarrow_{\mathcal{P}}$ *is denoted by* $\Rightarrow_{\mathcal{P}}^*$*. For a set of configurations* C*, we define* $pre^*{}_{\mathcal{P}}(C) = \{c' \mid \exists c \in C : c' \Rightarrow_{\mathcal{P}}^* c\}$ *and* $post^*{}_{\mathcal{P}}(C) = \{c' \mid \exists c \in C : c \Rightarrow_{\mathcal{P}}^* c'\}$*, which are just backward and forward reachability under the transition relation* $\Rightarrow$*. We drop the subscript* $\mathcal{P}$ *when there is no possibility of confusion.*

We restrict PDS rules to have at most two stack symbols on the right-hand side. This means that for every rule $r \in \Delta$ of the form $\langle p, \gamma \rangle \hookrightarrow_{\mathcal{P}} \langle p', u \rangle$, we have $|u| \leq 2$. This restriction does not decrease the power of pushdown systems because by increasing the number of stack symbols by a constant factor, an arbitrary pushdown system can be converted into one that satisfies this restriction [85].

The standard approach for modeling program control flow with a pushdown system is as follows: P contains a single state $\{p\}$, Γ corresponds to program locations, and Δ corresponds to transitions in the interprocedural control-flow graph (ICFG), as shown in Fig. 2.4. For

Rule	Control flow modeled
$\langle p, u\rangle \hookrightarrow \langle p, v\rangle$	Intraprocedural edge $u \to v$
$\langle p, c\rangle \hookrightarrow \langle p, e_f\ r\rangle$	Call to procedure f from node c that enters the procedure at e_f and returns to r
$\langle p, x_f\rangle \hookrightarrow \langle p, \varepsilon\rangle$	Return from procedure f at exit node x_f

Figure 2.4 The encoding of an ICFG's edges as PDS rules.

instance, the rules that encode the ICFG shown in Fig. 2.1 are:

$$\begin{array}{lll}
\langle p, e_{main}\rangle \hookrightarrow \langle p, n_1\rangle & \langle p, n_5\rangle \hookrightarrow \langle p, n_9\rangle & \langle p, e_f\rangle \hookrightarrow \langle p, n_{10}\rangle \\
\langle p, n_1\rangle \hookrightarrow \langle p, n_2\rangle & \langle p, n_6\rangle \hookrightarrow \langle p, n_7\rangle & \langle p, n_{10}\rangle \hookrightarrow \langle p, n_{11}\rangle \\
\langle p, n_2\rangle \hookrightarrow \langle p, n_3\rangle & \langle p, n_7\rangle \hookrightarrow \langle p, e_f\ n_8\rangle & \langle p, n_{11}\rangle \hookrightarrow \langle p, n_{12}\rangle \\
\langle p, n_3\rangle \hookrightarrow \langle p, e_f\ n_4\rangle & \langle p, n_8\rangle \hookrightarrow \langle p, n_9\rangle & \langle p, n_{12}\rangle \hookrightarrow \langle p, x_f\rangle \\
\langle p, n_4\rangle \hookrightarrow \langle p, n_5\rangle & \langle p, n_9\rangle \hookrightarrow \langle p, x_{main}\rangle & \langle p, n_{11}\rangle \hookrightarrow \langle p, n_{13}\rangle \\
\langle p, n_5\rangle \hookrightarrow \langle p, n_6\rangle & \langle p, x_{main}\rangle \hookrightarrow \langle p, \varepsilon\rangle & \langle p, n_{13}\rangle \hookrightarrow \langle p, x_f\rangle \\
 & & \langle p, x_f\rangle \hookrightarrow \langle p, \varepsilon\rangle
\end{array}$$

Under such an encoding of a program, a PDS configuration can be thought of as a CFG node with its calling context, i.e., the stack of return addresses of unfinished calls leading up to the node. A rule $r = \langle p, \gamma\rangle \hookrightarrow \langle p', u\rangle$, $u \in \Gamma^*$, is called a *pop* rule if $|u| = 0$, and a *push* rule if $|u| = 2$.

A PDS in which the set P is a singleton set is also referred to as a context-free process [16]. The state space P can be expanded to use multiple states to encode a finite abstraction of the global variables, and the stack alphabet can be expanded to encode local variables [85]. This technique will be discussed in more detail below.

Because the number of configurations of a pushdown system is unbounded, it is useful to use finite automata to describe certain infinite sets of configurations.

Definition 2.3.2. *If $\mathcal{P} = (P, \Gamma, \Delta)$ is a pushdown system then a* **$\mathcal{P}$-automaton** *is a finite automaton $(Q, \Gamma, \to, P, F)$ where $Q \supseteq P$ is a finite set of states, $\to \subseteq Q \times \Gamma \times Q$ is the*

transition relation, P is the set of initial states, and F is the set of final states of the automaton. We say that a configuration $\langle p, u\rangle$ is accepted by a $\mathcal{P}$-automaton if the automaton can accept u when it is started in the state p (written as $p \xrightarrow{u}{}^{*} q$, where $q \in F$). A set of configurations is called **regular** *if some $\mathcal{P}$-automaton accepts it. Without loss of generality, $\mathcal{P}$-automata are restricted to not have any transitions leading to an initial state.*

An important result is that for a regular set of configurations C, both $post^*(C)$ and $pre^*(C)$ are also regular sets of configurations [15, 85, 9, 30, 33].

2.3.1 Encoding Boolean programs using PDSs

To encode a Boolean program B using a PDS, the state alphabet P is expanded to encode the values of global variables, and the stack alphabet Γ is expanded to encode the values of local variables [85].

Let N_i be the set of CFG nodes of the i^{th} procedure of B. Let G and L be the set of global and local states of B, respectively, as defined in Section 2.2. Let Val_i be the set of valuations of local variables of the i^{th} procedure.

We set P to be G, and Γ to be the union of $N_i \times \text{Val}_i$ over all procedures. (Note that the set of local states L equals Γ^+.) The PDS rules for the i^{th} procedure are constructed as follows: (i) an intraprocedural ICFG edge $u \to v$ with statement `st` is encoded via a set of rules $\langle g, (u, l)\rangle \hookrightarrow \langle g', (v, l')\rangle$, for each $((g, l), (g', l')) \in [\![\texttt{st}]\!]$; (ii) a call edge $c \to r$ that calls procedure f, with enter node e_f, is encoded via a set of rules $\langle g, (c, l)\rangle \hookrightarrow \langle g, (e_f, l_0)\ (r, l)\rangle$, for each $(g, l) \in G \times \text{Val}_i$ and $l_0 \in \text{Val}_f$; (iii) a procedure return at node u is encoded via a set of rules $\langle g, (u, l)\rangle \hookrightarrow \langle g, \varepsilon\rangle$, for each $(g, l) \in G \times \text{Val}_i$;

Under such an encoding of a Boolean program as a PDS, a configuration $\langle p, \gamma_1\gamma_2 \cdots \gamma_n\rangle$ of the PDS is an element of $G \times L$ that describes the instantaneous state of the Boolean program. The state p encodes the values of global variables; γ_1 encodes the current program counter and the values of local variables in scope; and the rest of the stack encodes the list of unfinished calls with the values of local variables at the time the call was made. The reader

can verify that the PDS transition relation ($\Rightarrow$) is the same as the single-step execution relation ($\rightarrow$) defined in Fig. 2.3.

2.3.2 Solving Reachability on PDSs using Saturation-Based Algorithms

In this section, we show how to compute $post^*(C)$ and $pre^*(C)$ for a regular set of configurations C.[3] This will serve to lay out some of the concepts that we will use in designing algorithms for more advanced abstract models in later chapters.

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 configurations accepted by $\mathcal{A}$, they produce $\mathcal{P}$-automata $\mathcal{A}_{post^*}$ and $\mathcal{A}_{pre^*}$ that accept the sets of configurations $post^*(C)$ and $pre^*(C)$, respectively [9, 30, 33]. Both *poststar* and *prestar* can be implemented as *saturation procedures*; i.e., transitions are added to $\mathcal{A}$ according to some saturation rule until no more can be added.

Algorithm *prestar*: $\mathcal{A}_{pre^*}$ can be constructed from $\mathcal{A}$ using the following saturation rule: *If* $\langle p, \gamma\rangle \hookrightarrow \langle p', w\rangle$ *and* $p' \xrightarrow{w} q$ *in the current automaton, add a transition* (p, γ, q).

This algorithm is based on the intuition that if the automaton accepts a configuration c and a rule r allows the transition $c' \Rightarrow c$, then the automaton needs to accept c' as well: If there is an accepting path starting in state q that accepts u, then the automaton accepts the configuration $c = \langle p', wu\rangle$. The rule $r = \langle p, \gamma\rangle \hookrightarrow \langle p', w\rangle$ allows the transition $c' \Rightarrow c$, where $c' = \langle p, \gamma u\rangle$. The addition of the transition (p, γ, q) allows c' to be accepted by the automaton.

Termination of the algorithm follows from the fact that the number of states of the automaton does not increase (hence, the number of transitions is bounded).

Algorithm *poststar*: $\mathcal{A}_{post^*}$ can be constructed from $\mathcal{A}$ by performing Phase I and then saturating via the rules given in Phase II:

[3] The material in this section is adapted from [83].

- *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 $(\overset{\epsilon}{\rightarrow})^\star \overset{\gamma}{\rightarrow} (\overset{\epsilon}{\rightarrow})^\star$.)
  - If $\langle p, \gamma \rangle \hookrightarrow \langle p', \epsilon \rangle \in \Delta$ and $p \overset{\gamma}{\rightsquigarrow} q$ in the current automaton, add a transition (p', ϵ, q).
  - If $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \rangle \in \Delta$ and $p \overset{\gamma}{\rightsquigarrow} q$ in the current automaton, add a transition (p', γ', q).
  - If $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma'\gamma'' \rangle \in \Delta$ and $p \overset{\gamma}{\rightsquigarrow} q$ in the current automaton, add the transitions $(p', \gamma', p'_{\gamma'})$ and $(p'_{\gamma'}, \gamma'', q)$.

This algorithm is based on intuition similar to that for *prestar*. The difference is that *poststar* adds more states to the automaton. These states are needed to accommodate configurations that are added because of a push rule. One has to argue that reusing these states for different applications of (possibly distinct) call rules is correct. The interested reader is referred to the original papers for this proof [9, 30, 33].

Example 2.3.3. *Given the PDS that encodes the ICFG from Fig. 2.1 and the query automaton $\mathcal{A}$ shown in Fig. 2.5(a), which accepts the language $\{\langle p, e_{main} \rangle\}$, poststar produces the automaton $\mathcal{A}_{post^*}$ shown in Fig. 2.5(b),*

2.3.3 Solving Pre-Reachability on PDSs using Context-Free Grammars

While most implementations of PDS reachability use the saturation-based algorithms, there are different ways looking at the reachability problem.[4] In particular, we show how *context-free grammars* can be used to find not only the set of reachable configurations, but also the set of paths (rule sequences) that justify their reachability. In this section, fix a PDS $\mathcal{P} = (P, \Gamma, \Delta)$ and a $\mathcal{P}$-automaton $\mathcal{A} = (Q, \Gamma, \rightarrow, P, F)$.

[4]The material in this section is adapted from [83].

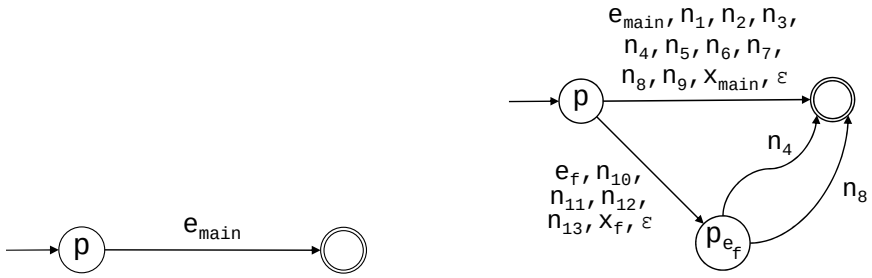


Figure 2.5 (a) Automaton for the input language of configurations $\{\langle p, e_{main}\rangle\}$; (b) automaton for $post^*(\{\langle p, e_{main}\rangle\})$ (computed for the PDS that encodes the ICFG from Fig. 2.1).

	Production	for each
(1)	$PopRuleSeq_{(q,\gamma,q')} \rightarrow r$	$r = \langle q, \gamma\rangle \hookrightarrow \langle q', \varepsilon\rangle \in \Delta$
(2)	$PopRuleSeq_{(p,\gamma,q)} \rightarrow r \;\; PopRuleSeq_{(p',\gamma',q)}$	$r = \langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\rangle \in \Delta, q \in P$
(3)	$PopRuleSeq_{(p,\gamma,q)} \rightarrow r \;\; PopRuleSeq_{(p',\gamma',q')} \;\; PopRuleSeq_{(q',\gamma'',q)}$	$r = \langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\gamma''\rangle \in \Delta, q, q' \in P$

Figure 2.6 The *PopRuleSeq* grammar for a PDS $\mathcal{P} = (P, \Gamma, \Delta)$.

Consider the *PopRuleSeq* grammar shown in Fig. 2.6. The non-terminals of the grammar are $PopRuleSeq_{(p,\gamma,q)}$ for all $p, q \in P$ and $\gamma \in \Gamma$, and the set of terminals is Δ. An important property of this grammar is as follows.

Lemma 2.3.4. *The set of strings derived by the non-terminal* $PopRuleSeq_{(p,\gamma,q)}$ *of the grammar shown in Fig. 2.6 is exactly the set* $paths_{\mathcal{P}}(\langle p, \gamma\rangle, \langle q, \varepsilon\rangle)$.

The proof of Lem. 2.3.4 follows quite easily from induction on the length of a rule sequence.

We extend Lem. 2.3.4 to capture the set of all rule sequences from a given configuration. First, observe that every rule sequence $\sigma \in paths(\langle p_1, \gamma_1\gamma_2\cdots\gamma_n\rangle, \langle p, \varepsilon\rangle)$ can be decomposed as $\sigma = \sigma_1\sigma_2\cdots\sigma_n$ (see Fig. 2.7) such that $\sigma_i \in paths(\langle p_i, \gamma_i\rangle, \langle p_{i+1}, \varepsilon\rangle)$ for $1 \leq i \leq n$, and

$$\begin{aligned}
\langle p_1, \gamma_1\ \gamma_2\ \gamma_3 \cdots \gamma_n \rangle &\Rightarrow^{\sigma_1} \quad \langle p_2, \gamma_2\ \gamma_3 \cdots \gamma_n \rangle \\
&\Rightarrow^{\sigma_2} \quad \langle p_3, \gamma_3 \cdots \gamma_n \rangle \\
&\cdots \\
&\Rightarrow^{\sigma_{n-1}} \quad \langle p_n, \gamma_n \rangle \\
&\Rightarrow^{\sigma_n} \quad \langle p_{n+1}, \varepsilon \rangle
\end{aligned}$$

Figure 2.7 A path in the transition relation of a PDS from the configuration $\langle p_1, \gamma_1\ \gamma_2\ \gamma_3 \cdots \gamma_n \rangle$ to the configuration $\langle p_{n+1}, \varepsilon \rangle$.

$p_{n+1} = p$. Intuitively, this holds because for a path to look at γ_2, it must first pop off γ_1, and then repeat this until the stack becomes empty. This implies the following two results.

Lemma 2.3.5. *For any PDS $\mathcal{P} = (P, \Gamma, \Delta)$, the set $paths_{\mathcal{P}}(\langle p_1, \gamma_1\gamma_2 \cdots \gamma_n \rangle, \langle p_{n+1}, \varepsilon \rangle)$ equals the union of the sets $(paths_{\mathcal{P}}(\langle p_1, \gamma_1 \rangle, \langle p_2, \varepsilon \rangle).paths_{\mathcal{P}}(\langle p_2, \gamma_2 \rangle, \langle p_3, \varepsilon \rangle) \ \ldots\ paths_{\mathcal{P}}(\langle p_n, \gamma_n \rangle, \langle p_{n+1}, \varepsilon \rangle))$, where "." denotes elementwise concatenation of sets of strings, over all possible choices of $p_2, p_3, \cdots, p_n \in P$.*

Corollary 2.3.6. *Let $\mathcal{L}(N)$ be the language that can be derived from a non-terminal N. For a PDS $\mathcal{P} = (P, \Gamma, \Delta)$, the set of rule sequences $paths_{\mathcal{P}}(\langle p_1, \gamma_1\gamma_2 \cdots \gamma_n \rangle, \langle p, \varepsilon \rangle)$ is the union of the sets $(\mathcal{L}(PopRuleSeq_{(p_1,\gamma_1,p_2)}).\mathcal{L}(PopRuleSeq_{(p_2,\gamma_2,p_3)}) \cdots \mathcal{L}(PopRuleSeq_{(p_n,\gamma_n,p)}))$ over all possible choices of $p_2, p_3, \cdots, p_n \in P$.*

Next, we show that Cor. 2.3.6 is sufficient to compute the pre^* set. Let $\mathcal{L}(\mathcal{A})$ be the set of configurations accepted by $\mathcal{A}$. We absorb $\mathcal{A}$ into the PDS $\mathcal{P}$ in order to only consider a single system as follows:

Definition 2.3.7. *Given a PDS $\mathcal{P} = (P, \Gamma, \Delta)$ and a $\mathcal{P}$-automaton $\mathcal{A} = (Q, \Gamma, \rightarrow, P, F)$, their combined PDS $\mathcal{PA}$ is defined to be $(Q, \Gamma, \Delta \cup \Delta')$, where Δ' contains a pop rule $\langle p, \gamma \rangle \hookrightarrow \langle q, \varepsilon \rangle$ for every transition (p, γ, q) in $\mathcal{A}$.*

The PDS $\mathcal{PA}$ can either operate like $\mathcal{P}$, by using a rule in Δ, or like $\mathcal{A}$, by using a rule in Δ'. Because $\mathcal{A}$ has no incoming transitions to states in P (Defn. 2.3.2), a valid rule sequence

σ of $\mathcal{PA}$ (i.e., $\sigma \in \mathit{paths}(c, c')$ for some configurations c and c') must be from the set $\Delta^*(\Delta')^*$, i.e., it is a sequence of rules from Δ followed by a sequence of rules from Δ'. This is because when a rule in Δ' is used, the target configuration must have a control state from the set $Q - P$, after which a rule in Δ cannot fire, and rules in Δ' keep the control state in $Q - P$. The first phase, consisting of rules from Δ, is also a valid rule sequence for $\mathcal{P}$; the second phase, consisting of rules from Δ', simulates running automaton $\mathcal{A}$. The following lemma is based on this observation.

Lemma 2.3.8. *Let $\mathcal{P} = (P, \Gamma, \Delta)$ be a PDS and $\mathcal{A} = (Q, \Gamma, \rightarrow, P, F)$ be a $\mathcal{P}$-automaton. Let $\mathcal{PA}$ be their combined PDS. Then $c' \Rightarrow_{\mathcal{P}} c$ for some $c \in C$, if and only if $c' \Rightarrow_{\mathcal{PA}} \langle q_f, \varepsilon \rangle$ for some $q_f \in F$. Consequently, $\mathit{pre}^*_{\mathcal{P}}(\mathcal{L}(\mathcal{A})) = \mathit{project}_P(\mathit{pre}^*_{\mathcal{PA}}(\{\langle q_f, \varepsilon \rangle \mid q_f \in F\}))$, where $\mathit{project}_P(S)$ consists of all configurations in S, whose control state is in P.*

Given a configuration c, we can compute the set of all rule sequences that take c to a configuration accepted by $\mathcal{A}$ as follows: For each $q_f \in F$, apply Cor. 2.3.6 to the *PopRuleSeq* grammar of $\mathcal{PA}$, with $p = q_f$, to obtain the set S_{q_f} of all paths from c to $\langle q_f, \varepsilon \rangle$ in $\mathcal{PA}$. Next, take a union of these sets over all states in F, and remove the rules in Δ'. The resultant set is the desired answer.

Because the set of such accepting paths can be unbounded, computing it explicitly may not be possible. However, the above technique does allow us to get a handle on the set of all accepting paths. By replacing the rules with other quantities, we can compute other values of interest. For instance, for weighted pushdown systems, the rules are replaced by weights to compute the net effect of all paths between two given configurations (Section 2.4.1). If one is only interested in the set of reachable configurations, then the rules can be replaced by ε, leading to the grammar shown in Fig. 2.8, which has some interesting properties.

In the *PopSeq* grammar, $\mathit{PopSeq}_{(p,\gamma,q)}$ can derive ε if and only if $\mathit{paths}_{\mathcal{PA}}(\langle p, \gamma \rangle, \langle q, \varepsilon \rangle)$ is non-empty. Also note the similarity of this grammar with the saturation-based algorithm *prestar* presented earlier: each grammar production, which was created because of the rule $r \in \Delta$, corresponds to an instance of saturation rule for r. For example, in *prestar*, the rule $r = \langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \rangle$ dictates the following: if (p', γ', q) is a transition in the current

	Production	for each
(1)	$PopSeq_{(q,\gamma,q')} \rightarrow \varepsilon$	$\langle q,\gamma\rangle \hookrightarrow \langle q',\varepsilon\rangle \in (\Delta \cup \Delta')$
(2)	$PopSeq_{(p,\gamma,q)} \rightarrow PopSeq_{(p',\gamma',q)}$	$\langle p,\gamma\rangle \hookrightarrow \langle p',\gamma'\rangle \in \Delta, q \in Q$
(3)	$PopSeq_{(p,\gamma,q)} \rightarrow PopSeq_{(p',\gamma',q')}$	$PopSeq_{(q',\gamma'',q)}$ $\langle p,\gamma\rangle \hookrightarrow \langle p',\gamma'\gamma''\rangle \in \Delta, q, q' \in Q$

Figure 2.8 The *PopSeq* grammar for PDS $\mathcal{PA}$.

automaton, then add a transition (p,γ,q); in the grammar, it states that if $PopSeq_{(p',\gamma',q)}$ can derive ε, then so can $PopSeq_{(p,\gamma,q)}$. This leads to the following result.

Lemma 2.3.9. *The non-terminal $PopSeq_{(p,\gamma,q)}$ of the grammar shown in Fig. 2.8 can derive ε if and only if the transition (p,γ,q) exists in the final automaton $\mathcal{A}_{pre^*}$ produced by prestar($\mathcal{A}$).*

Lem. 2.3.9 along with Cor. 2.3.6 justifies the correctness of the *prestar* algorithm, i.e., the fact that $\mathcal{L}(\mathcal{A}_{pre^*}) = pre^*(\mathcal{L}(\mathcal{A}))$.

2.3.4 Solving Post-Reachability on PDSs using Context-Free Grammars

The situation is similar for computing $post^*$ using context-free grammars, but slightly more complicated.[5] The complication arises from the fact that the PDS has to be massaged into a different form before we obtain a notion that is analogous to pop sequences.

Let Q_{mid} be a set that contains a state $p'_{\gamma'}$ for every push rule $\langle p,\gamma\rangle \hookrightarrow \langle p',\gamma'\gamma''\rangle$ in Δ. Let $\mathcal{P}_e = (P \cup Q_{\text{mid}}, \Gamma, \Delta_e)$, where Δ_e contains every rule from Δ with zero or one stack symbols on the right-hand side; and for every push rule $r = \langle p,\gamma\rangle \hookrightarrow \langle p',\gamma'\gamma''\rangle \in \Delta$, Δ_e contains two rules: $\eta_1(r) = \langle p,\gamma\rangle \hookrightarrow \langle p'_{\gamma'},\gamma''\rangle$ and $\eta_2(r) = \langle p'_{\gamma'},\varepsilon\rangle \hookrightarrow \langle p',\gamma'\rangle$. Note that this allows the addition of rules with no stack symbols on the left-hand side. Such rules can fire without consuming the top symbol of the stack, i.e., the rule $\langle p,\varepsilon\rangle \hookrightarrow \langle q,\gamma\rangle$ contributes to the transition relation of the PDS as follows: $\langle p,u\rangle \Rightarrow \langle q,\gamma u\rangle$ for every $u \in \Gamma^*$.

[5] Again, the material in this section is adapted from [83].

	Production		for each
(1)	$SameLevelRuleSeq_{(p',\varepsilon,q)}$	$\rightarrow\ PushRuleSeq_{(p,\gamma,q)}\ r$	$r = \langle p,\gamma\rangle \hookrightarrow \langle p',\varepsilon\rangle \in \Delta, q \in P_e$
(2)	$PushRuleSeq_{(p',\gamma',q')}$	$\rightarrow\ PushRuleSeq_{(q,\gamma',q')}\ SameLevelRuleSeq_{(p',\varepsilon,q)}$	$p' \in P,\ q, q' \in P_e$
(3)	$PushRuleSeq_{(p',\gamma',q)}$	$\rightarrow\ PushRuleSeq_{(p,\gamma,q)}\ r$	$r = \langle p,\gamma\rangle \hookrightarrow \langle p',\gamma'\rangle \in \Delta, q \in P_e$
(4)	$PushRuleSeq_{(p',\gamma',p'_{\gamma'})}$	$\rightarrow\ \eta_2(r)$	$r = \langle p,\gamma\rangle \hookrightarrow \langle p',\gamma'\ \gamma''\rangle \in \Delta$
(5)	$PushRuleSeq_{(p'_{\gamma'},\gamma'',q)}$	$\rightarrow\ PushRuleSeq_{(p,\gamma,q)}\ \eta_1(r)$	$r = \langle p,\gamma\rangle \hookrightarrow \langle p',\gamma'\ \gamma''\rangle \in \Delta, q \in P_e$

Figure 2.9 The *PushRuleSeq* grammar for PDS $\mathcal{P}_e$. The set P_e is defined as $(P \cup Q_{\text{mid}})$

The non-terminals of the grammar shown in Fig. 2.6 derived a language of rule sequences, each of which could pop off one symbol from the top of the stack. Now we define a grammar whose non-terminals derive rule sequences that can *push* one symbol on the top of the stack. This grammar is shown in Fig. 2.9.

Lemma 2.3.10. *The set of strings derived by the non-terminal* $PushRuleSeq_{(p,\gamma,q)}$ *of the grammar shown in Fig. 2.9 is exactly the set* $paths_{\mathcal{P}_e}(\langle q,\varepsilon\rangle, \langle p,\gamma\rangle)$.

Note that any for any rule sequence of $\mathcal{P}_e$ between two configurations $\langle p_1, u_1\rangle$ and $\langle p_2, u_2\rangle$ such that $p_1, p_2 \in P$, it must have $\eta_2(r)$ and $\eta_1(r)$ adjacent to each other. This is because once a state changes to be one in Q_{mid}, only a rule of the form $\eta_2(r)$ can fire. We can convert a valid rule sequence $\sigma_e \in \Delta_e^*$ to one in Δ^* by replacing the sequence of rules $(\eta_1(r);\ \eta_2(r))$ in σ_e with r. (This is possible because η_1 is invertible.)

Similar to Cor. 2.3.6, we can use the *PushRuleSeq* grammar to find all paths in the set $paths_{\mathcal{P}_e}(\langle p,\varepsilon\rangle, \langle p_{n+1}, \gamma_{n+1}\cdots\gamma_1\rangle)$.

We define the combined PDS-Automaton system as follows: the PDS $\mathcal{A}^R\mathcal{P}$ is the tuple $(Q \cup Q_{\text{mid}}, \Gamma, \Delta_e)$, where Δ_e contains every rule from Δ with zero or one stack symbols on the right-hand side; for every push rule $\langle p,\gamma\rangle \hookrightarrow \langle p',\gamma'\gamma''\rangle$, Δ_e contains two rules: $\langle p,\gamma\rangle \hookrightarrow \langle p'_{\gamma'},\gamma''\rangle$ and $\langle p'_{\gamma'},\varepsilon\rangle \hookrightarrow \langle p',\gamma'\rangle$; and for every transition (q,γ,q') in $\mathcal{A}$, Δ_e contains the rule $\langle q',\varepsilon\rangle \hookrightarrow \langle q,\gamma\rangle$. A rule sequence in this PDS first generates a configuration in the language

	Production		for each
(1)	$PushSeq_{(q,\gamma,q')}$	$\rightarrow \varepsilon$	$(q,\gamma,q') \in \rightarrow$
(2)	$SameLevelSeq_{(p',\varepsilon,q)}$	$\rightarrow PushSeq_{(p,\gamma,q)}$	$\langle p,\gamma\rangle \hookrightarrow \langle p',\varepsilon\rangle \in \Delta, q \in Q_e$
(3)	$PushSeq_{(p',\gamma',q')}$	$\rightarrow PushSeq_{(q,\gamma',q')}\ SameLevelSeq_{(p',\varepsilon,q)}$	$p' \in P,\ q,q' \in Q_e$
(4)	$PushSeq_{(p',\gamma',q)}$	$\rightarrow PushSeq_{(p,\gamma,q)}$	$\langle p,\gamma\rangle \hookrightarrow \langle p',\gamma'\rangle \in \Delta, q \in Q_e$
(5)	$PushSeq_{(p',\gamma',p'_{\gamma'})}$	$\rightarrow \varepsilon$	$\langle p,\gamma\rangle \hookrightarrow \langle p',\gamma'\ \gamma''\rangle \in \Delta$
(6)	$PushSeq_{(p'_{\gamma'},\gamma'',q)}$	$\rightarrow PushSeq_{(p,\gamma,q)}$	$\langle p,\gamma\rangle \hookrightarrow \langle p',\gamma'\ \gamma''\rangle \in \Delta, q \in Q_e$

Figure 2.10 The *PushSeq* grammar for PDS $\mathcal{A}^R\mathcal{P}$. The set Q_e is defined as $(Q \cup Q_{\text{mid}})$

of $\mathcal{A}$ and then fires a rule sequence from $\mathcal{P}_e$. The saturation-based algorithm *poststar* is in direct correspondence with the *PushRuleSeq* grammar when the rules are replaced with ε.

Lemma 2.3.11. *The non-terminal $PushSeq_{(p,\gamma,q)}$ of the grammar shown in Fig. 2.10 can derive ε if and only if the transition (p,γ,q) exists in the final automaton $\mathcal{A}_{post^*}$ produced by poststar($\mathcal{A}$).*

2.4 Weighted Pushdown Systems

A weighted pushdown system is obtained by supplementing a pushdown system with a weight domain that is a bounded idempotent semiring [82, 10]. Such semirings are powerful enough to encode finite-state data abstractions such as the one required to encode Boolean programs, as well as infinite-state data abstractions, such as copy-constant propagation and affine-relation analysis [60]. The basic idea is to use weights to encode the effect that each rule has on the data state of the program.

Definition 2.4.1. *A* **bounded idempotent semiring** *is a quintuple $(D,\oplus,\otimes,\overline{0},\overline{1})$, where D is a set whose elements are called* **weights**, *$\overline{0}$ and $\overline{1}$ are elements of D, and $\oplus$ (the combine operation) and $\otimes$ (the extend operation) are binary operators on D such that*

1. *$(D,\oplus)$ is a commutative monoid with $\overline{0}$ as its neutral element, and where $\oplus$ is idempotent. $(D,\otimes)$ is a monoid with the neutral element $\overline{1}$.*

2. $\otimes$ *distributes over* $\oplus$*, i.e., for all* $a, b, c \in D$ *we have*

$$a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c) \text{ and } (a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c) .$$

3. $\overline{0}$ *is an annihilator with respect to* $\otimes$*, i.e., for all* $a \in D$*,* $a \otimes \overline{0} = \overline{0} = \overline{0} \otimes a$*.*

4. *In the partial order* $\sqsubseteq$ *defined by* $\forall a, b \in D,\ b \sqsubseteq a$ *iff* $a \oplus b = a$*, there are no infinite ascending chains.*

In dataflow-analysis terms, D is a set of dataflow transformers, $\oplus$ is join, $\otimes$ is transformer composition, $\overline{0}$ is the infeasible transformer, and $\overline{1}$ is the identity transformer.

The *height* of a weight domain is defined to be the length of the longest ascending chain in the domain. For simplicity, when we discuss complexity results, we will assume that the height is bounded, but WPDSs, and the algorithms in this dissertation, can also be used in certain cases when the height is unbounded (as long as the condition in Defn. 2.4.1, item 4 is satisfied).

Definition 2.4.2. *A* **weighted pushdown system** *is a triple* $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$ *where* $\mathcal{P} = (P, \Gamma, \Delta)$ *is a pushdown system,* $\mathcal{S} = (D, \oplus, \otimes, \overline{0}, \overline{1})$ *is a bounded idempotent semiring and* $f : \Delta \to D$ *is a map that assigns a weight to each pushdown rule.*

Let $\sigma \in \Delta^*$ be a sequence of rules. Using f, we can associate a value to σ, i.e., if $\sigma = [r_1, \ldots, r_k]$, then we define $v(\sigma) \stackrel{\text{def}}{=} f(r_1) \otimes \ldots \otimes f(r_k)$. The set of all rule sequences from a configuration in S to a configuration in T is denoted as $paths(S, T)$.

Definition 2.4.3. *Let* $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$ *be a WPDS, where* $\mathcal{P} = (P, \Gamma, \Delta)$*, and let* $S, T \subseteq P \times \Gamma^*$ *be regular sets of configurations. The* ***interprocedural join-over-all-paths (IJOP)*** *value* $\mathrm{IJOP}(S, T)$ *is defined as* $\bigoplus\{v(\sigma) \mid s \Rightarrow^{\sigma} t, s \in S, t \in T\}$*. A set of* ***witnesses*** $\omega(S, T)$ *for this value is defined as a finite set of paths (rule sequences)* $\{\sigma_1, \cdots, \sigma_n\}, \sigma_i \in paths(S, T)$*, such that* $\oplus_i v(\sigma_i) = \mathrm{IJOP}(S, T)$*.*

The IJOP value describes the net transformation that occurs when going from one set of configurations to another. The set of witnesses gives a finite number of paths that together

justify the reported IJOP value. The WPDS reachability problems, which compute the set of forward and backward reachable states, are defined as follows.

Definition 2.4.4. *Let $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$ be a weighted pushdown system, where $\mathcal{P} = (P, \Gamma, \Delta)$, and let $C \subseteq P \times \Gamma^*$ be a regular set of configurations. The* **generalized pushdown predecessor problem** $\mathrm{GPP}(C)$ *is to find for each $c \in P \times \Gamma^*$:*

$$\delta(c, C) \stackrel{\text{def}}{=} \mathrm{IJOP}(\{c\}, C)$$

The **generalized pushdown successor (GPS) problem** $\mathrm{GPS}(C)$ *is to find for each $c \in P \times \Gamma^*$:*

$$\delta(C, c) \stackrel{\text{def}}{=} \mathrm{IJOP}(C, \{c\})$$

If S is the set of initial configurations of a program then $\mathrm{GPS}(S)$ solves for the set of all reachable states in the program. If T is the set of error configurations (such as ones that trigger an assertion violation), then $\mathrm{GPP}(T)$ is the set of all states that can lead to an error configuration. Checking program safety reduces to checking that $\mathrm{IJOP}(S, T) \neq \overline{0}$. In case it is non-$\overline{0}$, $\omega(S, T)$ gives a finite number of counterexamples — valid paths from a configuration in S to a configuration in T.

To illustrate the above definitions, we show how a Boolean program B with only global variables can be encoded using a WPDS $(\mathcal{P}, \mathcal{S}, f)$. Let G be the set of global states of B. The weight domain $\mathcal{S}$ is $(2^{G \times G}, \cup, ;, \emptyset, id)$, where the weights are relations (transformers) on the set G. Combine is set union, extend is relational composition (composition of transformers), $\overline{0}$ is the empty relation and $\overline{1} = id$ is the identity relation on G. The ICFG of B is encoded using the PDS $\mathcal{P}$ and a statement `st` that labels edge e of B is encoded as the weight $[\![\texttt{st}]\!]$ on the rule corresponding to e. An example is shown in Fig. 2.11(a).

The set of all data values that reach a node n can be calculated as follows: let S be the singleton configuration consisting of the program's enter node, and let T be the set $\{\langle p, n\ u\rangle \mid u \in \Gamma^*\}$. Let $w = \mathrm{IJOP}(S, T)$. If $w = \overline{0}$, then the node cannot be reached. Otherwise, w captures the net transformation on the global state from when the program started. The range of w, i.e., the set $\{g \in G \mid \exists g' \in G : (g', g) \in w\}$, is the set of valuations

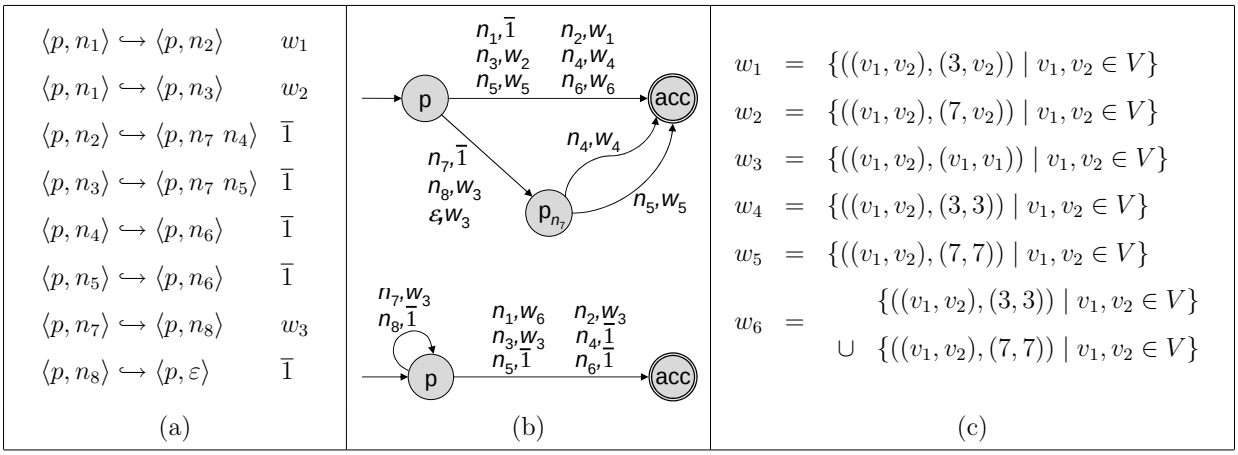


Figure 2.11 (a) A WPDS that encodes the Boolean program from Fig. 2.2(a). (b) The result of *poststar*($\langle p, n_1 \rangle$) and *prestar*($\langle p, n_6 \rangle$). The final state in each of the automata is *acc*. (c) Definitions of the weights used in the figure.

that reach node n. For example, in Fig. 2.11(a), the IJOP weight to node n_6 is the weight w_6 shown in Fig. 2.11(c). Its range shows that either $\mathtt{x} = 3$ and $\mathtt{y} = 3$, or $\mathtt{x} = 7$ and $\mathtt{y} = 7$.

Because T can be any regular set, one can also answer stack-qualified queries [83]. For example, the set of values that arise at node n when its procedure is called from call site m can be found by setting $T = \{\langle p, n\ m_r\ u \rangle \mid u \in \Gamma^*\}$, where m_r is the return site for call site m.

A WPDS with a weight domain that has a finite set of weights, such as the one described above for Boolean programs, can be encoded as a PDS. However, it is often useful to use weights because they can be symbolically encoded. Tools such as Moped [85] and Bebop [6] use BDDs [14] to encode sets of data values, which allows them to scale to a large number of variables. (Using PDSs for Boolean program verification, without any symbolic encoding, is generally not a feasible approach.)

Dataflow analysis can also be encoded using WPDSs. The control-flow is encoded using a PDS, as done for Boolean programs; the dataflow transformer associated an edge becomes the weight associated with the rule corresponding to that edge; combine is defined as join of transformers; and extend is defined as the reverse of function composition. In this case, JOVP_n (Defn. 2.1.2) is the same as $\text{IJOP}(\{n_0\}, \{\langle p, n\ u \rangle \mid u \in \Gamma^*\})(v_0)$, where n_0 is the

entry point of the main procedure and v_0 is the initial dataflow value. This encoding is valid under the restriction that all the dataflow transformers are distributive (over the lattice join) and do not have any infinite ascending chains. These are the same restrictions used in other work on interprocedural dataflow analysis [88].

2.4.1 Solving for the IJOP Value

There are two algorithms for solving backward and forward reachability on WPDSs, called *prestar* and *poststar*, respectively (in the unweighted case, these algorithms reduce to computing the pre^* and $post^*$ sets of configurations). Sets of weighted configurations are symbolically represented using *weighted automata*.

Definition 2.4.5. *Given a WPDS $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$, a **$\mathcal{W}$-automaton** $\mathcal{A}$ is a $\mathcal{P}$-automaton, where each transition in the automaton is labeled with a weight. The weight of a path in the automaton is obtained by taking an extend of the weights on the transitions in the path in either a forward or backward direction, depending on the context in which the automaton is used. The automaton is said to accept a configuration $\langle p, u\rangle$ with weight w, denoted by $\mathcal{A}(\langle p, u\rangle)$, if w is the combine of weights of all accepting paths for u starting from state p in the automaton. We call the automaton a **backward $\mathcal{W}$-automaton** if the weight of a path is read backwards, and a **forward $\mathcal{W}$-automaton** otherwise.*

Let $\mathcal{A}$ be an unweighted automaton and $\mathcal{L}(\mathcal{A})$ be the set of configurations accepted by it. Then, *prestar*($\mathcal{A}$) produces a forward weighted automaton $\mathcal{A}_{pre^*}$ as output, such that $\mathcal{A}_{pre^*}(c) = \text{IJOP}(\{c\}, \mathcal{L}(\mathcal{A}))$, whereas *poststar*($\mathcal{A}$) produces a backward weighted automaton $\mathcal{A}_{post^*}$ as output, such that $\mathcal{A}_{post^*}(c) = \text{IJOP}(\mathcal{L}(\mathcal{A}), \{c\})$ [83]. These algorithms are similar to those for PDSs; they only differ in the weight computations.

Notation. In a forward $\mathcal{W}$-automaton, we say that $p \xrightarrow{u} q$ with weight w if $u = \gamma_1\gamma_2\cdots\gamma_n$, and there are transitions (p_i, γ_i, p_{i+1}) with weight w_i, where $p = p_1$ and $q = p_{n+1}$, and $w = w_1 \otimes w_2 \otimes \cdots \otimes w_n$. The same holds for a backward automaton, except that w should equal $w_n \otimes \cdots \otimes w_2 \otimes w_1$. The operation of adding a transition t with weight w to a weighted automaton $\mathcal{A}$ is carried out as follows: if t does not exist in $\mathcal{A}$, then t is simply added to

the transition set of $\mathcal{A}$; otherwise, if t exists with weight w', then t's weight is updated to $w \oplus w'$.

Algorithm *prestar*: The forward weighted automaton $\mathcal{A}_{pre^*}$ can be constructed from $\mathcal{A}$ using the following saturation rule: *If* $r = \langle p, \gamma\rangle \hookrightarrow \langle p', u\rangle$ *is a rule in the PDS and* $p' \xrightarrow{u} q$ *in the current automaton with weight* w*, then add a transition* (p, γ, q) *to the automaton with weight* $(f(r) \otimes w)$.

Algorithm *poststar*: The backward weighted automaton $\mathcal{A}_{post^*}$ can be constructed from $\mathcal{A}$ by performing Phase I and then saturating via the rules given in Phase II:

- *Phase I.* For each pair (p', γ') such that $\mathcal{P}$ contains at least one rule of the form $\langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\gamma''\rangle$, add a new state $p'_{\gamma'}$.
- *Phase II (saturation phase).* (The symbol $\overset{\gamma}{\rightsquigarrow}$ denotes the relation $(\xrightarrow{\epsilon})^\star \xrightarrow{\gamma} (\xrightarrow{\epsilon})^\star$.)
  - If $r = \langle p, \gamma\rangle \hookrightarrow \langle p', \epsilon\rangle \in \Delta$ and $p \overset{\gamma}{\rightsquigarrow} q$ with weight w in the current automaton, add a transition (p', ϵ, q) with weight $w \otimes f(r)$.
  - If $r = \langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\rangle \in \Delta$ and $p \overset{\gamma}{\rightsquigarrow} q$ with weight w in the current automaton, add a transition (p', γ', q) with weight $w \otimes f(r)$.
  - If $\langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\gamma''\rangle \in \Delta$ and $p \overset{\gamma}{\rightsquigarrow} q$ in the current automaton, add the transitions $(p', \gamma', p'_{\gamma'})$ and $(p'_{\gamma'}, \gamma'', q)$ with weights $\overline{1}$ and $w \otimes f(r)$, respectively.

  An efficient implementation of this algorithm dynamically maintains the epsilon closure of the automaton so that weights under the transition relation $\overset{\gamma}{\rightsquigarrow}$ can be read off efficiently [83].

Examples are shown in Fig. 2.11(b). One thing to note here is how the *poststar* automaton works. The procedure `bar` is analyzed independently of its calling context (i.e., without knowing the exact value of `x`), which generates the transitions between p and p_{n_7}. The calling context of `bar`, which determines the input values to `bar`, is represented by the transitions that leave state p_{n_7}. This is how, for instance, the automaton records that `x` $= 3$ and `y` $= 3$ at node n_8 when `bar` is called from node n_2.

We now provide some intuition into why one needs both forwards and backwards automata. Consider the automaton shown in Fig. 2.11(c). For the *poststar* automaton, when one follows a path that accepts the configuration $\langle p, n_8\ n_4 \rangle$, the transition (p, n_8, q) comes before (q, n_4, acc). However, the former transition describes the transformation inside `bar`, which happens *after* the transformation performed in reaching the call site at n_4 (which is stored on (q, n_4, acc)). Because the transformation for the calling context happens earlier in the program, but its transitions appear later in the automaton, the weights are read backwards. For the *prestar* automaton, the weight on (p, n_4, acc) is the transformation for going from n_4 to n_6, which occurs after the transformation inside `bar`. Thus, it is a forwards automaton.

These saturation-based algorithms can be applied on weighted automata as well. In that case, one can prove the following. (Define $\mathcal{A}(c)$ to be $\overline{0}$ if $\mathcal{A}$ does not accept c.)

Lemma 2.4.6. *If $\mathcal{A}$ is a forward-weighted automaton and $\mathcal{A}_{pre^*}$ is the result of running prestar on $\mathcal{A}$, then for every configuration c:*

$$\mathcal{A}_{pre^*}(c) = \bigoplus_{c'} \{v(\sigma) \otimes \mathcal{A}(c') \mid \sigma \in \mathit{paths}(c, c')\}$$

If $\mathcal{A}$ is a backward-weighted automaton and $\mathcal{A}_{post^*}$ is the result of running poststar on $\mathcal{A}$, then for every configuration c:

$$\mathcal{A}_{post^*}(c) = \bigoplus_{c'} \{\mathcal{A}(c') \otimes v(\sigma) \mid \sigma \in \mathit{paths}(c', c)\}$$

The *path_summary* Algorithm

Once the weighted automata $\mathcal{A}_{pre^*}$ and $\mathcal{A}_{post^*}$ are computed, we still need to be able to compute the weight with which they accept a particular configuration, or a set of configurations. Recall that $\mathcal{A}(c)$ is defined as the combine of weights of all accepting paths for c. We define $\mathcal{A}(C) = \bigoplus\{\mathcal{A}(c) \mid c \in C\}$. This allows one to solve for $\text{IJOP}(S, T)$ for configuration

sets S and T by computing either *poststar*$(S)(T)$ or *prestar*$(T)(S)$. The algorithm that can read off these weights from a weighted automata is called the *path_summary* algorithm.

Definition 2.4.7. *For a weighted automaton $\mathcal{A}$, the weight path_summary$(\mathcal{A})$ is defined as $\mathcal{A}(\Gamma^*)$, i.e., the combine over the weights of all accepting paths of $\mathcal{A}$.*

The weight $\mathcal{A}(C)$ can be computed as *path_summary*$(\mathcal{A} \cap \mathcal{A}_C)$, where $\mathcal{A}_C$ is an unweighted automaton that accepts the set of configurations C and the intersection operation is carried out as for unweighted automata, except that the weights of $\mathcal{A}$ are retained.

The *path_summary* weight is computed in the same manner as intraprocedural dataflow analysis (Section 2.1.4). We restrict attention to forward weighted automata. The algorithm is similar for backward weighted automata. Let $\mathcal{A}$ be a forward $\mathcal{W}$-automaton; let Q be its set of states; and F be its set of final states. Let $l(q)$ be a weight associated with state $q \in Q$. Initialize $l(q)$ to $\overline{0}$ for all $q \in Q - F$ and $\overline{1}$ for $q \in F$. Then use the following saturation rule: for a transition (p, γ, q) with weight w, update $l(p)$ to $l(p) \oplus (w \otimes l(q))$. Once saturation finishes, i.e., no weight changes, then *path_summary*$(\mathcal{A}) = l(q_0)$, where q_0 is the unique initial state of $\mathcal{A}$.

Abstract Grammar Problems

Just as in the case of PDSs, context-free grammars can be used to gain more insight into WPDS reachability problems. The following presents a weighted problem on context-free grammars.

Definition 2.4.8. *[83] Let $(S, \sqcup)$ be a join semilattice. An* **abstract grammar** *over $(S, \sqcup)$ is a collection of context-free grammar productions, where each production θ has the form $X_0 \to g_\theta(X_1, \ldots, X_k)$. Parentheses, commas, and g_θ are terminal symbols. Every production θ is associated with a function $g_\theta \colon S^k \to S$. Thus, every string α of terminal symbols derived in this grammar denotes a composition of functions, and corresponds to a unique value in S, which we call val(α). Let $\mathcal{L}(X)$ denote the strings of terminals derivable from a non-terminal X. The* **abstract grammar problem** *is to compute, for each non-terminal X, the value*

(1) $t_1 \rightarrow g_1(\epsilon)$ $\quad g_1 = w_1$ $\quad$ (3) $t_2 \rightarrow g_3(t_1)$ $\quad g_3 = \lambda x.w_3 \otimes x$

(2) $t_1 \rightarrow g_2(t_2)$ $\quad g_2 = \lambda x.w_2 \otimes x$ $\quad$ (4) $t_3 \rightarrow g_4(t_2)$ $\quad g_4 = \lambda x.w_4 \otimes x$

Figure 2.12 A simple abstract grammar with four productions.

	Production	for each
(1)	$PopSeq_{(q,\gamma,q')} \rightarrow g_1(\epsilon)$	$(q,\gamma,q') \in \rightarrow_0$
	$g_1 = \overline{1}$	
(2)	$PopSeq_{(p,\gamma,p')} \rightarrow g_2(\epsilon)$	$r = \langle p,\gamma\rangle \hookrightarrow \langle p',\varepsilon\rangle \in \Delta$
	$g_2 = f(r)$	
(3)	$PopSeq_{(p,\gamma,q)} \rightarrow g_3(PopSeq_{(p',\gamma',q)})$	$r = \langle p,\gamma\rangle \hookrightarrow \langle p',\gamma'\rangle \in \Delta, q \in Q$
	$g_3 = \lambda x.f(r) \otimes x$	
(4)	$PopSeq_{(p,\gamma,q)} \rightarrow g_4(PopSeq_{(p',\gamma',q')}, PopSeq_{(q',\gamma'',q)})$	
	$g_4 = \lambda x.\lambda y.f(r) \otimes x \otimes y$	$r = \langle p,\gamma\rangle \hookrightarrow \langle p',\gamma'\gamma''\rangle \in \Delta, q, q' \in Q$

Figure 2.13 An abstract grammar problem for solving GPP.

$\mathrm{JOD}(X) = \bigsqcup_{\alpha \in \mathcal{L}(X)} val(\alpha)$. *This value is called the* **join-over-all-derivations** *value for* X.

We define abstract grammars over the meet semilattice $(D, \oplus)$, where D is a set of weights. An example is shown in Fig. 2.12. The non-terminal t_3 can derive the string $\alpha = g_4(g_3(g_1))$ and $val(\alpha) = w_4 \otimes w_3 \otimes w_1$.

The abstract grammar for solving GPP is shown in Fig. 2.13. The grammar has one non-terminal $PopSeq_t$ for each possible transition $t \in Q \times \Gamma \times Q$ of $\mathcal{A}_{pre^*}$. It is based on the unweighted grammar shown in Fig. 2.8, which was shown to capture all paths in a PDS. The following lemma follows from Lem. 2.3.9.

Lemma 2.4.9. *For a transition t in the automaton that results from running prestar$(\mathcal{A})$, the weight on t is exactly* $\mathrm{JOD}(PopSeq_t)$.

The abstract grammar for solving GPS is shown in Fig. 2.14. The grammar has one non-terminal $PushSeq_t$ for each possible transition $t \in (Q \cup Q_{\mathrm{mid}}) \times \Gamma \times (Q \cup Q_{\mathrm{mid}})$ of $\mathcal{A}_{post^*}$.

	Production		for each
(1)	$PushSeq_{(q,\gamma,q')}$	$\rightarrow\ h_1(\epsilon)$	$(q,\gamma,q') \in \rightarrow_0$
	$h_1 = \overline{1}$		
(2)	$SameLevelSeq_{(p',\varepsilon,q)}$	$\rightarrow\ h_2(PushSeq_{(p,\gamma,q)})$	$r = \langle p,\gamma\rangle \hookrightarrow \langle p',\varepsilon\rangle \in \Delta, q \in Q$
	$h_2 = \lambda x.x \otimes f(r)$		
(3)	$PushSeq_{(p',\gamma',q')} \rightarrow h_{2'}(PushSeq_{(q,\gamma',q')}, SameLevelSeq_{(p',\varepsilon,q)})$		
			$p' \in P,\ q,q' \in Q$
	$h_{2'} = \lambda x.\lambda y.x \otimes y$		
(4)	$PushSeq_{(p',\gamma',q)}$	$\rightarrow\ h_3(PushSeq_{(p,\gamma,q)})$	$r = \langle p,\gamma\rangle \hookrightarrow \langle p',\gamma'\rangle \in \Delta, q \in Q$
	$h_3 = \lambda x.x \otimes f(r)$		
(5)	$PushSeq_{(p',\gamma',p'_{\gamma'})}$	$\rightarrow\ h_4(\epsilon)$	$\langle p,\gamma\rangle \hookrightarrow \langle p',\gamma'\gamma''\rangle \in \Delta$
	$h_4 = \overline{1}$		
(6)	$PushSeq_{(p'_{\gamma'},\gamma'',q)}$	$\rightarrow\ h_5(PushSeq_{(p,\gamma,q)})$	$r = \langle p,\gamma\rangle \hookrightarrow \langle p',\gamma'\gamma''\rangle \in \Delta, q \in Q$
	$h_5 = \lambda x.x \otimes f(r)$		

Figure 2.14 An abstract grammar problem for solving GPS.

It is based on the unweighted grammar shown in Fig. 2.10. The following lemma follows from Lem. 2.3.11.

Lemma 2.4.10. *For a transition t in the automaton that results from running poststar($\mathcal{A}$), the weight on t is exactly* JOD($PushSeq_t$).

Complexity

The following lemma states the complexity of *poststar* by the algorithm of Reps et al. [83], which is the same as the one described earlier, but with a few optimizations. We will assume that the time to perform an $\otimes$ and a $\oplus$ are the same, and use the notation $O_s(.)$ to denote the time bound in terms of semiring operations. The *height* of a weight domain is defined to be the length of the longest ascending chain in the domain.

Lemma 2.4.11. *[83] Given a WPDS with PDS $\mathcal{P} = (P, \Gamma, \Delta)$, if $\mathcal{A} = (Q, \Gamma, \rightarrow, P, F)$ is a $\mathcal{P}$-automaton that accepts an input set of configurations, poststar produces a backward weighted automaton with at most $|Q| + |\Delta|$ states in time $O_s(|P||\Delta|(|Q_0| + |\Delta|)H + |P||\lambda_0|H)$, where $Q_0 = Q \backslash P$, $\lambda_0 \subseteq \rightarrow$ is the set of all transitions leading from states in Q_0, and H is the height of the weight domain.*

Approximate Analysis

Among the properties imposed by a weight domain, one important property is distributivity (Defn. 2.4.1, item 2). This is a common requirement for a precise analysis, which also arises in various coincidence theorems for dataflow analysis [44, 88, 52]. Sometimes this requirement is too strict and may be relaxed to monotonicity, i.e., for all $a, b, c \in D$, $a \otimes (b \oplus c) \sqsubseteq (a \otimes b) \oplus (a \otimes c)$ and $(a \oplus b) \otimes c \sqsubseteq (a \otimes c) \oplus (b \otimes c)$. In such cases, the IJOP computation may not be precise, but it will be *safe* under the partial order $\sqsubseteq$.

2.4.2 Weight Domains

This section gives a few weight domains (i.e., bounded idempotent semirings) and the kind of analysis they permit when used in a WPDS.

Definition 2.4.12. *Let $\mathbb{B}$ be the set of Boolean values $\{true, false\}$. The **Boolean weight domain** is defined as $(\mathbb{B}, \vee, \wedge, false, true)$.*

A Boolean weight domain is the most trivial example of a weight domain. A WPDS with such a weight domain effectively reduces to its underlying PDS (after deleting rules with $\overline{0}$ weight): $\text{IJOP}(c_1, c_2) = true$ if and only if there is a path in the PDS from c_1 to c_2. In this dissertation, when we present an algorithm for WPDSs, the same algorithm can be reworked for PDSs by considering this weight domain.

Definition 2.4.13. *If G is a finite set, then the **relational weight domain** on G is defined as $(2^{G \times G}, \cup, ;, \emptyset, id)$: weights are binary relations on G, combine is union, extend is relational composition (";"), $\overline{0}$ is the empty relation, and $\overline{1}$ is the identity relation on G.*

This weight domain is the one used for encoding Boolean programs, as shown in Section 2.4. Weights in such a semiring can be encoded symbolically, using BDDs [85]. The extend and combine operations can be implemented efficiently on BDDs.

Definition 2.4.14. *The **minpath semiring** is the weight domain* $\mathcal{M} = (\mathbb{N} \cup \{\infty\}, min, +, \infty, 0)$*: weights are non-negative integers including "infinity", combine is minimum, and extend is addition.*

If all rules of a WPDS are given the weight 1 from this semiring (different from the semiring weight $\overline{1}$, which is the integer 0), then the IJOP weight between two configurations is the length of the shortest valid path (shortest valid rule sequence) between them.

Another infinite weight domain, which is based on the minpath semiring, is given in [56] and was shown to be useful for debugging programs.

The minpath semiring can be combined with a relational weight domain, for example, to find the shortest (valid) path in a Boolean program (for finding the shortest trace that exhibits some property).

Definition 2.4.15. *A **weighted relation** on a set* S*, weighted with semiring* $(D, \oplus, \otimes, \overline{0}, \overline{1})$*, is a function from* $(S \times S)$ *to* D*. The composition of two weighted relations* R_1 *and* R_2 *is defined as* $(R_1; R_2)(s_1, s_3) = \oplus\{w_1 \otimes w_2 \mid \exists s_2 \in S : w_1 = R_1(s_1, s_2), w_2 = R_2(s_2, s_3)\}$*. The union of the two weighted relations is defined as* $(R_1 \cup R_2)(s_1, s_2) = R_1(s_1, s_2) \oplus R_2(s_1, s_2)$*. The identity relation is the function that maps each pair* (s, s) *to* $\overline{1}$ *and others to* $\overline{0}$*. The reflexive transitive closure is defined in terms of these operations, as before. If* $\rightarrow$ *is a weighted relation and* $(s_1, s_2, w) \in \rightarrow$*, then we write* $s_1 \xrightarrow{w} s_2$*.*

Definition 2.4.16. *If* $\mathcal{S}$ *is a weight domain with set of weights* D *and* G *is a finite set, then the relational weight domain on* $(G, \mathcal{S})$ *is defined as* $(2^{G \times G \rightarrow D}, \cup, ;, \emptyset, id)$*: weights are weighted relations on* G *and the operations are the corresponding ones for weighted relations.*

If G is the set of global states of a Boolean program, then the relational weight domain on $(G, \mathcal{M})$ can be used for finding the shortest trace: for each rule, if $R \subseteq G \times G$ is the

effect of executing the rule on the global state of the Boolean program, then associate the following weight with the rule:

$$\{g_1 \xrightarrow{1} g_2 \mid (g_1, g_2) \in R\} \cup \{g_1 \xrightarrow{\infty} g_2 \mid (g_1, g_2) \notin R\}.$$

Then, if $w = \mathrm{IJOP}(C_1, C_2)$, the length of the shortest path that starts with global state g from a configuration in C_1 and ends at global state g' in a configuration in C_2, is $w(g, g')$ (which would be ∞ if no path exists). (Moreover, if a finite-length path does exist, a witness trace can be obtained to identify the elements of the path.)

2.4.3 Verifying Finite-State Properties

In this section, we give an instance of how property verification can be converted to a reachability problem, which is a very basic form of assertion checking. We suppose that the property is given in the form of a finite-state machine and the abstract model of the program is a WPDS.

The property is supplied as a finite-state automaton that performs transitions on ICFG nodes. The automaton has a designated error state, and runs (i.e., ICFG paths) that drive it to the error state are considered potentially erroneous program executions. For instance, the automaton shown in Fig. 2.15 can be used to verify the absence of null-pointer dereferences (for a pointer `p` in the program) by matching automaton-edge labels against program statements on ICFG nodes. For example, we would associate the transition label `p = NULL` with every ICFG node that has this statement. The reader is referred to other papers for more examples of useful finite-state properties [4, 21].

More formally, a program is abstracted to a WPDS $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$, where $\mathcal{P} = (P, \Gamma, \Delta)$. Let the initial configuration of the program be c_0. A property automaton $\mathcal{A}$ is the tuple $(Q, \Gamma, \rightarrow, q_0, F)$, where Q is a finite set of control states, Γ is the transition alphabet, $\rightarrow \subseteq Q \times \Gamma \times Q$ is the transition relation, $q_0 \in Q$ is the initial state and $F \subseteq Q$ is the set of final states. A word (in Γ^*) accepted by $\mathcal{A}$ is considered to be an erroneous program execution. The verification problem is to find if there is a path σ in $\mathcal{W}$ with non-$\overline{0}$ weight, starting from c_0, such that the nodes visited by σ are in the language of $\mathcal{A}$.

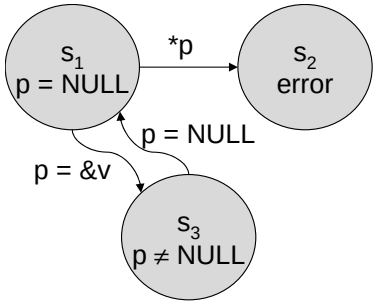


Figure 2.15 A finite-state machine for checking null-pointer dereferences in a program. The initial state of the machine is s_1. The label "`p = &v`" stands for the assignment of a non-null address to the pointer `p`. We assume that the machine stays in the same state when it has to transition on an undefined label.

This problem can be solved by taking a cross-product of $\mathcal{W}$ and $\mathcal{A}$. Define the relation $R_\gamma \subseteq Q \times Q$ to be $\{(q_1, q_2) \mid (q_1, \gamma, q_2) \in \rightarrow\}$, i.e., R_γ is the projection of the transition relation of $\mathcal{A}$ to only those that fire on γ. Define a WPDS $\mathcal{W}' = (\mathcal{P}, \mathcal{S}', f')$, where $\mathcal{S}'$ is the relational weight domain on $(Q, \mathcal{S})$ (Defn. 2.4.16), and $f'(r)$ is defined as follows: if $r = \langle p, \gamma \rangle \hookrightarrow \langle p', u \rangle$ then $f'(r) = \{q_1 \xrightarrow{f(r)} q_2 \mid (q_1, q_2) \in R_\gamma\}$.[6]

A path σ in $\mathcal{W}$, with weight w, is accepted by $\mathcal{A}$ if and only if the same path in $\mathcal{W}'$ has weight w' such that $(q_0, q_f, w) \in w'$ for some $q_f \in F$. Consequently, $\mathcal{W}$ has a path with non-$\overline{0}$ weight, starting from c_0, that is accepted by $\mathcal{A}$ if and only if $(q_0, q_f, w) \in \mathrm{IJOP}_{\mathcal{W}'}(\{c_0\}, \Gamma^*)$ for some $q_f \in F$, and $w \neq \overline{0}$. This shows that solving for the IJOP weight is sufficient to verify finite-state properties on WPDSs.

[6]This construction is due to David Melski, and was used in an experimental version of the Path Inspector [35].

Chapter 3

Extended Weighted Pushdown Systems

In Chapter 2, we covered some common abstract models and defined the join-over-all-paths (JOP) value, and several variants (JOVP or IJOP), on those models. The JOP value, for a node n, is the net transformation over all paths in the model that reach n. From this value, the set of all reachable states at n (in the model) can be computed, and used for checking assertions at node n. Thus, it is desirable to compute the precise JOP value. However, the definition of JOP is declarative in nature and cannot be directly computed because it involves combining the effect of an unbounded number of paths. However, under certain conditions, the JOP value can be computed precisely.

Chapter 2 presented two results in this direction. First, it presented the Kam and Ullman coincidence theorem [44] (Section 2.1) that provides sufficient conditions under which the JOP value can be calculated for single-procedure dataflow models. This result was extended to multiple-procedure models by Sharir and Pnueli [88]. Second, the weighted pushdown system (WPDS) model also specifies certain conditions (namely, that the weights should come from a bounded-idempotent semiring), which when satisfied, imply that the *poststar* and *prestar* algorithms can be used to precisely compute the JOP weight. However, with all these models, it is not clear how to encode programs with multiple procedures and local variables in such a way that all these conditions are satisfied. (Note that we only considered examples without local variables in Chapter 2.) In this chapter, we study an abstract model that provides a straightforward way of encoding programs with local variables, and show how to precisely compute JOP values in the model.

In dataflow analysis, this challenge of incorporating local variables was addressed by Knoop and Steffen [52] by extending Sharir and Pnueli's coincidence theorem to model the run-time stack of a program. Their work is summarized in Section 3.3. Alternative techniques for handling local variables have been proposed in [81, 84], but these lose certain relationships between local and global variables.

As shown in Chapter 2, WPDSs generalize dataflow analysis. For instance, in interprocedural dataflow analysis, the JOVP value for a program node represents the set of all possible reachable states at that node regardless of its calling context. Using WPDSs, one can answer "stack-qualified queries" that calculate the set of states that can occur at a program point for a given regular set of calling contexts. Moreover, the ability to represent reachable states along with their stack contents (in the form of a weighted automaton) will be critical in designing the algorithms and techniques presented in later chapters.

However, as with Sharir and Pnueli's coincidence theorem, it is not clear if WPDSs can handle local variables accurately. In this chapter, we extend the WPDS model to the Extended-WPDS or EWPDS model, which can accurately encode interprocedural analyses on programs with local variables and answer stack-qualified queries about them. The EWPDS model can be seen as generalizing WPDS in much the same way that Knoop and Steffen generalized Sharir and Pnueli's coincidence theorem.

The contributions of the work presented in this chapter can be summarized as follows:

- We give a way of handling local variables in the WPDS model. The advantage of using WPDSs is that they give a way of calculating IJOP weights that hold at a program node for a particular calling context (or set of calling contexts). They can also provide a set of witness program execution paths that justify a reported dataflow value.

- We show that the EWPDS model is powerful enough to capture Knoop and Steffen's coincidence theorem. In particular, this means that we can calculate the IJOP value for any distributive dataflow-analysis problem for which the domain of transfer functions has no infinite ascending chains.

- We have extended the WPDS library [49] to support EWPDSs and used it in an application that calculates affine relationships that hold between registers in x86 code [3].

- We show how to encode several abstract models using EWPDSs. These abstract models include Boolean programs (Section 3.5.1), affine programs (Section 3.5.2), and programs with single-level pointers [61] (Section 3.5.3). This shows that the analysis of all these models can be carried out using EWPDS reachability algorithms. It also gives us something new: a way of answering stack-qualified queries on all these models.

The rest of this chapter is organized as follows: Section 3.1 defines the EWPDS model. Section 3.2 presents algorithms to solve reachability queries in EWPDSs. Section 3.3 presents Knoop and Steffen's coincidence theorem for dataflow analysis and shows that the theorem can also be obtained using EWPDSs. Section 3.4 presents experimental results. Section 3.5 presents various applications of EWPDSs by showing how different abstract models can be encoded using EWPDSs. Section 3.6 describes related work. Section 3.7 has proofs of the theorems in this chapter.

3.1 Defining the EWPDS Model

We start by recalling the definitions of reachability problems on WPDSs.

Definition 3.1.1. *Let $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$ be a weighted pushdown system, where $\mathcal{P} = (P, \Gamma, \Delta)$, and let $C \subseteq P \times \Gamma^*$ be a regular set of configurations. The* **generalized pushdown predecessor (GPP) problem** *is to find for each $c \in P \times \Gamma^*$:*

$$\delta(c) \stackrel{\text{def}}{=} \bigoplus \{\, v(\sigma) \mid \sigma \in \mathit{paths}(c, c'), c' \in C \,\}$$

The **generalized pushdown successor (GPS) problem** *is to find for each $c \in P \times \Gamma^*$:*

$$\delta(c) \stackrel{\text{def}}{=} \bigoplus \{\, v(\sigma) \mid \sigma \in \mathit{paths}(c', c), c' \in C \,\}$$

We aim to solve each of these problems on the EWPDS model as well. These require computing the weight of a rule sequence. Rule sequences, in general, represent interprocedural paths in a program, and such paths can have unfinished procedure calls, e.g., when

the path ends in the middle of a called procedure. To compute the weight of such paths, we have to maintain information about local variables of all unfinished procedures that appear on the path.

We allow for local variables to be stored at call sites and then use special *merge functions* to appropriately merge them with the value returned by a procedure. Merge functions are defined as follows:

Definition 3.1.2. *A function $g : D \times D \to D$ is a* **merge function** *with respect to a bounded idempotent semiring $\mathcal{S} = (D, \oplus, \otimes, \overline{0}, \overline{1})$ if it satisfies the following properties.*

1. **Strictness.** *For all $a \in D$, $g(\overline{0}, a) = g(a, \overline{0}) = \overline{0}$.*
2. **Distributivity.** *The function distributes over $\oplus$ in each argument. For all $a, b, c \in D$,*
$$g(a \oplus b, c) = g(a, c) \oplus g(b, c) \text{ and } g(a, b \oplus c) = g(a, b) \oplus g(a, c)$$
3. **Path Extension**.[1] *For all $a, b, c \in D$, $g(a \otimes b, c) = a \otimes g(b, c)$.*

For a set of pushdown rules Δ, we use $\Delta_i \subseteq \Delta$ to denote the set of all rules with i stack symbols on the right-hand side.

Definition 3.1.3. *An* **extended weighted pushdown system** *is a quadruple $\mathcal{W}_e = (\mathcal{P}, \mathcal{S}, f, g)$ where $(\mathcal{P}, \mathcal{S}, f)$ is a weighted pushdown system and $g : \Delta_2 \to \mathcal{G}$ assigns a merge function to each rule in Δ_2, where $\mathcal{G}$ is the set of all merge functions on the semiring $\mathcal{S}$. We will write g_r as a shorthand for $g(r)$.*

Note that a push rule has both a weight and a merge function associated with it. The merge functions are used to combine the effects of a called procedure with those made by the calling procedure just before the call. As an example, refer to Fig. 3.1, which is similar to the dataflow model shown in Fig. 2.1, except that it has local variables as well. The dataflow values used for this model are environments of the form $Env = (Var \to \mathbb{Z}^\top) \cup \{\bot\}$, with join ($\sqcup$) defined pointwise. This model can be encoded as an EWPDS $(\mathcal{P}, \mathcal{S}, f, g)$ as

[1]This property can be too restrictive in some cases; Section 3.2.3 discusses how this property may be dispensed with. In most cases, however, the path-extension property does hold.

```
int y;

void main() {
  n1: int a = 5;
  n2: y = 1;
  n3,n4: f(a);
  n5: if(...) {
    n6: a = 2;
    n7,n8: f(a);
  }
  n9: ...;
}

void f(int b) {
  n10: if(...)
    n11: y = 2;
  else
    n12: y = b;
}
```

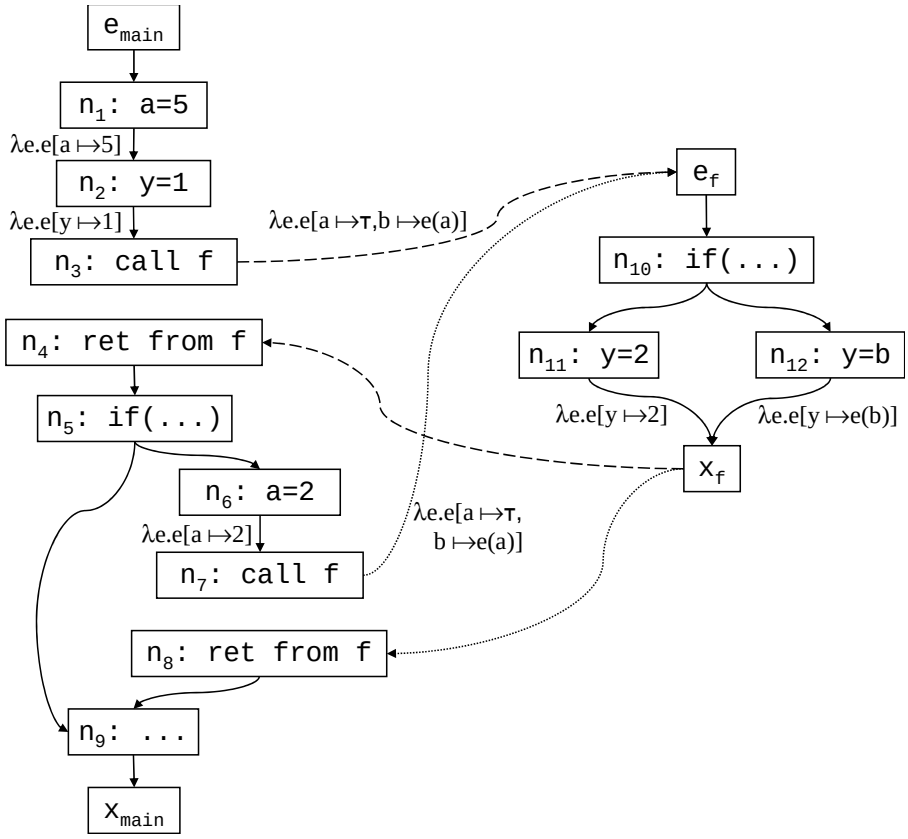


Figure 3.1 A program fragment and its ICFG. For all unlabeled edges, the environment transformer is $\lambda e.e$. The statements labeled "..." are assumed not to change any of the declared variables.

follows: the control-flow of the model is encoded using the PDS $\mathcal{P}$. The weight domain $\mathcal{S}$ is $(D, \oplus, \otimes, 0, 1)$ where $D = (Env \to Env)$ is the set of all environment transformers that are $\perp$-strict, i.e., for all $d \in D$, $d(\perp) = \perp$. The semiring operations and constants are defined as follows:

$$\begin{array}{lcllcl} \overline{0} & = & \lambda e.\perp & w_1 \oplus w_2 & = & \lambda e.w_1(e) \sqcup w_2(e) \\ \overline{1} & = & \lambda e.e & w_1 \otimes w_2 & = & w_2 \circ w_1 \end{array}$$

The weights for the PDS rules are the corresponding edge labels in Fig. 2.1. Merge functions are two-argument functions on weights. The merge function for call site n_3 will

receive two environment transformers: one that summarizes the effect of the caller from its entry point to the call site (e_{main} to n_3) and one that summarizes the effect of the called procedure (e_f to x_f). It then has to output the transformer that summarizes the effect of the caller from its entry point to the return site (e_{main} to n_4). We define it as follows:

$$\begin{aligned} g(w_1, w_2) \quad &= \quad \textbf{if } (w_1 = \overline{0} \text{ or } w_2 = \overline{0}) \textbf{ then } \overline{0} \\ &\qquad\quad \textbf{else } \lambda e.e[\mathtt{a} \mapsto w_1(e)(\mathtt{a}), \mathtt{y} \mapsto (w_1 \otimes w_2)(e)(\mathtt{y})] \end{aligned}$$

It copies over the value of the local variable `a` from the call site, and gets the value of `y` from the called procedure. Because the merge function has access to the environment transformer just before the call, we do not need to pass the value of local variable `a` into procedure *f*. This is achieved by the weight on the call rule at n_3 that maps `a` to $\top$. Moreover, the merge function also ensures that the local variables of `f`, which are present in weight w_2, do not get passed into `main`.

The main change that EWPDSs require over WPDSs is the way the weight of a path (rule sequence) is calculated because the merge functions have to be incorporated. The technical difference, formalized below, is that paths have to be "parsed" in EWPDSs to find matching calls and returns so that the appropriate weights are calculated to be passed to the merge functions. For instance, consider the path $[e_{main}, n_1, n_2, n_3, e_f, n_{10}, n_{11}, x_f, n_4]$. We first need to calculate the weights of the sub-paths $[e_f, n_{10}, n_{11}, x_f]$ and $[e_{main}, n_1, n_2, n_3]$, and then pass these weights to the merge function. In WPDSs, there was no such order imposed in calculating the weight of a path.

To formalize this notion, we redefine the generalized pushdown predecessor and successor problem by changing how we define the value of a rule sequence. If $\sigma \in \Delta^*$ with $\sigma = [r_1, r_2, \cdots, r_k]$ then let $(r\ \sigma)$ denote the sequence $[r, r_1, \cdots, r_k]$. Also, let $[\ ]$ denote the empty sequence. Consider the context-free grammar shown in Fig. 3.2. σ_s is simply R_1^*. σ_b represents a *balanced* sequence of rules that have matched calls (R_2) and returns (R_0) with any number of rules from Δ_1 in between. σ_i is just $(R_2 \mid \sigma_b)^+$ in regular-language terminology, and represents sequences that increase stack height. σ_d is $(R_0 \mid \sigma_b)^+$ and

represents rule sequences that decrease stack height. σ_a can derive any rule sequence. We will use this grammar to define the value of a rule sequence.

$$
\begin{array}{lcllcllcl}
R_0 & \to & r \ \ (r \in \Delta_0) & \sigma_s & \to & [\,] \mid R_1 \mid \sigma_s \ \sigma_s & \sigma_i & \to & R_2 \mid \sigma_b \mid \sigma_i \ \sigma_i \\
R_1 & \to & r \ \ (r \in \Delta_1) & \sigma_b & \to & \sigma_s \mid \sigma_b \ \sigma_b & \sigma_d & \to & R_0 \mid \sigma_b \mid \sigma_d \ \sigma_d \\
R_2 & \to & r \ \ (r \in \Delta_2) & & \mid & R_2 \ \sigma_b \ R_0 & \sigma_a & \to & \sigma_d \ \sigma_i
\end{array}
$$

Figure 3.2 Grammar used for parsing rule sequences. The start symbol of the grammar is σ_a.

Definition 3.1.4. *Given an EWPDS $\mathcal{W}_e = (\mathcal{P}, \mathcal{S}, f, g)$ and a rule sequence $\sigma \in \Delta^*$, we define its* **value** *$v(\sigma)$ by parsing σ according to the grammar of Fig. 3.2 and computing the weight using its derivation tree as follows:*

$$
\begin{array}{lllll}
1. & v(r) = f(r) & & 5. & v(R_2 \ \sigma_b \ R_0) = g_{R_2}(\overline{1}, v(\sigma_b) \otimes v(R_0)) \\
2. & v([\,]) = 1 & & 6. & v(\sigma_d \ \sigma_d) = v(\sigma_d) \otimes v(\sigma_d) \\
3. & v(\sigma_s \ \sigma_s) = v(\sigma_s) \otimes v(\sigma_s) & & 7. & v(\sigma_i \ \sigma_i) = v(\sigma_i) \otimes v(\sigma_i) \\
4. & v(\sigma_b \ \sigma_b) = v(\sigma_b) \otimes v(\sigma_b) & & 8. & v(\sigma_d \ \sigma_i) = v(\sigma_d) \otimes v(\sigma_i)
\end{array}
$$

Here we have used g_{R_2} as a shorthand for g_r where r is the terminal derived by R_2.

The main thing to note in the above definition is the application of merge functions on balanced sequences. The path-extension property of merge functions allow us to compute $g(w_1, w_2)$ as $w_1 \otimes g(\overline{1}, w_2)$. An alternative grammar is given in Section 3.2.3 when the path extension property does not hold. Because the grammar presented in Fig. 3.2 is ambiguous, there might be many parsings of the same rule sequence, but all of them would produce the same value because the extend operation is associative and there is a unique way to balance R_2s with R_0s.

The generalized pushdown problems GPP and GPS for EWPDSs are the same as those for WPDSs except for the changed definition of the value of a rule sequence. If we let each merge function be $g_r(w_1, w_2) = w_1 \otimes f(r) \otimes w_2$, then the EWPDS reduces to a WPDS. From

now on, whenever we talk about generalized pushdown problems in this chapter, we mean it in the context of EWPDSs.

3.2 Solving Reachability Problems in EWPDSs

In this section, we present algorithms to solve the generalized reachability problems for EWPDSs. Let $\mathcal{W}_e = (\mathcal{P}, \mathcal{S}, f, g)$ be an EWPDS where $\mathcal{P} = (P, \Gamma, \Delta)$ is a pushdown system and $\mathcal{S} = (D, \oplus, \otimes, 0, 1)$ is the weight domain. Let C be a fixed regular set of configurations that is recognized by a $\mathcal{P}$-automaton $\mathcal{A} = (Q, \Gamma, \rightarrow_0, P, F)$. We will follow the above notation throughout this section. As in the case of WPDSs, we will construct a weighted automaton that represents the set of reachable configurations along with their weights. The automaton will be the same as the automaton constructed for WPDS reachability (Section 2.4.1) except for the weights. We will not show the calculation of witness annotations because they are obtained in exactly the same way as for WPDSs [83]. The reason why the computation is unchanged is because witnesses record the paths that justify a weight and not how the values of those paths were calculated.

3.2.1 Solving GPP

To solve GPP, we take as input a $\mathcal{P}$-automaton $\mathcal{A}$ that describes the starting set of configurations. As output, we create a weighted automaton $\mathcal{A}_{pre^*}$. The algorithm is based on the saturation rule shown in Fig. 3.3. Starting with the automaton $\mathcal{A}$, we keep applying this rule until it no longer causes any changes. Termination is guaranteed because there are a finite number of transitions and there are no infinite ascending chains in a weight domain. For each transition in the automaton being created, we store the weight on it using function l. The saturation rule is the same as that for predecessor reachability in ordinary pushdown systems, except for the weights, and is different from the one for weighted pushdown systems only in the last case, where a merge function is applied.

Theorem 3.2.1. *The saturation rule shown in Fig. 3.3 solves* GPP *for EWPDSs, i.e., for a configuration* $c = \langle p, \gamma_1\gamma_2\cdots\gamma_n\rangle$, $\delta(c, \mathcal{L}(\mathcal{A}))$, *defined as* $\mathrm{IJOP}(\{c\}, \mathcal{L}(\mathcal{A}))$, *is* $\mathcal{A}_{pre^*}(c)$.

- If $r = \langle p, \gamma \rangle \hookrightarrow \langle p', \varepsilon \rangle$, then update the weight on $t = (p, \gamma, p')$ to $l(t) \leftarrow l(t) \oplus f(r)$.
- If $r = \langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \rangle$ and there is a transition $t = (p', \gamma', q)$, then update the weight on $t' = (p, \gamma, q)$ to $l(t') \leftarrow l(t') \oplus (f(r) \otimes l(t))$.
- If $r = \langle p, \gamma \rangle \hookrightarrow \langle p', \gamma'\gamma'' \rangle$ and there are transitions $t = (p', \gamma', q_1)$ and $t' = (q_1, \gamma'', q_2)$, then update the weight on $t'' = (p, \gamma, q_2)$ to

$$l(t'') \leftarrow l(t'') \oplus \begin{cases} f(r) \otimes l(t) \otimes l(t') & \text{if } q_1 \notin P \\ g_r(1, l(t)) \otimes l(t') & \text{otherwise} \end{cases}$$

Figure 3.3 Saturation rule for constructing $\mathcal{A}_{pre^*}$ from $\mathcal{A}$. In each case, if a transition t does not yet exist, it is treated as if $l(t)$ equals $\overline{0}$.

A proof of this theorem is given in Section 3.7.

3.2.2 Solving GPS

For this section, we shall assume that we can have at most one rule of the form $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma'\gamma'' \rangle$ for each combination of p',γ', and γ''. This involves no loss of generality because we can replace a rule $r = \langle p, \gamma \rangle \hookrightarrow \langle p', \gamma'\gamma'' \rangle$ with two rules: (a) $r' = \langle p, \gamma \rangle \hookrightarrow \langle p_r, \gamma'\gamma'' \rangle$ with weight $f(r)$ and merge function g_r, and (b) $r'' = \langle p_r, \gamma' \rangle \hookrightarrow \langle p', \gamma' \rangle$ with weight $\overline{1}$, where p_r is a new state. This replacement does not change the reachability problem's answers. Let *lookupPushRule* be a function that returns the unique push rule associated with a triple (p', γ', γ'') if there is one.

Before presenting the algorithm, let us consider an operational definition of the value of a rule sequence. The importance of this alternative definition is that it shows the correspondence with the call semantics of a program. For each interprocedural path in a program, we define a stack of weights that contains a weight for each unfinished call in the path. Elements of the stack are from the set $D \times D \times \Delta_2$ (recall that Δ_2 was defined as the set of all push rules in Δ), where (w_1, w_2, r) signifies that (i) a call was made using rule r, (ii) the weight at the time of the call was w_1, and (iii) w_2 was the weight on the call rule.

Let $STACK = D.(D \times D \times \Delta_2)^*$ be the set of all nonempty stacks where the topmost element is from D and the rest are from $(D \times D \times \Delta_2)$. We will write an element $(w_1, w_2, r) \in D \times D \times \Delta_2$ as $(w_1, w_2)_r$. For each rule $r \in \Delta$ of the form $\langle p, \gamma \rangle \hookrightarrow \langle p', u \rangle$, $u \in \Gamma^*$, we will associate a function $[\![r]\!] : STACK \rightarrow STACK$. Let $S \in (D \times D \times \Delta_2)^*$.

- If r has one symbol on the right-hand side ($|u| = 1$), then accumulate its weight on the top of the stack.

$$[\![r]\!]\ (w_1\ S) = ((w_1 \otimes f(r))\ S)$$

- If r has two symbols on the right-hand side ($|u| = 2$), then save the weight of the push rule as well as the push rule itself on the stack and start a fresh entry on the top of the stack.

$$[\![r]\!]\ (w_1\ S) = (\overline{1}\ (w_1, f(r))_r\ S)$$

- If r has no symbols on the right-hand side ($|u| = 0$), then apply the appropriate merge function if there is a something pushed on the stack. Otherwise, r represents an unbalanced pop rule and simply accumulate its weight on the stack.

$$\begin{aligned} [\![r]\!]\ (w_1\ (w_2, w_3)_{r_1}\ S) &= ((g_{r_1}(w_2, w_1 \otimes f(r))\ S) \\ [\![r]\!]\ (w_1) &= (w_1 \otimes f(r)) \end{aligned} \tag{3.1}$$

  Note that we drop the weight w_3 of the push rule r_1 when we apply the merge function. This is in accordance with case 5 of Defn. 3.1.4.

For a sequence of rules $\sigma = [r_1, r_2, \cdots, r_n]$, define $[\![\sigma]\!] = [\![[r_2, \cdots, r_n]]\!] \circ [\![r_1]\!]$. Let $\mathit{flatten} : STACK \rightarrow D$ be an operation that computes a weight from a stack as follows:

$$\mathit{flatten}(w_1\ S) = \mathit{flatten}'(S) \otimes w_1 \qquad \begin{aligned} \mathit{flatten}'((\ )) &= \overline{1} \\ \mathit{flatten}'((w_1, w_2)_r\ S) &= \mathit{flatten}'(S) \otimes (w_1 \otimes w_2) \end{aligned}$$

Example 3.2.2. *Consider the rule sequence σ corresponding to the path in Fig. 3.1 that goes from e_{main} to n_3 to n_{11} to x_f. If we apply $[\![\sigma]\!]$ to a stack containing just $\overline{1}$, we get a stack of height 2 as follows:* $[\![\sigma]\!](\overline{1}) = ((\lambda e.e[\mathtt{y} \mapsto 2])\ (\lambda e.e[\mathtt{a} \mapsto 5, \mathtt{y} \mapsto 1], \lambda e.e[\mathtt{a} \mapsto \top, \mathtt{b} \mapsto e(\mathtt{a})])_r)$,

where r is the push rule that calls procedure f at node n_3. The top of the stack is the weight computed inside procedure f, and the bottom of the stack contains a pair of weights: the first component is the weight computed in main just before the call; the second component is just the weight of the call rule r. If we apply the flatten operation on this stack, we get the weight $\lambda e.e[\mathtt{a} \mapsto \top, \mathtt{y} \mapsto 2, \mathtt{b} \mapsto 5]$, which is exactly the value $v(\sigma)$. When we apply the pop rule r' corresponding to the procedure return at x_f to this stack, we get:

$$\begin{aligned} [\![\sigma\ r']\!](\overline{1}) &= [\![r']\!] \circ [\![\sigma]\!](\overline{1}) \\ &= (g_r(\lambda e.e[\mathtt{a} \mapsto 5, \mathtt{y} \mapsto 1], \lambda e.e[\mathtt{y} \mapsto 2])) \\ &= (\lambda e.e[\mathtt{a} \mapsto 5, \mathtt{y} \mapsto 2]) \end{aligned}$$

Again, applying flatten on this stack gives us $v(\sigma\ r')$.

The following lemma formalizes the equivalence between $[\![\sigma]\!]$ and $v(\sigma)$.

Lemma 3.2.3. *For any valid sequence of rules σ ($\sigma \in paths(c, c')$ for some configurations c and c'), $[\![\sigma]\!]\ (\overline{1}) = S$ such that flatten$(S) = v(\sigma)$.*

Corollary 3.2.4. *Let C be a set of configurations. For a configuration c, let $\delta_S(c) \subseteq STACK$ be defined as follows:*

$$\delta_S(C, c) = \{[\![\sigma]\!](\overline{1}) \mid \sigma \in paths(c', c),\ c' \in C\}.$$

Then: $\delta(C, c)$, defined in Defn. 2.4.4, is $\oplus\{flatten(S) \mid S \in \delta_S(C, c)\}$.

The above corollary shows that $\delta_S(C, c)$ has enough information to compute $\delta(C, c)$ directly. To solve the pushdown successor problem, we take the input $\mathcal{P}$-automaton $\mathcal{A}$ that describes the starting set of configurations and create a weighted automaton $\mathcal{A}_{post^*}$ from which we can read off the value of $\delta(\mathcal{L}(\mathcal{A}), c)$ for any configuration c. The algorithm is again based on a saturation rule. For each transition in the automaton being created, we have a function l that stores the weight on the transition. Based on the above operational definition of the value of a path, we create $\mathcal{A}_{post^*}$ on pairs of weights, that is, over the semiring $(D \times D, \oplus, \otimes, (\overline{0}, \overline{0}), (\overline{1}, \overline{1}))$ where $\oplus$ and $\otimes$ are defined component-wise. Also, we introduce a new state for each push rule. So the states of $\mathcal{A}_{post^*}$ are $Q \cup Q_{\text{mid}}$, where $Q_{\text{mid}} = \{p'_{\gamma'} \mid \langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\gamma''\rangle \in \Delta\}$. The saturation rule is shown in Fig. 3.4. To see what the saturation rule does, consider a path in $\mathcal{A}_{post^*}$: $\tau = q_1 \xrightarrow{\gamma_1} q_2 \xrightarrow{\gamma_2} \cdots \xrightarrow{\gamma_n} q_{n+1}$. As an invariant

of our algorithm, we have $q_1 \in (P \cup Q_{\text{mid}})$; $q_2, \cdots, q_k \in Q_{\text{mid}}$; and $q_{k+1}, \cdots, q_{n+1} \in (Q - P)$ for some $0 \leq k \leq n+1$. This is because of the fact that we never create transitions to a state in P or from a state in $Q - P$ to a state in Q_{mid}. Define a new transition label $l'(t)$ as follows:

$$l'(p, \gamma, q) = \mathit{lookupPushRule}(p', \gamma', \gamma) \text{ if } p \equiv p'_{\gamma'}$$

Another invariant of our algorithm is that every transition t to a state in Q_{mid} has $l'(t)$ defined. Then the path τ describes the *STACK* $\mathit{vpath}(\tau) = (l_1(t_1) \;\; l(t_2)_{l'(t_2)} \cdots \; l(t_k)_{l'(t_k)})$ where $t_i = (q_i, \gamma_i, q_{i+1})$ and $l_1(t)$ is the first component projected out of the weight-pair $l(t)$. This means that each path in $\mathcal{A}_{post^*}$ represents a *STACK* and all the saturation algorithm does is to make the automaton rich enough to encode all *STACK*s in $\delta_S(\mathcal{L}(\mathcal{A}), c)$ for all configurations c. The first and third cases of the saturation rule can be seen as applying $[\![r]\!]$ for rules with one and two stack symbols on the right-hand side, respectively. Applying the fourth case immediately after the second case can be seen as applying $[\![r]\!]$ for pop rules. We now have the following theorem.

Theorem 3.2.5. *The saturation rule shown in Fig. 3.4 solves* GPS *for EWPDSs. For a configuration $c = \langle p, u \rangle$, we have,*

$$\delta(\mathcal{L}(\mathcal{A}), c) = \oplus\{\mathit{flatten}(\mathit{vpath}(\sigma_t)) \mid \sigma_t \in \mathit{paths}(p, u, q_f), q_f \in F\}$$

where $\mathit{paths}(p, u, q_f)$ denotes the set of all paths of transitions in $\mathcal{A}_{post^*}$ that go from p to q_f on input u, i.e. $p \xrightarrow{u}{}^* q_f$.

A proof of this theorem is given in Section 3.7.

An easy way to compute the combine in Thm. 3.2.5 is to replace the annotation $l(t)$ on each transition t with $l_1(t) \otimes l_2(t)$, the extend of the two weight components of $l(t)$, and then use the *path_summary* algorithm.

3.2.3 Relaxing Merge Function Requirements

Defn. 3.1.2 requires merge functions to satisfy three properties. The first requirement (strictness) can be easily satisfied and the second requirement of distributivity is essential for saturation algorithms to work for the GPP and GPS problems. However, in some cases

- If $r = \langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \rangle$ and there is a transition $t = (p, \gamma, q)$ with annotation $l(t)$, then update the annotation on transition $t' = (p', \gamma', q)$ to $l(t') \leftarrow l(t') \oplus (l(t) \otimes (f(r), \overline{1}))$.

- If $r = \langle p, \gamma \rangle \hookrightarrow \langle p', \varepsilon \rangle$ and there is a transition $t = (p, \gamma, q)$ with annotation $l(t)$, then update the annotation on transition $t' = (p', \varepsilon, q)$ to $l(t') \leftarrow l(t') \oplus (l(t) \otimes (f(r), \overline{1}))$.

- If $r = \langle p, \gamma \rangle \hookrightarrow \langle p', \gamma'\gamma'' \rangle$ and there is a transition $t = (p, \gamma, q)$ with annotation $l(t)$ then let $t' = (p', \gamma', q_{p',\gamma'})$, $t'' = (q_{p',\gamma'}, \gamma'', q)$ and update annotations on them.

$$\begin{aligned} l(t') &\leftarrow l(t') \oplus (\overline{1}, \overline{1}) \\ l(t'') &\leftarrow l(t'') \oplus (l(t) \otimes (\overline{1}, f(r))) \end{aligned}$$

- If there are transitions $t = (p, \varepsilon, q)$ and $t' = (q, \gamma', q')$ with annotations $l(t) = (w_1, w_2)$ and $l(t') = (w_3, w_4)$ then update the annotation on the transition $t'' = (p, \gamma', q')$ to $l(t'') \leftarrow l(t'') \oplus w$ where w is defined as follows:

$$w = \begin{cases} (g_{lookupPushRule(p',\gamma',\gamma'')}(w_3, w_1), \overline{1}) & \text{if } q \equiv p'_{\gamma'} \\ l(t') \otimes l(t) & \text{otherwise} \end{cases}$$

Figure 3.4 Saturation rule for constructing $\mathcal{A}_{post^*}$ from $\mathcal{A}$. In each case, if a transition t' (or t'') does not yet exist, it is treated as if $l(t')$ (or $l(t'')$) equals $(\overline{0}, \overline{0})$.

we might not be able to satisfy the third property of path-extension (Section 3.3 presents one such case). Let us now consider what happens when merge functions do not satisfy this property.

The *prestar* algorithm of Section 3.2.1 (used for creating $\mathcal{A}_{pre^*}$) would still be correct because it parses rule sequences exactly as described in Defn. 3.1.4, but the *poststar* algorithm of Section 3.2.2 (used for creating $\mathcal{A}_{post^*}$) would not work as it utilizes a different

parsing and relies on the path-extension property for computing the correct value. Instead of trying to modify the *poststar* algorithm, we introduce an alternative definition of the value of a rule sequence that is suited for the cases when merge functions do not satisfy the path-extension property. The definition involves presenting a slightly more complicated but intuitive grammar for parsing rule sequences.

Definition 3.2.6. *Given an EWPDS $\mathcal{W}_e = (\mathcal{P}, \mathcal{S}, f, g)$ where the merge functions do not satisfy the path-extension property, the* **value** *of a rule sequence $\sigma \in \Delta^*$ is calculated in the same manner as given in Defn. 3.1.4, but we change the productions and valuations of balanced sequences as follows:*

$$\begin{array}{lcll} \sigma_{b'} & \rightarrow & [\,] \mid \sigma_{b'}\ \sigma_{b'} & v(\sigma_{b'}\ \sigma_{b'}) = v(\sigma_{b'}) \otimes v(\sigma_{b'}) \\ & \mid & \sigma_b\ R_2\ \sigma_b\ R_0 & v(\sigma_b\ R_2\ \sigma_b\ R_0) = g_{R_2}(v(\sigma_b), v(\sigma_b) \otimes v(R_0)) \qquad (2) \\ \sigma_b & \rightarrow & \sigma_{b'}\ \sigma_s & v(\sigma_{b'}\ \sigma_s) = v(\sigma_{b'}) \otimes v(\sigma_s) \end{array}$$

The value of a rule sequence as defined above is the same as the value defined by Defn. 3.1.4 when merge functions satisfy the path-extension property. In the absence of the property, we need to make sure that each occurrence of a merge function is applied to the weight computed in the calling procedure just before the call and the weight computed by the called procedure. We enforce this using Eqn. (2) values that we calculate for rule sequences in Section 3.2.2 also do the same in Eqn. (3.1)[Pg. 63]. This means that Lem. 3.2.3 still holds and the *poststar* algorithm correctly solves this more general version of GPS. However, the *prestar* algorithm is closely based on Defn. 3.1.4 and the way that it solves solves the generalized version of GPP is not based on the above alternative definition.

3.3 Knoop and Steffen's Coincidence Theorem

In this section, we show how EWPDSs can encode Knoop and Steffen's coincidence theorem [52] about interprocedural dataflow analysis in the presence of local variables. We refer to the IJOP value defined by Knoop and Steffen as the interprocedural-local-join-over-all-paths (LJOP) value. (Note that we are using different terminology than that used in [52]. This is to avoid confusion with the other terms defined in the previous chapters.)

We are given a join semilattice $(\mathcal{C}, \sqcup)$ that describes dataflow facts and the interprocedural-control-flow graph of a program $(\mathcal{N}, \mathcal{E})$, where $\mathcal{N}_C, \mathcal{N}_R \subseteq \mathcal{N}$ are the sets of call and return nodes, respectively. We are also given a semantic transformer for each node in the program: $[\![\ ]\!] : \mathcal{N} \to (\mathcal{C} \to \mathcal{C})$, which represents the effect of executing a statement in the program. Let $STK = \mathcal{C}^+$ be the set of all nonempty stacks with elements from $\mathcal{C}$. STK is used as an abstract representation of the run-time stack of a program. Define the following operations on stacks.

$newstack : \mathcal{C} \to STK$	creates a new stack with a single element
$push : STK \times \mathcal{C} \to STK$	pushes a new element on top of the stack
$pop : STK \to STK$	removes the topmost element of the stack
$top : STK \to \mathcal{C}$	returns the topmost element of the stack

We can now describe the interprocedural semantic transformer for each program node: $[\![\ ]\!]^* : \mathcal{N} \to (STK \to STK)$. For $stk \in STK$,

$$[\![n]\!]^*(stk) = \begin{cases} push(pop(stk), [\![n]\!](top(stk))) & \text{if } n \in \mathcal{N} - (\mathcal{N}_C \cup \mathcal{N}_R) \\ push(stk, [\![n]\!](top(stk))) & \text{if } n \in \mathcal{N}_C \\ push(pop(pop(stk)), \mathcal{R}_n(top(pop(stk)), [\![n]\!](top(stk)))) & \\ & \text{if } n \in \mathcal{N}_R \end{cases}$$

where $\mathcal{R}_n : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ is a merge function like we have in EWPDSs. It is applied to the dataflow value computed by the called procedure ($[\![n]\!](top(stk))$) and the value computed by the caller at the time of the call ($top(pop(stk))$). The definition assumes that a dataflow fact in $\mathcal{C}$ contains all information that is required by a procedure so that each transformer has to look at only the top of the stack passed to it – except for return nodes, where the transformer looks at the top two elements of the stack. We define a path transformer as follows: if $p = [n_1\ n_2 \cdots n_k]$ is a valid interprocedural path in the program then $[\![p]\!]^* = [\![[n_2 \cdots n_k]]\!]^* \circ [\![n_1]\!]^*$. This leads to the following definition.

Definition 3.3.1. *[52] If $s \in \mathcal{N}$ is the starting node of a program, then for $c_0 \in \mathcal{C}$ and $n \in \mathcal{N}$, the interprocedural-local-join-over-all-paths value is defined as follows:*

$$\text{LJOP}_{c_0}(n) = \sqcup\{[\![p]\!]^*(newstack(c_0)) \mid p \in VPaths(s, n)\}$$

where VPaths(s, n) is the set of all valid interprocedural paths from s to n and join of stacks is just the join of their topmost values: $stk_1 \sqcup stk_2 = top(stk_1) \sqcup top(stk_2)$.

We now construct an EWPDS $\mathcal{W}_e = (\mathcal{P}, \mathcal{S}, f, g)$ to compute this value when $\mathcal{C}$ has no infinite ascending chains, all semantic transformers $[\![n]\!]$ are distributive, and all merge relations $\mathcal{R}_n$ are distributive in each of their arguments. Define a semiring $\mathcal{S} = (D, \oplus, \otimes, \overline{0}, \overline{1})$ as $D = [\mathcal{C} \to \mathcal{C}] \cup \{\overline{0}\}$, which consists of the set of all distributive functions on $\mathcal{C}$ and a special function $\overline{0}$. The semiring operations are defined as follows. For $a, b \in D$,

$$a \oplus b = \begin{cases} a & \text{if } b = \overline{0} \\ b & \text{if } a = \overline{0} \\ (a \sqcup b) & \text{otherwise} \end{cases} \qquad a \otimes b = \begin{cases} 0 & \text{if } a = \overline{0} \text{ or } b = \overline{0} \\ (b \circ a) & \text{otherwise} \end{cases}$$

$$\overline{1} = \lambda c.c$$

The pushdown system $\mathcal{P}$ is $(\{p\}, \mathcal{N}, \Delta)$, where Δ is constructed by including a rule for each edge in $\mathcal{E}$. First, let $\mathcal{E}_{intra} \subseteq \mathcal{E}$ be the intraprocedural edges and $\mathcal{E}_{inter} \subseteq \mathcal{E}$ be the interprocedural (call and return) edges. Then include the following rules in Δ.

1. For $(n, m) \in \mathcal{E}_{intra}$, include the rule $r = \langle p, n\rangle \hookrightarrow \langle p, m\rangle$ with $f(r) = [\![n]\!]$.

2. For $n \in \mathcal{N}_C$ and $(n, m) \in \mathcal{E}_{inter}$, where $n_R \in \mathcal{N}_R$ is the return site for the call at n, include the rule $r = \langle p, n\rangle \hookrightarrow \langle p, m\ n_R\rangle$ with $f(r) = [\![n]\!]$ and
$$g_r(a, b) = \lambda c.\mathcal{R}_n(a(c), (a \otimes [\![n]\!] \otimes b \otimes [\![n_R]\!])(c)).$$

3. For $n \in \mathcal{N}$, if it is an exit node of a procedure, include the rule $r = \langle p, n\rangle \hookrightarrow \langle p, \varepsilon\rangle$ with $f(r) = [\![n]\!]$.

The merge functions defined above need not satisfy the path-extension property given in Defn. 3.1.2, but the techniques presented in Section 3.2.3 still allow us to solve GPS. This leads us to the following theorem.

Theorem 3.3.2. *Let $\mathcal{A}$ be a $\mathcal{P}$-automaton that accepts just the configuration $\langle p, s\rangle$, where s is the starting point of the program, and let $\mathcal{A}_{post^*}$ be the automaton obtained by using the saturation rule shown in Fig. 3.4 on $\mathcal{A}$.*

(1) If $\delta(\mathcal{L}(\mathcal{A}), c)$ *is read off* $\mathcal{A}_{post^*}$ *in accordance with Thm. 3.2.5 and* $c_0 \in \mathcal{C}$ *and* $n \in \mathcal{N}$*, we have:*

$$\mathrm{LJOP}_{c_0}(n) = [\oplus\{\delta(\mathcal{L}(\mathcal{A}), \langle q, n\ u\rangle) \mid u \in \Gamma^*\}](c_0).$$

(2) If $L \subseteq \Gamma^*$ *is a regular language of stack configurations then* $\mathrm{LJOP}_{c_0}(n, L)$*, which is the* LJOP *value restricted to only those paths that end in configurations described by* L*, can be calculated as follows:*

$$\mathrm{LJOP}_{c_0}(n, L) = [\oplus\{\delta(\mathcal{L}(\mathcal{A}), \langle q, n\ u\rangle) \mid u \in L\}](c_0).$$

Result (1) in Thm. 3.3.2 shows how an EWPDS can capture Knoop and Steffen's result. Result (2) is an extension of their theorem; it gives us a way of performing stack-qualified queries in the presence of local variables.

In case the semantic transformers $[\![.]\!]$ and $\mathcal{R}_n$ are not distributive but only monotonic, then, in the two equations of Thm. 3.3.2, the right-hand sides safely approximate $\mathrm{LJOP}_{c_0}(n)$ and $\mathrm{LJOP}_{c_0}(n, L)$, respectively.

3.4 EWPDS Experiments

In [3], Balakrishnan and Reps present an algorithm to analyze memory accesses in x86 code. Its goal is to determine an over-approximation of the set of values/memory-addresses that each register and memory location holds at each program point. The core dataflow-analysis algorithm used, called value-set analysis (VSA), is not relational, i.e., it does not keep track of the relationships that hold among registers and memory locations. However, when interpreting conditional branches, specifically those that implement loops, it is important to know such relationships. Hence, a separate affine-relation analysis (ARA) is performed to recover affine relations that hold among the registers at conditional branch points; those affine relations are then used to interpret conditional branches during VSA. ARA recovers affine relations involving registers only, because recovering affine relations involving memory locations would require points-to information, which is not available until the end of VSA. ARA is implemented by encoding the x86 program as an EWPDS using the weight domain

from [68]. It is based on machine arithmetic, i.e., arithmetic module 2^{32}, and is able to take care of overflow. (The encoding is similar to the one described in Section 3.5.2.)

Before each call instruction, a subset of the registers is saved on the stack, either by the caller or the callee, and restored at the return. Such registers are called the *caller-save* and *callee-save* registers. Because ARA only keeps track of information involving registers, when ARA is implemented using a WPDS, all affine relations involving caller-save and callee-save registers are lost at a call. We used an EWPDS to preserve them across calls by treating caller-save and callee-save registers as local variables at a call; i.e., the values of caller-save and callee-save registers after the call are set to the values before the call and the values of other registers are set to the values at the exit node of the callee.

The results are shown in Tab. 3.1. The column labeled 'Branches with useful information' refers to the number of branch points at which ARA recovered at least one affine relation. The last column shows the number of branch points at which ARA implemented via an EWPDS recovered more affine relations when compared to ARA implemented via a WPDS. Tab. 3.1 shows that the information recovered by EWPDS is better in 30% to 63% of the branch points that had useful information. The EWPDS version is somewhat slower, but uses less space; this is probably due to the fact that the dataflow transformer from [68] for 'spoiling' the affine relations that involve a given register uses twice the space of a transformer that preserves such relations.

3.5 Applications of EWPDSs

In this section, we show how different problems can be encoded using EWPDSs. We give encodings for Boolean programs, affine-relation analysis, and single-level pointer analysis. All of these encodings benefit from the use of merge functions.

3.5.1 Boolean Programs

Let B be a Boolean program. Let $\mathcal{W}_e = (\mathcal{P}, \mathcal{S}, f, g)$ be an EWPDS. We will use $\mathcal{W}_e$ to encode B. Without loss of generality, we assume that each procedure of B has the same

					Memory (MB)		Time (s)		Branches with useful information		
Prog	Insts	Procs	Branches	Calls	WPDS	EWPDS	WPDS	EWPDS	WPDS	EWPDS	Improvement
mplayer2	58452	608	4608	2481	27	6	8	9	137	192	57 (42%)
print	96096	955	8028	4013	61	19	20	23	601	889	313 (52%)
attrib	96375	956	8076	4000	40	8	12	13	306	380	93 (30%)
tracert	101149	1008	8501	4271	70	22	24	27	659	1021	387 (59%)
finger	101814	1032	8505	4324	70	23	24	30	627	999	397 (63%)
lpr	131721	1347	10641	5636	102	36	36	46	1076	1692	655 (61%)
rsh	132355	1369	10658	5743	104	36	37	45	1073	1661	616 (57%)
javac	135978	1397	10899	5854	118	43	44	58	1376	2001	666 (48%)
ftp	150264	1588	12099	6833	121	42	43	61	1364	2008	675 (49%)
winhlp32	179488	1911	15296	7845	156	58	62	98	2105	2990	918 (44%)
regsvr32	297648	3416	23035	13265	279	117	145	193	3418	5226	1879 (55%)
notepad	421044	4922	32608	20018	328	124	147	390	3882	5793	1988 (51%)
cmd	482919	5595	37989	24008	369	144	175	444	4656	6856	2337 (50%)

Table 3.1 Comparison of ARA results implemented using EWPDS versus WPDS.

number of local variables. Let G be the set of valuations of the global variables and Val be the set of valuations of local variables. The actions of program statements and conditions are now binary relations on $G \times \text{Val}$; thus, the weight domain $\mathcal{S}$ is a relational weight domain on the finite set $G \times \text{Val}$ (Defn. 2.4.13). The PDS $\mathcal{P}$ and weight assignments f are done in a manner similar to the encoding for dataflow models: $\mathcal{P}$ encodes the control flow, and the weight of a rule that is produced from an edge e is the binary relation of the statement on e. The weight on a call rule forgets the values of the local variables (under the assumption that in a Boolean program, local variables of a procedure are uninitialized at the start of the procedure):

$$\textit{weight on call rule} = \{(g, l_1, g, l_2) \mid g \in G, l_1, l_2 \in \text{Val}\}$$

This avoid passing the values of local variables from a caller to the callee. The weight on a return rule does the same to avoid passing values of local variables from the callee to the caller:

$$\textit{weight on return rule} = \{(g, l_1, g, l_2) \mid g \in G, l_1, l_2 \in \text{Val}\}$$

What remains to be defined are the merge functions.

Because different weights can refer to local variables from different procedures, one cannot take relational composition of weights from different procedures. The *project* function is used to change the scope of a weight. It existentially quantifies out the current transformation on local variables and replaces it with an identity relation. Formally, it can be defined as follows:

$$\textit{project}(w) = \{(g_1, l_1, g_2, l_1) \mid (g_1, l_1, g_2, l_2) \in w\}.$$

Once the summary of a procedure is calculated as a weight w involving local variables of the procedure, the *project* function is applied to it, and the result $\textit{project}(w)$ is passed to the callers of that procedure. This makes sure that local variables of one procedure do not interfere with those of another procedure. Thus, merge functions for Boolean programs all have the form

$$g(w_1, w_2) = w_1 \otimes \textit{project}(w_2).$$

For encoding Boolean programs with other abstractions, such as finding the shortest trace, one can use the relational weight domain on $(G \times \text{Val}, \mathcal{M})$ (Defn. 2.4.16), where $\mathcal{M}$ is the minpath semiring (Defn. 2.4.14). The weights on rules change as follows: if w is the weight of a rule obtained from the Boolean program (as defined earlier) then replace it with the following weight that attaches the value $1 \in \mathcal{M}$ to it:

$$\lambda(g_1, l_1, g_2, l_2).\ \text{if } (g_1, l_1, g_2, l_2) \in w \text{ then } 1 \text{ else } \infty$$

The *project* function on weights from this domain can be defined as follows:

$$\begin{aligned} \textit{project}(w) = \lambda(g_1, l_1, g_2, l_2). \quad & \text{if } (l_1 \neq l_2) \text{ then } \overline{0}_{\mathcal{S}} \\ & \text{else } \textstyle\bigoplus_{\mathcal{M}} \{w(g_1, l_1, g_2, l) \mid l \in L\} \end{aligned}$$

Again, the merge functions all have the form $g(w_1, w_2) = w_1 \otimes \textit{project}(w_2)$.

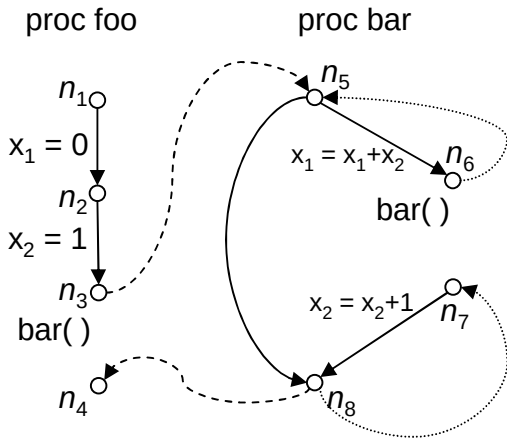


Figure 3.5 An affine program that starts execution at node $\mathtt{n}_1$. There are two global variables: $\mathtt{x}_1$ and $\mathtt{x}_2$.

3.5.2 Affine Relation Analysis

An affine relation is a linear-equality constraint between integer-valued variables. Affine-relation analysis (ARA) tries to find all affine relationships that hold in the program. An example is shown in Fig. 3.5. For example, for this program, ARA would infer that $\mathtt{x}_2 = \mathtt{x}_1+1$ at program node $\mathtt{n}_4$.

ARA for single-procedure programs was first addresses by Karr [45]. ARA generalizes other analyses, including copy-constant propagation, linear-constant propagation [84], and induction-variable analysis [45]. We have used ARA on machine code to find induction-variable relationships between machine registers (see Section 3.4).

Affine Programs

Interprocedural ARA can be performed precisely on *affine programs*, and has been the focus of several papers [67, 68, 36]. Affine programs are similar to Boolean programs, but with integer-valued variables. First, we restrict our attention to affine programs with only global variables and show how they can be encoded using WPDSs, and then show how the addition of locals variables is handled using merge functions.

If $\{\mathtt{x}_1, \mathtt{x}_2, \cdots, \mathtt{x}_n\}$ is the set of global variables of the program, then all assignments in an affine program have the form $\mathtt{x}_j := a_0 + \sum_{i=1}^{n} a_i \mathtt{x}_i$, where $a_0, \cdots, a_n$ are integer constants. An assignment can also be non-deterministic, denoted by $\mathtt{x}_j := ?$, which may assign any integer to $\mathtt{x}_j$. (This is typically used for abstracting assignments that cannot be modeled as an affine transformation of the variables.) All branch conditions in affine programs are non-deterministic.

ARA Weight Domain

We briefly describe the weight domain based on the linear-algebra formulation of ARA from [67]. An affine relation $a_0 + \sum_{i=1}^{n} a_i \mathtt{x}_i = 0$ is represented using a column vector of size $n+1$: $\vec{a} = (a_0, a_1, \cdots, a_n)^t$. A valuation of program variables $\overline{\mathtt{x}}$ is a map from the set of global variables to the integers. The value of $\mathtt{x}_i$ under this valuation is written as $\overline{\mathtt{x}}(i)$.

A valuation $\overline{\mathtt{x}}$ satisfies an affine relation $\vec{a} = (a_0, a_1, \cdots, a_n)^t$ if $a_0 + \sum_{i=1}^{n} a_i \overline{\mathtt{x}}(i) = 0$. An affine relation $\vec{a}$ represents the set of all valuations that satisfy it, written as $\textsc{Pts}(\vec{a})$. An affine relation $\vec{a}$ holds at a program node if the set of valuations reaching that node (in the concrete collecting semantics) is a subset of $\textsc{Pts}(\vec{a})$.

An important observation about affine programs is that if affine relations $\vec{a}_1$ and $\vec{a}_2$ hold at a program node, then so does any linear combination of $\vec{a}_1$ and $\vec{a}_2$. For example, one can verify that $\textsc{Pts}(\vec{a}_1 + \vec{a}_2) \supseteq \textsc{Pts}(\vec{a}_1) \cap \textsc{Pts}(\vec{a}_2)$, i.e., the affine relation $\vec{a}_1 + \vec{a}_2$ (componentwise addition) holds at a program node if both $\vec{a}_1$ and $\vec{a}_2$ hold at that node. The set of affine relations that hold at a program node forms a (finite-dimensional) vector space [67]. This implies that a (possibly infinite) set of affine relations can be represented by any of its bases; each such basis is always a finite set.

For reasoning about affine programs, Müller-Olm and Seidl defined an abstraction that is able to find all affine relationships in an affine program: each statement is abstracted by a set of matrices of size $(n+1) \times (n+1)$. A statement $\mathtt{x}_j := a_0 + \sum_{i=1}^{n} a_i \mathtt{x}_i$ can be written as $\overline{\mathtt{x}} := A\overline{\mathtt{x}} + b$, where $\overline{\mathtt{x}}$ is interpreted as a column vector of size n, A is an $(n \times n)$ matrix, and b is a column vector of size n:

$$A = \begin{array}{|c|c|}\hline I_{j-1} & 0 \\ \hline a_1 \;\; a_2 \;\; \cdots \;\; a_n & \\ \hline 0 & I_{n-j} \\ \hline \end{array} \qquad b = \left(\begin{array}{c} 0 \\ \hline a_0 \\ \hline 0 \end{array} \right)$$

where I_k is the identity matrix of size $(k \times k)$, and a_0 appears in the j^{th} row of b. Then this statement is abstracted by a singleton set consisting of the following matrix of size $(n+1) \times (n+1)$:

$$\begin{array}{|c|c|}\hline 1 & b^t \\ \hline 0 & A^t \\ \hline \end{array}$$

We refer the reader to [67] for the abstraction of other kinds of statements. The set of matrices obtained in this way form the weakest-precondition transformer on affine relations for a statement: if a statement is abstracted as the set $\{m_1, m_2, \cdots, m_r\}$, then the affine relation $\vec{a}$ holds after the execution of the statement if and only if the affine relations $(m_1\vec{a}), (m_2\vec{a}), \cdots, (m_r\vec{a})$ hold before the execution of the statement.

Under such an abstraction of program statements, one can define the extend operation as matrix multiplication of each member of the first set with each member of the second set, and the combine operation as set union. This is correct semantically, but it does not give an effective algorithm because the matrix sets can grow in size without bound. However, the observation that affine relations form a vector space carries over to a set of matrices as well. One can show that the transformer $\{m_1, m_2, \cdots, m_r\}$ is semantically equivalent to the transformer $\{m_1, m_2, \cdots, m_r, m\}$, where m is any linear combination of the m_i matrices. Thus, a set of matrices can be abstracted as the (infinite) set of matrices spanned by them. Once we have a vector space, we can represent it using any of its bases to get a finite and bounded representation: a vector space over matrices of size $(n+1) \times (n+1)$ cannot have more that $(n+1)^2$ matrices in any basis.

If M is a set of matrices, let $\textsc{Span}(M)$ be the vector space spanned by them. Let β be the basis operation that takes a set of matrices and returns a basis of their span. We

can now define the weight domain. A weight w is a vector space of matrices, which is represented using any of its bases. Extend of vector spaces w_1 and w_2 is the vector space $\{(m_1 m_2) \mid m_i \in w_i\}$. Combine of w_1 and w_2 is the vector space $\{(m_1 + m_2) \mid m_i \in w_i\}$, which is the smallest vector space containing both w_1 and w_2. $\overline{0}$ is the empty set, and $\overline{1}$ is the span of the singleton set consisting of the identity matrix. The extend and combine operations, as defined above, are operations on infinite sets. They can be implemented by the corresponding operations on any basis of the weights. The following properties show that it is semantically correct to operate on the elements in the basis instead of all the elements in the vector space spanned by them:

$$\begin{aligned} \beta(w_1 \oplus w_2) &= \beta(\beta(w_1) \oplus \beta(w_2)) \\ \beta(w_1 \otimes w_2) &= \beta(\beta(w_1) \otimes \beta(w_2)) \end{aligned}$$

These properties are satisfied because of the linearity of extend (matrix multiplication distributes over addition) and combine operations.

Under such a weight domain, $\mathrm{IJOP}(S, T)$ is a weight that is the net weakest-precondition transformer between S and T. Suppose that this weight has the basis $\{m_1, \cdots, m_r\}$. The affine relation that indicates that any variable valuation might hold at S is $\vec{0} = (0, 0, \cdots, 0)$. Thus, $\vec{0}$ holds at S, and the affine relation $\vec{a}$ holds at T iff $m_1\vec{a} = m_2\vec{a} = \cdots = m_r\vec{a} = \vec{0}$. In other words, $\vec{a}$ is in the nullspaces of each of the m_i. The set of all affine relations that hold at T can be found as the intersection of the null spaces of the matrices $m_1, m_2, \cdots, m_r$.

Incorporating Local Variables

If an affine program has n global variables and no local variables, then the matrices have size $(n + 1) \times (n + 1)$. Assume, without loss of generality, that each procedure has l local variables. The statements are abstracted as a set of matrices exactly in the same manner as before, except that the matrices have size $(n + l + 1) \times (n + l + 1)$. These matrices can be divided into four quadrants, as shown below.

I	II
$(n+1) \times (n+1)$	$(n+1) \times l$
III	IV
$l \times (n+1)$	$l \times l$

The four quadrants of a matrix describe four pieces of the transformation from pre-state to post-state: the first quadrant encodes the contribution of pre-state values of global variables to post-state values of global variables; the second quadrant encodes the contribution of pre-state globals to post-state locals; the third quadrant encodes the contribution of pre-state locals to post-state globals; and the fourth quadrant encodes the contribution of pre-state locals to post-state locals.

As for Boolean programs, we will define a *merge* function using a *project* function that quantifies out the local variables and replaces them with the identity transformation. This is carried out by zero-ing out the second and third quadrants, and changing the fourth quadrant to the identity matrix. If w is a set of matrices, $project(w)$ is defined as the application of the following operation on all matrices of w:

$$\begin{array}{|c|c|} \hline m_1 & m_2 \\ \hline m_3 & m_4 \\ \hline \end{array} \quad \mapsto \quad \begin{array}{|c|c|} \hline m_1 & 0 \\ \hline 0 & a\ I_l \\ \hline \end{array}$$

Here a is the topmost-leftmost element of m_1. It is used to make the above operation linear, which, in turn, makes $merge(w_1, w_2) = w_1 \otimes project(w_2)$ distribute over combine. (A justification of this operation for quantifying out the local variables can be found in [67].)

Extensions to ARA

ARA can also be performed for modular arithmetic [68] to precisely model machine arithmetic (which is modulo 2 to the power of the word size). The weight domain is similar to the one described above.

3.5.3 Single-Level Pointer Analysis

In this section, we define an EWPDS to find variable aliasing in programs written in a C-like imperative language that is restricted to single-level pointers (i.e., one cannot have pointers to pointers).[2] This problem was defined and solved by Landi and Ryder [61]. We first discuss some of the results from [61], and then move on to describe an EWPDS that finds aliasing in a program. This encoding shows the power of EWPDSs for solving different kinds of problems. Moreover, it gives us something new: a way of answering stack-qualified aliasing queries.

We will only to describe the weight domain and merge functions for the EWPDS, because we already know how to model the control flow of a program as a PDS (Fig. 2.4).

We say that two access expressions a and b are aliased (written as $\langle a, b\rangle$) at a particular program point n if in *some* program execution they refer to the same memory location when execution reaches n. We limit access expressions to variables and pointer dereferences (written as $*p$ for an address-valued variable p). Given a program, we want to determine an overapproximation of all alias pairs that hold at each program point. This problem is also referred to as *may-aliasing*. In [61], this is computed in two stages. First, *conditional may-aliasing* information is computed, which answers questions of the form: "if all alias pairs in the set $\mathcal{A}$ hold at a program point n_1, does the pair $\langle a, b\rangle$ hold at point n_2?" The second stage then uses this information to build up the final may-aliasing table.

An important property that results from the fact that we only have single-level pointers is that for all program points n_1 and n_2, where n_1 is the enter node of the procedure containing n_2, if the alias pair $\langle a, b\rangle$ holds at n_2 under the assumption that the set $\mathcal{A} = \{A_1, \cdots, A_m\}$ of alias pairs holds at n_1, then either (i) we can prove that $\langle a, b\rangle$ holds at n_2, assuming that no alias pair holds at n_1; or (ii) there exists a k, $1 \leq k \leq m$, such that assuming that just A_k holds at n_1 suffices to prove that $\langle a, b\rangle$ holds at n_2. In other words, we only need to compute

[2]For languages in which more than one level of indirection is possible, the algorithm for single-level pointers still provides a safe solution (i.e., an overapproximation) [61].

conditional may-alias information for each *alias pair* $A_k \in \mathcal{A}$, rather than for each *subset* of $\mathcal{A}$.

We say that the alias pair $\langle a, . \rangle$ holds at program point n if a is aliased to some access expression that is not visible (i.e., out of scope) in the procedure containing n. It is not necessary to know the particular invisible access expression to which a is aliased because a procedure will always have the same effect on all alias pairs that contain access expression a and any invisible access expression [61].

For a given program, let V denote the set of all its variables and pointer dereferences. Assume that all variables have different names (local variables can be prefixed by the name of the procedure that contains them) so that there are no name conflicts. The set $\mathcal{AP} = (V \times V) \cup (V \times \{.\}) \cup (\{.\} \times V)$ is the set of all possible alias pairs. Let $\mathcal{AP}_\perp = \mathcal{AP} \cup \{\perp\}$, where $\perp$ represents the absence of an alias pair.

We now construct a weight domain over the set $D = (\mathcal{AP}_\perp \rightarrow 2^{\mathcal{AP}})$ of all functions w from $\mathcal{AP}_\perp$ to the power set of $\mathcal{AP}$ with the following monotonicity restriction: for all $x \in \mathcal{AP}$, $w(\perp) \subseteq w(x)$. Operations on weights will maintain the invariant that alias relations are symmetric (i.e., if $\langle a, b \rangle$ holds, so does $\langle b, a \rangle$). Each weight $w \in D$ can be efficiently represented as a one-to-many map from $\mathcal{AP}_\perp$ to $\mathcal{AP}$.

An interprocedural path P with weight w means that if we assume $\langle a, b \rangle$ to hold at the beginning of P then all pairs in $w(\langle a, b \rangle)$ hold at the end of path P when the program execution follows P. The special element $\perp$ handles the case when no pair is assumed to hold at the beginning of the path; $w(\perp)$ is the set of all alias pairs that hold at the end of the path without assuming that any pair holds at the beginning of the path. Thus, a weight represents conditional may-aliasing information, which motivates the monotonicity condition introduced above.

For all $w_1 \neq \overline{0} \neq w_2$, the semiring operations are defined as follows. For $x \in \mathcal{AP}_\perp$,

$$\begin{aligned}
(w_1 \oplus w_2)(x) &= w_1(x) \cup w_2(x) \\
(w_1 \otimes w_2)(x) &= w_2(\perp) \cup (\cup_{y \in w_1(x)} w_2(y)) \\
\overline{1}(x) &= \begin{cases} \emptyset & \text{if } x = \perp \\ \{x\} & \text{otherwise} \end{cases}
\end{aligned}$$

If path P_1 has weight w_1 and path P_2 has weight w_2, then the weight $w_1 \otimes w_2$ summarizes the conditional alias information of the path P_1 followed by P_2. In particular, $(w_1 \otimes w_2)(x)$ consists of the alias pairs that hold from w_2, regardless of the value of w_1, together with the alias pairs that hold from w_2 given $w_1(x)$. When P_1 and P_2 have the same starting and ending points, the weight $w_1 \oplus w_2$ stores conditional aliasing information when the program execution follows P_1 *or* P_2.

(The semiring constant $\overline{0}$ cannot be naturally described in terms of conditional aliasing, but we can add it to D as a special value that satisfies all properties of Defn. 2.4.1.)

We now consider how to associate a weight to each pushdown rule in the EWPDS that encodes the program. For a node n that contains a statement of the form $x = y$, where x and y are pointers, the weight associated with each rule of the form $\langle p, n\rangle \hookrightarrow \cdots$ is a map, where for each $x \in \mathcal{AP}_\perp$, the first applicable mapping that appears in the list below is used:

$$\begin{aligned}
\langle *y, b\rangle &\mapsto \{\langle *x, b\rangle\} \\
\langle a, *y\rangle &\mapsto \{\langle a, *x\rangle\} \\
\langle *x, b\rangle &\mapsto \emptyset \\
\langle a, *x\rangle &\mapsto \emptyset \\
\langle a, b\rangle &\mapsto \{\langle a, b\rangle\} \\
\perp &\mapsto \{\langle a, a\rangle \mid a \in V\} \cup \{\langle *x, *y\rangle, \langle *y, *x\rangle\}
\end{aligned}$$

Roughly speaking, this generates the alias pairs $\langle *x, *y\rangle$ and $\langle *y, *x\rangle$, makes the aliases of $*y$ into aliases of $*x$, and removes the previously existing alias pairs of $*x$ (except for $\langle *x, *x\rangle$). To enforce monotonicity on weights, the following closure operation is applied to the map: $cl(w) = \lambda x.(w(x) \cup w(\perp))$. The weights on other rules that represent intraprocedural edges can be defined similarly (see [61]).

For a *push* rule, the weight is determined according to the binding that occurs at the call site; the definition is presented in Fig. 3.6. All pop rules have the weight $\overline{1}$.

The merge functions associated with push rules reflect the way conditional aliasing information is computed for return nodes in [61]. Consider the push rule $\langle p, call_{foo}\rangle \hookrightarrow \langle p, enter_{bar}\ return_{foo}\rangle$, which is a call to procedure *bar* from *foo*, and suppose that $bind_{call}$ is the weight associated with this rule. For local access expressions l_1, l_2 of *foo* and global access expressions g_1, g_2, the following must hold.

- The alias pair $\langle l_1, l_2\rangle$ holds at $return_{foo}$ only if the pair $\langle l_1, l_2\rangle$ holds at the call node $call_{foo}$.
- The alias pair $\langle g_1, g_2\rangle$ holds at $return_{foo}$ only if the pair holds at $exit_{bar}$.
- The alias pair $\langle g_1, l_1\rangle$ holds at $return_{foo}$ only if $\langle g_1, .\rangle$ holds at $exit_{bar}$ and the invisible variable is l_1. This happens when a pair $\langle o_1, l_1\rangle$ that held at $call_{foo}$ caused $\langle o_2, .\rangle$ to hold at $enter_{bar}$ because of the call bindings ($\langle o_2, .\rangle \in bind_{call}(\langle o_1, l_1\rangle)$) and this pair, in turn, caused $\langle g_1, .\rangle$ to hold at $exit_{bar}$.

To encode these facts as weights for an algorithmic description of the merge functions, we need to define certain weights and operations on them.

- **Projection.** For a set $S \subseteq (V \cup \{.\})$, let w_S be a weight that only preserves alias pairs in $S \times S$: $w_S(\bot) = \emptyset$ and

$$w_S(\langle a, b\rangle) = \begin{cases} \{\langle a, b\rangle\} & \text{if } a, b \in S \\ \emptyset & \text{otherwise} \end{cases}$$

- **Restoration.** For an access expression $v \in V$, let w_S^v be a weight that changes alias pairs when v comes back in scope conditional on the set $S \subseteq (V \cup \{.\})$: $w_S^v(\bot) = \emptyset$ and

$$bind_n(\bot) = \left(\begin{array}{ll} & \{\langle *f_i, *f_j\rangle \mid [f_i, a_i], [f_j, a_j], a_i = a_j\} \\ \cup & \{\langle *f_i, *a_i\rangle \mid [f_i, a_i], visible_p(a_i)\} \\ \cup & \{\langle *a_i, *f_i\rangle \mid [f_i, a_i], visible_p(a_i)\} \\ \cup & \{\langle *f_i, .\rangle \mid [f_i, a_i], \neg visible_p(a_i)\} \\ \cup & \{\langle ., *f_i\rangle \mid [f_i, a_i], \neg visible_p(a_i)\} \end{array} \right)$$

$$bind_n(\langle a, b\rangle) = \left(\begin{array}{ll} & bind_n(\bot) \\ \cup & \{\langle a, b\rangle \mid visible_p(a), visible_p(b)\} \\ \cup & \{\langle a, .\rangle \mid visible_p(a), \neg visible_p(b)\} \\ \cup & \{\langle ., b\rangle \mid \neg visible_p(a), visible_p(b)\} \\ \cup & \{\langle a, *f_i\rangle \mid visible_p(a), [f_i, a_i], *a_i = b\} \\ \cup & \{\langle ., *f_i\rangle \mid \neg visible_p(a), [f_i, a_i], *a_i = b\} \\ \cup & \{\langle *f_i, b\rangle \mid visible_p(b), [f_i, a_i], *a_i = a\} \\ \cup & \{\langle *f_i, .\rangle \mid \neg visible_p(b), [f_i, a_i], *a_i = a\} \\ \cup & \{\langle *f_i, *f_j\rangle \mid [f_i, a_i], [f_j, a_j], *a_i = a, *a_j = b\} \end{array} \right)$$

Figure 3.6 A function that models parameter binding for a call at program point n to a procedure named p. For brevity, we write $[f, a]$ to denote the fact that f is a pointer-valued formal parameter bound to actual a. Also, $visible_p(a)$ is *true* if a is visible in procedure p.

$$w_S^v(\langle a, b\rangle) = \begin{cases} \{\langle a, v\rangle\} & \text{if } b = . \text{ and } a \in S \\ \{\langle v, b\rangle\} & \text{if } a = . \text{ and } b \in S \\ \emptyset & \text{otherwise} \end{cases}$$

- **Conditional Extend.** For an alias pair $\langle a, b\rangle$, define $\otimes_{\langle a,b\rangle}$ to be a binary operation on weights that calculates the alias pairs that hold at the end of a path as a result of the fact that $\langle a, b\rangle$ held at a point inside the path. For $x \in \mathcal{AP}_\bot$,

$$(w_1 \otimes_{\langle a,b\rangle} w_2)(x) = \begin{cases} w_2(\langle a, b\rangle) & \text{if } \langle a, b\rangle \in w_1(x) \\ w_2(\bot) & \text{otherwise} \end{cases}$$

We can now define the merge functions. If G is the set of global access expressions of the program, then for a call from a procedure with local access expressions L and binding weight $bind_{call}$ (i.e., the weight on the push rule), the merge function is defined as follows (where L_e denotes $L \cup \{.\}$):

$$g(w_1, w_2) = \textbf{if}(w_1 = \overline{0} \ or \ w_2 = \overline{0}) \ \textbf{then} \ \overline{0}$$
$$\textbf{else} \left(\begin{array}{ll} & (w_1 \otimes w_{L_e}) \\ \oplus & (w_1 \otimes bind_{call} \otimes w_2 \otimes w_G) \\ \oplus & \bigoplus_{\langle a,l \rangle \in V \times L_e} ((w_1 \otimes_{\langle a,l \rangle} (bind_{call} \otimes w_2)) \otimes w_G^l) \\ \oplus & \bigoplus_{\langle l,a \rangle \in L_e \times V} ((w_1 \otimes_{\langle l,a \rangle} (bind_{call} \otimes w_2)) \otimes w_G^l) \end{array} \right)$$

The first term in the combine copies over from the call site the pairs for local access expressions. The second term copies over from the called procedure's exit site the pairs for global access expressions. The third and fourth terms, which are combines over all pairs in $V \times L_e$ and $L_e \times V$, respectively, account for global-local access expressions, following the strategy discussed earlier in this section.

After the EWPDS is constructed, we can run a single GPS query on the configuration set $C = \{\langle p, enter_{main} \rangle\}$ (where p is the single control location of the EWPDS), and obtain the may-alias pairs as follows,

$$may\text{-}alias(n) = \text{IJOP}(C, \{\langle p, n \ u \rangle \mid u \in \Gamma^*\})(\bot).$$

In addition to computing the Landi-Ryder may-alias pairs, we can also answer stack-qualified queries about may-alias relationships. For instance, we can find out the may-alias pairs that hold at n_1 when execution ends in the stack configuration $\langle p, n_1 n_2 \cdots n_k \rangle$. Such queries allow us to obtain more precise information than what is obtained by merely computing a may-aliasing query for paths that end at n_1 with *any* stack configuration.

3.6 Related Work

Weighted pushdown systems have been used for finding uninitialized variables, live variables, linear constant propagation, and the detection of affine relationships. In each of these

cases, local variables are handled by introducing special paths in the transition system of the PDS that models the program. These paths skip call sites to avoid passing local variables to the callee. This leads to imprecision by breaking existing relationships between local and global variables. Besides dataflow analysis, WPDSs have also been used for generalized authorization problems [87].

A library for WPDSs is available as part of MOPED [50]. We have developed our own implementations of WPDSs: WPDS++ [49] and WALi [47], both of which now support EWPDSs as well.

MOPED [32, 50] has been used for performing relational dataflow analysis, but only for finite abstract domains. Its basic approach is to embed the abstract transformer of each program statement into the rules of the pushdown system that models the program. This contrasts with WPDSs, where the abstract transformer is a separate weight associated with a pushdown rule. MOPED associates global variables with states of the PDS and local variables with its stack symbols. Then the stack of the PDS simulates the run-time stack of the program and maintains a different copy of the local variables for each procedure invocation. A simple pushdown reachability query can be used to compute the required dataflow facts. The disadvantage of that approach is that it cannot handle infinite-size abstract domains because then associating an abstract transformer with a pushdown rule would create infinite pushdown rules. In contrast, an EWPDS is capable of performing an analysis on infinite-size abstract domains. The domain used for copy-constant propagation in Section 3.1 is one such example.

Besides dataflow analysis, model-checking of pushdown systems has also been used for verifying security properties in programs [31, 42, 21]. Like WPDSs, we can use EWPDS for this purpose, but with added precision that comes due to the presence of merge functions.

The idea behind the transition from a WPDS to an EWPDS is that we attach extra meaning to each run of the pushdown system. We look at a run as a *tree* of matching calls and returns that push and pop values on the run-time stack of the program. This treatment of a program run has also been explored by Müller-Olm and Seidl [67] in an interprocedural

dataflow-analysis algorithm to identify the set of all affine relationships that hold among program variables at each program node. They explicitly match calls and returns to avoid passing relations involving local variables to different procedures. This allowed us to to directly translate their work into an EWPDS, which we have used for the experiments in Section 3.4.

3.7 Proofs

In this section, we give proofs for Thms. 3.2.1 and 3.2.5. In each case, we give an abstract grammar problem G, similar to the ones shown in Section 2.4.1 for WPDSs, and then show the following: (i) computing JOD values for G is sufficient for computing IJOP on EWPDSs; (ii) the saturation-based algorithms compute JOD values for G.

Fix $\mathcal{W}_e = (\mathcal{P}, \mathcal{S}, f, m)$ to be an EWPDS, where $\mathcal{P} = (P, \Gamma, \Delta)$ is the underlying PDS. Fix $\mathcal{A} = (Q, \Gamma, \rightarrow_0, P, F)$ to be a $\mathcal{P}$-automaton. Let m_r be the merge function associated with rule r.

Proof of Thm. 3.2.1

The *PopRuleSeq* grammar shown in Fig. 2.6 characterizes the set of all paths in a PDS. For WPDSs, we saw how replacing the rules in the *PopRuleSeq* grammar with weights led to an abstract grammar problem (shown in Fig. 2.13) that solves GPP for WPDSs. We follow a similar strategy for EWPDSs, but we need to consider how a rule sequence is parsed by the *PopRuleSeq* grammar, identify balanced rule sequences, and insert merge functions accordingly.

We say that a non-terminal N_1 over-approximates a non-terminal N_2 (possibly from a different context-free grammar) when $\mathcal{L}(N_1) \supseteq \mathcal{L}(N_2)$. A grammar G_1 over-approximates a grammar G_2 if for every non-terminal of G_2, there is a non-terminal of G_1 that over-approximates it.

Consider the grammar shown in Fig. 3.7. We call it the *accepting rule-sequence* grammar for GPP. The productions from case 1 to 4 are from the *PopRuleSeq* grammar for the

	Production		for each
(1)	$PopRuleSeq_{(q,\gamma,q')}$	$\rightarrow \varepsilon$	$(q,\gamma,q') \in \rightarrow_0$
(2)	$PopRuleSeq_{(p,\gamma,p')}$	$\rightarrow r$	$r = \langle p,\gamma\rangle \hookrightarrow \langle p',\varepsilon\rangle \in \Delta$
(3)	$PopRuleSeq_{(p,\gamma,q)}$	$\rightarrow r \;\; PopRuleSeq_{(p',\gamma',q)}$	$r = \langle p,\gamma\rangle \hookrightarrow \langle p',\gamma'\rangle \in \Delta, q \in P$
(4)	$PopRuleSeq_{(p,\gamma,q)}$	$\rightarrow \;\; r \;\; PopRuleSeq_{(p',\gamma',q')}$	$PopRuleSeq_{(q',\gamma'',q)}$
			$r = \langle p,\gamma\rangle \hookrightarrow \langle p',\gamma'\gamma''\rangle \in \Delta, q, q' \in P$
(5)	$AcceptingRuleSeq[p\gamma_1\gamma_2\cdots\gamma_n]$	$\rightarrow \;\; PopRuleSeq_{(p,\gamma_1,q_1)} PopRuleSeq_{(q_1,\gamma_2,q_2)} \cdots PopRuleSeq_{(q_{n-1},\gamma_n,q_n)}$	
			$p \in P, \gamma_i \in \Gamma, q_i \in Q$ for $1 \leq i \leq n, q_n \in F$

Figure 3.7 The *AcceptingRuleSeq* grammar for GPP, given PDS $\mathcal{P}$ and automaton $\mathcal{A}$.

PDS $\mathcal{PA}$ (Defn. 2.3.7), but the set of terminals is restricted to be only Δ, i.e., we replace the rules that are produced from $\mathcal{A}$ with ε (in case 1 of the grammar). The productions from case 5 add a new set of non-terminals. From Cor. 2.3.6 and Lem. 2.3.8, one can show that $\mathcal{L}(AcceptingRuleSeq[p\gamma_1\gamma_2\cdots\gamma_n])$ equals the set of all rule sequences that take the configuration $\langle p,\gamma_1\cdots\gamma_n\rangle$ to a configuration in $\mathcal{A}$, i.e., it equals $\{paths_{\mathcal{P}}(\langle p,\gamma_1\cdots\gamma_n\rangle, c) \mid c \in \mathcal{L}(\mathcal{A})\}$.

The grammar shown in Fig. 3.8 (call it G_{over}) over-approximates the accepting rule-sequence grammar when we remove the *AcceptingRuleSeq* non-terminals. This is easy to prove. The grammar G_{over} is obtained as follows: for each production in Fig. 3.7, except for the ones in case 5, replace the non-terminal $PopRuleSeq_t$ with A, if $t \in (P \times \Gamma \times P)$, or with B if $t \in P \times \Gamma \times (Q - P)$, or with C if $t \in (Q - P) \times \Gamma \times (Q - P)$; a PDS rule r is replaced with R_i if it has i stack symbols in its right-hand side.

One can show that $\mathcal{L}(A) \subseteq (\mathcal{L}(\sigma_b)\ R_0)$ and $\mathcal{L}(B) \subseteq (R_2 \cup \mathcal{L}(\sigma_b))^*$, where σ_b is the non-terminal from Fig. 3.2 that derives balanced sequences. Thus, $\mathcal{L}(PopRuleSeq_t) \subseteq (\mathcal{L}(\sigma_b)\ R_0)$ for $t \in P \times \Gamma \times P$, and $\mathcal{L}(PopRuleSeq_t) \subseteq (R_2 \cup \mathcal{L}(\sigma_b))^*$ for $t \in P \times \Gamma \times (Q - P)$. One can also show that $\mathcal{L}(AcceptingRuleSeq[c]) \subseteq (\mathcal{L}(A)^*\mathcal{L}(B)^+)$. Moreover, $(R_2 A)$ can only derive balanced sequences. The two instances of $R_2 A$ in the grammar G_{over} are where merge functions have to be slipped in—and both such instances come from case 4 productions of the accepting rule-sequence grammar.

Non-terminal	Over-approximates $PopRuleSeq_t$ for t in
A	$P \times \Gamma \times P$
B	$P \times \Gamma \times (Q - P)$
C	$(Q - P) \times \Gamma \times (Q - P)$

$$
\begin{array}{lcl}
A & \rightarrow & R_0 \mid R_1\ A \mid R_2\ A\ A \\
B & \rightarrow & \varepsilon \mid R_1\ B \mid R_2\ B\ C \mid R_2\ A\ B \\
C & \rightarrow & \varepsilon
\end{array}
$$

Figure 3.8 A grammar that over-approximates the grammar shown in Fig. 3.7.

Based on the above observations about the rule sequences that can be derived from the non-terminals of the accepting rule-sequence grammar, we construct the abstract grammar shown in Fig. 3.9. It is similar to the abstract grammar for solving GPP on WPDSs (Fig. 2.13). Case 4 in Fig. 3.9 corresponds to the productions $A \rightarrow R_2\ A\ A$ and $B \rightarrow R_2\ A\ B$ of G_{over}, thus, uses a merge function. Case 5 corresponds to the production $B \rightarrow R_2\ B\ C$, and does not use a merge function because the call rule R_2 cannot have any matching return rule in the rule sequence derived from $(B\ C)$.

By our construction, $\text{JOD}(PopSeq_{(p,\gamma,q)}) = \text{IJOP}_{\mathcal{W}_e}(paths(\langle p, \gamma\rangle, \langle q, \varepsilon\rangle))$. Moreover, $\text{JOD}(AcceptingSeq[p\gamma_1\gamma_2 \cdots \gamma_n]) = \delta_{\mathcal{W}_e}(paths(\langle p, \gamma_1 \cdots \gamma_n\rangle, \mathcal{L}(\mathcal{A})))$

The saturation procedure shown in Fig. 3.3 solves the abstract grammar of Fig. 3.9 for *PopSeq* non-terminals. Thus, when the saturation procedure finishes, the weight on transition t is w_t if and only if $w_t = \text{JOD}(PopSeq_t)$. (See also Lem. 2.4.9.) Moreover, if $\mathcal{A}_{pre^*}$ is the resulting automaton, then $\mathcal{A}_{pre^*}(\langle p, \gamma_1 \cdots \gamma_n\rangle) = \text{JOD}(AcceptingSeq[p\gamma_1\gamma_2 \cdots \gamma_n])$ because the productions for *AcceptingSeq* are only computing the weight of accepting paths in $\mathcal{A}_{pre^*}$. This proves that $\mathcal{A}_{pre^*}(c) = \delta_{\mathcal{W}_e}(c, \mathcal{L}(\mathcal{A}))$.

Proof of Thm. 3.2.5

The proof of Thm. 3.2.5 uses an argument similar to the one used in the proof of Thm. 3.2.1. To simplify the proof, we assume that the weight on a call rule is always

	Production	for each
(1)	$PopSeq_{(q,\gamma,q')} \rightarrow g_1(\varepsilon)$	$(q,\gamma,q') \in \rightarrow_0$
	$g_1 = \overline{1}$	
(2)	$PopSeq_{(p,\gamma,p')} \rightarrow g_2(\varepsilon)$	$r = \langle p,\gamma\rangle \hookrightarrow \langle p',\varepsilon\rangle \in \Delta$
	$g_2 = f(r)$	
(3)	$PopSeq_{(p,\gamma,q)} \rightarrow g_3(PopSeq_{(p',\gamma',q)})$	$r = \langle p,\gamma\rangle \hookrightarrow \langle p',\gamma'\rangle \in \Delta, q \in Q$
	$g_3 = \lambda x.f(r) \otimes x$	
(4)	$PopSeq_{(p,\gamma,q)} \rightarrow g_4(PopSeq_{(p',\gamma',q')}, PopSeq_{(q',\gamma'',q)})$	
		$r = \langle p,\gamma\rangle \hookrightarrow \langle p',\gamma'\gamma''\rangle \in \Delta, q \in Q, q' \in P$
	$g_4 = \lambda x.\lambda y.m_r(\overline{1},x) \otimes y$	
(5)	$PopSeq_{(p,\gamma,q)} \rightarrow g_5(PopSeq_{(p',\gamma',q')}, PopSeq_{(q',\gamma'',q)})$	
		$r = \langle p,\gamma\rangle \hookrightarrow \langle p',\gamma'\gamma''\rangle \in \Delta, q, q' \in (Q-P)$
	$g_5 = \lambda x.\lambda y.f(r) \otimes x \otimes y$	
(6)	$AcceptingSeq[p\gamma_1\gamma_2\cdots\gamma_n] \rightarrow g_6(PopSeq_{(p,\gamma_1,q_1)}, PopSeq_{(q_1,\gamma_2,q_2)}, \cdots, PopSeq_{(q_{n-1},\gamma_n,q_n)})$	
		$p \in P, \gamma_i \in \Gamma, q_i \in Q$ for $1 \leq i \leq n, q_n \in F$
	$g_6 = \lambda x_1 \cdots \lambda x_n.x_1 \otimes \cdots \otimes x_n$	

Figure 3.9 An abstract grammar problem for solving GPP on EWPDSs.

$\overline{1}$. An EWPDS $\mathcal{W}_e^1$ that does not satisfy this restriction can always be converted to one, say $\mathcal{W}_e^2$, that does satisfy it as follows:

1. The semiring of $\mathcal{W}_e^2$ is over pairs of weights with the operations defined componentwise, i.e., the semiring is $((D,D),\oplus_p,\otimes_p,(\overline{0},\overline{0}),(\overline{1},\overline{1}))$, where both $\otimes_p$ and $\oplus_p$ are componentwise $\otimes$ and $\oplus$, respectively.

2. For every call rule $r = \langle p,\gamma\rangle \hookrightarrow \langle p',\gamma'\ \gamma''\rangle$ in $\mathcal{W}_e^1$ with weight $f(r)$ and merge function m_r, add the rules $r_1 = \langle p,\gamma\rangle \hookrightarrow \langle p,\gamma_r\rangle$ and $r_2 = \langle p,\gamma_r\rangle \hookrightarrow \langle p',\gamma'\ \gamma''\rangle$ to $\mathcal{W}_e^2$ with weights $(\overline{1},f(r))$ and $(\overline{1},\overline{1})$, respectively. The rule r_2 has the merge function $\lambda(x_1,x_2).\lambda(y_1,y_2).(m_r(x_1,y_1),\overline{1})$.

	Production			for each
(1)	$PushRuleSeq_{(q,\gamma,q')}$	$\rightarrow$	ε	$(q,\gamma,q') \in \rightarrow_0$
(2)	$SameLevelRuleSeq_{(p',\varepsilon,q)}$	$\rightarrow$	$PushRuleSeq_{(p,\gamma,q)}\ r$	$r = \langle p,\gamma\rangle \hookrightarrow \langle p',\varepsilon\rangle \in \Delta, q \in Q_e$
(3)	$PushRuleSeq_{(p',\gamma',q')}$	$\rightarrow$	$PushRuleSeq_{(q,\gamma',q')}\ SameLevelRuleSeq_{(p',\varepsilon,q)}$	$p' \in P,\ q,q' \in Q_e$
(4)	$PushRuleSeq_{(p',\gamma',q)}$	$\rightarrow$	$PushRuleSeq_{(p,\gamma,q)}\ r$	$r = \langle p,\gamma\rangle \hookrightarrow \langle p',\gamma'\rangle \in \Delta, q \in Q_e$
(5)	$PushRuleSeq_{(p',\gamma',p'_{\gamma'})}$	$\rightarrow$	ε	$r = \langle p,\gamma\rangle \hookrightarrow \langle p',\gamma'\ \gamma''\rangle \in \Delta$
(6)	$PushRuleSeq_{(p'_{\gamma'},\gamma'',q)}$	$\rightarrow$	$PushRuleSeq_{(p,\gamma,q)}\ r$	$r = \langle p,\gamma\rangle \hookrightarrow \langle p',\gamma'\ \gamma''\rangle \in \Delta, q \in Q_e$
(7)	$AcceptingRuleSeq[p\gamma_1\gamma_2\cdots\gamma_n]$	$\rightarrow$	$PushRuleSeq_{(p,\gamma_1,q_1)}\ \cdots PushRuleSeq_{(q_{n-1},\gamma_n,q_n)}$	$p \in P, \gamma_i \in \Gamma, q_i \in Q_e,$ for $1 \le i \le n, q_n \in F$

Figure 3.10 The *AcceptingRuleSeq* grammar for GPS, given PDS $\mathcal{P}$ and automaton $\mathcal{A}$. The set Q_e is defined as $(Q \cup Q_{\text{mid}})$

3. For every other rule r of $\mathcal{W}_e^1$, add r to $\mathcal{W}_e^2$ with weight $(f(r), \overline{1})$.

The pairing of weights that occurs in *poststar* is because of the above construction (but the transitions on the new stack symbols γ_r remain implicit).

The accepting rule-sequence grammar for GPS is shown in Fig. 3.10. The grammar that overapproximates it is shown in Fig. 3.11, and its language is shown in Fig. 3.12. The occurrences of $(D\ E)$ and $(C\ E)$ is where merge functions have to be applied. The abstract grammar that solves GPS is shown in Fig. 3.13. Again, this abstract grammar justifies the saturation procedure of Fig. 3.4: the latter is simply computing JOD values for the former.

Non-terminal	Over-approximates $PushRuleSeq_t$ for t in
A	$P \times \Gamma \times (Q - P)$
B	$P \times \Gamma \times Q_{\text{mid}}$
C	$Q_{\text{mid}} \times \Gamma \times (Q - P)$
D	$Q_{\text{mid}} \times \Gamma \times Q_{\text{mid}}$
E	$P \times \{\epsilon\} \times Q_{\text{mid}}$
F	$P \times \{\epsilon\} \times (Q - P)$
G	$(Q - P) \times \Gamma \times (Q - P)$

$$
\begin{aligned}
A &\rightarrow \varepsilon \mid A\ R_1 \mid C\ E \mid F \\
B &\rightarrow \varepsilon \mid B\ R_1 \mid D\ E \\
C &\rightarrow A\ R_2 \\
D &\rightarrow B\ R_2 \\
E &\rightarrow G\ B\ R_0 \\
F &\rightarrow G\ A\ R_0 \\
G &\rightarrow \varepsilon
\end{aligned}
$$

Figure 3.11 A grammar that over-approximates the grammar shown in Fig. 3.10.

Non-terminal	Language is over-approximated by
A	$(R_0 \mid \sigma_b)^*$
B	σ_b
C	$(R_0 \mid \sigma_b)^*\ R_2$
D	$\sigma_b\ R_2$
E	$\sigma_b\ R_0$
F	$(R_0 \mid \sigma_b)^*\ R_0$

Figure 3.12 The language of strings derivable from the non-terminals of the grammar shown in Fig. 3.11. Here σ_b is non-terminal of Fig. 3.2 that derives balanced sequences.

	Production		for each
(1)	$PushSeq_{(q,\gamma,q')}$	$\rightarrow\ g_1(\varepsilon)$	$(q,\gamma,q') \in \rightarrow_0$
	$g_1 = \overline{1}$		
(2)	$SameLevelSeq_{(p',\varepsilon,q)}$	$\rightarrow\ g_2(PushSeq_{(p,\gamma,q)})$	$r = \langle p,\gamma\rangle \hookrightarrow \langle p',\varepsilon\rangle \in \Delta, q \in Q_e$
	$g_2 = \lambda x.x \otimes f(r)$		
(3a)	$PushSeq_{(p',\gamma'',q')}$	$\rightarrow\ \ g_3(PushSeq_{(q,\gamma'',q')}, SameLevelSeq_{(p',\varepsilon,q)})$	
		$p' \in P,\ q' \in Q_e, q = p_{\gamma'} \in Q_{\text{mid}}, r = lookupPushRule(p,\gamma',\gamma'')$	
	$g_3 = \lambda x.\lambda y.m_r(y,x)$		
(3b)	$PushSeq_{(p',\gamma'',q')}$	$\rightarrow\ \ g'_3(PushSeq_{(q,\gamma'',q')}, SameLevelSeq_{(p',\varepsilon,q)})$	
		$p' \in P,\ q' \in Q_e, q \notin Q_{\text{mid}}$	
	$g'_3 = \lambda x.\lambda y.y \otimes x$		
(4)	$PushSeq_{(p',\gamma',q)}$	$\rightarrow\ g_4(PushSeq_{(p,\gamma,q)})$	$r = \langle p,\gamma\rangle \hookrightarrow \langle p',\gamma'\rangle \in \Delta, q \in Q_e$
	$g_4 = \lambda x.x \otimes f(r)$		
(5)	$PushSeq_{(p',\gamma',p'_{\gamma'})}$	$\rightarrow\ g_5(\varepsilon)$	$r = \langle p,\gamma\rangle \hookrightarrow \langle p',\gamma'\ \gamma''\rangle \in \Delta$
	$g_5 = \overline{1}$		
(6)	$PushSeq_{(p'_{\gamma'},\gamma'',q)}$	$\rightarrow\ g_6(PushSeq_{(p,\gamma,q)})$	$r = \langle p,\gamma\rangle \hookrightarrow \langle p',\gamma'\ \gamma''\rangle \in \Delta, q \in Q_e$
	$g_6 = \lambda x.x$		
(7)	$AcceptingSeq[p\gamma_1\gamma_2\cdots\gamma_n]$	$\rightarrow\ \ g_7(PushSeq_{(p,\gamma_1,q_1)}, \cdots, PushSeq_{(q_{n-1},\gamma_n,q_n)})$	
		$p \in P, \gamma_i \in \Gamma, q_i \in Q_e,$ for $1 \le i \le n, q_n \in F$	
	$g_7 = \lambda x_1 \cdots \lambda x_n.x_n \otimes \cdots \otimes x_1$		

Figure 3.13 An abstract grammar problem for solving GPS in an EWPDS. m_r is the merge function associated with rule r.

Chapter 4

Faster Interprocedural Analysis Using WPDSs

The previous chapter described various abstract models and their analyses for finding the set of reachable states. At the heart of all these analyses, and some others [81, 84, 88, 9, 33, 85], is a *chaotic-iteration strategy*. These analyses are saturation based: they use some set of rules in order to saturate the currently inferred set of reachable states. The analyses simply state that the rules can be applied in any order (e.g., see Section 2.3.2). Thus, their implementations are free to choose the rules in any order. The strategy of choosing any rule at random is called the chaotic-iteration strategy.

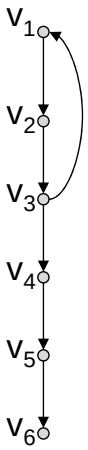


Figure 4.1 A simple dataflow model that has a graph with a loop.

For instance, many standard algorithms for dataflow analysis are worklist-based. They start with an initial value at the entry node and, at each step, propagate changes to a successor node by picking an outgoing edge at random. Consider running this strategy on the dataflow model shown in Fig. 4.1. Suppose each edge in the graph is labeled with a dataflow transformer, and we want to solve for the JOP value for all paths from node v_1 to node v_6. Also suppose that the loop (v_1 v_2 v_3) requires 10 iterations around the loop to reach a fixpoint.

If the algorithm starts at node v_1 and propagates changes to v_2, then v_3 and so on to v_6 before taking the backedge from v_3 to v_1, the algorithm would end up performing $6 \times 10 = 60$ operations (assuming one operation for propagating changes across a single edge). The ideal way of computing the JOP value is to first saturate the loop and then

go outside it, i.e., the changes are propagated within the loop until a fixpoint is reached, and then propagated to nodes v_4, v_5 and v_6 just once. This requires only $3 \times 10 + 3 = 33$ operations.

The general observation is that the iteration order matters for the total running time of a saturation-based analysis. Tarjan gave an efficient iteration order for finite graphs [91, 90] that applies to single-procedure dataflow models. We extend that algorithm to the interprocedural setting. To provide a common setting to discuss most of the above-mentioned analyses, we use WPDSs to describe our improvement to the chaotic-iteration strategy. Our techniques also apply to EWPDSs. Besides speeding up reachability, our techniques also help in witness generation, differential propagation, and incremental computation (Section 4.2).

Tarjan's algorithm [91, 90] works by efficiently converting a graph into a regular expression. Evaluating the regular expression (under an appropriate interpretation in which expression concatenation is interpreted as $\otimes$ and expression union is interpreted as $\oplus$) is sufficient to solve for the desired JOP weight. Our technique generalizes this algorithm to programs with multiple procedures as follows: for every procedure p, we introduce a variable X_p, which represents the summary of the procedure, i.e., the net effect of all (valid) paths that go from the entry of the procedure to its exit. Next, we replace procedure calls with their summary, i.e., a call to a procedure p in the ICFG is replaced with an intraprocedural edge labeled with X_p. This results in a collection of graphs, one for each procedure. Next, we use Tarjan's algorithm to obtain a set of equations: the graph for procedure p is converted to an equation $X_p = r_p$, where r_p is the regular expression for the graph. The expressions may depend on unknown variables. For example, if procedure p calls procedures q and s, then r_p may contain variables X_q and X_s.

The resulting equations can be solved using a chaotic-iteration strategy: for each procedure j, initialize the variable X_j to $\bot$ (or the semiring weight $\overline{0}$); next, pick an equation $X_i = r_i$, evaluate the expression r_i and update the value of X_i; and repeat until the values of all variables stop changing. This strategy would, however, give up most of the benefit of

using the regular expressions. We give an order in which the equations should be solved, and also show how to speed up multiple evaluations of the same expression. This results in an efficient interprocedural-analysis algorithm.

We also show how to reduce a WPDS reachability problem (Defn. 2.4.4) to a problem on graphs that can be solved in a similar fashion as above. When the PDS underlying the WPDS is obtained from an ICFG using the standard encoding (Fig. 2.4) then the graphs coincide with the control-flow graphs of the procedures in the ICFG.

The contributions of the work presented in this chapter can be summarized as follows:

- We present a new reachability algorithm for WPDSs and EWPDSs that improves on previously known algorithms for PDS reachability. The algorithm is asymptotically faster when the PDS is *regular* (decomposes into a single graph), and offers substantial improvement in the general case as well. (Section 4.1)

- The algorithm is demand-driven, and computes only that information needed for answering a particular user query. It has an implicit slicing stage where it disregards parts of the program not needed for answering the user query.

- We show that other analysis questions, namely witness tracing, differential propagation and incremental analysis, carry over to the new approach. (Section 4.2)

- We carried out experiments on three very different applications that use WPDSs and obtained substantial speedups for each of them. (Section 4.3)

The rest of this chapter is organized as follows: Section 4.1 presents our algorithm for solving reachability queries on WPDSs and EWPDSs. Section 4.2 describes algorithms for witness tracing, differential propagation and incremental analysis. Section 4.3 presents experimental results. Section 4.4 describes related work.

4.1 Solving WPDS Reachability Problems

In this section, we show how to speed up backward reachability on WPDSs (i.e., GPP; Defn. 2.4.4). Solving forward reachability is similar, but slightly more complicated.

Recall that solving GPP involves computing the join-over-all-derivations (JOD) value over the abstract grammar shown in Fig. 2.13. We will convert the JOD problem into one of computing join-over-all-valid-paths (JOVP) over a graph similar to the ICFG, and then use graph-based techniques. Our technique applies to solving GPS in exactly the same fashion by using the abstract grammar for GPS. (We use GPP in this section because its abstract grammar is smaller.)

In this section, fix $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$ as the WPDS, where $\mathcal{P} = (P, \Gamma, \Delta)$ is a pushdown system and $\mathcal{S} = (D, \oplus, \otimes, \overline{0}, \overline{1})$ is the weight domain. Let the initial set of configurations be ones that are accepted by the $\mathcal{P}$-automaton $\mathcal{A} = (Q, \Gamma, \rightarrow_0, P, F)$.

Definition 4.1.1. *A (directed)* ***hypergraph*** *is a generalization of a directed graph in which generalized edges, called* ***hyperedges****, can have multiple sources, i.e., the source of an edge is an ordered set of vertices. A* **transition dependence graph (TDG)** *for a grammar G is a hypergraph whose vertices are the non-terminals of G. There is a hyperedge from $(t_1, \cdots, t_n)$ to t if G has a production with t on the left-hand side and $t_1 \cdots t_n$ are the non-terminals that appear (in order) on the right-hand side.*

If we construct the TDG of the grammar shown in Fig. 2.13 when the underlying PDS is obtained from an ICFG, and the initial set of configurations is $\{\langle p, \varepsilon\rangle \mid p \in P\}$ (or $\rightarrow_0 = \emptyset$), then the TDG is almost identical to the ICFG (with the edges reversed). There are two differences in the way procedure calls are represented: the TDG has no analog of exit-node-to-return-node edges, and one of the predecessors of a call-node is the corresponding return-node. Fig. 4.2 shows an example (disregard the edge labels, nodes t_{s1} and t_{s2} and the dotted edges in Fig. 4.2(c) for now). This can be observed from the fact that except for the PDS states in Fig. 2.13, the transition dependencies are almost identical to the dependencies encoded in the pushdown rules, which in turn come from ICFG edges; e.g., in Fig. 4.2, the

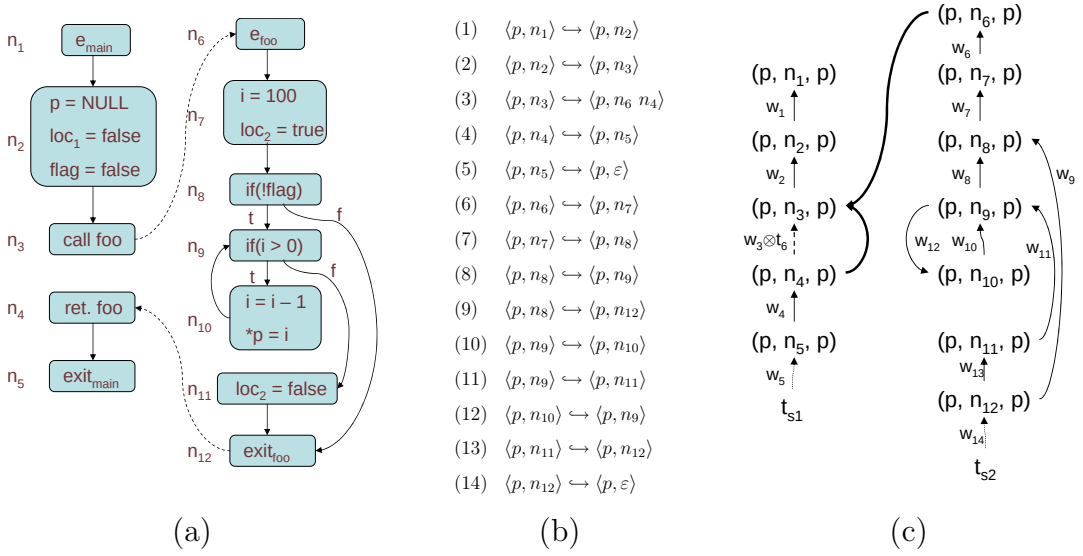


Figure 4.2 (a) An ICFG. The e and *exit* nodes represent entry and exit points of procedures, respectively. The program statement are only written for illustration purposes. Dashed edges represent interprocedural control flow. (b) A PDS system that models the control flow of the ICFG. (c) The TDG for the WPDS whose underlying PDS is shown in (b), assuming that rule number i has weight w_i. The non-terminal $PopSeq_{\langle p,\gamma,p'\rangle}$ is shown as simply (p, γ, p'). Let t_j stand for the node (p, n_j, p). The thick bold arrows form a single hyperedge. Nodes t_{s1} and t_{s2} are root nodes, and the dashed arrow is a summary edge.

ICFG edge (n_1, n_2) corresponds to the transition dependence $((t_2), t_1)$, and the call-return pair (n_3, n_6) and (n_{12}, n_4) in the ICFG corresponds to the hyperedge $((t_4, t_6), t_3)$.

For such PDSs, which are obtained from ICFGs, constructing the TDGs might seem unnecessary (because the ICFG was already available) but it allows us to generalize to an arbitrary initial set of configurations, which defines a region of interest in the program. Moreover, PDSs can encode a larger range of constructs than an ICFG, such as setjmp/longjmp in C programs. However, it is still convenient to think of a TDG as an ICFG. In the rest of this chapter, we illustrate the issues using the TDG of the grammar in Fig. 2.13. We reduce the join-over-all-derivation problem on the grammar to a join-over-all-valid-paths problem on its TDG.

4.1.1 Intraprocedural Iteration

We first consider TDGs of a special form: consider the intraprocedural case, i.e., there are no hyperedges in the TDG (and correspondingly no push rules in the PDS). As an example, assume that the TDG in Fig. 4.2 has only the part corresponding to procedure `foo()` without any hyperedges. In such a TDG, if an edge $((t_1), t)$ was inserted because of the production $t \to g(t_1)$ for $g = \lambda x.x \otimes w$ for some weight w, then label this edge with w. Next, insert a special node t_s into the TDG, and for each production of the form $t \to g(\epsilon)$ with $g = w$, insert the edge $((t_s), t)$ and label it with weight w. t_s is called a root node. This gives us a graph with weights on each edge.[1] Define the weight of a path in this graph in the standard (but reversed) way: the weight of a path is the extend of weights on its constituent edges in the reverse order. It is easy to see that $\mathrm{JOD}(t) = \bigoplus\{v(\eta) \mid \eta \in \mathit{paths}(t_s, t)\}$, where $\mathit{paths}(t_s, t)$ is the set of all paths from t_s to t in the TDG and $v(\eta)$ is the weight of the path η. To solve for JOD, we could still use chaotic iteration, but instead we will make use of Tarjan's path-expression algorithm [90].

Problem 1. *Given a directed graph G and a fixed vertex s, the* ***single-source path expression*** *(SSPE) problem is to compute a regular expression that represents $\mathit{paths}(s, v)$ for all vertices v in the graph. The syntax of regular expressions is as follows: $r ::= \emptyset \mid \varepsilon \mid e \mid r_1 \cup r_2 \mid r_1.r_2 \mid r^*$, where e stands for an edge in G.*

We can use any algorithm for SSPE to compute regular expressions for $\mathit{paths}(t_s, t)$, which gives us a complete description of the set of paths that we need to consider. Moreover, the Kleene-star operator in the regular expressions identifies loops in the TDG. Let $\otimes^c$ be the reverse of $\otimes$, i.e., $w_1 \otimes^c w_2 = w_2 \otimes w_1$. To compute $\mathrm{JOD}(t)$, we interpret the regular expression for $\mathit{paths}(t_s, t)$ as an expression over the weight domain: replace each edge e with its weight, $\emptyset$ with $\overline{0}$, ε with $\overline{1}$, $\cup$ with $\oplus$, . with $\otimes^c$; and then evaluate the expression. The weight w^* is computed as $\overline{1} \oplus w \oplus (w \otimes w) \oplus \cdots$; because of the no-infinite-ascending-chain property of the semiring, this iteration converges. The two main advantages of using regular

[1] A hypergraph reduces to a graph when all hyperedges have a single source node.

expressions to compute JOD(t) are: First, loops are identified in the expression, and the evaluation of an expression is forced to saturate a loop before exiting it. Second, we can compute w^* faster than using the normal iteration sequence. For this, observe that

$$(\overline{1} \oplus w)^n = \overline{1} \oplus w \oplus w^2 \oplus \cdots \oplus w^n$$

where exponentiation is defined using $\otimes$, i.e., $w^0 = \overline{1}$ and $w^i = w \otimes w^{(i-1)}$. Then w^* can be computed by repeatedly squaring $(\overline{1} \oplus w)$ until it converges. If $w^* = \overline{1} \oplus w \oplus \cdots \oplus w^n$ then it can be computed in $O(\log n)$ operations. A chaotic-iteration strategy would take $O(n)$ steps to compute the same value. In other words, having a closed representation of loops provides an exponential speedup.[2]

Tarjan's path-expression algorithm solves the SSPE problem efficiently. It uses *dominators* to construct the regular expressions for SSPE. This has the effect of computing the weight on the dominators of a node before computing the weight on the node itself. This avoids unnecessary propagation of weights to the node (which is the case, for instance, when one exits a loop too early). Given a graph with m edges (or m grammar productions in our case) and n nodes (or non-terminals), regular expressions for $\mathit{paths}(t_s, t)$ can be computed for all nodes t in time $O(m \log n)$ when the graph is *reducible*. Evaluating these expressions will take an additional $O(m \log n \log h)$ semiring operations, where h is the height of the semiring.[3] These expressions are represented using shared DAGs, i.e., expressions for $\mathit{paths}(t_s, t_1)$ and $\mathit{paths}(t_s, t_2)$ can share common sub-expressions, even when $t_1 \neq t_2$. The combined size of all the regular expressions is bounded by the time taken to find the expressions; i.e., the combined size is $O(m \log n)$.

Because most high-level languages are well-structured, their ICFGs are mostly reducible. When the graph is not reducible, the running time degrades to $O((m \log n + k) \log h)$ semiring operations, where k is the sum of the cubes of the sizes of *dominator-strong components* of the

[2]This assumes that each semiring operation takes the same amount of time. In the absence of any assumption on the semiring being used, we aim to decrease the number of semiring operations. In some cases, e.g., BDD-based weight domains, repeated squaring may not reduce the overall running time. However, the user can supply a procedure for computing w^* whenever there is a more efficient way of computing it than by using simple iteration sequence [63].

[3]As usual, we assume the height to be bounded while discussing complexity results.

graph. In the worst case, k can be $O(n^3)$. In our experiments, we seldom found irreducibility to be a problem: k/n was a small constant. A pure chaotic-iteration strategy would take $O(m\ h)$ semiring operations in the worst case. Comparing these complexities, we can expect the algorithm that uses path expressions to be much faster than chaotic iteration, and the benefit will be greater as the height of the semiring increases.

4.1.2 Interprocedural Iteration

We now generalize our algorithm to any TDG. For each hyperedge $((t_1, t_2), t)$, delete it from the graph and replace it with the edge $((t_1), t)$. This new edge is called a *summary edge*, and node t_2 is called an *out-node*. Out-nodes will be used to represent the summary weight of a procedure. For example, in Fig. 4.2, we would delete the hyperedge $((t_4, t_6), t_3)$ and replace it with $((t_4), t_3)$. The new edge is called a summary edge because it crosses a call-site (from a return node to a call node) and will be used to summarize the effect of a procedure call. Node t_6 is an out-node and will supply the summary weight of procedure `foo`. The resultant TDG is a collection of connected graphs, with each graph roughly corresponding to a procedure. In Fig. 4.2, the transitions that correspond to procedures `main` and `foo` get split. Each connected graph is called an *intragraph*. For each intragraph, we introduce a root node as before, and add edges from the root node to all nodes that have ϵ-productions. The weight labels are also added as before. For a summary edge $((t_1), t)$ obtained from a hyperedge $((t_1, t_2), t)$ with associated production function $g = \lambda x.\lambda y.w \otimes x \otimes y$, label it with $w \otimes t_2$, or $t_2 \otimes^c w$.

This gives us a collection of intragraphs with edges labeled with either a weight or a simple expression over an out-node. To solve for the JOD value, we construct a set of *regular equations*, which we call *out-node equations*. For an intragraph G, let t_G be its unique root node. Then, for each out-node t_o in G, construct the regular expression for all paths in G from t_G to t_o, i.e., for $paths(t_G, t_o)$. In this expression, replace each edge with its corresponding label. If the resulting expression is r and it contains labels t_1 to t_n, then add the equation $t_o = r(t_1, \cdots, t_n)$ to the set of out-node equations. Repeat this for

all intragraphs. For example, for the TDG shown in Fig. 4.2, assuming that t_1 is also an out-node, we would obtain the following out-node equations.[4]

$$\begin{aligned} t_6 &= w_{14}.(w_9 \oplus w_{13}.w_{11}.(w_{12}.w_{10})^*.w_8).w_7.w_6 \\ t_1 &= w_5.w_4.(t_6.w_3).w_2.w_1 \end{aligned}$$

Here we have used . as a shorthand for $\otimes^c$.

The resulting set of out-node equations describe all hyperpaths in the TDG to an out-node from the collection of all root nodes. Hence, the JOD value of the out-nodes is the least fix-point of these equations (with respect to $\sqsubseteq$ of Defn. 2.4.1(4)).

One way to solve these equations is by using chaotic iteration: start by initializing each out-node with $\overline{0}$ (the least element in the semiring) and update the values of out-nodes by repeatedly solving the equations until they converge. However, learning from our previous observations, we give a direction to the iteration strategy. This can be done using regular expressions on the dependence graph of the equations as follows. For each equation $t_o = r(t_1, \cdots, t_n)$, produce the edges $t_i \rightarrow t_o, 1 \leq i \leq n$ and construct a graph from these edges. Label each edge with the expression (r) that it came from. Assume any out-node to be the source node and construct a regular expression to all other nodes using SSPE again. These expressions give the order in which equations have to be evaluated. For example, consider the following set of equations on three out-nodes:

$$t_1 = r_1(t_1, t_3) \quad t_2 = r_2(t_1) \quad t_3 = r_3(t_2)$$

Then a possible regular expression for paths from t_1 to itself would be $(r_1 \cup r_2.r_3.r_1)^*$. This suggests that to solve for t_1 we should use the following evaluation strategy: evaluate r_1, update t_1, then evaluate r_2, r_3, and r_1, and update t_1 again — repeating this until the solution converges.

In our implementation, we use a simpler strategy that still turns out to be efficient in practice. We take a *strongly connected component* (SCC) decomposition of the dependence graph and solve all equations in one component, using chaotic-iteration, before moving on to the equations in the next component (in a topological order). This is efficient because

[4]The equations might be different depending on how the SSPE problem was solved, but all such equations would have the same solution.

SCCs in the dependence graph correspond to a set of mutually recursive procedures and these groups tend to be quite small in practice.

Each regular expression in the out-node equations summarizes all paths in an intragraph, and can be quite large. Therefore, we want to avoid evaluating them repeatedly while solving the equations. To this end, we incrementally evaluate the regular expressions: only that part of an expression is reevaluated that contains a modified out-node. The algorithm is given in Fig. 4.4. Whenever this algorithm is used on a regular expression, the whole expression may be traversed, but it only performs weight operations on nodes such that the sub-expression rooted at that node contains a modified out-node.

A regular expression is represented using its abstract-syntax tree (AST), where leaves are weights or out-nodes, and internal nodes correspond to $\oplus$, $\otimes$, or *. A possible AST for the regular expression for out-node t_1 of Fig. 4.2 is shown in Fig. 4.3. Whenever the value of out-node t_6 is updated, one only needs to reevaluate the weight of subtrees at a_4, a_3, and a_1, and update the value of out-node t_1 to the weight at a_1.

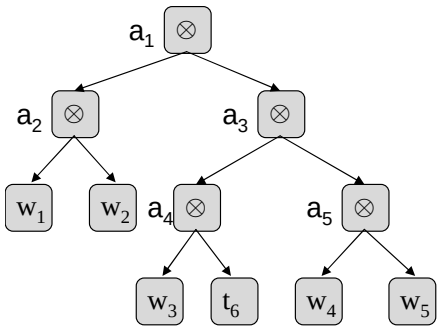


Figure 4.3 An AST for $w_5.w_4.(w_3 \otimes t_6).w_2.w_1$. Internal nodes for $\otimes^c$ are converted into $\otimes$ nodes by reversing the order of its children. Internal nodes in this AST have been given names a_1 to a_5.

One complication that we face here is that the ASTs actually have a shared-DAG representation to allow different expressions to share common sub-expressions. (This is a requirement of Tarjan's path-expression algorithm.) Thus, our algorithm needs to take care of two aspects: it should benefit from the sharing in the DAGs as much as possible, and it must also be able to identify the part of an expression that is modified when the weight of an

out-node is updated. For this, we maintain, at each DAG node, two integers, `last_change` and `last_seen`, as well as the weight `weight` of the subdag rooted at the node. We assume that all regular expressions share the same leaves for out-nodes. We keep a global counter `update_count` that is incremented each time the weight of some out-node is updated. Our incremental evaluation algorithm is shown in Fig. 4.4. After calling *evaluate*(`r`), the weight `r.weight` is the correct updated weight of expression `r`, no matter how many times the weights of out nodes were updated since the last call to *evaluate* on `r`.

For a node, the counter `last_change` records the last value of `update_count` for which the weight of its subdag changed, and the counter `last_seen` records the value of `update_count` when the subdag was reevaluated. When the weight of an out-node is changed, its corresponding leaf node is updated with that weight, `update_count` is incremented, and both of the out-node's counters (`last_change` and `last_seen`) are set to `update_count`.

This incremental-evaluation algorithm is used as follows: we solve the out-node equations in the same order as described earlier, but as the algorithm iterates over the equations, whenever it picks an equation $t = r$, it calls *evaluate*(r) to compute the weight of r incrementally; next, it updates the value of t to this weight and increments `update_count`. These steps are repeated for each out-node equation.

Once we solve for the values of the out-nodes, we can change the out-node labels on summary edges in the intragraphs and replace them with their corresponding weight. Then the JOD values for other nodes in the TDG can be obtained as in the intraprocedural version by considering each intragraph in isolation.

The time required for solving this system of equations depends on the reducibility of the intragraphs. Let S_G be the time required to solve SSPE on intragraph G, i.e., $S_G = O(m \log n + k)$ where k is $O(n^3)$ in the worst-case, but is ignorable in practice. If the equations do not have any cyclic dependencies (corresponding to no mutually recursive procedures) then the running time is $\sum_G S_G \log h$, where the sum ranges over all intragraphs, because each equation has to be solved exactly once. In the presence of recursion, we use the observation that the weight of each subdag in a regular expression can change at most h times while

```
1  procedure evaluate(r)
2  begin
3    if r.last_seen == update_count then
4      return;
5    case r = w, r = t_o return;
6    case r = r_1^*
7      evaluate(r_1)
8      if r_1.last_change > r.last_seen then
9        w = (r_1.weight)^*
10       if r.weight ≠ w then
11         r.last_change = r_1.last_change
12         r.weight = w
13     r.last_seen = update_count
14   case r = r_1 ⊙ r_2
15     evaluate(r_1)
16     evaluate(r_2)
17     m = max{r_1.last_change, r_2.last_change}
18     if m > r.last_seen then
19       w = r_1.weight ⊙ r_2.weight
20       if r.weight ≠ w then
21         r.last_change = m
22         r.weight = w
23     r.last_seen = update_count
24 end
```

Figure 4.4 Incremental evaluation algorithm for regular expressions. Here $\odot$ stands for either $\oplus$ or $\otimes$.

the equations are being solved. Because the size of a regular expression obtained from an intragraph G is bounded by S_G, the worst-case time for solving the equations is $\sum_G S_G\ h$.

This bound is very pessimistic and is actually worse than that of chaotic iteration. Here we did not make use of the fact that incrementally reevaluating regular expressions is much faster than reevaluating them from scratch. For a regular expression with one modified out-node, we only need to perform semiring operations for each node from the out-node leaf to the root of the expression. For a nearly balanced regular expression tree, this path to the root can be as small as $\log S_G$. Empirically, we found that incrementally reevaluating the expression required many fewer operations than reevaluating the expression from scratch.

Unlike with chaotic iteration, where the weights of all TDG nodes are computed, we only need to compute the weights on out-nodes. The weights for the rest of the nodes can be computed lazily by evaluating their corresponding regular expression when needed. For applications that just require the weight for a few TDG nodes, this gives us additional savings. We also limit the computation of weights of out-nodes to only those intragraphs that contain a TDG node whose weight is required. This corresponds to slicing the out-node equations with respect to the user query, which rules out computation in procedures that are irrelevant to the query. Moreover, the algorithm can be executed on multi-processor machines by assigning each intragraph to a different processor. The only communication required between the processors would be the weights on out-nodes while they are being saturated.

4.1.3 Solving EWPDS Reachability Problems

Reachability problems for EWPDSs are also based on abstract grammars, similar to the ones for WPDSs. Thus, we can easily adapt our algorithm to EWPDSs. The abstract grammar for GPP and GPS on EWPDSs are shown in Figs. 3.9 and 3.13, respectively.

These grammars only differ from those for WPDSs in the application of the merge function. This difference can be handled as follows: to solve GPP, for hyperedges in the TDG corresponding to case 4 of Fig. 3.9, if t_o is the out-node, then label the corresponding summary edge with $m_r(\overline{1}, t_o)$ (in keeping with the production function g_4). We use EWPDSs in our experiments.

4.2 Solving other WPDS Problems

In this section, we give algorithms for some important PDS problems: witness tracing, differential propagation, and incremental analysis. Of these three, only witness tracing and differential propagation have been discussed before for WPDSs [83].

4.2.1 Witness Tracing

For program-analysis tools, if a program does not satisfy a property, it is often useful to provide a justification of why the property was not satisfied. In terms of WPDSs, it amounts to reporting a set of paths, or rule sequences, that together justify the reported weight for a configuration. Formally, using the notation of Defn. 2.4.4, the witness tracing problem for GPP(C) is to find, for each configuration c, a set $\omega(c) \subseteq \bigcup_{c' \in C} \mathit{paths}(c, c')$ such that $\bigoplus_{\sigma \in \omega(c)} v(\sigma) = \delta(c, C)$. This definition of witness tracing does not impose any restrictions on the size of the reported witness set because any compact representation of the set suffices for most applications. The algorithm for witness tracing for GPP [83] requires $O(|Q|^2\ |\Gamma|\ h)$ memory. Our algorithm only requires $O(|ON|\ D\ h)$ memory, where $|ON|$ is the number of out-nodes and D is the maximum number of out-nodes that appear on the right-hand side of an out-node equation. Typically, $|ON|$ is the number of procedures, which is much smaller than $|\Gamma|$, and D is the maximum number of call sites in any procedure, which is usually a small constant. One can consider $(|ON|D)$ to be roughly the size of the call graph of a program. Essentially, the idea behind our algorithm is to perform a two-level *staging*, where only a subset of the witness information needs to be kept in memory, and the rest can be computed on demand.

In our new GPP algorithm, we already compute regular expressions that describe all paths in an intragraph. In the intragraphs, we label each edge with not just a weight, but also the rule that justifies the edge. Push rules will be associated with summary edges and pop rules with edges that originate from a root node. Edges from the root node that were inserted because of production (1) in Fig. 2.13 are not associated with any rule (or with an

empty rule sequence). After solving SSPE on the intragraphs, we can replace each edge with the corresponding rule label. This gives us, for each out-node, a regular expression in terms of other out-nodes that captures the set of all rule sequences that can reach that out-node.

While solving the out-node equations, we record the weights on out-nodes; i.e., when we solve the equation $t_o = r(t_1, \cdots, t_n)$, we record the weights on $t_1, \cdots, t_n$ — say $w_1, \cdots, w_n$ — whenever the weight on t_o changes to, say, w_o, by saving the tuple $(t_o, w_o, t_1, w_1, \cdots, t_n, w_n)$ to memory. Then the set of rule sequences to create transition t_o with weight w_o is given by the expression r (where we replace TDG edges with their rule labels) by replacing each out-node t_i with the regular expression for all rule sequences used to create t_i with weight w_i (obtained recursively). This gives a regular expression for the witness set of each out-node. Witness sets for other transitions can be obtained by solving SSPE on the intragraphs by replacing out-node labels with their witness-set expression.

We only require $O(|ON|\ D\ h)$ space for recording witnesses because we just have to remember the history of weights on out-nodes, and each piece of information is at most a $(2D + 2)$-ary tuple.

4.2.2 Differential Propagation

The general framework of WPDSs can sometimes be inefficient for certain analysis. While executing GPP, when the weight of a transition changes from w_1 to $w_2 = w_1 \oplus w$, the new weight w_2 is propagated to other transitions. However, because the weight w_1 had already been propagated, this will do extra work by propagating w_1 again when only w (or a part of w) needs to be propagated. This simple observation can be incorporated into WPDSs when the semiring weight domain has a special subtraction operation (called *diff*, denoted by $\dot{-}$) [83]. The *diff* operator must satisfy the following properties: For each $a, b, c \in D$,

$$a \oplus (b \dot{-} a) = a \oplus b \tag{4.1}$$

$$(a \dot{-} b) \dot{-} c = a \dot{-} (b \oplus c) \tag{4.2}$$

$$a \oplus b = a \iff b \dot{-} a = \overline{0} \tag{4.3}$$

For example, for the relational weight domain (Defn. 2.4.13), set difference (when relations are considered to be sets of tuples) satisfies all of the above properties.

We make use of the *diff* operation while solving the set of regular equations. In addition to incrementally computing the regular expressions, we also incrementally compute the weights. When the weight of an out-node changes from w_1 to w_2, we associate its corresponding leaf node with the change $w_2 \dot{-} w_1$. This change is then propagated to other nodes. If the weight of expressions r_1 and r_2 are w_1 and w_2, respectively,and they change by d_1 and d_2, then the weights of the following kinds of expressions change as follows:

$$\begin{aligned} r_1 \cup r_2 &: d_1 \oplus d_2 \\ r_1.r_2 &: (d_1 \otimes^c d_2) \oplus (d_1 \otimes^c w_2) \oplus (w_1 \otimes^c d_2) \\ r_1^* &: (w_1 \oplus d_1)^* \dot{-} w_1^* \end{aligned}$$

There is no better way of computing the change for Kleene-star (chaotic iteration suffers from the same problem), but we can use the *diff* operator to compute the Kleene-star of a weight as shown in Fig. 4.5.

Theorem 4.2.1. *The procedure Kleene-star, defined in Fig. 4.5, when applied to weight w, returns w^*.*

Proof. The proof is by induction. But, first, we need some auxiliary properties of *diff*.

$$a \dot{-} b = \overline{0} \text{ and } b \dot{-} a = \overline{0} \implies (a = b) \tag{4.4}$$

$$(a \oplus b) \dot{-} a = (b \dot{-} a) \tag{4.5}$$

Eqn. (4.4): This follows from Eqn. (4.3): $a \dot{-} b = \overline{0}$ implies $(b \oplus a) = b$ and $b \dot{-} a = \overline{0}$ implies that $(a \oplus b) = a$. Because $\oplus$ is commutative, $a = b$.

```
1  procedure Kleene-star(w)
2  begin
3     wstar = del = 1̄
4     while del ≠ 0̄
5        temp = del ⊗ w
6        del = temp ⊖ wstar
7        wstar = wstar ⊕ temp
8     return wstar
9  end
```

Figure 4.5 Procedure for computing the Kleene-star of a weight using the *diff* operation on weights.

Eqn. (4.5): To prove this equality, we will show that $(lhs \dot{-} rhs) = \overline{0}$ and $(rhs \dot{-} lhs) = \overline{0}$. Then Eqn. (4.4) shows that $lhs = rhs$.

$$
\begin{aligned}
((a \oplus b) \dot{-} a) \dot{-} (b \dot{-} a) &= (a \oplus b) \dot{-} (a \oplus (b \dot{-} a)) && \text{by Eqn. (4.2)} \\
&= (a \oplus b) \dot{-} (a \oplus b) && \text{by Eqn. (4.1)} \\
&= \overline{0} && \text{by Eqn. (4.3)} \\
(b \dot{-} a) \dot{-} ((a \oplus b) \dot{-} a) &= b \dot{-} (a \oplus ((a \oplus b) \dot{-} a)) && \text{by Eqn. (4.2)} \\
&= b \dot{-} (a \oplus b) && \text{by Eqn. (4.1)} \\
&= \overline{0} && \text{by Eqn. (4.3)}
\end{aligned}
$$

Let $S_n(w) = \overline{1} \oplus w \oplus w^2 \oplus \cdots \oplus w^n$, where $S_0(w) = \overline{1}$, and $S_{-1}(w) = \overline{0}$. Then note that $S_n(w) = \overline{1} + (w \otimes S_{n-1}(w))$ for all $n \geq 0$. The invariant in Fig. 4.5 is that whenever execution reaches line 4 for the n^{th} time, wstar $= S_n(w)$ and del $= (S_n(w) \dot{-} S_{n-1}(w))$. We will prove this invariant by induction, but it is easy to see that if this invariant holds, and the while loop terminates, then wstar $= w^*$.

The base case is $n = 1$, and is easy to establish. The inductive case is proved as follows. The variable wstar is updated in the loop body to wstar $\oplus$ (del $\otimes\, w$). This equals:

$$
\left.
\begin{array}{rl}
 & S_n(w) \oplus (S_n(w) \mathbin{\dot{-}} S_{n-1}(w)) \otimes w \\
= & (\overline{1} \oplus w \otimes S_{n-1}(w)) \oplus (S_n(w) \mathbin{\dot{-}} S_{n-1}(w)) \otimes w \\
= & \overline{1} \oplus (w \otimes (S_{n-1}(w) \oplus (S_n(w) \mathbin{\dot{-}} S_{n-1}(w)))) \\
= & \overline{1} \oplus (w \otimes (S_{n-1}(w) \oplus S_n(w))) \\
= & \overline{1} \oplus (w \otimes S_n(w)) \\
= & S_{n+1}(w)
\end{array}
\right\} (*)
$$

The variable del is updated in the loop body to $(\mathrm{del} \otimes w) \mathbin{\dot{-}} \mathrm{wstar}$. This equals:

$$
\begin{array}{rll}
 & ((S_n(w) \mathbin{\dot{-}} S_{n-1}(w)) \otimes w) \mathbin{\dot{-}} S_n(w) & \\
= & (S_n(w) \oplus ((S_n(w) \mathbin{\dot{-}} S_{n-1}(w)) \otimes w)) \mathbin{\dot{-}} S_n(w) & \text{by Eqn. (4.5)} \\
= & S_{n+1}(w) \mathbin{\dot{-}} S_n(w) & \text{by } (*)
\end{array}
$$

□

4.2.3 Incremental Analysis

An incremental algorithm for verifying finite-state properties on ICFGs was given by Conway et al. [22]. We can use the methods presented in this chapter to generalize their algorithm to WPDSs. An incremental approach to verification has the advantage of amortizing the verification time across program development or debugging time.

We consider two cases: addition of new rules and deletion of existing ones. In each case, we work at the granularity of intragraphs. Let $\mathcal{W}$ be the original WPDS for which we have already computed the out-node equations E and solved them. Let $\mathcal{W}'$ be the WPDS obtained from $\mathcal{W}$ after making some changes to it.

First, consider the addition of new rules. In this case, the fix-point solution of the out-node equations monotonically increases and we can reuse all of the existing computation. We identify the intragraphs that changed (i.e., they have more edges) because of the new rules. Next, we recompute the regular expressions for out-nodes in those intragraphs and

add them to the set of out-node equations E.[5] Then we solve the equations as described in Section 4.1.2, but after setting different initial weights for the out-nodes. In the algorithm described in Section 4.1.2, the initial weights of all out-nodes were $\overline{0}$. For the incremental algorithm, set the initial weights of out-nodes that appear in E to be the weight obtained after solving E, and set the initial weights of new out-nodes (i.e., ones that did not appear in E) to $\overline{0}$. This leverages the existing information to reach a fix-point sooner.

Deletion of a rule requires more work. Again, we identify the changed intragraphs and recompute the out-node equations for them. We call out-nodes in these intragraphs *modified* out-nodes. Next, we construct the dependence graph of the out-node equations as described in Section 4.1.2. We perform an SCC decomposition of this graph and topologically sort the SCCs. Then the weights for all out-nodes that appear before the first SCC that has a modified out-node need not be changed. Thus, we set the value of these out-nodes to be the weights obtained after solving E. We recompute the solution for other out-nodes in topological order, and stop as soon as the new weights agree with previous weights. This is done as follows. We start with out-nodes in the first SCC that has a modified out-node; initialize the weights of all out-nodes in this SCC to be $\overline{0}$, and solve the out-nodes equations for the SCC. If the new weight of an out-node is different from its previously computed weight, all out-nodes in later SCCs that are dependent on it are marked as modified. We repeat this procedure until there are no more modified out-nodes.

The advantage of doing incremental analysis in our framework is that very little information has to be stored between analysis runs: we only need to store the computed weights for out-nodes.

4.3 Experiments

We compare our algorithm from Section 4.1 against the ones from [83] and Section 3.2 (for WPDSs and EWPDSs, respectively), which are implemented in WPDS++ [49]. We refer to

[5]There are incremental algorithms for SSPE as well, but we have not used them because solving SSPE for a single intragraph is usually very fast.

the implementation of our algorithm as FWPDS (F stands for "fast"). WPDS++ supports an optimized iteration strategy (over chaotic iteration) where the user can supply a priority-ordering on stack symbols, which is used by chaotic iteration to choose the transition with least priority first. We refer to this version as BFS-WPDS++ and supply it with a breadth-first ordering on the ICFG obtained by treating it as an ordinary graph. BFS-WPDS++ almost always performs better than WPDS++ with chaotic iteration.

To measure end-to-end performance, FWPDS only computes the weight on transitions of the output automaton (corresponding to TDG nodes) that are required by the application. We also report the time taken to compute the weight on all transitions and refer to this as FWPDS-Full. A comparison with FWPDS-Full will give an indication of "application-independent" improvement provided by our approach because it computes the same amount of information as the previous WPDS algorithms. However, we measure speedups using FWPDS running times to show the potential of using lazy-evaluation in a real setting. FWPDS-Full uses a left-associative evaluation order for computing weights of regular expressions. It is also worth noting that repeated squaring for computing w^* did not cause any appreciable difference compared with using a simple iterative method.

We tested FWPDS on three applications that use (E)WPDSs. In each, we perform GPS on the (E)WPDS with the entry point of the program as the initial configuration. The first application performs affine-relation analysis (ARA) on x86 programs [60]. An x86 program is translated into a WPDS to find affine relationships between machine registers. The application only requires affine relationships at branch points [3]. The results are shown in Tab. 4.1. Over all the experiments we performed, FWPDS provided an average speedup of 1.8× (i.e., reduced running time by 44%) over BFS-WPDS++.

The second application, BTrace, is for debugging [56]. It performs path optimization on C programs: given a set of ICFG nodes, called critical nodes, it tries to find a shortest ICFG path that touches the maximum number of these nodes. The path starts at the entry point of the program and stops at a given failure point in the program. FWPDS only computes

			Time (s)				Speedup
Prog	Insts	# Procs	WPDS++	BFS-WPDS++	FWPDS-Full	FWPDS	
mplayer2	40052	385	2.11	1.30	1.17	0.69	1.88
print	75539	697	1.23	1.02	0.77	0.41	2.49
find	76240	703	11.03	8.17	6.99	4.58	1.78
attrib	76380	703	2.52	2.11	1.57	0.89	2.37
doskey	77983	716	2.27	1.83	1.15	0.75	2.44
xcopy	87000	780	22.28	15.78	13.68	8.80	1.79
sort	89291	840	13.47	11.16	10.00	6.34	1.76
more	90792	860	17.42	11.92	10.54	7.02	1.70
tracert	95459	870	9.83	8.16	7.03	4.45	1.83
finger	96123	893	11.14	7.94	7.13	4.44	1.79
rsh	100935	941	18.31	13.17	11.65	7.47	1.76
javac	101369	944	20.12	16.20	14.65	9.25	1.75
lpr	110301	1011	14.83	11.75	10.57	7.06	1.66
java	112305	1049	24.77	20.19	19.01	11.97	1.69
ftp	130255	1253	22.84	15.13	14.23	8.98	1.68
winhlp32	157634	1612	25.51	19.61	17.32	11.00	1.78
regsvr32	225857	2789	58.70	38.83	37.15	24.65	1.58
cmd	230481	2317	69.19	46.33	52.38	34.87	1.33
notepad	239408	2911	54.08	40.80	41.85	26.50	1.54

Table 4.1 Comparison of ARA results. The last column show the speedup (ratio of running times) of FWPDS versus BFS-WPDS++. The programs are common Windows executables, and the experiments were run on 3.2 Ghz P4 machine with 4GB RAM.

the weight at the failure point. As shown in Tab. 4.2, FWPDS performs much better than BFS-WPDS++ for this application, and the overall speedup was 3.6×.

The third application is Moped [50], which is a model checker for Boolean programs. It uses its own WPDS library for performing reachability queries (which is, again, based on the chaotic-iteration strategy). Weights are binary relations on valuations of Boolean variables, and are represented using BDDs. We measure the performance of FWPDS against

			Time (s)			Speedup
Prog	ICFG nodes	# Procs	BFS-WPDS++	FWPDS-Full	FWPDS	
uucp	16973	139	4.7	3.3	2.9	1.60
mc	78641	676	5.4	5.2	3.1	1.72
make	40667	204	15.1	7.7	5.8	2.58
indent	28155	104	19.6	28.2	15.9	1.24
less	33006	359	22.4	8.6	5.3	4.19
patch	27389	133	70.2	23.2	17.1	4.09
gawk	86617	401	72.7	64.5	45.1	1.61
wget	44575	399	318.4	58.9	27.0	11.77

Table 4.2 Comparison of BTrace results. The last column shows speedup of FWPDS over BFS-WPDS++. The critical nodes were chosen at random from ICFG nodes and the failure site was set as the exit point of the program. The programs are common Unix utilities, and the experiments were run on 2.4 GHz P4 machine with 4GB RAM.

this library using a set of programs (and an error configuration for each program) supplied by S. Schwoon. We compute the set of all variable valuations that can hold at the error configuration by computing its JOP weight. As shown in Tab. 4.3, FWPDS is 2 to 5 times faster than Moped.

Moped can also be asked to stop as soon as it finds out that the error configuration is reachable (instead of exploring all paths that lead to the error configuration). In that case, when the error configuration was reachable, Moped performed much better than FWPDS, often completing in less than a second. This is expected because the evaluation strategy used by FWPDS is oriented towards finding the complete weight (JOD value) on a transition. For example, it might be better to avoid saturating a loop completely and propagate partially computed weights in the hope of finding out that the error configuration is reachable. However, when the error configuration is unreachable, or when the abstraction-refinement mode in Moped is turned on, it explores all paths in the program and computes the JOD value of all transitions. In such situations, it is likely to be better to use FWPDS.

Prog	MOPED	FWPDS-Full	FWPDS	Speedup
bugs5	13.11	13.03	7.25	1.81
slam-fixed	32.67	19.23	13.3	2.46
slam	6.32	5.21	3.27	1.93
unified-serial	37.10	19.65	12.46	2.98
iscsi1	29.15	27.12	14.08	2.07
iscsi10	178.22	59.63	31.29	5.70

Table 4.3 MOPED results. The last column shows speedup of FWPDS over MOPED. The programs were provided by S. Schwoon, and are not yet publically available.

Incremental Analysis

We also measure the advantage of doing an incremental analysis for BTRACE. Similar to the experiments performed in [22], we delete a procedure from a program, solve GPS, then reinsert the procedure and look at the time that it takes to solve GPS incrementally. We compare this time with the time that it takes to compute the solution from scratch. We repeated this for all procedures in a given program, and discarded those runs that did not affect at least one other procedure. The results are shown in Tab. 4.4, which shows an average speed up by a factor of 6.5.

Prog	Procs	#Recomputed	Incremental (sec)	Scratch (sec)	Improvement
less	359	91	1.66	8.6	5.18
mc	676	70	0.41	5.2	12.68
uucp	139	36	2.00	3.3	1.65

Table 4.4 Results for incremental analysis for BTRACE. The third column gives the average number of procedures for which the solution had to be recomputed. The fourth and fifth columns report the time taken by the incremental approach and by recomputation from scratch (using FWPDS), respectively.

4.4 Related Work

The basic strategy of using a regular expression to describe a set of paths has been used previously for dataflow analysis of single-procedure programs [91]. The only work that we are aware of that uses this technique for multi-procedure programs is by Ramalingam [79]. However, there the regular expressions were used for a particular analysis (namely, execution-frequency analysis) and the technique was motivated by the special requirements of execution-frequency analysis when creating procedure summaries, rather than efficiency.

There has been other work on improving over the chaotic-iteration strategy, but these have mostly been restricted to single-procedure programs. The work on node-listing algorithms [46] and Bourdoncle's weak topological ordering (wto) [13] assign a priority to each node of a graph such that nodes with lower priority in the worklist must be processed before nodes with higher priority. In the tool CS/x86, G. Balakrishnan extended Bourdoncle's technique to interprocedural analysis using a two-part priority scheme: one part was the wto priority; the other part was based on the call graph [2].

The focus of our work has been on addressing interprocedural analysis. Our techniques apply to any problem that can be encoded as a WPDS, and showed how various enhancements (incremental computation of regular expressions, computing lazily, etc.) contribute to creating a faster analysis. At the intraprocedural level, we chose to make use of Tarjan's path-expression algorithm instead of the other techniques mentioned above. This was because we were able to leverage the compactness of the regular-expression representation at the interprocedural level as well (by computing them incrementally, lazily, etc.). It would be interesting to explore how node-listing algorithms and Bourdoncle's technique could be used interprocedurally.

There has been a host of previous work on incremental program analysis as well as on interprocedural automata-based analysis [22]. The incremental algorithm we have presented is similar to the algorithm in [22], but generalizes it to WPDSs and is thus applicable in domains other than finite-state property verification. A key difference with their algorithm

is that they explore the property automaton on-the-fly as the program is explored. Encoding the property automaton into a WPDS (Section 2.4.3) requires the whole automaton before the program is explored. While such an encoding has the benefit of being amenable to symbolic approaches, it can be disadvantageous when the property automaton is large but only a small part of the property space is relevant for the program.

Chapter 5

Error Projection

Abstraction refinement has been shown to be useful both for finding bugs and for establishing properties of programs (Section 1.1.1). This technique has been implemented in a number of verification tools, including SLAM [4], BLAST [37], and MAGIC [18]. In this chapter, we show how to improve the abstraction-refinement process by making maximum possible use of a given abstraction before moving to more refined abstractions.

We accomplish this by computing *error projections* and *annotated error projections*. An error projection is the set of program nodes N such that for each node $n \in N$, there exists an error path that starts from the entry point of the program and passes through n. By definition, an error projection describes all of the nodes that are members of paths that lead to a specified error in the model, and no more. This allows an automated verification tool (or a human debugging code manually) to focus their efforts on only the nodes in the error projection: every node not in the error projection does not contribute to the (apparent) error (with respect to the property being verified). Tools such as SLAM only need to refine the part of the program that is inside the projection.

Annotated error projections are an extension of error projections. An *annotated error projection* adds two annotations to each node n in the error projection: 1) A counterexample (i.e., a path that fails) that passes through n; 2) a set of data values (memory-configuration descriptors) that describes the conditions necessary at n for the program to fail. The goal is to give back to the user—either an automated tool or human debugger—more of the information discovered during the verification process.

From a theoretical standpoint, an error projection solves a combination of forward and backward analyses. The forward analysis computes the set of program states S_{fwd} that are reachable from program entry; the backward analysis computes the set of states S_{bck} that can reach an error at certain pre-specified nodes. Under a sound abstraction of the program, each of these sets provides a strong guarantee: only states in S_{fwd} can ever arise in the program, and only states in S_{bck} can ever lead to error. Error projections ask the natural question of combining these guarantees to compute the set of states $S_{\text{err}} = S_{\text{fwd}} \cap S_{\text{bck}}$ containing all states that can both arise during program execution, and lead to error. In this sense, an error projection is making maximum use of the given abstraction—by computing the smallest envelope of states that may contribute to program failure.

Computation of this intersection turns out to be non-trivial because the two sets S_{fwd} and S_{bck} may be infinite. In Section 5.2 and Section 5.3, we show how to compute this set efficiently and precisely for WPDSs. The techniques that we use are general, and apart from the application of finding error projections, we discuss additional applications in Section 5.5.

The contributions of the work presented in this chapter can be summarized as follows:

- We define the notions of error projection and annotated error projection. These projections divide the program into a correct and an incorrect part such that further analysis need only be carried out on the incorrect part.

- We give a novel combination of forward and backward analyses for multi-procedural programs using weighted automata and use it for computing (annotated) error projections (Section 5.2 and Section 5.3). We also show that our algorithms can be used for solving various other problems in program verification (Section 5.5).

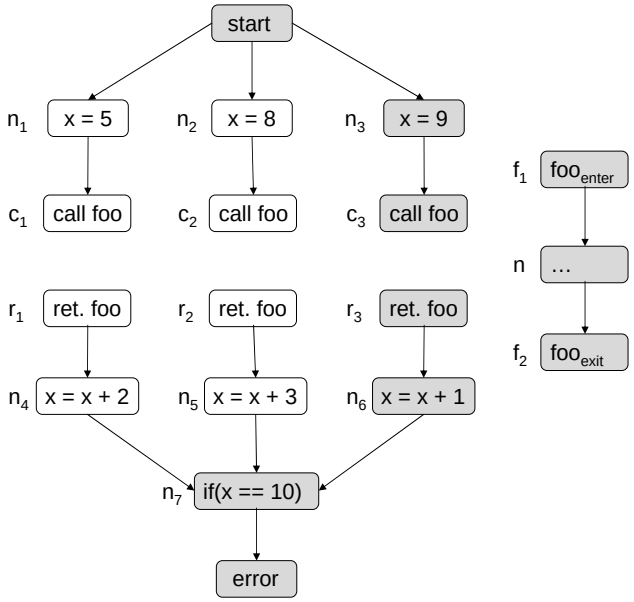
- Our experiments show that we can efficiently compute error projections (Section 5.4).

The remainder of this chapter is organized as follows: Section 5.1 motivates the difficulty in computing (annotated) error projections and illustrates their utility. Section 5.2 and Section 5.3 give the algorithms for computing error projections and annotated error

projections, respectively. Section 5.4 presents our initial experiments. Section 5.5 covers other applications of our algorithms. Section 5.6 discusses related work.

5.1 Examples

Consider the program shown in Fig. 5.1. Here `x` is a global unsigned integer variable, and assume that procedure `foo` does not change the value of `x`. Also assume that the program abstraction is a Boolean abstraction in which integers (only `x` in this case) are modeled using 8 bits, i.e., the value of `x` can be between 0 and 255 with saturated arithmetic. This type of abstraction is used by MOPED [85], and happens to be a precise abstraction for this example.



(a)

(1)	$\langle p, \text{start}\rangle \hookrightarrow \langle p, n_1\rangle$	id
(2)	$\langle p, n_1\rangle \hookrightarrow \langle p, c_1\rangle$	$\{(\_, 5)\}$
(3)	$\langle p, c_1\rangle \hookrightarrow \langle p, f_1\ r_1\rangle$	id
(4)	$\langle p, r_1\rangle \hookrightarrow \langle p, n_4\rangle$	id
(5)	$\langle p, n_4\rangle \hookrightarrow \langle p, n_7\rangle$	$\{(i, i+2)\}$
(6)	$\langle p, \text{start}\rangle \hookrightarrow \langle p, n_2\rangle$	id
(7)	$\langle p, n_2\rangle \hookrightarrow \langle p, c_2\rangle$	$\{(\_, 8)\}$
(8)	$\langle p, c_2\rangle \hookrightarrow \langle p, f_1\ r_2\rangle$	id
(9)	$\langle p, r_2\rangle \hookrightarrow \langle p, n_5\rangle$	id
(10)	$\langle p, n_5\rangle \hookrightarrow \langle p, n_7\rangle$	$\{(i, i+3)\}$
(11)	$\langle p, \text{start}\rangle \hookrightarrow \langle p, n_3\rangle$	id
(12)	$\langle p, n_3\rangle \hookrightarrow \langle p, c_3\rangle$	$\{(\_, 9)\}$
(13)	$\langle p, c_3\rangle \hookrightarrow \langle p, f_1\ r_3\rangle$	id
(14)	$\langle p, r_3\rangle \hookrightarrow \langle p, n_6\rangle$	id
(15)	$\langle p, n_6\rangle \hookrightarrow \langle p, n_7\rangle$	$\{(i, i+1)\}$
(16)	$\langle p, n_7\rangle \hookrightarrow \langle p, \text{error}\rangle$	$\{(10, 10)\}$
(17)	$\langle p, f_1\rangle \hookrightarrow \langle p, n\rangle$	id
(18)	$\langle p, n\rangle \hookrightarrow \langle p, f_2\rangle$	id
(19)	$\langle p, f_2\rangle \hookrightarrow \langle p, \varepsilon\rangle$	id

(b)

Figure 5.1 (a) An example program and (b) its corresponding WPDS. Weights, shown in the last column, are explained in Section 5.2.

The program has an error if node `error` is reached. The two paths on the left that set the value of `x` to 5 or 8 are correct paths, and the one on the right, which sets the value of `x`

to 9, goes to `error`. The error projection is shaded in the figure. An error projection need not be restricted to a single trace (e.g., if `foo` had multiple paths then the error projection would include multiple traces). An annotated error projection will additionally tell us that the value of `x` at node `n` inside `foo` has to be 9 on an error path passing through this node. Note that the value of `x` can be 5 or 8 on other paths that pass through `n`, but they do not lead to the error node.

It is non-trivial to deduce these facts about the value of `x` at node `n`. An interprocedural forward analysis starting from `start` will show that the value of `x` at node `n` is in the set $\{5, 8, 9\}$. A backward interprocedural analysis starting from `error` concludes that the value of `x` at `n` has to be in the set $\{7, 8, 9\}$ in order to reach `error`. Intersecting the sets obtained from forward and backward analysis only gives an over-approximation of the annotated error projection values. In this case, the intersection is $\{8, 9\}$, but `x` can never be 8 on a path leading to `error`. The over-approximation occurs because, in the forward analysis, the value of `x` is 8 only when the call to `foo` occurs at call site c_2, but in the backward analysis, a path that reaches `n` with `x` $= 8$ and goes to `error` must have had the call to `foo` from call site c_1. This mismatch in the calling context leads to the observed over-approximation.

Such a complication also occurs while computing non-annotated error projections: to see this, assume that the edge leading to node `n` is predicated by the condition `if(x!=9)`. Then, node `n` can be reached from `start`, and there is a path starting at `n` that leads to `error`, but both of these cannot occur together.

Formally, a node is in the error projection if and only if the associated value set computed for the annotated projection is non-empty. In this sense, computing an error projection is a special case of computing the annotated version. However, we still discuss error projections separately because (*i*) computing them is easier, as we see later (computing annotations requires one extra trick), and (*ii*) they can very easily be cannibalized by existing tools such as SLAM in their abstraction-refinement phase: when an abstraction needs to be refined, only the portion inside the error projection needs to be rechecked. We illustrate this point in more detail in the next example.

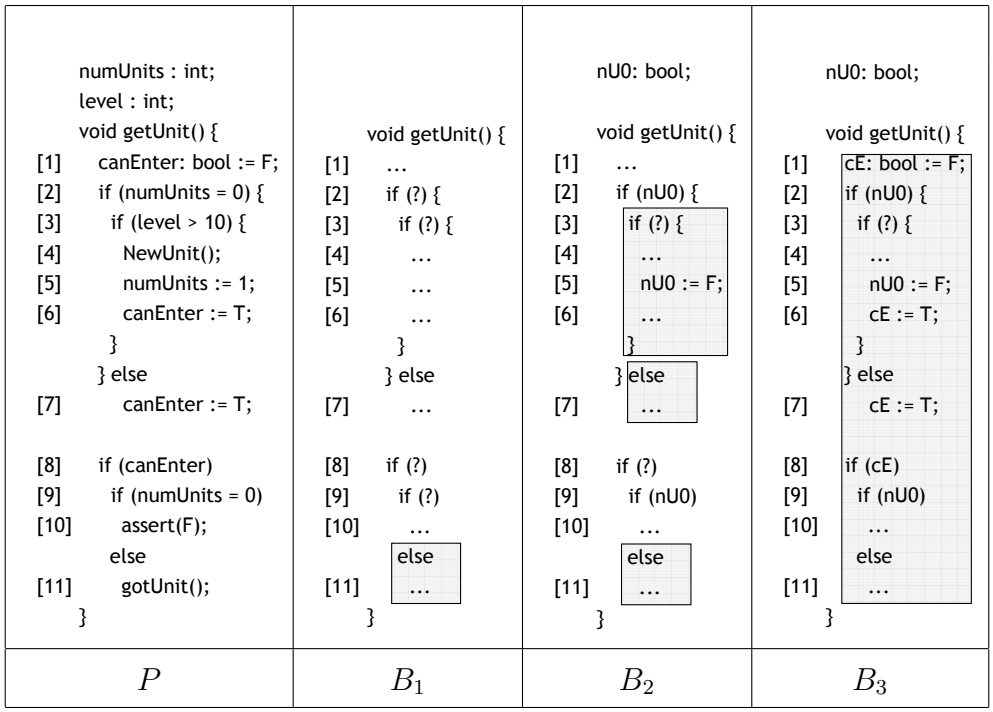


Figure 5.2 An example program P and its abstractions as Boolean programs. The "..." represents a "skip" or a no-op. The part outside the error projection is shaded in each case.

Fig. 5.2 shows an example program and several abstractions that SLAM might produce. This example is given in [7] to illustrate the SLAM refinement process. SLAM uses predicate abstraction to create Boolean programs as described earlier in Section 1.1.1. We will show the utility of error projections for abstraction refinement using this example.

First, recall the SLAM refinement process. In Fig. 5.2, the property of interest is the assertion on line 10. We want to verify that line 10 is never reached ("`assert`(F)" always triggers an assertion violation). The first abstraction B_1 is created without any predicates. It only reflects the control structure of P. Reachability analysis on B_1 (assuming `getUnit` is program entry) shows that the assertion is reachable. This results in a counterexample, whose subsequent analysis reveals that the predicate {`numUnits = 0`} is important. Program B_2 tracks that predicate using variable `nU0`. Reachability analysis on B_2 reveals that the assertion is still reachable. Now predicate {`canEnter = T`} is added, to produce B_3, which

tracks the predicate's value using variable `cE`. Reachability analysis on B_3 reveals that the assertion is not reachable, hence it is not reachable in P.

The advantage of using error projections is that the whole program need not be abstracted when a new predicate is added. Analysis on B_1 and B_2 fails to prove that the whole program is correct, but error projections may reveal that at least some part of the program is correct. The parts outside the error projections (and hence correct) are shaded in the figure. Error projection on B_1 shows that line 11 cannot contribute to the bug, and need not be considered further. Therefore, when constructing B_2, we need not abstract that statement with the new predicate. Error projection on B_2 further reveals that lines 3 to 6 and line 7 do not contribute to the bug (the empty else branch to the conditional at line 3 still can). Thus, when B_3 is constructed, this part need not be abstracted with the new predicate. B_3, with the shaded region of B_2 excluded, reduces to a very simple program, resulting in reduced effort for its construction and analysis.

Annotated error projections can further reduce the analysis cost. Suppose there was some code between lines 1 and 2, possibly relevant to proving the program to be correct, that does not modify `numUnits`. After constructing B_2, the annotated error projection would tell us that in this region of code, `nU0` can be assumed to be *true*, because otherwise the assertion cannot be reached. This might save half of the theorem prover calls needed to abstract that region of code when using multiple predicates (because every predicate whose value is not fixed doubles the cost of abstracting program statements).

While this example did not require an interprocedural analysis, placing any piece of code inside a procedure would necessitate its use. We show how to compute error projections when WPDSs or EWPDSs are used as the model of a program. Because Boolean programs can be encoded using EWPDSs, our techniques would be able to find the error projections shown in Fig. 5.2.

Standard interprocedural analyses do not say anything about calling contexts associated with different reachable values of variables. As we saw earlier, a mismatch in the calling context can lead to an over-approximation in the error projection. Because of the need

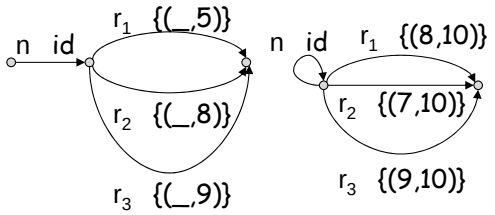


Figure 5.3 Parts of the *poststar* and *prestar* automaton, respectively.

for reasoning about the calling contexts, the automata-based reachability algorithms for (E)WPDSs offer a distinct advantage over other algorithms. The next section shows how to combine the automata obtained from backward and forward reachability analysis on WPDSs to compute (annotated) error projections. This technique is generalized for EWPDSs in Section 5.2.1.

5.2 Computing an Error Projection

As a running example, we use the WPDS of Fig. 5.1(b) that models the program shown in Fig. 5.1(a). This WPDS uses a relational weight domain over the set $V = \{0, 1, \cdots, 255\}$, corresponding to an 8-bit encoding for the values of variable `x` (as explained in Section 5.1). The weight $\{(\_, 5)\}$ is shorthand for the set $\{(i, 5) \mid i \in V\}$; $\{(i, i+1)\}$ stands for $\{(i, i+1) \mid i \in V\}$ (with saturated arithmetic); and *id* stands for the identity relation on V.

Let $\mathcal{A}_S$ and $\mathcal{A}_T$ be (unweighted) automata that accept the sets S and T, respectively. Recall that $\text{IJOP}(S, T) = \mathit{poststar}(\mathcal{A}_S)(T) = \mathit{prestar}(\mathcal{A}_T)(S)$. For the program shown in Fig. 5.1, parts of the automata produced by $\mathit{poststar}(\{\texttt{start}\})$ and $\mathit{prestar}(\texttt{error}\ \Gamma^*)$ are shown in Fig. 5.3 (only the part important for node `n` is shown). Using these, we get $\text{IJOP}(\{\texttt{start}\}, \texttt{n}\ \Gamma^*) = \{(\_, 5), (\_, 8), (\_, 9)\}$ and $\text{IJOP}(\texttt{n}\ \Gamma^*, \texttt{error}\ \Gamma^*) = \{(7, 10), (8, 10), (9, 10)\}$. Here, $(\gamma\ \Gamma^*)$ stands for the set $\{\gamma\ c \mid c \in \Gamma^*\}$.

We now define an error projection using WPDSs as our model of programs. Usually, a WPDS created from a program has a single PDS state. Even when this is not the case, the states can be pushed inside the weights to get a single-state WPDS. We use this to simplify the discussion: PDS configurations are just represented as stacks (Γ^*).

Also, we concern ourselves with assertion checking. We assume that we are given a target set of control configurations T such that the program model exhibits an error only if it can reach a configuration in that set. One way of accomplishing this is to convert every assertion of the form "`assert(`$\mathcal{E}$`)`" into a condition "`if(!`$\mathcal{E}$`) then goto error`" (assuming $!\mathcal{E}$ is expressible under the current abstraction), and instantiate T to be the set of configurations (`error` Γ^*). We also assume that the weight abstraction has been constructed such that a path σ in the PDS is *infeasible* if and only if its weight $v(\sigma)$ is $\overline{0}$. Therefore, under this model, the program has an error only when it can reach a configuration in T with a path of non-$\overline{0}$ weight.

Definition 5.2.1. *Given S, the set of starting configurations of the program, and a target set of configurations T, a program node $\gamma \in \Gamma$ is in the* ***error projection*** *$EP(S,T)$ if and only if there exists a path $\sigma = \sigma_1\sigma_2$ such that $v(\sigma) \neq \overline{0}$ and $s \Rightarrow^{\sigma_1} c \Rightarrow^{\sigma_2} t$ for some $s \in S, c \in \gamma\Gamma^*, t \in T$.*

We calculate the error projection by computing a constrained form of the join-over-all-paths value, which we call a weighted chopping query.

Definition 5.2.2. *Given regular sets of configurations S (source), T (target), and C (chop); a* ***weighted chopping query*** *is to compute the following weight:*

$$WC(S,C,T) = \bigoplus\{v(\sigma_1\sigma_2) \mid s \Rightarrow^{\sigma_1} c \Rightarrow^{\sigma_2} t, s \in S, c \in C, t \in T\}$$

It is easy to see that $\gamma \in \mathrm{EP}(S,T)$ if and only if $\mathrm{WC}(S, \gamma\,\Gamma^*, T) \neq \overline{0}$. We now show how to solve these queries. First, note that $\mathrm{WC}(S,C,T) \neq \mathrm{IJOP}(S,C) \otimes \mathrm{IJOP}(C,T)$. For example, in Fig. 5.1, if `foo` was not called from $\mathtt{c}_3$, and $S = \{\mathtt{start}\}, T = (\mathtt{error}\ \Gamma^*), C = (\mathtt{n}\ \Gamma^*)$ then $\mathrm{IJOP}(S,C) = \{(\_,5),(\_,8)\}$ and $\mathrm{IJOP}(C,T) = \{(7,10),(8,10)\}$, and their extend is non-empty, whereas $\mathrm{WC}(S,C,T) = \emptyset$. This is exactly the problem mentioned in Section 5.1.

A first attempt at solving weighted chopping is to use the identity $\mathrm{WC}(S,C,T) = \bigoplus\{\mathrm{IJOP}(S,c) \otimes \mathrm{IJOP}(c,T) \mid c \in C\}$. However, this only works when C is a finite set of configurations, which is not the case if we want to compute an error projection. We can

solve this problem using the automata-theoretic constructions described in the previous section. Let $\mathcal{A}_S$ be an unweighted automaton that represents the set S, and similarly for $\mathcal{A}_C$ and $\mathcal{A}_T$. The following two algorithms, given in different columns, are valid ways of solving a weighted chopping query.

Algorithm Double-*poststar*	**Algorithm Double-*prestar***
1. $\mathcal{A}_1 = \mathit{poststar}(\mathcal{A}_S)$	1. $\mathcal{A}_1 = \mathit{prestar}(\mathcal{A}_T)$
2. $\mathcal{A}_2 = (\mathcal{A}_1 \cap \mathcal{A}_C)$	2. $\mathcal{A}_2 = (\mathcal{A}_1 \cap \mathcal{A}_C)$
3. $\mathcal{A}_3 = \mathit{poststar}(\mathcal{A}_2)$	3. $\mathcal{A}_3 = \mathit{prestar}(\mathcal{A}_2)$
4. $\mathcal{A}_4 = \mathcal{A}_3 \cap \mathcal{A}_T$	4. $\mathcal{A}_4 = \mathcal{A}_3 \cap \mathcal{A}_S$
5. $\mathrm{WC}(S, C, T) = \mathit{path\_summary}(\mathcal{A}_4)$	5. $\mathrm{WC}(S, C, T) = \mathit{path\_summary}(\mathcal{A}_4)$

Proof. We prove correctness of the double-*poststar* algorithm. A proof for double-*prestar* is similar. From the properties of *poststar* (Lem. 2.4.6), we know that:

$$
\begin{array}{lll}
 & \mathcal{A}_2(c) & = \begin{cases} \overline{0} & \text{if } c \notin C \\ \bigoplus\{v(\sigma_1) \mid s \Rightarrow^{\sigma_1} c, s \in S\} & \text{if } c \in C \end{cases} \\
\Rightarrow & \mathcal{A}_3(t) & = \bigoplus\{\mathcal{A}_2(c) \otimes v(\sigma_2) \mid c \Rightarrow^{\sigma_2} t\} \\
 & & = \bigoplus\{\bigoplus\{v(\sigma_1) \otimes v(\sigma_2) \mid s \Rightarrow^{\sigma_1} c, s \in S\} \mid c \in C, c \Rightarrow^{\sigma_2} t\} \\
 & & = \bigoplus\{v(\sigma_1) \otimes v(\sigma_2) \mid s \Rightarrow^{\sigma_1} c \Rightarrow^{\sigma_2} t, s \in S, c \in C\} \\
 & & = \bigoplus\{v(\sigma_1\sigma_2) \mid s \Rightarrow^{\sigma_1} c \Rightarrow^{\sigma_2} t, s \in S, c \in C\} \\
\Rightarrow & \mathit{path\_summary}(\mathcal{A}_4) & = \mathcal{A}_3(T) \\
 & & = \bigoplus\{v(\sigma_1\sigma_2) \mid s \Rightarrow^{\sigma_1} c \Rightarrow^{\sigma_2} t, s \in S, c \in C, t \in T\} \\
 & & = \mathrm{WC}(S, C, T)
\end{array}
$$

□

The running time of these algorithms is proportional to the size of $\mathcal{A}_C$, not the size of C.

An error projection is computed by solving a separate weighted chopping query for each node γ in the program. This means that the source set S and the target set T remain fixed, but the chop set C keeps changing. Unfortunately, the two algorithms given above have a major shortcoming: only their first steps can be carried over from one chopping query to the

next; the rest of the steps have to be recomputed for each node γ. As shown in Section 5.4, this approach is very slow, and the algorithm discussed next is about 3 orders of magnitude faster.

To derive a better algorithm for weighted chopping that is more suited for computing error projections, let us first look at the unweighted case (i.e., the weighted case where the weight domain just contains the weights $\overline{0}$ and $\overline{1}$). Then $\mathrm{WC}(S, C, T) = \overline{1}$ if and only if $(post^*(S) \cap pre^*(T)) \cap C \neq \emptyset$. This procedure just requires a single intersection operation for different chop sets. Computation of both $post^*(S)$ and $pre^*(T)$ have to be done just once. We generalize this approach to the weighted case.

First, we need to define what we mean by intersecting weighted automata. Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two weighted automata. Define their *intersection* $\mathcal{A}_1 \lhd \mathcal{A}_2$ to be a function from configurations to weights, which we later compute in the form of a weighted automaton, such that $(\mathcal{A}_1 \lhd \mathcal{A}_2)(c) = \mathcal{A}_1(c) \otimes \mathcal{A}_2(c)$.[1] Define $(\mathcal{A}_1 \lhd \mathcal{A}_2)(C) = \bigoplus\{(\mathcal{A}_1 \lhd \mathcal{A}_2)(c) \mid c \in C\}$, as before. Based on this definition, if $\mathcal{A}_{post^*} = poststar(\mathcal{A}_S)$ and $\mathcal{A}_{pre^*} = prestar(\mathcal{A}_T)$, then $\mathrm{WC}(S, C, T) = (\mathcal{A}_{post^*} \lhd \mathcal{A}_{pre^*})(C)$.

Let us give some intuition into why intersecting weighted automata is hard. For $\mathcal{A}_1$ and $\mathcal{A}_2$ as above, the intersection is defined to read off the weight from $\mathcal{A}_1$ first and then extend it with the weight from $\mathcal{A}_2$. A naive approach would be to construct a weighted automaton $\mathcal{A}_{12}$ as the concatenation of $\mathcal{A}_1$ and $\mathcal{A}_2$ (with epsilon transitions from the final states of $\mathcal{A}_1$ to the initial states of $\mathcal{A}_2$) and let $(\mathcal{A}_1 \lhd \mathcal{A}_2)(c) = \mathcal{A}_{12}(c\ c)$. However, computing $(\mathcal{A}_1 \lhd \mathcal{A}_2)(C)$ for a regular set C requires computing join-over-all-paths in $\mathcal{A}_{12}$ over the set of paths that accept the language $\{(c\ c) \mid c \in C\}$ because the *same* path (i.e., c) must be followed in both $\mathcal{A}_1$ and $\mathcal{A}_2$. This language is neither regular nor context-free, and we do not know of any method that computes join-over-all-paths over a non-context-free set of paths.

The trick here is to recognize that weighted automata have a direction in which weights are read off. We need to intersect $\mathcal{A}_{post^*}$ with $\mathcal{A}_{pre^*}$, where $\mathcal{A}_{post^*}$ is a backward automaton

[1] Note that the operator $\lhd$ is not commutative in general, but we still call it *intersection* because the construction of $\mathcal{A}_1 \lhd \mathcal{A}_2$ resembles the one for intersection of unweighted automata.

and $\mathcal{A}_{pre^*}$ is a forward automaton. If we concatenate these together but reverse the second one (i.e., reverse all its transitions and interchange its initial and final states), then we get a purely backward weighted automaton and we only need to solve for join-over-all-paths over the language $\{(c\ c^R) \mid c \in C\}$ where c^R is c written in the reverse order. This language can be defined using a linear context-free grammar with production rules of the form "$X \to \gamma Y \gamma$", where X and Y are non-terminals. The following section uses this intuition to derive an algorithm for intersecting two weighted automata.

Intersecting Weighted Automata

Let $\mathcal{A}_b = (Q_b, \Gamma, \to_b, P, F_b)$ be a backward weighted automaton and $\mathcal{A}_f = (Q_f, \Gamma, \to_f, P, F_f)$ be a forward weighted automaton. We proceed with the standard automata-intersection algorithm: Construct a new automaton $\mathcal{A}_{bf} = (Q_b \times Q_f, \Gamma, \to, P, F_b \times F_f)$, where we identify the state $(p, p), p \in P$ with p, i.e., the P-states of $\mathcal{A}_{bf}$ are states of the form $(p, p), p \in P$. The transitions of this automaton are computed by matching on stack symbols. If $t_b = (q_1, \gamma, q_2)$ is a transition in $\mathcal{A}_b$ with weight w_b and $t_f = (q_3, \gamma, q_4)$ is a transition in $\mathcal{A}_f$ with weight w_f, then add transition $t_{bf} = ((q_1, q_3), \gamma, (q_2, q_4))$ to $\mathcal{A}_{bf}$ with weight $\lambda z.(w_b \otimes z \otimes w_f)$. We call this type of weight a *functional weight* and use the capital letter W (possibly subscripted) to distinguish them from normal weights. Functional weights are special functions on weights: given a weight w and a functional weight $W = \lambda z.(w_1 \otimes z \otimes w_2)$, $W(w) = (w_1 \otimes w \otimes w_2)$. The automaton $\mathcal{A}_{bf}$ is called a *functional automaton.*

We define extend on functional weights as reversed function composition. That is, if $W_1 = \lambda z.(w_1 \otimes z \otimes w_2)$ and $W_2 = \lambda z.(w_3 \otimes z \otimes w_4)$, then $W_1 \otimes W_2 = W_2 \circ W_1 = \lambda z.((w_3 \otimes w_1) \otimes z \otimes (w_2 \otimes w_4))$, and is thus also a functional weight. However, the combine operator, defined as $W_1 \oplus W_2 = \lambda z.(W_1(z) \oplus W_2(z))$, does not preserve the form of functional weights. Hence, functional weights do not form a semiring. We now show that this is not a handicap, and we can still compute $\mathcal{A}_b \lhd \mathcal{A}_f$ as required.

Because $\mathcal{A}_{bf}$ is obtained from an intersection operation, every path in it that is of the form $(q_1, q_2) \xrightarrow{c}{}^* (q_3, q_4)$ is in one-to-one correspondence with paths $q_1 \xrightarrow{c}{}^* q_3$ in $\mathcal{A}_b$ and

$q_2 \xrightarrow{c}{}^* q_4$ in $\mathcal{A}_f$. Using this fact, we get that the weight of a path in $\mathcal{A}_{bf}$ will be a function of the form $\lambda z.(w_b \otimes z \otimes w_f)$, where w_b and w_f are the weights of the corresponding paths in $\mathcal{A}_b$ and $\mathcal{A}_f$, respectively. In this sense, $\mathcal{A}_{bf}$ is constructed based on the intuition given in the previous section: the functional weights resemble grammar productions "$X \to \gamma Y \gamma$" for the language $\{(c\ c^R)\}$ with weights replacing the two occurrences of γ, and their composition resembles the derivation of a string in the language. (Note that in "$X \to \gamma Y \gamma$", the first γ is a letter in c, whereas the second γ is a letter in c^R. In general, the letters will be given different weights in $\mathcal{A}_b$ and $\mathcal{A}_f$.)

Formally, for a configuration c and a weighted automaton $\mathcal{A}$, define the predicate $accPath(\mathcal{A}, c, w)$ to be true if there is an accepting path in $\mathcal{A}$ for c that has weight w, and false otherwise (note that we only need the extend operation to compute the weight of a path). Similarly, $accPath(\mathcal{A}, C, w)$ is true iff $accPath(\mathcal{A}, c, w)$ is true for some $c \in C$. Then we have:

$$\begin{aligned}
(\mathcal{A}_b \lhd \mathcal{A}_f)(c) &= \mathcal{A}_b(c) \otimes \mathcal{A}_f(c) \\
&= \bigoplus\{w_b \otimes w_f \mid accPath(\mathcal{A}_b, c, w_b), accPath(\mathcal{A}_f, c, w_f)\} \\
&= \bigoplus\{w_b \otimes w_f \mid accPath(\mathcal{A}_{bf}, c, \lambda z.(w_b \otimes z \otimes w_f))\} \\
&= \bigoplus\{\lambda z.(w_b \otimes z \otimes w_f)(\overline{1}) \mid accPath(\mathcal{A}_{bf}, c, \lambda z.(w_b \otimes z \otimes w_f))\} \\
&= \bigoplus\{W(\overline{1}) \mid accPath(\mathcal{A}_{bf}, c, W)\}
\end{aligned}$$

Similarly, we have $(\mathcal{A}_b \lhd \mathcal{A}_f)(C) = \bigoplus\{W(\overline{1}) \mid accPath(\mathcal{A}_{bf}, C, W)\} = \bigoplus\{W(\overline{1}) \mid accPath(\mathcal{A}_{bf} \cap \mathcal{A}_C, \Gamma^*, W)\}$, where $\mathcal{A}_C$ is an unweighted automaton that accepts the set C, and this can be obtained using a procedure similar to *path_summary*. The advantage of the way we have defined $\mathcal{A}_{bf}$ is that we can intersect it with $\mathcal{A}_C$ (via ordinary intersection) and then run *path_summary* over it, as we show next.

Functional weights distribute over (ordinary) weights, i.e., $W(w_1 \oplus w_2) = W(w_1) \oplus W(w_2)$. Thus, $path\_summary(\mathcal{A}_{bf})$ can be obtained merely by solving an intraprocedural join-over-all-paths over distributive transformers starting with the weight $\overline{1}$, which is completely standard: Initialize $l(q) = \overline{1}$ for initial states, and set $l(q) = \overline{0}$ for other states. Then, until a

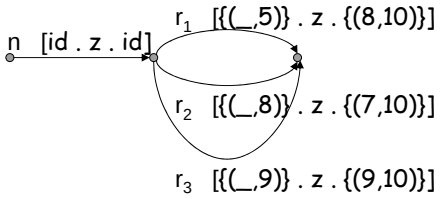


Figure 5.4 Functional automaton obtained after intersecting the automata of Fig. 5.3.

fixpoint is reached, for a transition (q, γ, q') with weight W, update the weight on state q' by $l(q') := l(q') \oplus W(l(q))$. Then *path_summary*$(\mathcal{A}_{bf})$ is the combine of the weights on the final states. Termination is guaranteed because we still have weights associated with states, and functional weights are monotonic. Because of the properties satisfied by $\mathcal{A}_{bf}$, we use $\mathcal{A}_{bf}$ as a representation for $(\mathcal{A}_b \lhd \mathcal{A}_f)$.

This allows us to solve $\mathrm{WC}(S, C, T) = (\mathcal{A}_{post^*} \lhd \mathcal{A}_{pre^*})(C)$. That is, after a preparation step to create $(\mathcal{A}_{post^*} \lhd \mathcal{A}_{pre^*})$, one can solve $\mathrm{WC}(S, C, T)$ for different chop sets C just using intersection with $\mathcal{A}_C$ followed by *path_summary*, as shown above. Fig. 5.4 shows an example. For short, the weight $\lambda z.(w_1 \otimes z \otimes w_2)$ is denoted by $[w_1.z.w_2]$. Note how the weights for different call sites get appropriately paired in the functional automaton.

It should be noted that this technique applies only to the intersection of a forward weighted automaton with a backward one, because in this case we are able to get around the problem of computing join-over-all-paths over a non-context-free set of paths. Algorithms for intersecting two forward or two backward automata will be discussed in Section 6.5.1.

5.2.1 Computing Error Projections for EWPDSs

Computing error projections for EWPDSs is slightly harder than for WPDSs for the following reason: for a rule sequence $\sigma = \sigma_1\sigma_2$, $v(\sigma) \neq v(\sigma_1) \otimes v(\sigma_2)$ because an unbalanced call at the end of σ_1 may match with an unbalanced return in the beginning of σ_2, in which case, a merge function has to be applied. Thus, $\mathrm{WC}(S, \{c\}, T) \neq \mathrm{IJOP}(S, \{c\}) \otimes \mathrm{IJOP}(\{c\}, T)$.

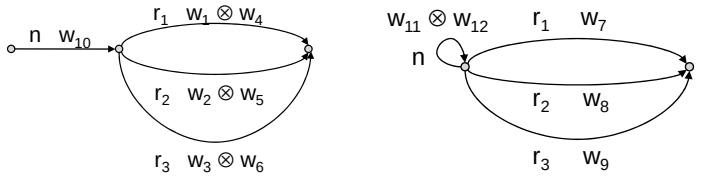
Our solution for computing an error projection for EWPDSs is also based on "intersecting" the weighted automata $\mathcal{A}_{post^*} = \mathit{poststar}(S)$ and $\mathcal{A}_{pre^*} = \mathit{prestar}(T)$. However, it turns

out that, in general, $\mathcal{A}_{post^*}$ does not retain enough information about the unbalanced calls. We restrict the set S to be just the starting configuration of the program, i.e., $S = \{\langle p, e_{\texttt{main}}\rangle\}$ (or any singleton set with a configuration with one stack symbol).

The intersection operation for weighted automata is carried out in the same manner as for WPDSs, but instead of always creating functional weights of the form $\lambda z.(w_1 \otimes z \otimes w_2)$, we may also create weights of the form $\lambda z.(m(w_1, z) \otimes w_2)$ as well, where m is a merge function. We explain our strategy through an example, and then give the algorithm.

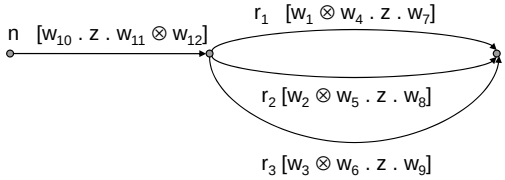
(1) $\langle p, \texttt{start}\rangle \hookrightarrow \langle p, c_1\rangle \quad w_1$
(2) $\langle p, \texttt{start}\rangle \hookrightarrow \langle p, c_2\rangle \quad w_2$
(3) $\langle p, \texttt{start}\rangle \hookrightarrow \langle p, c_3\rangle \quad w_3$
(4) $\langle p, c_1\rangle \hookrightarrow \langle p, f_1\ r_1\rangle \quad w_4$
(5) $\langle p, c_2\rangle \hookrightarrow \langle p, f_1\ r_2\rangle \quad w_5$
(6) $\langle p, c_3\rangle \hookrightarrow \langle p, f_1\ r_3\rangle \quad w_6$
(7) $\langle p, r_1\rangle \hookrightarrow \langle p, \texttt{error}\rangle \quad w_7$
(8) $\langle p, r_2\rangle \hookrightarrow \langle p, \texttt{error}\rangle \quad w_8$
(9) $\langle p, r_3\rangle \hookrightarrow \langle p, \texttt{error}\rangle \quad w_9$
(10) $\langle p, f_1\rangle \hookrightarrow \langle p, n\rangle \quad w_{10}$
(11) $\langle p, n\rangle \hookrightarrow \langle p, f_2\rangle \quad w_{11}$
(12) $\langle p, f_2\rangle \hookrightarrow \langle p, \varepsilon\rangle \quad w_{12}$

(a)

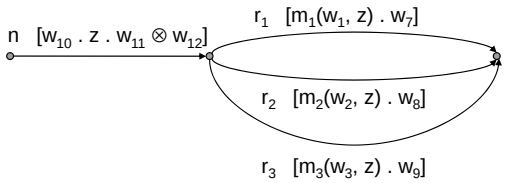


(b)

(c)



(d)



(e)

Figure 5.5 (a) A WPDS. (b),(c) Parts of the *poststar* and *prestar* automata, respectively. (d) Functional automaton obtained after intersecting the automata shown in (b) and (c). (e) Functional automaton for an EWPDS when the call rule at c_i is associated with merge function m_i.

Consider the example shown in Fig. 5.5, which is a reworking of the example shown in Fig. 5.1. Let $S = \{\langle p, \texttt{start}\rangle\}$ and $T = \{\langle p, \texttt{error}\rangle\}$. The weights have been left unspecified so that we can track their contribution to the weights in the functional automaton that is shown in Fig. 5.5(d). Let us call this automaton $\mathcal{A}_{\text{func}}$.

The functional weight $\mathcal{A}_{\text{func}}(\langle p, n\ r_1\rangle)$ applied to $\overline{1}$ equals $(w_1 \otimes w_4 \otimes w_{10} \otimes w_{11} \otimes w_{12} \otimes w_7)$ (which equals $\text{WC}(S, \{\langle p, n\ r_1\rangle\}, T)$). The weight on the transition with stack symbol n summarizes the set of all paths from f_1 to n (weight w_{10}) and n to the exit of the procedure, including the return rule (weight $w_{11} \otimes w_{12}$). Putting these together, the weight $(w_{10} \otimes w_{11} \otimes w_{12})$ is the summary of the procedure starting at f_1. Similarly, the two components of the functional weight on the transition for r_1 are the weights of the paths from `start` to c_1, including the call (weight $w_1 \otimes w_4$), and from r_1 to `error` (weight w_7), respectively. This functional weight summarizes paths from `start` to `error` with a hole, which is filled by the variable z, and represents the summary of a called procedure. Thus, the functional weight on the transition for r_1 must apply a merge function.

Let $\mathcal{W}_e$ be an EWPDS obtained by associating merge function m_i with call rule rule $\langle p, c_i\rangle \hookrightarrow \langle p, f_1\ \ r_i\rangle$ in the WPDS shown in Fig. 5.5(a). For $\mathcal{W}_e$, the weight $\text{WC}(S, \{\langle p, n\ r_1\rangle\}, T)$ is $m_1(w_1, w_{10} \otimes w_{11} \otimes w_{12}) \otimes w_7$. The functional automaton shown in Fig. 5.5(e) computes exactly this weight for $\langle p, n\ r_1\rangle$. Next, we outline the algorithm for constructing this automaton.

Let $\mathcal{A}_{poststar} = \mathit{poststar}(S)$ and $\mathcal{A}_{pre^*} = \mathit{prestar}(T)$. The set of states of $\mathcal{A}_{pre^*}$ is $(P \cup Q_T)$, where Q_T is the set of states of the automaton that represents T. The set of states of $\mathcal{A}_{post^*}$ is $(P \cup Q_{\text{mid}} \cup Q_S)$, where Q_S is the set of states of the automaton that represents S. To distinguish the two occurrences of P in these sets, we label the former as P_T and the latter as P_S. To simplify the discussion, we assume that the weight on a call rule is always $\overline{1}$. (The construction given in Section 3.7 shows how one can convert an EWPDS that does not satisfy this restriction into one that does satisfy it.) The functional automaton is constructed as before, except for the weights on transitions. For each transition $t_1 = (q_1, \gamma, q_2)$ of $\mathcal{A}_{post^*}$

with weight w_1 and transition $t_2 = (q_3, \gamma, q_4)$ of $\mathcal{A}_{pre^*}$ with weight w_2, add a transition to $(\mathcal{A}_{post^*} \lhd \mathcal{A}_{pre^*})$ as follows:

1. If $t_1 \in (Q_{\text{mid}} \times \Gamma \times Q_{\text{mid}})$ or $t_1 \in (Q_{\text{mid}} \times \Gamma \times Q_S)$ then let $q_1 = p'_{\gamma'}$ and $r = \mathit{lookupPushRule}(p', \gamma'\gamma)$. Add transition $((q_1, q_3), \gamma, (q_2, q_4))$ to the functional automaton with weight $\lambda z.(m_r(w_1, z) \otimes w_2)$.

2. In all other cases, add transition $((q_1, q_3), \gamma, (q_2, q_4))$ to the functional automaton with weight $\lambda z.(w_1 \otimes z \otimes w_2)$.

The reader can verify that this algorithm will produce the automaton shown in Fig. 5.5(e), after removing the states that cannot be reached from the initial state, or cannot reach a final state.

A justification of this algorithm is based on the types of rule sequences captured by a transition in $\mathcal{A}_{post^*}$ and $\mathcal{A}_{pre^*}$, which are given in Section 3.7 (Figs. 3.8 and 3.11). We recall the results of that section and show them pictorially in Fig. 5.6(a) and (b). (They have been simplified using the restriction imposed on S.) Fig. 5.6(a) can be read as follows: the weight on a transition $t \in (Q_{\text{mid}} \times \Gamma \times Q_{\text{mid}})$ summarizes the weights of rule sequences derivable from $(\sigma_b R_2)$, i.e., ones that have a balanced sequence followed by a call rule; a transition $t \in P_S \times \Gamma \times Q_S$ summarizes rule sequences derivable from σ_b; and so on. Similarly for Fig. 5.6(b). Taking the intersection of these automata, one gets the automaton shown in Fig. 5.6(c). Let us call this automaton $\mathcal{A}_e$.

The initial states of $\mathcal{A}_e$ are $P_S \times P_T$ and the final states belong to $Q_S \times Q_T$. The states $P_S \times Q_T$ cannot be reached from the initial states, and $Q_S \times P_T$ cannot reach a final state. As a consequence, these states, and their transitions, have not been shown in the figure.

We will show that a merge function needs to be applied if and only if a transition starting from a state in $Q_{\text{mid}} \times P_T$ is taken. (Because these are exactly the transitions created in item 1 of the algorithm outlined above, proving this claim shows that our construction is correct.) Intuitively, our claim holds because a merge function is applied during *poststar* when a state is in Q_{mid} (last case of Fig. 3.4) and is applied during *prestar* when a state is

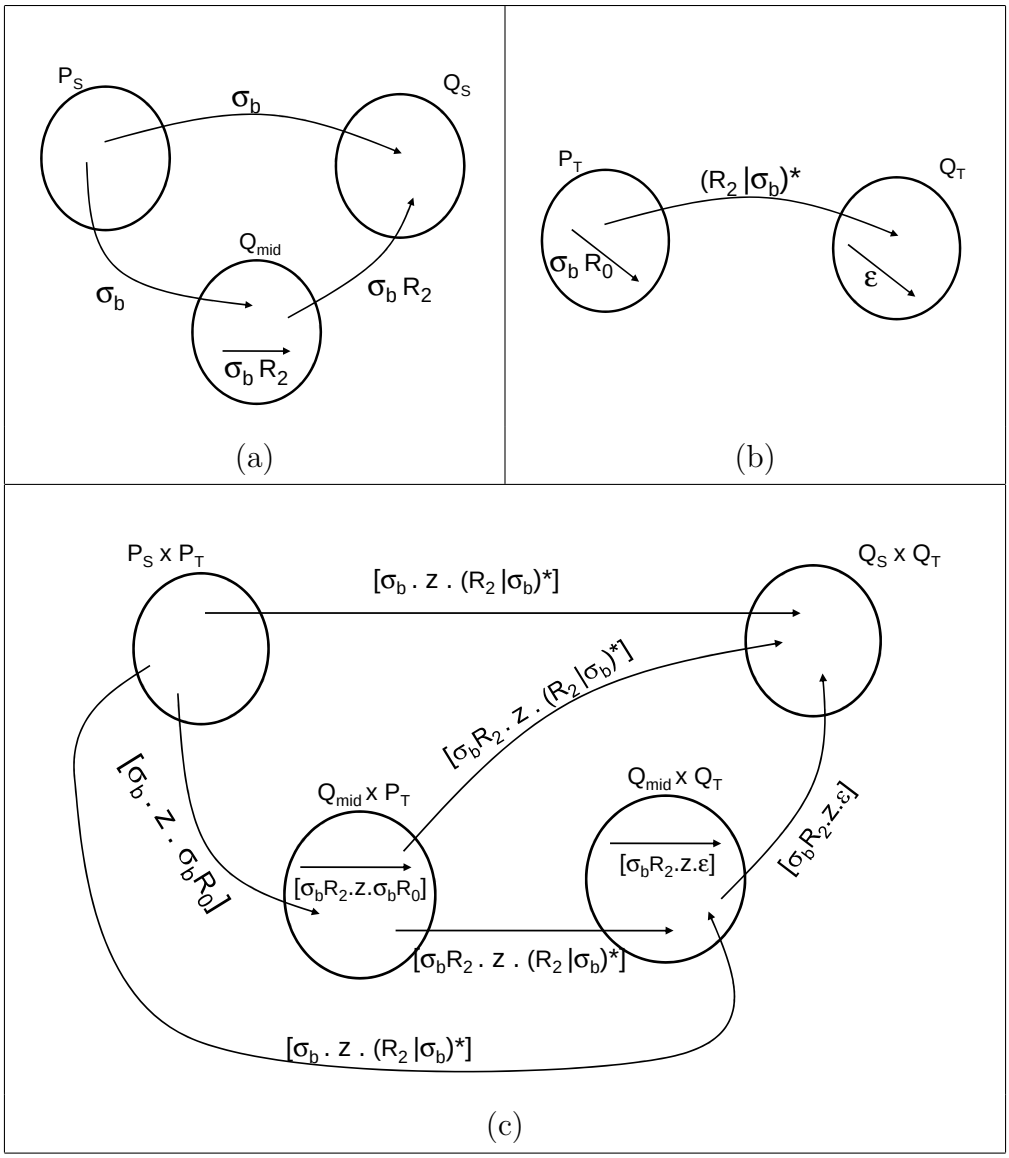


Figure 5.6 (a) Rule sequences for $\mathcal{A}_{post^*}$. (b) Rule sequences for $\mathcal{A}_{pre^*}$. (c) Rule sequences for the functional automaton $\mathcal{A}_{post^*} \triangleleft \mathcal{A}_{pre^*}$.

in P (last case of Fig. 3.3). Thus, in the functional automaton, both conditions have to be satisfied (i.e., a state must lie in $Q_{\text{mid}} \times P_T$) for a merge function to be applied.

The annotations on the transitions of $\mathcal{A}_e$ are like functional weights. Every path in $\mathcal{A}_e$ can be associated with a rule sequence that the path represents, in a manner similar to the way one calculates the weight of a path in a functional automaton. However, instead of

starting with weight $\overline{1}$ (which is done for reading weights out of a functional automaton), one starts with the empty rule sequence ε. For instance, if one takes a transition t_1 from a state in $P_S \times P_T$ to a state in $Q_{\text{mid}} \times P_T$ and then t_2 to another state in $Q_{\text{mid}} \times P_T$, the net rule sequence for $(t_1 t_2)$ is the following:

$$\begin{aligned} & (\lambda z.(\sigma_b R_2\ z\ \sigma_b R_0)\ \ (\lambda z.(\sigma_b\ z\ \sigma_b R_0)\ \ \varepsilon)) \\ = \ & (\lambda z.(\sigma_b R_2\ z\ \sigma_b R_0)\ \ (\sigma_b \sigma_b R_0)) \\ = \ & (\lambda z.(\sigma_b R_2\ z\ \sigma_b R_0)\ \ (\sigma_b R_0)) \\ = \ & (\sigma_b R_2 (\sigma_b R_0) \sigma_b R_0) \\ = \ & (\sigma_b R_0) \end{aligned}$$

Here we have simplified expressions using the grammar of Fig. 3.2: we replace $(\sigma_b\ \sigma_b)$ with σ_b, and $(R_2\ \sigma_b\ R_0)$ with σ_b, because each denote a balanced sequence.

One can prove that for all paths in $\mathcal{A}_e$ that start from an initial state and end in a state in $Q_{\text{mid}} \times P_T$, the rule sequence of that path is $(\sigma_b\ R_0)$, i.e., it has one unbalanced return in the end. When one takes any transition starting in $Q_{\text{mid}} \times P_T$, the rule sequence becomes $(\lambda z.(\sigma_b R_2\ z\ \sigma_b R_0)\ \ (\sigma_b\ R_0))$ or $(\lambda z.(\sigma_b R_2\ z\ (R_2|\sigma_b)^*)\ \ (\sigma_b\ R_0))$. In each case, the leftmost R_2 of the functional weight matches with the R_0 of the incoming rule sequence, and a merge function needs to be applied.

5.3 Computing an Annotated Error Projection

An annotated error projection adds more information to an error projection by associating each node in the error projection with (i) at least one counterexample that goes through that node and (ii) the set of data values that may arise on a path doomed to fail in the future.

5.3.1 Computing Witnesses

Given source set S and target set T, previous work on WPDSs allows the computation of a finite set of paths, called *witnesses*, $\{\sigma_i \mid 1 \leq i \leq n\}$ such that $\oplus_i \{v(\sigma_i)\} = \text{IJOP}(S, T)$

[83]. The same result holds for *path_summary* on weighted as well as functional automata: we can find a finite set of paths in the automaton that justifies the weight returned by *path_summary* (we say that a set of paths justifies a weight w if the combine of their weights is equal to w). We make use of this technology in this section.

Suppose we find that γ is in the error projection. Then, we know that $\mathrm{WC}(S, C, T) \neq \overline{0}$, where $C = \gamma\Gamma^*$. We will find a path from some configuration $s \in S$ to some configuration $t \in T$ that goes through some $c \in C$, with non-$\overline{0}$ weight, in two stages. In the first stage, we find c. In the second stage, we find the path through c.

Let $\mathcal{A}_C$ be the unweighted automaton that accepts the language C, and $\mathcal{A}_\cap = (\mathcal{A}_{post^*} \lhd \mathcal{A}_{pre^*}) \cap \mathcal{A}_C$. Then *path_summary*$(\mathcal{A}_\cap) = \mathrm{WC}(S, C, T) \neq \overline{0}$. Using witness generation, we can find at least one path in $\mathcal{A}_\cap$ whose weight is not $\overline{0}$. A path in this automaton corresponds to a configuration c with $\mathcal{A}_\cap(c) \neq \overline{0}$. This, in turn, implies that $c \in C$ and there is a path in the WPDS from S to T through c with non-$\overline{0}$ weight.

Again, using standard witness generation, we can find a set of witness $\{\sigma_i\}_{1 \leq i \leq n}$ for $\mathcal{A}_{post^*}(c) = \mathrm{IJOP}(S, c)$ and a set of witnesses $\{\rho_j\}_{1 \leq j \leq m}$ for $\mathcal{A}_{pre^*}(c) = \mathrm{IJOP}(c, T)$, respectively. The concatenation of these witnesses $\{\sigma_i\rho_j\}_{1 \leq i \leq n}^{1 \leq j \leq m}$ justifies $\mathrm{IJOP}(S, c) \otimes \mathrm{IJOP}(c, T)$. (The concatenation is a constant-time operation because a witness set is represented using a DAG.) Therefore, one of these witnesses is a path with non-$\overline{0}$ weight and serves as the desired witness for node γ. The same procedure can be repeated for each node in the error projection. Finding witnesses is not a very expensive operation, but it adds a fair amount of overhead to the execution of *poststar* and *prestar* (although their worst-case running times do not change).

One optimization that witnesses allow is that if we obtain σ as a witness for a node γ in the error projection, then for every node γ' such that a configuration $c \in \gamma'\Gamma^*$ occurs in σ, γ' must also be in the error projection. Therefore, while computing an error projection, if we find γ to be in the error projection, then we can find a witness for it and immediately include in the error projection every such γ'.

5.3.2 Computing Data Values

In this section, we discuss algorithms for computing the data values for nodes in an error projection. The technique that we present applies to relational weight domains (Defn. 2.4.13). Note that the value of $\mathrm{WC}(S, C, T)$ does not say anything about the required set of values at C: for Fig. 5.1, $\mathrm{WC}(S, \mathtt{n}\ \Gamma^*, T) = \{(\_, 10)\}$ but the required memory configuration at $\mathtt{n}$ is $\{9\}$.

Let V be a finite set of memory configurations, i.e., an element of V abstracts a collection of valuations of program variables. In terms of dataflow analysis, V is a set of dataflow facts. In terms of Boolean programs, V is the set of valuations of Boolean variables. If the Boolean program is obtained using predicate abstraction, an element of V is a valuation v of all predicates, which represents an abstraction for all program states that satisfy those predicates or their negated form, according to v (see Section 1.1.2).

Let $(D, \oplus, \otimes, \overline{0}, \overline{1})$ be the relational weight domain on V. For weights $w, w_1, w_2 \in D$, define $\mathtt{Rng}(w)$ to be the range of w, $\mathtt{Dom}(w)$ to be the domain of w and $\mathtt{Com}(w_1, w_2) = \mathtt{Rng}(w_1) \cap \mathtt{Dom}(w_2)$. For a node $\gamma \in \mathrm{EP}(S, T)$, we compute the following subset of V: $V_\gamma = \{v \in \mathtt{Com}(v(\sigma_1), v(\sigma_2)) \mid s \Rightarrow^{\sigma_1} c \Rightarrow^{\sigma_2} t, s \in S, c \in \gamma\Gamma^*, t \in T\}$. If $v \in V_\gamma$, then there must be a path in the program model that leads to an error such that the abstract store v arises at node γ.

5.3.2.1 An Explicit Algorithm

First, we show how to check for membership in the set V_γ. Conceptually, we place a bottleneck at node γ, using a special weight, to see if there is a feasible path that can pass through the bottleneck at γ with abstract store v, and then continue on to the error configuration. Let $w_v = \{(v, v)\}$. Note that $v \in \mathtt{Com}(w_1, w_2)$ iff $w_1 \otimes w_v \otimes w_2 \neq \overline{0}$. Let $\mathcal{A}_{post^*} = \mathit{poststar}(\mathcal{A}_S), \mathcal{A}_{pre^*} = \mathit{prestar}(\mathcal{A}_T)$ and $\mathcal{A}_\lhd$ be their intersection. Then $v \in V_\gamma$ iff there is a configuration $c \in \gamma\Gamma^*$ such that $\mathrm{IJOP}(S, c) \otimes w_v \otimes \mathrm{IJOP}(c, T) \neq \overline{0}$ or, equivalently, $\mathcal{A}_{post^*}(c) \otimes w_v \otimes \mathcal{A}_{pre^*}(c) \neq \overline{0}$. To check this, we use the functional automaton $\mathcal{A}_\lhd$ again. It is not hard to check that the following holds for any weight w:

$$\mathcal{A}_{post^*}(c) \otimes w \otimes \mathcal{A}_{pre^*}(c) = \bigoplus\{W(w) \mid accPath(\mathcal{A}_{\triangleleft}, c, W)\}$$

Then $v \in V_\gamma$ iff $\bigoplus\{W(w_v) \mid accPath(\mathcal{A}_{\triangleleft}, \gamma\Gamma^*, W)\} \neq \overline{0}$. Again, this is computable using *path_summary*: Intersect $\mathcal{A}_{\triangleleft}$ with an unweighted automaton that accepts $\gamma\Gamma^*$, then run *path_summary*, but initialize the weight on the initial states of $\mathcal{A}_{\triangleleft}$ with w_v instead of $\overline{1}$.

This gives us an algorithm for computing V_γ, but its running time is proportional to $|V|$, which might be very large. In the case of predicate abstraction, $|V|$ is exponential in the number of predicates, but the weights (transformers) can be efficiently encoded using BDDs. For example, the identity transformer on V can be encoded with a BDD of size $\log|V|$. To avoid losing the advantages of using BDDs, we now present a symbolic algorithm.

5.3.2.2 A Symbolic Algorithm

Let $Y = \{y_v \mid v \in V\}$ be a fresh set of variables. We switch our weight domain from being $V \times V$ to $V \times Y \times V$. We write weights in the new domain with superscript e. Intuitively, the triple (v_1, y, v_2) denotes the transformation of v_1 to v_2 provided the variable y is "true". Combine is still defined to be union and extend is defined as follows: $w_1^e \otimes w_2^e = \{(v_1, y, v_2) \mid (v_1, y, v_3) \in w_1^e, (v_3, y, v_2) \in w_2^e\}$. Also, $\overline{1}^e = \{(v, y, v) \mid v \in V, y \in Y\}$ and $\overline{0}^e = \emptyset$. Define a symbolic identity id_s^e as $\{(v, y_v, v) \mid v \in V\}$. Let $Var(w^e) = \{v \mid (v_1, y_v, v_2) \in w^e$ for some $v_1, v_2 \in V\}$, i.e., the set of values whose corresponding variable appears in w^e. Given a weight in $V \times V$, define $\texttt{ext}(w) = \{(v_1, y, v_2) \mid (v_1, v_2) \in w, y \in Y\}$, i.e., all variables are added to the middle dimension. Note that $\overline{1}^e = \texttt{ext}(\overline{1})$. We will use the middle dimension to remember the "history" when composition is performed: for weights $w_1, w_2 \in V \times V$, it is easy to prove that $\texttt{Com}(w_1, w_2) = Var(\texttt{ext}(w_1) \otimes \mathrm{id}_s^e \otimes \texttt{ext}(w_2))$. Therefore, $V_\gamma = Var(w_\gamma^e)$ where, $w_\gamma^e = \bigoplus\{\texttt{ext}(v(\sigma_1)) \otimes id_s^e \otimes \texttt{ext}(v(\sigma_2)) \mid s \Rightarrow^{\sigma_1} c \Rightarrow^{\sigma_2} t, s \in S, c \in \gamma\Gamma^*, t \in T\}$. This weight is computed by replacing all weights w in the functional automaton with $\texttt{ext}(w)$ and running *path_summary* over paths accepting $\gamma\Gamma^*$, and initializing initial states with weight id_s^e. The advantages of this algorithm are: the weight $\texttt{ext}(w)$ can be represented using the same-sized BDD as the one for w (the middle dimension is "don't-care"); and the weight id_s^e can be represented using a BDD of size $O(\log|V|)$.

For our example, the weight $w^e_{\mathtt{n}}$ read off from the functional automaton shown in Fig. 5.4 is $\{(\_, y_9, 10)\}$, which gives us $V_{\mathtt{n}} = \{9\}$, as desired.

5.4 Experiments

We carried out experiments to measure two aspects of using error projections. First, we measured the efficiency of our error-projection algorithm. To put the numbers in perspective, we compared the time taken to compute an error projection against the time taken by a reachability query, which provides a measure of the amount of overhead that the error-projection computation can add to an abstraction-refinement loop. Second, we measured the sizes of the computed error projections. An error-projection size of 50% implies, roughly, a 2× speedup for all further rounds of refinement, because half of the program was proved correct and would not be considered subsequently. Our results were encouraging in both respects.

We added the error-projection algorithm to MOPED [85], a model checker for Boolean programs. We changed the implementation of MOPED so that it first encodes a Boolean program as an EWPDS, and then uses reachability queries to check assertions in the program.

We measured the time needed to solve $\mathrm{WC}(S, n\Gamma^*, T)$ for all program nodes n using the algorithms from Section 5.2: one that uses functional automata and the double-*prestar* algorithm. Although we report the size of the error projection, we could not validate how useful it was because only the model (and not the source code) was available to us.

The results are shown in Tab. 5.1. The table can be read as follows: the first two columns give the program names, and the number of nodes in the program. The Boolean programs were provided to us by S. Schwoon. They were created by SLAM as a result of performing predicate abstraction on real driver source code, but the original source code was not available to us.

The next three columns give the error-projection size relative to program size, and times to compute *poststar*(S) and *prestar*(T), respectively. Columns six and seven give the running time for solving $\mathrm{WC}(S, n\Gamma^*, T)$ for all nodes n using functionals and using double-*prestar*,

respectively, after the initial computation of *poststar*(S) and *prestar*(T) was completed, i.e., the time reported for functionals is the time taken to intersect $\mathcal{A}_{post^*}$ and $\mathcal{A}_{pre^*}$ and read off values from it; and the time reported for double-*prestar* is the time taken by lines 2–5 of the algorithm. Because the double-*prestar* method is so slow, we did not run these examples to completion; instead, we report the time for solving the weighted chop query for only 1% of the blocks and multiply the resulting number by 100. Column eight shows the ratio of the running time for using functionals (column six) against the time taken to compute $post^*(S) + pre^*(T)$ (column four + column five). The last column shows the ratio of the running time for the entire functional computation (column four + column five + column six) against the entire double-*prestar* computation (column five + column seven). All running times are in seconds. The experiments were run on a 3GHz P4 machine with 2GB RAM.

					WC($S, n\Gamma^*, T$)		Functional vs.	
Prog	Nodes	Error Proj.	$post^*(S)$ (sec)	$pre^*(T)$ (sec)	Functional (sec)	Double-pre^* (sec)	Reach (sec)	Double-pre^* (sec)
iscsiprt16	4884	0%	79	1.8	3.5	5800	0.04	69
pnpmem2	4813	0%	7	4.1	8.8	16000	0.79	804
iscsiprt10	4824	46%	0.28	0.36	1.6	1200	2.5	536
pnpmem1	4804	65%	7.2	4.5	9.2	17000	0.79	814
iscsi1	6358	84%	53	110	140	750000	0.88	2476
bugs5	36972	99%	13	2	170	85000	11.3	459

Table 5.1 MOPED results: The Boolean programs were provided by S. Schwoon. S is the entry point of the program, and T is the error configuration set. An error projection of size 0% means that the program is correct.

Discussion

As can be seen from the table, using functionals is about three orders of magnitude faster than using the double-pre^* method. Also, as shown in column eight, computation of the error projection compares fairly well with running a single forward or backward analysis (at least

for the smaller programs). To some extent, this implies that error-projection computation can be incorporated into model checkers without adding significant overhead.

The sizes of the error projections indicate that they might be useful in model checkers. Simple slicing, which only deals with the control structure of the program (and no weights) produced more than 99% of the program in each case, even when the program was correct.

The result for the last program `bugs5`, however, does not seem as encouraging due to the large size of the error projection. We do not have the source code for this program, but investigating the model reveals that there is a loop that calls several procedures that contain most of the code, and the error can occur inside the loop. If the loop resets its state when looping back, the error projection would include everything inside the loop or called from it. This is because for every node, there is a path from the loop head that goes through the node, then loops back to the head, with the same data state, and then goes to error.

This seems to be a limitation of error projections and perhaps calls for similar techniques that only focus on acyclic paths (paths that do not repeat a program state). However, for use inside a refinement process, error projections still give the minimal set of nodes that is sound with respect to the property being verified (focusing on acyclic paths need not be sound, i.e., the actual path that leads to error might actually be cyclic in an abstract model).

5.5 Additional Applications

The techniques presented in Section 5.2 and Section 5.3 give rise to several other applications of our ideas. In each case, we run one *poststar* query and one *prestar* query to obtain automata $\mathcal{A}_b$ and $\mathcal{A}_f$, respectively, and then create $\mathcal{A}_{bf} = (\mathcal{A}_b \lhd \mathcal{A}_f)$. Let $\mathrm{BW}(w_{\mathrm{bot}}, \gamma)$ be the weight obtained from the functional automaton $\mathcal{A}_{bf}$ intersected with $(\gamma\ \Gamma^*)$ and bottleneck weight w_{bot}. (The bottleneck weight used in Section 5.3.2.1 was w_v and the one used in Section 5.3.2.2 was id_s^e, respectively.) This weight can be computed for all nodes γ in roughly the same time as the error projection (which computes $\mathrm{BW}(\overline{1}, \gamma)$).

Multi-threaded Programs

KISS [78] is a system that can detect errors in concurrent programs that arise in at most two context switches. The two-context-switch bound enables verification using a tool that can only handle sequential programs. To convert a concurrent program into one suitable for a sequential analysis, KISS adds nondeterministic function calls to the `main` method of thread 2 after each statement of thread 1. Likewise it adds nondeterministic function returns after each statement of thread 2. It also ensures that a function call from thread 1 to thread 2 is only performed once. This technique essentially results in a sequential program that mimics the behavior of a concurrent program (with two threads) for two context switches.

Using our techniques, we can extend KISS to determine all of the nodes in thread 1 where a context switch can occur that leads to an error later in thread 1. One way to do this is to use nondeterministic calls and returns as KISS does and then compute the error projection. However, due to the automata-theoretic techniques we employ, we can omit the extra additions. The following algorithm shows how to do this:

1. Create $\mathcal{A}_{\lhd} = \mathcal{A}_{post^*} \lhd \mathcal{A}_{pre^*}$ for thread 1.

2. Let $\mathcal{A}_2$ be the result of a poststar query from `main` for process 2. Let $w = path\_summary(\mathcal{A}_2)$; w represents the state transformation caused by the execution steps spent in thread 2.

3. For each program node γ of thread 1, let $w_\gamma = \mathrm{BW}(w, \gamma)$ be the weight obtained from functional automaton $\mathcal{A}_{\lhd}$ of thread 1. By using w as the bottleneck weight, we account for the two context switches (from thread 1 to 2 and from 2 back to 1); w summarizes the effect produced while thread 2 has control. If $w_\gamma \neq \overline{0}$ then an error can occur in the program when the first context switch occurs at node γ in thread 1.

Then this process can be repeated after interchanging the roles of thread 1 and thread 2. This allows thread 2 the first chance to execute. Using this algorithm, we can determine all the nodes where a context switch must occur for an error to (eventually) arise.

Error Reporting

The model checker SLAM [4] used a technique presented in [5] to identify error causes from counterexample traces. Their main idea is to remove "correct" transitions from the error trace; the remaining transitions indicate the cause of the error. These correct transitions were obtained by a backward analysis from non-error configurations. However, no restrictions were imposed that these transitions also be reachable from the entry point of the program. Thus, their technique may remove too many transitions and fail to localize the error. Using annotated error projections, we can limit the correct transitions to ones that are both forward reachable from program entry and backward reachable from the non-error configurations.

5.6 Related Work

The combination of forward and backward analysis has a long history in abstract interpretation, going back to Cousot's thesis [23]. It has been also been used in model checking [62] and in interprocedural analysis [41]. In this chapter, we have shown how forward and backward approaches can be combined precisely in the context of interprocedural analysis performed with WPDSs; our experiments show that this approach is significantly faster than a more straightforward one.

With model checkers becoming more popular, there has been considerable work on explaining the results obtained from a model checker in an attempt to localize the fault in the program [20, 5]. These approaches are complimentary to ours. They build on information obtained from reachability analysis performed by the model checker and use certain heuristics to isolate the root cause of the bug. Error projections seek to maximize information that can be obtained from the reachability search so that other tools can take advantage of this gain in precision. This chapter focused on using error projections inside an abstraction-refinement loop. The second application in Section 5.5 briefly shows how they can be used for fault localization. It would be interesting to explore further use of error projections for fault localization.

Such error-reporting techniques have also been used outside model checking. Kremenek et al. [54] use statistical analysis to rank counterexamples found by the `xgcc`[28] compiler. Their goal is to present to the user an ordered list of counterexamples sorted by their confidence rank.

The goal of both program slicing [93] and our work on error projection is to compute a set of nodes that exhibit some property. In our work, the property of interest is membership in an error path, whereas in the case of program slicing, the property of interest is membership in a path along data and control dependence edges. Slicing and chopping have certain advantages—for instance, chopping filters out statements that do not transmit effects from source s to target t. These techniques have been generalized by Hong et al. [39], who show how to perform more precise versions of slicing and chopping using predicate-abstraction and model checking. However, their methods are intraprocedural, whereas our work addresses interprocedural analysis.

Mohri et al. investigated the intersection of weighted automata in their work on natural-language recognition [64, 65]. For their weight domains, the extend operation must be commutative. We do not require this restriction.

Chapter 6

Interprocedural Analysis of Concurrent Programs Under a Context Bound

The analysis of concurrent programs is a challenging problem. While, in general, the analysis of both concurrent and sequential programs is undecidable, what makes concurrency hard is the fact that even for simple program models, the presence of concurrency makes their analysis computationally very expensive. When the model of each thread is a finite-state automaton, the analysis of such systems is PSPACE-complete; when the model is a pushdown system, the analysis becomes undecidable [80]. This is unfortunate because it does not allow the advancements made on such models in the sequential setting, i.e., when the program has only one thread, to be applied in the presence of concurrency.

Another consequence of the above result is that the analysis of concurrent programs that may contain recursion is undecidable. Even in the absence of recursion, designing an *interprocedural* analysis, i.e., an analysis that can precisely reason about the call-return semantics of a procedure call, becomes hard. As a consequence, to deal with concurrency soundly, most analyses give up precise handling of procedures and become *context-insensitive*. Alternatively, tools can use inlining to unfold multi-procedure programs into single-procedure ones. This approach cannot handle recursive programs, and can cause an exponential blowup in the size for non-recursive ones.

Because interprocedural analyses have proven to be very useful for sequential programs [4, 83, 81, 84], it is desirable to have the same kind of precision even for concurrent programs.

A different way to sidestep the undecidability of analyzing concurrent recursive programs is to limit the amount of concurrency by bounding the number of *context switches*, where a context switch is defined as the transfer of control from one thread to another. Such an abstraction has proven to be useful for program analysis because many bugs can be found in a few context switches [78, 77, 70, 59]. We use the term *context-bounded analysis* (CBA) to refer to the general approach of analyzing recursive and concurrent programs under a context bound.

CBA does not impose any bound on the execution length between context switches. Thus, even under a context bound, the analysis still has to consider the possibility that the next switch takes place in any one of the (possibly infinite) states that may be reached after a context switch. Because of this, CBA still considers many concurrent behaviors [70].

In previous work, Qadeer and Rehof [77] showed that CBA is decidable when program threads are modeled using pushdown systems (i.e., for recursive programs under a finite-state abstraction of program data). In this chapter, we generalize their result to weighted pushdown systems (i.e., to recursive programs under infinite-state data abstractions), and also provide a new symbolic algorithm for the finite-state case.

Our goal is to be able to take any abstraction used for interprocedural analysis of sequential programs and directly extend it to handle context-bounded concurrency. Our main result follows in the spirit of *coincidence theorems* in dataflow analysis (for sequential programs) [44, 88, 52]. We give conditions on the abstractions under which CBA can be precisely solved, along with an algorithm. In addition to the usual conditions required for precise interprocedural analysis of sequential programs, we require the existence of a *tensor product* (defined in Section 6.5). We show that these conditions are satisfied by a class of abstractions, thus giving precise algorithms for CBA with those abstractions. These include finite-state abstractions, such as the ones used for verification of Boolean programs in model checking [4], as well as infinite-state abstractions, such as affine-relation analysis (ARA) [67]. Note that without a context bound, reasoning about concurrent programs under these abstractions is undecidable [80, 66].

We show that when a WPDS is used to model each thread of a concurrent program, CBA can be precisely carried out for the program, provided tensor products exist for the weights.

Motivation

Context-bounded analysis is not sound because it does not capture all of the behaviors of a program; however, it has been shown to be useful for program analysis. KISS [78], a verification tool for CBA with a fixed context bound of 2, found numerous bugs in device drivers. A study with an explicit-state model checker [70], which works on programs with a closed environment, found more bugs with slightly higher context bounds. It also showed that the amount of additional state space covered decreases with each increment of the context bound. Thus, even a small context bound is sufficient to cover many program behaviors, and proving safety under a context bound should provide confidence towards the reliability of the program.

Unlike the above-mentioned work, this dissertation addresses CBA with any given context bound and with different program abstractions (including ones that would cause explicit-state model checkers not to terminate).

In Chapter 7, we add to the above results on the utility of CBA. Using a symbolic model checker that works on programs with an open environment, we showed that many bugs can be found in a few context switches. Motivated by these reasons, our goal is to develop analyses that are sound under a context bound.

Previous work has only considered CBA for a restricted set of abstractions. Having the ability to do CBA with other abstractions can be useful for analyzing concurrent programs. For example, it can be useful to combine CBA with ARA. This is illustrated by the program snippet in Fig. 6.1. Here, multiple threads share the circular buffer `q` in a producer (`enq`) consumer (`deq`) fashion. Using CBA with ARA with modular arithmetic [68], one can prove (under a given context bound) that `(hd - tl - cnt) % SIZE = 0` provided `SIZE` is a prime power. ARA generalizes analyses like copy-constant propagation, linear-constant

propagation, and induction-variable analysis. It can be used to find invariants, such as the one shown above, to do verification or to increase the precision of other analyses.

```
Elem q[SIZE];
int hd = cnt = tl = 0;

void enq(Elem e) {
  while(true) {
  atomic {
    if( cnt < SIZE) {
      q[tl] = e;
      tl = (tl+1)%SIZE;
      cnt ++;
      break;
    }
}}}

Elem deq() {
  while(true) {
  atomic {
    if(cnt > 0) {
      Elem e = q[hd];
      hd = (hd+1)%SIZE;
      cnt--;
      return e;
    }
}}}
```

Figure 6.1 A concurrent program that manages a circular queue.

The context bound used for CBA can be increased iteratively to consider more effects of concurrency and to analyze more program behaviors. This has the added advantage of finding bugs in the fewest context switches needed to trigger them. It is reasonable to consider a bug that arises only after a greater number of context switches to be "harder" than a bug that requires fewer context switches. Thus, CBA allows additional concurrency to be considered "on-demand" by increasing the context bound.

Challenges and Techniques

Between consecutive context switches, a concurrent program acts like a sequential program because only one thread is executing. However, a recursive thread can reach an infinite number of states before the next context switch because it has an unbounded stack. CBA has to consider the possibility of a context switch occurring at any one of these states.

The Qadeer-Rehof (QR) algorithm uses PDSs to encode program threads. The set of reachable states of a PDS can be represented using an automaton [15] (see Section 2.3.2). The QR algorithm makes use of this result to get a handle on all reachable states between context switches. However, to explore all possible context switches, it crucially relies on the finiteness of the data abstraction because it enumerates all reachable data states at a context switch.

Our first step is to develop a new algorithm for the case of PDSs. Our motivation is to have an algorithm that is more likely to generalize to handle other abstractions. The new algorithm (Section 6.3) represents the effect of executing a thread (a PDS) from any arbitrary state using a *finite-state transducer*. The transducer accepts a pair (c_1, c_2) if a thread, when started in state c_1, can reach state c_2. Caucal [17] showed that such transducers can be constructed for PDSs. This is a more general result than the one on reachability in PDSs that was used in the QR algorithm. Next, to describe the behavior of the entire program with multiple threads, these transducers are composed. One transducer composition is performed for each context switch.

We then generalize this algorithm for WPDSs (Section 6.4 and Section 6.5). The weights (or the data abstraction) add several complications. We define *weighted transducers* to capture the reachability relation of WPDSs. We show that a weighted transducer can always be constructed for a WPDS (no such result was known previously). The next step is to compose these transducers. While weighted automata and transducers have been considered in the literature before, the weights are assumed to have much stronger properties (especially commutativity, which defeats the purpose of CBA by making thread interleavings redundant, as we shall see later). For program analysis, we only have weaker properties on weights. To compose weighted transducers, we require that weight domains provide a *tensor-product* operation (Section 6.5). Tensor products have been used previously in program analysis for combining abstractions [71]. However, we use them in a different context and for a completely different purpose. In particular, previous work has used them for combining abstractions that are to be performed in *lock-step*; in contrast, we use them to stitch together the data state *before* a context switch with the data state *after* a context switch. This is non-trivial because the data state is correlated with an (unbounded) program stack.

By using WPDSs, not only do we obtain new algorithms for infinite-state abstractions, but also symbolic algorithms for finite-state abstractions. The latter algorithms avoid the enumeration that the QR algorithm performs at a context switch.

The contributions of the work presented in this chapter can be summarized as follows:

- We give sufficient conditions under which CBA is decidable, along with an algorithm. This generalizes previous work on CBA of PDSs [77]. Our result also proves that CBA can be decided for affine-relation analysis, i.e., we can precisely find all affine relationships between program variables that hold at a particular point in the (concurrent) program. We use WPDSs as our program model, and the weights encode the program's data abstraction. By using WPDSs, we can also answer "stack-qualified" queries [83], which ask for the set of values that may arise at a program point in a given calling context, or in a regular set of calling contexts.

- We show that for WPDSs, the reachability relation can be encoded using a weighted transducer (Section 6.4), generalizing a previous result for PDSs by Caucal [17].

- We give precise algorithms for composing weighted transducers (Section 6.5), when tensor products exist for the weights. This generalizes previous work on manipulating weighted automata and transducers [64, 65]. We also show a class of abstractions that satisfies this property.

- We discuss implementation issues for realizing CBA in Section 6.6. We show that for PDSs, CBA is NP-complete. Our algorithm, based on transducers, does have a large complexity, but it is more amenable to symbolic techniques such as using BDDs (in the finite-state case) than the QR algorithm.

The rest of the chapter is organized as follows. In Section 6.1, we formally define CBA. In Section 6.2, we discuss previous work on CBA under a finite-state data abstraction. In Section 6.3, we present our algorithm for PDSs, which is based on transducers. The later sections generalize this result to WPDSs. In Section 6.4, we give an efficient construction for transducers for WPDSs. In Section 6.5, we show how weighted transducers can be composed. In Section 6.6, we discuss implementation issues for CBA. In Section 6.7, we discuss related work. A further generalization of CBA is discussed in Chapter 7.

6.1 Problem Definition

Notation. A binary relation on a set S is a subset of $S \times S$. If R_1 and R_2 are binary relations on S, then their relational composition $(R_1; R_2)$ is defined as $\{(s_1, s_3) \mid \exists s_2 \in S, (s_1, s_2) \in R_1, (s_2, s_3) \in R_2\}$. If R is a binary relation, R^i is the relational composition of R with itself i times, and R^0 is the identity relation on S. $R^* = \cup_{i=0}^{\infty} R^i$ is the reflexive-transitive closure of R.

The Unweighted Case

First, we define CBA under a finite-state data abstraction, i.e., when PDSs are used as the abstract model of threads. To distinguish this case from the general one, and in keeping with the nomenclature used in previous work, we call CBA under a finite-state data abstraction *context-bounded model checking* (CBMC).

The abstract model for CBMC is a *concurrent PDS*, defined as sequence of PDSs, $(\mathcal{P}_1, \mathcal{P}_2, \cdots, \mathcal{P}_n)$, $\mathcal{P}_i = (P, \Gamma_i, \Delta_i)$, which have the same set of control locations (P). A configuration of this model is a tuple $(p, u_1, \cdots, u_n)$, where $p \in P$, $u_i \in \Gamma_i^*$. The transition system of this model, which is a binary relation on the set of all such configurations, is defined as follows. Let the transition relation of $\mathcal{P}_i$ be $\Rightarrow_i$. If $\langle p, u_i \rangle \Rightarrow_i \langle p', u_i' \rangle$, then we say $(p, u_1, \cdots, u_i, \cdots, u_n) \Rightarrow_i^c (p', u_1 \cdots, u_i', \cdots, u_n)$. The union of $\Rightarrow_i^c$ for all $i = 1$ to n defines the transition relation for the CBMC model, i.e., it defines a single execution step of the model.

Concurrent PDSs can encode finite-state data abstractions of recursive concurrent programs. Consider a *concurrent Boolean program*, defined as a set of Boolean programs, one for each thread, in which the global variables are shared between the threads. (Thus, any of the threads can modify the global variables and their own copy of the local variables, but they cannot directly read from or write to local variables of other threads.) Synchronization between threads can be easily implemented, e.g., by using global variables to implement locks.

Such models are a natural description of recursive concurrent programs with a finite-state data abstraction. We do not consider dynamic creation of threads in our model.[1]

Concurrent Boolean programs can be encoded using a concurrent PDS. Let B be a concurrent Boolean program with n threads $t_1, t_2, \cdots, t_n$. Let G be the set of global states of B (valuations of global variables) and L_i be the set of local states of t_i, which includes valuation of local variables as well as the program stack (as described in Section 2.2). Then the state space of B consists of the global state paired with local states of each of the threads, i.e., the set of states is $G \times L_1 \times \cdots \times L_n$.

Let $\mathcal{P}_{t_i}$ be the PDS that encodes the Boolean program t_i (Section 2.3). Because different threads share the same global variables, the PDSs $\mathcal{P}_{t_i}$ have the same set of control locations (which is G). Then the concurrent PDS $\mathcal{P}_B = (\mathcal{P}_{t_1}, \mathcal{P}_{t_2}, \cdots, \mathcal{P}_{t_n})$ encodes B. It is easy to see that the transition relation of $\mathcal{P}_B$ describes a single execution step of B.

The CBMC problem is to find the set of reachable states of a concurrent PDS $(\mathcal{P}_1, \mathcal{P}_2, \cdots, \mathcal{P}_n)$ under a bound on the number of context switches. Formally, let k be the bound on context switches. The execution of a concurrent program can be decomposed to a sequence of *execution contexts*. In an execution context, one thread has control and it executes a finite number of steps. The execution context changes at a *context switch* and control is passed to a different thread. For k context switches, there must be $k+1$ execution contexts. Let $\Rightarrow^{\text{ec}}$ be $(\cup_{i=1}^{n}(\Rightarrow_i^c)^*)$, the transition relation that describes the effect of one execution context. Then the CBMC problem is to find the set of reachable states in the transition relation given by $(\Rightarrow^{\text{ec}})^{k+1}$. Note that while a bound is placed on the number of context switches, no bound is placed on the length of an individual execution context.

Analysis of concurrent Boolean programs is undecidable [80], i.e., it is not possible to verify if a given state is reachable under the transition system $(\Rightarrow^{\text{ec}})^*$ or not, but Qadeer and Rehof showed that CBMC, i.e., reachability under $(\Rightarrow^{\text{ec}})^{k+1}$ for a fixed k, is decidable.

[1] Dynamic creation up to n threads can be encoded in the model [77]. Moreover, for CBA that considers k context switches, n can be bounded by k because other threads would never get a chance to run.

The Weighted Case

Now we define CBA in the general case, i.e., when WPDSs are used as the abstract model for each thread.

The transition relation of a WPDS is a weighted relation (Defn. 2.4.15) over the set of PDS configurations. For configurations c_1 and c_2, if $r_1, \cdots, r_m$ are all the rules such that $c_1 \Rightarrow^{r_i} c_2$, then $(c_1, c_2, \oplus_i f(r_i))$ is in the weighted relation of the WPDS. In a slight abuse of notation, we will use $\Rightarrow$ and its variants for the weighted transition relation of a WPDS. Note that the weighted relation $\Rightarrow^*$ maps the configuration pair (c_1, c_2) to $\text{IJOP}(\{c_1\}, \{c_2\})$.

The CBA problem is the same as the one for CBMC, except that all relations are weighted. This means that each thread is modeled as a WPDS and their underlying PDSs have the same set of control states.

Given the weighted relation $R = (\Rightarrow^{\text{ec}})^{k+1}$, the set of initial configurations S and a set of final configurations T, we want to be able to solve for $R(S,T) = \oplus\{R(s,t) \mid s \in S, t \in T\}$. This captures the net transformation on the data state between S and T: it is the combine over the values of all paths involving at most k context switches that go from a configuration in S to a configuration in T. Our results from Section 6.4 and Section 6.5 allow us to solve for this value when S and T are regular sets of configurations.

This problem definition allows one to precisely encode concurrent Boolean programs (with variations such as finding the shortest trace), as well as concurrent affine programs, when each only have global variables. (The extension to local variables requires that the threads be modeled using EWPDSs, which is left as future work.)

For example, consider two copies of the program in Fig. 2.2(a) running in parallel. Let the CFG nodes of the second copy be $\Gamma' = \{n'_1, \cdots, n'_8\}$, to distinguish them from those of the first copy. With $k = 2$, $S = \{\langle p, n_1, n'_1\rangle\}$ (the starting configuration of the program), $T = \{\langle p, n_6, u'\rangle \mid u' \in (\Gamma')^*\}$ (thread 1 is at n_6 and thread 2 can have any stack), and R as above, the weight $R(S,T)$ is a relation with range $\{(3,3),(3,7),(7,3),(7,7)\}$, meaning that these valuations of $(\mathtt{x}, \mathtt{y})$ are possible at some configuration in T.

6.2 Context Bounded Model Checking

In this section, we describe the Qadeer-Rehof (QR) algorithm for CBMC. It works under the assumption that the set G is finite. Under such an abstraction, the only source of unboundedness is the program stack.

The algorithm proceeds by iteratively increasing the number of execution contexts. Within one execution context, the global state can be considered local to the executing thread because it is the only thread that accesses it. At a context switch, the global state is synchronized with other threads so that they have the same view of the shared memory. The algorithm needs G to be finite to be able to explore all possibilities at a context switch. We only give an overview of the QR algorithm, presenting it mostly in terms of explicit state spaces and just touch on a few aspects of a PDS-based implementation. A complete description of the PDS-based implementation is given in [77].

If $S_i \subseteq L_i$ is a set of local states, then let $(g, S_1, S_2, \cdots, S_n)$ be the set of states $\{(g, l_1, \cdots, l_n) \mid l_i \in S_i\}$. We use the symbol η as a shorthand for such a set of states. The QR algorithm is a worklist-based algorithm. An item on the worklist is a pair (η, i), denoting that the set of states η is reachable in up to i context switches. Initially, the worklist contains $(\eta_{\text{init}}, 0)$, where η_{init} is the starting set of states for the program. Then the algorithm repeats the following steps until the worklist is empty.

1. Select and remove an item (η, i) from the worklist. If $i = k$, then the context bound has been reached, so pick another item.

2. Let $\eta = (g, S_1, \cdots, S_n)$. For each j from 1 to n, repeat steps 3 and 4.

3. Using a thread-local analysis on t_j, find the set of states that t_j can reach when started from the set of states (g, S_j). Let this set be R_j, i.e., $(g, S_j) \Rightarrow^*_{t_j} R_j$. In PDS terms, $R_j = post^*{}_{t_j}((g, S_j))$. Write R_j as $\cup_{p=1}^{m}(g_p, R_j^p)$. This implies that thread t_j can change the global state from g to g_p and itself reach some local state in R_j^p.

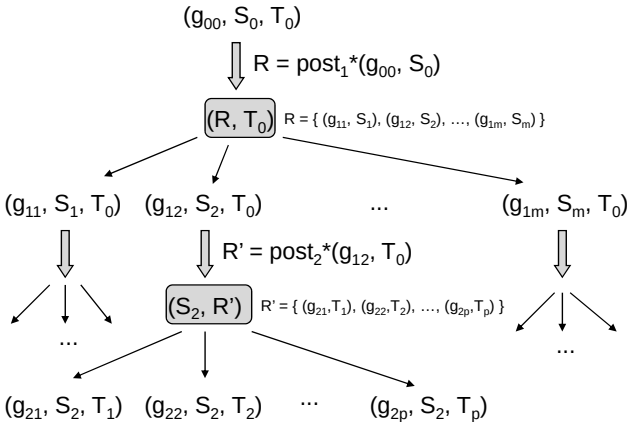


Figure 6.2 The computation of the QR algorithm, for two threads, shown schematically in the form of a tree. The shaded boxes are just temporary placeholders and are not inserted into the worklist. The thick arrows correspond to Step 3 and other arrows correspond to Step 4. The set of tuples at level i of the tree correspond to all states reached in i context switches.

4. For each g_p produced in the above step, the set of states $\eta_p = (g_p, S_1, \cdots, S_{j-1}, R_j^p, S_{j+1}, \cdots, S_n)$ are reachable in up to $i+1$ context switches. Insert $(\eta_p, i+1)$ into the worklist.

Steps 3 and 4 take a starting set of states η and produce all states that are reachable in one execution context. First, a thread t_j is picked that gets to execute in that context. Then step 3 finds all states that execution of t_j can produce. For each of the global states g_p that can be produced, it is passed to all other threads at the context switch in step 4. The set of tuples (η, i) with $i = k$ represent the set of all reachable states. The computation performed by this algorithm is depicted in Fig. 6.2 in the form of a tree.

An important aspect of the algorithm is the way it manipulates set of states. An item on the worklist is of the form $(g, S_1, \cdots, S_n)$, representing a set of states. The global state g is kept explicit because it is required for synchronization across threads at a context switch. The local states need not be kept explicit, and they are collected in the sets S_i. This is important because the set of local states can be infinite. In the PDS-based implementation, the sets S_i are kept in symbolic form using automata (Defn. 2.3.2). The *poststar* algorithm

works on these representations, mapping automata (capturing starting configurations) to automata (capturing reachable configurations).

6.3 A New Algorithm for CBMC Using Transducers

The QR algorithm fails to generalize to infinite-state abstractions because of its requirement to keep the global state explicit in the worklist items. After each context switch, the algorithm does a "fan-out" proportional to the size of the global state space $|G|$ (see Fig. 6.2) to pass the global state to all other threads. This is also true for the PDS-based implementation of the QR algorithm. The algorithm presented in this section avoids such a fan-out (and will be extended to infinite-state abstractions in Section 6.4 and Section 6.5).

The QR algorithm makes several calls to *poststar* to compute the forward reachable states in a single thread. This is crucial to be able to work with infinite sets of configurations. However, the disadvantage is that *poststar* requires a starting set of configurations to find all of the reachable configurations. Creation of this starting set is what forces the fan-out operation to alternate with calls to *poststar*.

A similar problem arises in interprocedural analysis of sequential programs: a procedure can get called from multiple places with multiple different input values. Instead of reanalyzing the procedure for each input value, it is analyzed independently of the calling context to create a *summary*. This summary concisely describes the effect of executing the procedure in any calling context, in terms of the *relation* between input to the procedure and its output. Similarly, instead of reanalyzing a thread every time it receives control after a context switch, we create a summary for it. The difficulty is that the "input" here is a starting set of configurations, and the "output" is the reachable sets of configurations; again, the summary must be relation-valued. Because both of these sets can be infinite, we need the summary to be representable symbolically.

Our approach to generalizing the QR algorithm (for both finite-state and infinite-state data abstractions) is based on the following observation:

Observation 1. *One can construct an appropriate summary of a thread's behavior using a finite-state transducer (an automaton with input and output tapes).*

Definition 6.3.1. *A **finite-state transducer** τ is a tuple $(Q, \Sigma_i, \Sigma_o, \lambda, I, F)$, where Q is a finite set of states, Σ_i and Σ_o are input and output alphabets, $\lambda \subseteq Q \times (\Sigma_i \cup \{\varepsilon\}) \times (\Sigma_o \cup \{\varepsilon\}) \times Q$ is the transition relation, $I \subseteq Q$ is the set of initial states, and $F \subseteq Q$ is the set of final states. If $(q_1, a, b, q_2) \in \lambda$, written as $q_1 \xrightarrow{a/b} q_2$, we say that the transducer can go from state q_1 to q_2 on input a, and outputs the symbol b. Given a state $q \in I$, we say that the transducer can accept a string $\sigma_i \in \Sigma_i^*$ with output $\sigma_o \in \Sigma_o^*$ if there is a path from state q to a final state that takes input σ_i and outputs σ_o. The **language** of the transducer $\mathcal{L}(\tau)$ is defined as the following subset of $\Sigma_i^* \times \Sigma_o^*$: $\{(\sigma_i, \sigma_o) \mid$ the transducer can output string σ_o when the input is $\sigma_i\}$.*

Given a PDS $\mathcal{P}$, one can construct a transducer $\tau_{\mathcal{P}}$ whose language equals $\Rightarrow^*$, the transitive closure of $\mathcal{P}$'s transition relation: The transducer accepts a pair (c_1, c_2) if a thread, when started in state c_1, can reach state c_2. This result was first given by Caucal [17], but it was not accompanied with a complexity result, except that it was polynomial time. Our construction of transducers for WPDSs (strictly more general than Caucal's result) makes use of recent advancements in the analysis of (W)PDSs [9, 30, 85, 83] for an efficient construction. Because such transducers are of general importance, we give a complexity result. The following theorem is derived from Thm. 6.4.4 given in Section 6.4.

Theorem 6.3.2. *Given a PDS $\mathcal{P} = (P, \Gamma, \Delta)$, a transducer $\tau_{\mathcal{P}}$ can be constructed such that it accepts input $(p_1\ u_1)$ and outputs $(p_2\ u_2)$ if and only if $\langle p_1, u_1 \rangle \Rightarrow^* \langle p_2, u_2 \rangle$. Moreover, this transducer can be constructed in time $O(|P||\Delta|(|P||\Gamma|+|\Delta|))$ and has at most $|P|^2|\Gamma|+|P||\Delta|$ states.*

The advantage of using transducers is that they are closed under relational composition.

Lemma 6.3.3. *Given transducers τ_1 and τ_2 with input and output alphabet Σ, one can construct a transducer $(\tau_1; \tau_2)$ such that $\mathcal{L}(\tau_1; \tau_2) = \mathcal{L}(\tau_1); \mathcal{L}(\tau_2)$, where the latter ";" denotes*

composition of relations. Similarly, if $\mathcal{A}$ is an automaton with alphabet Σ, one can construct an automaton $\tau_1(\mathcal{A})$ such that its language is the image of $\mathcal{L}(\mathcal{A})$ under $\mathcal{L}(\tau_1)$, i.e., the set $\{u \in \Sigma^* \mid \exists u' \in \mathcal{L}(\mathcal{A}), (u', u) \in \mathcal{L}(\tau_1)\}$.

Both of these constructions are carried out in a manner similar to automaton intersection [40]. For composing transducers, for each transition $p \xrightarrow{a/b} q$ in τ_1 and transition $p' \xrightarrow{b/c} q'$ in τ_2, add the transition $(p, p') \xrightarrow{a/c} (q, q')$ to their composition. For transducer-automaton application, each transition $p \xrightarrow{a/b} q$ in τ_1 is matched with transition $p' \xrightarrow{a} q'$ in $\mathcal{A}$ to produce transition $(p, p') \xrightarrow{b} (q, q')$ in $\tau_1(\mathcal{A})$. One can also take the union of transducers (union of their languages) in a manner similar to union of automata.

Coming back to CBMC, each thread is represented using a PDS. Thus, we can construct a transducer τ_{t_i} for the transition relation of thread t_i, i.e., for $\Rightarrow_i^*$. By extending τ_{t_i} to perform the identity transformation on stack symbols of threads other than t_i (using transitions of the form $p \xrightarrow{\gamma/\gamma} p$), we obtain a transducer $\tau_{t_i}^c$ for $(\Rightarrow_i^c)^*$. Next, a union of these transducers gives τ^{ec}, which represents $\Rightarrow^{\text{ec}}$. Performing the composition of τ^{ec} k times with itself gives us a transducer τ that represents $(\Rightarrow^{\text{ec}})^{k+1}$. If an automaton $\mathcal{A}$ captures the set of starting states of the concurrent program, $\tau(\mathcal{A})$ gives a single automaton for the set of all reachable states in the program (under the context bound).

We believe that the above algorithm provides a better basis for implementing a tool for CBMC than the QR algorithm. In particular, the new algorithm avoids the fan-out step, which—as we show below—allows it to be extended to infinite-state data abstractions. To make this extension, we represent (recursive) programs with infinite-state abstractions using WPDSs. Extending our algorithm to WPDSs presents two challenges: one is the construction of a weighted transducer for a WPDS, and the other is the composition of two weighted transducers. These issues are addressed in Section 6.4 and Section 6.5, respectively.

6.4 Weighted Transducers

In this section, we show how to construct a weighted transducer for the weighted relation $\Rightarrow^*$ of a WPDS. We defer the definition of a weighted transducer to a little later in this

$$\begin{aligned}
\langle p_1, \gamma_1\ \gamma_2\ \gamma_3 \cdots \gamma_n \rangle &\Rightarrow^{\sigma_1} \quad \langle p_2, \gamma_2\ \gamma_3 \cdots \gamma_{k+1}\ \gamma_{k+2} \cdots \gamma_n \rangle \\
&\Rightarrow^{\sigma_2} \quad \langle p_3, \gamma_3 \cdots \gamma_{k+1}\ \gamma_{k+2} \cdots \gamma_n \rangle \\
&\cdots \\
&\Rightarrow^{\sigma_k} \quad \langle p_{k+1}, \gamma_{k+1}\ \gamma_{k+2} \cdots \gamma_n \rangle \\
&\Rightarrow^{\sigma_{k+1}} \quad \langle p_{k+2}, u_1\ u_2 \cdots u_j\ \gamma_{k+2} \cdots \gamma_n \rangle
\end{aligned}$$

Figure 6.3 A path in the PDS's transition relation; $u_i \in \Gamma, j \geq 1, k < n$.

section (Defn. 6.4.3). Our solution uses the following observation about paths in a PDS's transition relation. Every path $\sigma \in \Delta^*$ that starts from a configuration $\langle p_1, \gamma_1\gamma_2 \cdots \gamma_n \rangle$ can be decomposed as $\sigma = \sigma_1\sigma_2 \cdots \sigma_k\sigma_{k+1}$ (see Fig. 6.3) such that $\langle p_i, \gamma_i \rangle \Rightarrow^{\sigma_i} \langle p_{i+1}, \varepsilon \rangle$ for $1 \leq i \leq k$, and $\langle p_{k+1}, \gamma_{k+1} \rangle \Rightarrow^{\sigma_{k+1}} \langle p_{k+2}, u_1u_2 \cdots u_j \rangle$: every path has zero or more *pop phases* $(\sigma_1, \sigma_2, \cdots, \sigma_k)$ followed by a single *growth phase* (σ_{k+1}):

1. **Pop-phase:** A path such that the net effect of the pushes and pops performed along the path is to take $\langle p, \gamma u \rangle$ to $\langle p', u \rangle$, without looking at $u \in \Gamma^*$. Equivalently, it can take $\langle p, \gamma \rangle$ to $\langle p', \varepsilon \rangle$.

2. **Growth-phase:** A path such that the net effect of the pushes and pops performed along the path is to take $\langle p, \gamma u \rangle$ to $\langle p', u'u \rangle$ with $u' \in \Gamma^+$, without looking at $u \in \Gamma^*$. Equivalently, it can take $\langle p, \gamma \rangle$ to $\langle p', u' \rangle$.

Intuitively, this holds because for a path to look at γ_2, it must pop off γ_1. If it does not pop off γ_1, then the path is in a growth phase starting from γ_1. Otherwise, the path just completed a pop phase. We construct the transducer for a WPDS by computing the net transformation (weight) implied by these phases. First, we define two procedures:

1. $pop : P \times \Gamma \times P \to D$ is defined as follows:

$$pop(p, \gamma, p') = \bigoplus \{ v(\sigma) \mid \langle p, \gamma \rangle \Rightarrow^{\sigma} \langle p', \varepsilon \rangle \}$$

2. $grow : P \times \Gamma \to ((P \times \Gamma^+) \to D)$ is defined as follows:

$$grow(p, \gamma)(p', u) = \bigoplus \{ v(\sigma) \mid \langle p, \gamma \rangle \Rightarrow^{\sigma} \langle p', u \rangle \}$$

Note that $grow(p, \gamma) = poststar(\langle p, \gamma \rangle)$, where the latter is interpreted as a function from configurations to weights that maps a configuration to the weight with which it is accepted, or $\overline{0}$ if it is not accepted. The following lemmas give efficient algorithms for computing the above quantities.

Lemma 6.4.1. *Let $\mathcal{A} = (P, \Gamma, \emptyset, P, P)$ be a $\mathcal{P}$-automaton that represents the set of configurations $C = \{\langle p, \varepsilon \rangle \mid p \in P\}$. Let $\mathcal{A}_{pop}$ be the forward weighted-automaton obtained by running prestar on $\mathcal{A}$. Then $pop(p, \gamma, p')$ is the weight on the transition (p, γ, p') in $\mathcal{A}_{pop}$. We can generate $\mathcal{A}_{pop}$ in time $O_s(|P|^2|\Delta|H)$, and it has at most $|P|$ states.*

Proof. This follows directly from Lem. 2.3.4 and its weighted version described in Section 2.4.1. However, we give a slightly informal, but intuitive, proof here. We use the fact that the saturation-based implementation of *prestar* (Section 2.4.1) is correct.

The lemma runs *prestar* on the empty automaton, i.e., one with no transitions, which represents the configuration set $C = \{\langle p, \varepsilon \rangle \mid p \in P\}$. Let β be a stack symbol not in Γ, and $\mathcal{A}^p_\beta$ be an automaton with two states $\{p, q\}$, $q \notin P$ and a single transition (p, β, q). Let q be the final state of this automaton. Because $\beta \notin \Gamma$, running *prestar* on $\mathcal{A}^p_\beta$ will return the same automaton as the one returned by running *prestar* on the empty automaton, except for the extra transition (p, β, q) (because no rule can match β). $\mathcal{A}^p_\beta$ represents the configuration set $\{\langle p, \beta \rangle\}$, and therefore, $\mathcal{A}^p_\beta(\langle p', \gamma\ \beta \rangle) = pop(p', \gamma, p)$ according to the definition of *pop*. However, $\mathcal{A}^p_\beta(\langle p', \gamma\ \beta \rangle)$ is exactly the weight on the transition (p', γ, p) because the only path in $\mathcal{A}^p_\beta$ that accepts $(\gamma\ \beta)$ starting in state p' is the one that follows transitions (p', γ, p) and (p, β, q). The result follows by repeating the argument for all $p \in P$. □

Lemma 6.4.2. *Let $\mathcal{A}_F = (Q, \Gamma, \rightarrow, P, F)$ be a $\mathcal{P}$-automaton, where $Q = P \cup \{q_{p,\gamma} \mid p \in P, \gamma \in \Gamma\}$ and $p \xrightarrow{\gamma} q_{p,\gamma}$ for each $p \in P, \gamma \in \Gamma$. Then $\mathcal{A}_{\{q_{p,\gamma}\}}$ represents the configuration $\langle p, \gamma \rangle$. Let $\mathcal{A}$ be this automaton where we leave the set of final states undefined. Let $\mathcal{A}_{grow}$ be the backward weighted-automaton obtained from running poststar on $\mathcal{A}$ (poststar does not need to know the final states). If we restrict the final states in $\mathcal{A}_{grow}$ to be just $q_{p,\gamma}$ (and remove all states that do not have an accepting path to the final state), we obtain a backward*

weighted-automaton $\mathcal{A}_{p,\gamma} = poststar(\langle p, \gamma\rangle) = grow(p, \gamma)$. *We can compute* $\mathcal{A}_{grow}$ *in time* $O_s(|P||\Delta|(|P||\Gamma| + |\Delta|)H)$, *and it has at most* $|P||\Gamma| + |\Delta|$ *states.*

Proof. The proof is similar to the one given for Lem. 6.4.1. Let $\beta \notin \Gamma$ be a new stack symbol. Let $\mathcal{A}_\beta^{p,\gamma}$ be the automaton $\mathcal{A}$ with an extra state q_f and an extra transition $(q_{p,\gamma}, \beta, q_f)$. Let q_f be the final state of this automaton. $\mathcal{A}_\beta^{p,\gamma}$ represents the configuration set $\{\langle p, \gamma\ \beta\rangle\}$. The automaton returned by $poststar(\mathcal{A}_\beta^{p,\gamma})$ would then represent the configuration set $grow(p, \gamma)$ with β appended at the end of the stack. The proof follows from the fact that running *poststar* on $\mathcal{A}_\beta^{p,\gamma}$ is the same as running it on $\mathcal{A}$ (for all p and γ) with the exception of the extra β-transition. □

The advantage of the construction presented in Lem. 6.4.2 is that it just requires a single *poststar* query to compute all of the $\mathcal{A}_{p,\gamma}$, instead of one query for each $p \in P$ and $\gamma \in \Gamma$. Because the standard *poststar* algorithm builds an automaton that is larger than the input automaton (Lem. 2.4.11), $\mathcal{A}_{grow}$ has many fewer states than those in all of the individual $\mathcal{A}_{p,\gamma}$ automata put together.

Fig. 6.4(b) and (c) show the $\mathcal{A}_{grow}$ and $\mathcal{A}_{pop}$ automata for a simple WPDS constructed over the minpath semiring (Defn. 2.4.14).

The idea behind our approach is to use $\mathcal{A}_{pop}$ to simulate the first phase where the PDS pops off stack symbols. With reference to Fig. 6.3, the transducer consumes $\gamma_1 \cdots \gamma_k$ from the input tape. When the transducer (non-deterministically) decides to switch over to the growth phase, and is in state p_{k+1} in $\mathcal{A}_{pop}$ with γ_{k+1} being the next symbol in the input, it passes control to $\mathcal{A}_{p_{k+1},\gamma_{k+1}}$ to start generating the output $u_1 \cdots u_j$. Then it moves into an accept phase where it copies the untouched part of the input stack ($\gamma_{k+2} \cdots \gamma_n$) to the output.

This can be optimized by avoiding a separate copy of $\mathcal{A}_{p,\gamma}$ for each γ. Let $\mathcal{A}_p$ be the same as $\mathcal{A}_{grow}$, but with final states restricted to $\{q_{p,\gamma} \mid \gamma \in \Gamma\}$, and unreachable states appropriately pruned (see Fig. 6.4(d) and (e)). The transducer we construct will non-deterministically guess the stack symbol γ from which the growth phase starts, pass control to $\mathcal{A}_p$, and then

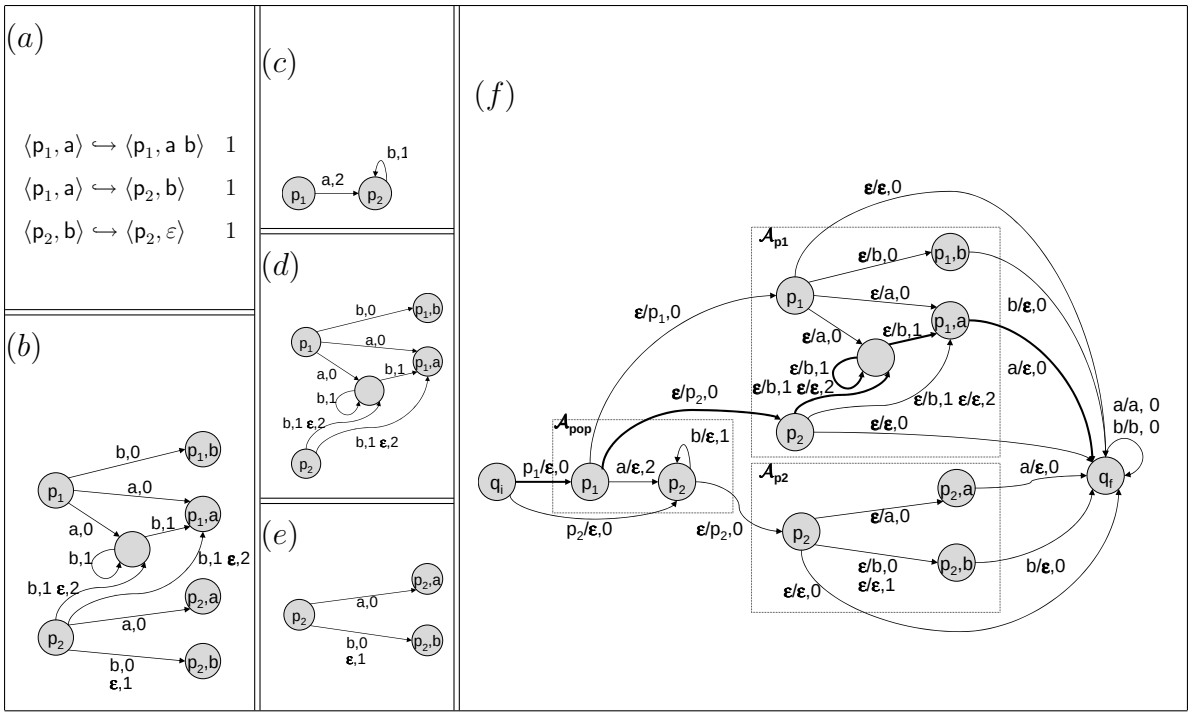


Figure 6.4 Weighted transducer construction: (a) A simple WPDS with the minpath semiring. (b) The $\mathcal{A}_{grow}$ automaton. Edges are labeled with their stack symbol and weight. (c) The $\mathcal{A}_{pop}$ automaton. (d) The $\mathcal{A}_{p_1}$ automaton obtained from $\mathcal{A}_{grow}$. (e) The $\mathcal{A}_{p_2}$ automaton obtained from $\mathcal{A}_{grow}$. The unnamed state in (c) and (d) is an extra state added by the *poststar* algorithm used in Lem. 6.4.2. (f) The weighted transducer. The boxes represent "copies" of $\mathcal{A}_{pop}$, $\mathcal{A}_{p_1}$ and $\mathcal{A}_{p_2}$ as required by steps 2 and 3 of the construction. The transducer paths that accept input $(p_1\ a)$ and output $(p_2\ b^n)$, for $n \geq 2$, with weight n are highlighted in bold.

verify that the guess was correct when it reaches the final state $q_{p,\gamma}$ in $\mathcal{A}_p$. As a result, we just need $|P|$ copies of $\mathcal{A}_{grow}$.

Note that $\mathcal{A}_{pop}$ is a forward-weighted automaton, whereas $\mathcal{A}_{grow}$ is a backward-weighted automaton. Therefore, when we mix them together to build a transducer, we must allow it to switch directions for computing the weight of a path. Consider Fig. 6.3; a PDS rule sequence consumes the input configuration from left to right (in the pop phase), but produces the output stack configuration u from right to left (as it pushes symbols on the stack). Because we need the transducer to output $u_1 \cdots u_j$ from left to right, we need to switch directions for computing the weight of a path. For this, we define *partitioned* transducers.

Definition 6.4.3. *A **partitioned weighted finite-state transducer** τ is a tuple $(Q, \{Q_i\}_{i=1}^2, \mathcal{S}, \Sigma_i, \Sigma_o, \lambda, I, F)$ where Q is a finite set of states, $\{Q_1, Q_2\}$ is a partition of Q, $\mathcal{S} = (D, \oplus, \otimes, \overline{0}, \overline{1})$ is a bounded idempotent semiring, Σ_i and Σ_o are input and output alphabets, $\lambda \subseteq Q \times D \times (\Sigma_i \cup \{\varepsilon\}) \times (\Sigma_o \cup \{\varepsilon\}) \times Q$ is the transition relation, $I \subseteq Q_1$ is the set of initial states, and $F \subseteq Q_2$ is the set of final states. We restrict the transitions that cross the state partition: if $(q, w, a, b, q') \in \lambda$ and $q \in Q_l, q' \in Q_k$ and $l \neq k$, then $l = 1, k = 2$ and $w = \overline{1}$. Given a state $q \in I$, the transducer accepts a string $\sigma_i \in \Sigma_i^*$ with output $\sigma_o \in \Sigma_o^*$ if there is a path from state q to a final state that takes input σ_i and outputs σ_o.*

For a path η that goes through states $q_1, \cdots, q_m$, such that the weight of the i^{th} transition is w_i, and all states q_i are in Q_j for some j, then the weight of this path $v(\eta)$ is $w_1 \otimes w_2 \otimes \cdots \otimes w_m$ if $j = 1$ and $w_m \otimes w_{m-1} \otimes \cdots \otimes w_1$ if $j = 2$, i.e., the state partition determines the direction in which we perform extend. For a path η that crosses partitions, i.e., $\eta = \eta_1\eta_2$ such that each η_j is a path entirely inside Q_j, then $v(\eta) = v(\eta_1) \otimes v(\eta_2)$.

In this chapter, we refer to partitioned weighted transducers as weighted transducers, or simply transducers when there is no possibility of confusion. Note that when the extend operator is commutative, partitioning is unnecessary.

Let $St(\mathcal{A})$ denote the set of states of an automaton $\mathcal{A}$. Because each of $\mathcal{A}_{pop}$ and $\mathcal{A}_p$ have P as a subset of their set of states, we distinguish them by referring to a state $q \in St(\mathcal{A}_{pop})$ by q_{pop} and $q \in St(\mathcal{A}_p)$ by q_p.

Given a WPDS $\mathcal{W}$, we construct the desired weighted transducer $\tau_{\mathcal{W}}$ using the steps given below. $\tau_{\mathcal{W}}$ has states $\{q_i, q_f\} \cup St(\mathcal{A}_{pop}) \cup (\bigcup_{p \in P} St(\mathcal{A}_p))$, input alphabet $P \cup \Gamma$, output alphabet $P \cup \Gamma$, weight domain the same as $\mathcal{W}$, initial state q_i, and final state q_f. Its state partition is $Q_1 = \{q_i\} \cup St(\mathcal{A}_{pop})$ and $Q_2 = \{q_f\} \cup (\bigcup_{p \in P} St(\mathcal{A}_p))$. The part of the transducer contained in Q_1 simulates the pop phase, and the part contained in Q_2 simulates the growth phase, including the part where the untouched part of the stack is copied to the output tape. Transitions to $\tau_{\mathcal{W}}$ are added as follows (an example is given in Fig. 6.4):

1. For each state $p \in P$, add the transition $(q_i, p/\varepsilon, p_{pop})$ with weight $\overline{1}$ to $\tau_{\mathcal{W}}$.

2. For each transition $(p^1_{pop}, \gamma, p^2_{pop})$ with weight w in $\mathcal{A}_{pop}$ add the transition $(p^1_{pop}, (\gamma/\varepsilon), p^2_{pop})$ with the same weight to $\tau_{\mathcal{W}}$, i.e., copy over $\mathcal{A}_{pop}$.

3. For each transition (q_p, γ', q'_p) in each automaton $\mathcal{A}_p$ add the transition $(q_p, (\varepsilon/\gamma'), q'_p)$ with the same weight to $\tau_{\mathcal{W}}$, i.e., copy over each of the $\mathcal{A}_p$.

4. For each $q, q' \in P$, add the transition $(q_{pop}, (\varepsilon/q'), q'_p)$ with weight $\overline{1}$ to $\tau_{\mathcal{W}}$. This transition permits a switch from the pop phase to the growth phase. At this point, we just know that the growth phase begins in state q and ends in state q'. This step guesses the stack symbol from which the growth phase starts. The next step verifies that our guess was correct.

5. For each final state $q_{p,\gamma} \in St(\mathcal{A}_p)$, add the transition $(q_{p,\gamma}, (\gamma/\varepsilon), q_f)$ with weight $\overline{1}$ to $\tau_{\mathcal{W}}$. This transition verifies that γ was on the input tape, and we just completed the growth phase starting from γ.

6. For each $p, q \in P$, add the transition $(q_p, (\varepsilon/\varepsilon), q_f)$ with weight $\overline{1}$ to $\tau_{\mathcal{W}}$. This transition allows us to skip the growth phase by going directly to the final state.

7. For each $\gamma \in \Gamma$, add the transition $(q_f, (\gamma/\gamma), q_f)$ with weight $\overline{1}$ to $\tau_{\mathcal{W}}$. This part of the transducer copies over the untouched part of the input tape to the output tape.

Theorem 6.4.4. *When the transducer $\tau_{\mathcal{W}}$, as constructed above, is given input $(p\ z)$, $p \in P, z \in \Gamma^*$, then the combine over the values of all paths in $\tau_{\mathcal{W}}$ that output the string $(p'\ z')$ is precisely* $\mathrm{IJOP}(\{\langle p, z\rangle\}, \{\langle p', z'\rangle\})$. *Moreover, this transducer can be constructed in time $O_s(|P||\Delta|(|P||\Gamma| + |\Delta|)H)$, has at most $|P|^2|\Gamma| + |P||\Delta|$ states and at most $|P|^2|\Delta|^2$ transitions.*

Proof. The proof is based on the observation made in Fig. 6.3. Suppose that we have a path in the PDS transition relation from $\langle p, \gamma_1\gamma_2\cdots\gamma_n\rangle$ to $\langle p_{k+1}, u\gamma_{k+2}\cdots\gamma_n\rangle$ that can be broken down as shown in Fig. 6.5. Then in the transducer, we can take the path starting at q_i that first takes the transition $(q_i, (p/\varepsilon), p_{pop})$ (Step 1 of the construction)

$$
\begin{array}{lll}
\langle p, \gamma_1\gamma_2\cdots\gamma_n\rangle & \Rightarrow^* & \langle p_1, \gamma_2\cdots\gamma_n\rangle \qquad\quad w_1 \\
& \Rightarrow^* & \langle p_2, \gamma_3\cdots\gamma_n\rangle \qquad\quad w_2 \\
& \Rightarrow^* & \cdots \\
& \Rightarrow^* & \langle p_k, \gamma_{k+1}\cdots\gamma_n\rangle \qquad\quad w_k \\
& \Rightarrow^* & \langle p_{k+1}, u\gamma_{k+2}\cdots\gamma_n\rangle \quad w_{k+1}
\end{array}
$$

Figure 6.5 A path in the PDS's transition relation with corresponding weights of each step.

and moves into state p of $\mathcal{A}_{pop}$. Then it successively takes the transitions $(p_1, (\gamma_2/\varepsilon), p_2)$, $(p_2, (\gamma_3/\varepsilon), p_3), \cdots, (p_{k-1}, (\gamma_k/\varepsilon), p_k)$ (Step 2), all the time staying inside $\mathcal{A}_{pop}$. If the weight of the i^{th} such transition is w^i, then $w_i \sqsubseteq w^i$. This follows from Lem. 6.4.1. Next, the transducer can take transition $(p_k, (\varepsilon/p_{k+1}), p_{k+1})$ (Step 4) and move into $\mathcal{A}_{p_k}$. Then it can take a path that outputs u and move into state $q_{p_k,\gamma_{k+1}}$. There is one such path because $\mathcal{A}_{p_k}$ can accept u starting in state p_{k+1} (representing the configuration $\langle p_{k+1}, u\rangle$) when the final state is $q_{p_k,\gamma_{k+1}}$ (Lem. 6.4.2). Moreover, $w_{k+1} \sqsubseteq$ the combine of weights of all such paths in the transducer. After this, the transducer can take transition $(q_{p_{k+1},\gamma_{k+1}}, (\gamma_{k+1}/\varepsilon), q_f)$ (Step 5) and copy the stack $(\gamma_{k+2}\cdots\gamma_n)$ on to the output tape in the final state q_f (Step 7). The path we just described took input $\langle p, z\rangle = (p\ \gamma_1\gamma_2\cdots\gamma_n)$ and output $\langle p', z'\rangle = (p_{k+1}\ u\gamma_{k+2}\cdots\gamma_n)$ as required, and the weight of the path shown in Fig. 6.5 $(w_1 \otimes w_2 \otimes \cdots \otimes w_{k+1})$ is $\sqsubseteq$ combine of weights of all paths in the transducer with this behavior. Note that there is a corresponding path in the transducer (that uses transitions inserted in Step 6) when the path shown in Fig. 6.5 has no growth phase. Thus, $\text{IJOP}(\langle p, z\rangle, \langle p', z'\rangle) \sqsubseteq \tau_{\mathcal{W}}((p\ z), (p'\ z'))$.

To argue the other direction, the reasoning is similar. A path in the transducer must start in state q_i, then move into $\mathcal{A}_{pop}$, then into $\mathcal{A}_p$ (for some $p \in P$) and then move to state q_f. Keeping track of the input and output required for this path, we can build the WPDS path as in Fig. 6.5. Using Lemmas 6.4.1 and 6.4.2, $\text{IJOP}(\langle p, z\rangle, \langle p', z'\rangle) \sqsupseteq$ the weight of such a path in the transducer. Thus, $\text{IJOP}(\langle p, z\rangle, \langle p', z'\rangle) \sqsupseteq \tau_{\mathcal{W}}((p\ z), (p'\ z'))$. □

Usually the WPDSs used for modeling programs have $|P| = 1$ and $|\Gamma| < |\Delta|$. In that case, constructing a transducer has similar complexity and size as running a single *poststar* query.

6.5 Composing Weighted Transducers

Composition of unweighted transducers is straightforward, but this is not the case with weighted transducers. For a weighted transducer τ, let $\mathcal{L}(\tau)$ be a weighted relation (Defn. 2.4.15) that contains (c_i, c_o, w) if and only if w is the combine of the weights of all paths in τ that take input c_i and output c_o.

Composition of weighted transducers is defined as follows: given two weighted transducers τ_1 and τ_2, construct another weighted transducer τ_3 such that $\mathcal{L}(\tau_3) = \mathcal{L}(\tau_1); \mathcal{L}(\tau_2)$, where the composition operator ";" denotes composition of weighted relations (Defn. 2.4.15).

We begin with a slightly simpler problem on weighted automata. The machinery that we develop for this problem will be used for composing weighted transducers.

6.5.1 The Sequential Product of Two Weighted Automata

Given forward-weighted automata $\mathcal{A}_1$ and $\mathcal{A}_2$, we define their *sequential product* as another weighted automaton $\mathcal{A}_3$ such that for any configuration c, $\mathcal{A}_3(c) = \mathcal{A}_1(c) \otimes \mathcal{A}_2(c)$. More generally, we want the following identity for any regular set of configurations C: $\mathcal{A}_3(C) = \bigoplus\{\mathcal{A}_3(c) \mid c \in C\} = \bigoplus\{\mathcal{A}_1(c) \otimes \mathcal{A}_2(c) \mid c \in C\}$. (In this section, we assume that configurations consist of just the stack and $|P| = 1$.) This problem is the special case of transducer composition when a transducer only has transitions of the form (γ/γ). For the Boolean weight domain (Defn. 2.4.12), it reduces to unweighted automaton intersection (with words accepted with weight $\overline{0}$ being considered as words not accepted by the automaton).

Note that this is a different version of the weighted-automaton intersection problem that was solved for computing an error projection (Section 5.2). For computing error projections, we only needed to take the sequential product of a forward-weighted automaton (obtained as a result of running *poststar*) with a backward-weighted automaton (obtained as a result of

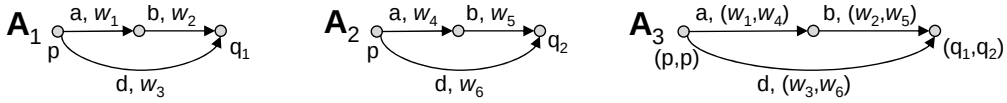


Figure 6.6 Forward-weighted automata. Their final states are q_1, q_2, and (q_1, q_2), respectively.

running *prestar*). Moreover, the result of this operation was a functional automaton, which is not a weighted automaton. In this section, we work with same-direction automata, and the resulting automaton $\mathcal{A}_3$ that we compute is still a valid weighted automaton (but with a different weight domain), thus enabling us to take its sequential product with other weighted automata (over the same weight domain). The solution in this section is more general than the one presented earlier in Section 5.2. We show later, in Section 6.5.3, that this technique also generalizes to take the sequential production of automata with different directions.

To take the sequential product of weighted automata, we start with the algorithm for intersecting unweighted automata. This is done by taking transitions (q_1, γ, q_2) and (q_1', γ, q_2') in the respective automata to produce $((q_1, q_1'), \gamma, (q_2, q_2'))$ in the new automaton. We would like to do the same with weighted transitions: given weights of the matching transitions, we want to compute a weight for the created transition. In Fig. 6.6, intersecting automata $\mathcal{A}_1$ and $\mathcal{A}_2$ produces $\mathcal{A}_3$ (ignore the weights for now). Automaton $\mathcal{A}_3$ should accept (`a b`) with weight $\mathcal{A}_1(\texttt{a b}) \otimes \mathcal{A}_2(\texttt{a b}) = w_1 \otimes w_2 \otimes w_4 \otimes w_5$.

One way of achieving this is to pair the weights while intersecting (as shown in $\mathcal{A}_3$ in Fig. 6.6). Matching the transitions with weights w_1 and w_4 produces a transition with weight (w_1, w_4). For reading off weights, we need to define operations on paired weights. Define extend on pairs ($\otimes_p$) to be componentwise extend ($\otimes$). Then $\mathcal{A}_3(\texttt{a b}) = (w_1, w_4) \otimes_p (w_2, w_5) = (w_1 \otimes w_2, w_4 \otimes w_5)$. Taking an extend of the two components produces the desired answer. Thus, this $\mathcal{A}_3$ together with a read-out operation in the end (that maps a weight pair to a weight) is a first attempt at constructing the sequential product of $\mathcal{A}_1$ and $\mathcal{A}_2$.

Because the number of accepting paths in an automaton may be infinite, one also needs a combine ($\oplus_p$) on paired weights. The natural attempt is to define it componentwise. However, this is not precise. For example, if $C = \{c_1, c_2\}$ then $\mathcal{A}_3(C)$ should be

$(\mathcal{A}_1(c_1) \otimes \mathcal{A}_2(c_1)) \oplus (\mathcal{A}_1(c_2) \otimes \mathcal{A}_2(c_2))$. However, using componentwise combine, we would get $\mathcal{A}_3(C) = \mathcal{A}_3(c_1) \oplus_p \mathcal{A}_3(c_2) = (\mathcal{A}_1(c_1) \oplus \mathcal{A}_1(c_2), \mathcal{A}_2(c_1) \oplus \mathcal{A}_2(c_2))$. Applying the read-out operation (extend of the components) gives four terms $\bigoplus\{(\mathcal{A}_1(c_i) \otimes \mathcal{A}_2(c_j)) \mid 1 \leq i, j \leq 2\}$, which includes cross terms like $\mathcal{A}_1(c_1) \otimes \mathcal{A}_2(c_2)$. The same problem arises also for a single configuration c if $\mathcal{A}_3$ has multiple accepting paths for it.

Under certain circumstances there is an alternative to pairing that lets us compute precisely the desired sequential product of weighted automata:

Definition 6.5.1. *The n^{th} **sequentializable tensor product** (n-STP) of a weight domain $\mathcal{S} = (D, \oplus, \otimes, \overline{0}, \overline{1})$ is defined as another weight domain $\mathcal{S}_t = (D_t, \oplus_t, \otimes_t, \overline{0}_t, \overline{1}_t)$ with operations $\odot : D^n \to D_t$ (called the tensor operation) and $DeTensor : D_t \to D$ such that for all $w_j, w'_j \in D$ and $t_1, t_2 \in D_t$,*

1. $\odot(w_1, w_2, \cdots, w_n) \otimes_t \odot(w'_1, w'_2, \cdots, w'_n) = \odot(w_1 \otimes w'_1, w_2 \otimes w'_2, \cdots, w_n \otimes w'_n)$
2. $DeTensor(\odot(w_1, w_2, \cdots, w_n)) = (w_1 \otimes w_2 \otimes \cdots \otimes w_n)$ *and*
3. $DeTensor(t_1 \oplus_t t_2) = DeTensor(t_1) \oplus DeTensor(t_2)$.

When $n = 2$, we write the tensor operator as an infix operator. Note that because of the first condition in the above definition, $\overline{1}_t = \odot(\overline{1}, \cdots, \overline{1})$ and $\overline{0}_t = \odot(\overline{0}, \cdots, \overline{0})$. Intuitively, one may think of the tensor product of i weights as a kind of generalized i-tuple of those weights. The first condition above implies that extend of weight-tuples must be carried out componentwise. The *DeTensor* operation is the "read-out" operation that puts together the tensor product by taking extend of its components. The third condition is the key. It distinguishes the tensor product from a simple tupling operation. It requires that the *DeTensor* operation distribute over the combine of the tensored domain, which pairing does not satisfy.

If a 2-STP exists for a weight domain, then we can take the product of weighted automata for that domain: if $\mathcal{A}_1$ and $\mathcal{A}_2$ are the two input automata, then for each transition (p_1, γ, q_1) with weight w_1 in $\mathcal{A}_1$, and transition (p_2, γ, q_2) with weight w_2 in $\mathcal{A}_2$,

add the transition $((p_1, p_2), \gamma, (q_1, q_2))$ with weight $(w_1 \odot w_2)$ to $\mathcal{A}_3$. The resulting automaton satisfies the property: $\mathit{DeTensor}(\mathcal{A}_3(c)) = \mathcal{A}_1(c) \otimes \mathcal{A}_2(c)$, and more generally, $\mathit{DeTensor}(\mathcal{A}_3(C)) = \bigoplus\{\mathcal{A}_1(c) \otimes \mathcal{A}_2(c) \mid c \in C\}$. Thus, with the application of the *DeTensor* operation, $\mathcal{A}_3$ behaves like the desired automaton for the product of $\mathcal{A}_1$ and $\mathcal{A}_2$. A similar construction and proof hold for taking the product of n automata at the same time, when an n-STP exists.

The proof follows from the definitions. Let $\mathit{accPath}(\mathcal{A}_i, \sigma_u, w)$ be a predicate that denotes that σ_u is a path in $\mathcal{A}_i$ from its initial state to a final state that accepts the word u, and w is the weight of the path (computed by performing extends of weights on transitions in the path, in order). Because of the way the automata-intersection algorithm is carried out, we know that paths that accept a word u in $\mathcal{A}_3$ are in one-to-one correspondence with paths that accept u in $\mathcal{A}_1$ and paths that accept u in $\mathcal{A}_2$. If σ_u^i is an accepting path for u in $\mathcal{A}_i$ $(i = 1, 2)$, then we can uniquely determine an accepting path $\langle \sigma_u^1, \sigma_u^2 \rangle$ for u in $\mathcal{A}_3$, and vice versa. These properties can be used to prove that if $\mathit{accPath}(\mathcal{A}_3, \langle \sigma_u^1, \sigma_u^2 \rangle, w)$ holds, then $w = w_1 \odot w_2$ such that $\mathit{accPath}(\mathcal{A}_i, \sigma_u^i, w_i)$ hold for $i = 1, 2$. This gives us:

$$
\begin{aligned}
&\mathit{DeTensor}(\mathcal{A}_3(C)) \\
&= \mathit{DeTensor}(\oplus_t\{w \mid \mathit{accPath}(\mathcal{A}_3, \sigma_c, w), c \in C\}) \\
&= \oplus\{\mathit{DeTensor}(w) \mid \mathit{accPath}(\mathcal{A}_3, \sigma_c, w), c \in C\} \\
&= \oplus\{\mathit{DeTensor}(w_1 \odot w_2) \mid \mathit{accPath}(\mathcal{A}_i, \sigma_c^i, w_i), c \in C, \\
&\qquad\qquad\qquad\qquad i = 1, 2, \sigma_c = \langle \sigma_c^1, \sigma_c^2 \rangle\} \\
&= \oplus\{w_1 \otimes w_2 \mid \mathit{accPath}(\mathcal{A}_i, \sigma_c^i, w_i), c \in C, i = 1, 2\} \\
&= \oplus\{\mathcal{A}_1(c) \otimes \mathcal{A}_2(c) \mid c \in C\}
\end{aligned}
$$

With the application of the *DeTensor* operation at the end, $\mathcal{A}_3$ behaves like the desired automaton for the product of $\mathcal{A}_1$ and $\mathcal{A}_2$. A similar construction and proof hold for taking the product of n automata at the same time, when an n-STP exists.

Before generalizing to composition of transducers, we show that n-STP exists, for all n, for a class of weight domains. This class includes the one needed to perform affine-relation analysis (Section 3.5.2).

6.5.2 Sequentializable Tensor Product

We say that a weight domain is commutative if its extend is commutative. STP is easy to construct for commutative domains (tensor is extend, and *DeTensor* is identity). This result is somewhat expected: the difficulty in taking the sequential product of weighted automata $\mathcal{A}_1$ and $\mathcal{A}_2$ is that while the input word (or configuration) is read synchronously by them, their weights have to be read off in sequence (first $\mathcal{A}_1(c)$, then $\mathcal{A}_2(c)$)). When extend is commutative, the weights can be read off synchronously as well.

However, commutative domains are not useful for CBA. Under a commutative extend, interference from other threads can have no effect on the execution of a thread. However, such domains still play an important role in constructing STPs. We show that STPs can be constructed for *matrix domains* built on top of a commutative domain.

Definition 6.5.2. *Let $\mathcal{S}_c = (D_c, \oplus_c, \otimes_c, \overline{0}_c, \overline{1}_c)$ be a commutative weight domain. Then a* ***matrix weight domain*** *on $\mathcal{S}_c$ of order n is a weight domain $\mathcal{S} = (D, \oplus, \otimes, \overline{0}, \overline{1})$ such that D is the set of all matrices of size $n \times n$ with elements from D_c; $\oplus$ on matrices is element-wise $\oplus_c$; $\otimes$ of matrices is matrix multiplication; $\overline{0}$ is the matrix in which all elements are $\overline{0}_c$; $\overline{1}$ is the identity matrix ($\overline{1}_c$ on the primary diagonal and $\overline{0}_c$ everywhere else).*

The reader can verify that $\mathcal{S}$, as defined above, is indeed a bounded idempotent semiring (even when $\mathcal{S}_c$ is not commutative). Let $\mathcal{B}$ be the Boolean weight domain with elements $\overline{1}_{\mathcal{B}}$ and $\overline{0}_{\mathcal{B}}$. The relational weight domain (Defn. 2.4.13) on a set $G = \{g_1, g_2, \cdots, g_{|G|}\}$, is a matrix weight domain on $\mathcal{B}$ of order $|G|$: a binary relation on G can be represented as a matrix such that the (i, j) entry of the matrix is $\overline{1}_{\mathcal{B}}$ if and only if (g_i, g_j) is in the relation. Relational composition then corresponds to matrix multiplication. Similarly, the relational weight domain on $(G, \mathcal{S}_c)$ (Defn. 2.4.16) is a matrix weight domain on $\mathcal{S}_c$ of order $|G|$, provided $\mathcal{S}_c$ is commutative.

The advantage of looking at weights as matrices is that it gives us essential structure to manipulate for constructing the STP. We need the following operation on matrices: the *Kronecker product* [95] of two matrices A and B, of sizes $n_1 \times n_2$ and $n_3 \times n_4$, respectively, is a matrix C of size $(n_1\ n_3) \times (n_2\ n_4)$ such that $C(i,j) = A(i \operatorname{div} n_3, j \operatorname{div} n_4) \otimes B(i \bmod n_3, j \bmod n_4)$, where matrix indices start from zero, and "div" is integer division. It is much easier to understand this definition pictorially (writing $A(i,j)$ as a_{ij}):

$$C = \begin{pmatrix} a_{00}B & \cdots & a_{0(n_2-1)}B \\ \vdots & \ddots & \vdots \\ a_{(n_1-1)0}B & \cdots & a_{(n_1-1)(n_2-1)}B \end{pmatrix}$$

The Kronecker product, written as $\odot$, is an associative operation. Moreover, it is well known that for matrices A, B, C, D with elements that have commutative multiplication, $(A \odot B) \otimes (C \odot D) = (A \otimes C) \odot (B \otimes D)$.

Note that the Kronecker product $m = m_1 \odot m_2$ has all pairwise products of elements from m_1 and m_2. We will rearrange the entries of $m_1 \odot m_2$, using only linear transformations to obtain $m_1 \otimes m_2$. Let m_1 and m_2 be matrices of size $k \times k$, so that m is of size $k^2 \times k^2$.

Let $p_{(l,i)}$ be a matrix of size $k \times k^2$ such that all of its entries are $\overline{0}$, except for the $(l,i)^{\text{th}}$ entry, which is $\overline{1}$. Let $q_{(j,r)}$ be a matrix of size $k^2 \times k$ such that all of its entries are $\overline{0}$, except for the $(j,r)^{\text{th}}$ entry, which is $\overline{1}$. Then the matrix $m^{(l,r)}_{(i,j)} = (p_{(l,i)}\ m\ q_{(j,r)})$, where juxtaposition denotes matrix multiplication, selects the $(i,j)^{\text{th}}$ entry of m and moves it to the $(l,r)^{\text{th}}$ entry of a $k \times k$ matrix. All other entries of the resultant matrix are $\overline{0}$. Moreover, the transformation from m to $m^{(l,r)}_{(i,j)}$ is linear, i.e., it distributes over matrix addition (combine).

Let $\mathbb{Z}_n = \{0, 1, \cdots, n-1\}$. Let S be a subset of $(\mathbb{Z}_{k^2} \times \mathbb{Z}_{k^2}) \times (\mathbb{Z}_k \times \mathbb{Z}_k)$. We define S to map from the index of an entry in $m = m_1 \odot m_2$ to its position in the product $m_1 \otimes m_2$, if it exists. For instance, $m(2k, k-1) = m_1(2,0) \otimes m_2(0, k-1)$, which contributes to the entry $(m_1 \otimes m_2)(2, k-1)$, i.e., it is one of the summands in the sum that defines $(m_1 \otimes m_2)(2, k-1)$. Thus, we include $((2k, k-1), (2, k-1))$ in S. Formally, S is defined as follows:

$$\{((i,j),(l,r)) \mid l = (i \operatorname{div} k), r = (j \bmod k), \text{ and } (j \operatorname{div} k = i \bmod k)\}$$

We are now ready to define the *DeTensor* operation. For any matrix m, define the expression θ_m to be the following:

$$\theta_m = \bigoplus_{((i,j),(l,m))\in S} m_{(i,j)}^{(l,m)}$$

By construction, $\theta_{m_1 \odot m_2} = (m_1 \otimes m_2)$. This can be generalized to multiple matrices to obtain an expression θ_m of the same form as above, such that $\theta_{m_1 \odot \cdots \odot m_n} = m_1 \otimes \cdots \otimes m_n$. The advantage of having an expression of this form is that $\theta_{m_1 \oplus m_2} = \theta_{m_1} \oplus \theta_{m_2}$ (because matrix multiplication distributes over their addition, or combine).

Theorem 6.5.3. *A n-STP exists on matrix domains for all n. If $\mathcal{S}$ is a matrix domain of order r, then its n-STP is a matrix domain of order r^n with the following operations: the tensor product of weights is defined as their Kronecker product, and the DeTensor operation is defined as $\lambda m.\theta_m$.*

The necessary properties for the tensor operation follow from those for Kronecker product and the expression θ_m. Commutativity of the underlying semiring is needed to show property 1 of Defn. 6.5.1: it is necessary to rearrange a product $(w_1 \otimes_c w_1' \otimes_c w_2 \otimes_c w_2')$ as $(w_1 \otimes_c w_2 \otimes_c w_1' \otimes_c w_2')$. This also implies that the tensor operation is associative and one can build weights in the n^{th} STP from a weight in the $(n-1)^{th}$ STP and the original matrix weight domain by taking the Kronecker product. This, in turn, implies that the sequential product of n automata can be built from that of the first $(n-1)$ automata and the last automaton. The same holds for composing n transducers. Therefore, the context-bound can be increased incrementally, and the transducer constructed for $(\Rightarrow^{\text{ec}})^k$ can be used to construct one for $(\Rightarrow^{\text{ec}})^{k+1}$.

The weight domain for ARA (Section 3.5.2) is not quite a matrix weight domain, but it is similar. The weights are sets of matrices over integers, which have a commutative multiplication. Extend is elementwise matrix multiplication and combine is elementwise

matrix addition. Therefore, defining the tensor and *DeTensor* operations as for the matrix domains (but elementwise), we obtain most of the desired properties. However, just as for interprocedural ARA one needed to prove two properties to show that combine and extend can be carried out on the basis instead of the whole vector space, one needs to prove the same for tensor and *DeTensor*: for weights w_1, w_2,

$$\begin{aligned} \beta(w_1 \odot w_2) &= \beta(\beta(w_1) \odot \beta(w_2)) \\ \beta(DeTensor(w_1)) &= \beta(DeTensor(\beta(w_1))) \end{aligned}$$

These properties follow quite trivially from the linearity of Kronecker product and the *DeTensor* operator (both distribute over addition).

6.5.3 Composing Transducers

If our weighted transducers were unidirectional (completely forwards or completely backwards) then composing them would be the same as taking the product of weighted automata: the weights on matching transitions would get tensored together. However, our transducers are partitioned, and have both a forwards component and a backwards component. To handle the partitioning, we need additional operations on weights.

Definition 6.5.4. *Let $\mathcal{S} = (D, \oplus, \otimes, \overline{0}, \overline{1})$ be a weight domain. Then a **transpose** operation on this domain is defined as $(.)^T : D \to D$ such that for all $w_1, w_2 \in D$, $w_1^T \otimes w_2^T = (w_2 \otimes w_1)^T$ and it is its self inverse: $(w_1^T)^T = w_1$. An **n-transposable STP** (TSTP) on $\mathcal{S}$ is defined as an n-STP along with another de-tensor-like operation: $TDeTensor : D^n \to D$ such that $TDeTensor(\odot(w_1, w_2, \cdots, w_n)) = w_1 \otimes w_2^T \otimes w_3 \otimes w_4^T \otimes \cdots w_n'$, where $w_n' = w_n$ if n is odd and w_n^T if n is even.*

TSTPs always exist for matrix domains: the transpose operation is just the matrix-transpose operation, and the *TDeTensor* operation can be defined using an expression similar to that for *DeTensor*. We can use TSTPs to remove the partitioning. Let τ be a partitioned weighted transducer on $\mathcal{S}$, for which a transpose exists, as well as a 2-TSTP. The partitioning on the states of τ naturally defines a partitioning on its transitions as well (a transition is

said to belong to the partition of its source state). Replace weights w_1 in the first (forwards) partition with $(w_1 \odot \overline{1})$, and weights w_2 in the second (backwards) partition with $(\overline{1} \odot w_2^T)$. This gives a completely forwards transducer τ' (without any partitioning). The invariant is that for any sets of configurations S and T, $\tau(S,T)$, which is the combine over all weights with which the transducer accepts $(s,t), s \in S, t \in T$, equals $TDeTensor(\tau'(S,T))$.

This can be extended to compose partitioned weighted transducers. Composing n transducers requires a $2n$-TSTP. First, each transducer is converted to a non-partitioned one over the 2-TSTP domain. Then input/output labels are matched just as for unweighted transducers, and the weights are tensored together.

Theorem 6.5.5. *Given n weighted transducers $\tau_1, \cdots, \tau_n$ on a weight domain with $2n$-TSTP, the above construction produces a weighted transducer τ such that for any sets of configurations S and T, $TDeTensor(\tau(S,T)) = R(S,T)$, where R is the weighted composition of $\mathcal{L}(\tau_1), \cdots, \mathcal{L}(\tau_n)$.*

Putting it all together

Using the construction from Section 6.4, we can construct a transducer τ_i for the (weighted) transition relation of thread t_i, i.e., for $\Rightarrow_i^*$. By extending τ_i to perform the identity transformation on stack symbols of threads other than t_i (using transitions of the form $p \xrightarrow{\gamma/\gamma} p$ with weight $\overline{1}$), we obtain a transducer τ_i^c for $(\Rightarrow_i^c)^*$. Next, a union of these transducers gives τ^{ec}, which represents $\Rightarrow^{\text{ec}}$. Performing the weighted composition of τ^{ec} k times with itself gives us a transducer τ that represents $(\Rightarrow^{\text{ec}})^{k+1}$.

If automaton $\mathcal{A}_S$ represents the set of starting states of a program, $\tau(\mathcal{A}_S)$ provides a weighted automaton $\mathcal{A}$ that describes all reachable states (under the context bound), i.e., the weight $\mathcal{A}(t)$ gives the net transformation in data state in going from S to t ($\overline{0}$ if t is not reachable).

For instance, to see how all this works out, consider a concurrent program with two threads. Furthermore, suppose that the WPDSs for the two threads have a single PDS control state p. In this case, the composition $(\tau_1^e; \tau_2^e)$ represents all behaviors with one

context switch, in which t_1 executes before t_2. We know that $\tau_1^e(\langle p, c_1, c_2\rangle, \langle p, c_1', c_2\rangle) = \mathrm{IJOP}_{t_1}(\{\langle p, c_1\rangle\}, \{\langle p, c_1'\rangle\})$, and similarly for τ_2^e. Next, the transducer $(\tau_1^e; \tau_2^e)$ accepts the composition of the weighted languages of τ_1^e and τ_2^e. Thus,

$$(\tau_1^e; \tau_2^e)(\langle p, c_1, c_2\rangle, \langle p, c_1', c_2'\rangle) = \mathrm{IJOP}_{t_1}(\{\langle p, c_1\rangle\}, \{\langle p, c_1'\rangle\}) \otimes \mathrm{IJOP}_{t_2}(\{\langle p, c_2\rangle\}, \{\langle p, c_2'\rangle\})$$

which exactly characterizes the set of all behaviors with one context switch in which t_1 executes before t_2. Next, consider the transducer $((\tau_1^e; \tau_2^e); (\tau_1^e; \tau_2^e))$. This accepts the input-output pair $(\langle p, c_1, c_2\rangle, \langle p, c_1', c_2'\rangle)$ with the following weight:

$$\begin{array}{rl} & \bigoplus_{c_1'', c_2''} \quad (\tau_1^e; \tau_2^e)(\langle p, c_1, c_2\rangle, \langle p, c_1'', c_2''\rangle) \otimes (\tau_1^e; \tau_2^e)(\langle p, c_1'', c_2''\rangle, \langle p, c_1', c_2'\rangle) \\ = & \bigoplus_{c_1'', c_2''} \quad \left(\begin{array}{rl} & \mathrm{IJOP}_{t_1}(\{\langle p, c_1\rangle\}, \{\langle p, c_1''\rangle\}) \otimes \mathrm{IJOP}_{t_2}(\{\langle p, c_2\rangle\}, \{\langle p, c_2''\rangle\}) \\ \otimes & \mathrm{IJOP}_{t_1}(\{\langle p, c_1''\rangle\}, \{\langle p, c_1'\rangle\}) \otimes \mathrm{IJOP}_{t_2}(\{\langle p, c_2''\rangle\}, \{\langle p, c_2'\rangle\}) \end{array} \right) \end{array}$$

This weight summarizes the next effect of all paths with three context switches in which the threads execute in the order: t_1, t_2, t_1, t_2.

6.6 Implementing CBA

This chapter developed novel machinery that shows how precise CBA can be carried out for various abstractions, including infinite-state abstractions. These algorithms may have practical value, as well. The QR algorithm requires an explicit fan-out proportional to $|G|$ for each context switch, which can be very large. To some extent, this huge complexity is unavoidable, as shown by the following result.

Theorem 6.6.1. *The language $\{\langle M, 0^k, c_1, c_2\rangle \mid M$ is a concurrent PDS, c_1 and c_2 are configurations of M, and $c_1(\Rightarrow^{ec})^{k+1}c_2\}$ is NP-complete.*

Proof. [Sketch] The proof follows from two earlier pieces of work. Ramalingam [80] showed that reachability in multi-threaded programs with synchronization primitives is undecidable by giving a reduction from Post's correspondence problem (PCP) [74]. We also know that

bounded-PCP is NP-complete [34, Problem SR11]. It is easy to see that a program can use shared memory and a bounded number of context switches to simulate a similar number of synchronization steps. Thus, Ramalingam's reduction can be used to give a reduction from bounded-PCP to CBMC. This proves NP-hardness of CBMC.

Next, we show that CBMC is in NP. First, guess a k-tuple of PDS control states (or global states) $(p_1, \cdots, p_k)$, which will represent the history of global states at all the context switches. Next, we run the QR algorithm, but instead of performing a fan-out on the set of all reachable global states, restrict the fan-out at level $i+1$ of the computation tree (Fig. 6.2) to just the states that contain the global state p_i. This means that the computation tree is pruned to one single branch. The running time of QR then becomes polynomial (because it only requires $k+1$ *poststar* queries). If the set of reachable states reported by this algorithm includes the target c_2, then output "yes", otherwise output "no". It is easy to see that this non-deterministic algorithm solves CBMC in polynomial time. □

Note that the analysis of sequential Boolean programs is PSPACE-complete (in the size of the Boolean program; the above result is in terms of the size of the PDS), but tools [85, 6, 37] are able to handle them efficiently, essentially, by using BDDs to encode weights (or binary relations). The fan-out operation of the QR algorithm requires explicit enumeration of global states, which destroys the sharing that existed in the BDDs. Our algorithm, based on transducers, requires no fan-out, and BDD-encoded valuations never need to be enumerated.

We used matrix domains only to prove the existence of STPs. Weights need not be represented using matrices. If binary relations are represented using BDDs, then taking their tensor product reduces to concatenating BDDs (and doubling the number of BDD variables), which is a linear-time operation. Composing k transducers would produce BDDs with k times the variables (a linear increase). The disadvantage of our algorithm is that the transducers we create have $|\Gamma|$ number of states (where Γ is the set of program control locations) and, consequently, the final transducer may have $|\Gamma|^k$ number of states. However, considering the fact that solving CBA just requires one query on this large transducer, we

can use techniques such as building it lazily [64] or exploiting the symmetric structure of compositions (the same transducer is composed each time).

The next chapter builds on some of the ideas discussed in this chapter to design a scalable algorithm for CBA that is able to do verification of real concurrent programs.

6.7 Related Work

CBA of bounded-heap-manipulating Boolean programs is given in [11]. It encodes such Boolean programs using PDSs, and then uses the QR algorithm. The same encoding could be used with either of our (unweighted or weighted) transducer-based algorithms, instead of the QR algorithm.

Reachability analysis of concurrent recursive programs has also been considered in [10, 72, 19]. These tackle the problem by computing over-approximations of the execution paths of the program, whereas here we compute under-approximations (bounded context) of the reachable configurations. Analysis under restricted communication policies (in contrast to shared memory) has also been considered [12, 43].

Constructing transducers. As mentioned in the introduction, a transducer construction for solving reachability in PDSs was given earlier by Caucal [17]. However, the construction was given for prefix-rewriting systems in general and is not accompanied by a complexity result, except for the fact that it runs in polynomial time. Our construction for PDSs, obtained as a special case of the construction given in Section 6.4, is quite efficient. The technique, however, seems to be related. Caucal constructed the transducer by exploiting the fact that the language of the transducer is a union of the relations $(pre^*(\langle p, \gamma \rangle), post^*(\langle p, \gamma \rangle))$ for all $p \in P$ and $\gamma \in \Gamma$, with an identity relation appended onto them to accept the untouched part of the stack. This is similar to our decomposition of PDS paths (see Fig. 6.3). Construction of a transducer for WPDSs has not been considered before. This was crucial for developing an algorithm for general CBA.

The *pop*-function used in Section 6.4 represents summary information about paths, and is similar to the use of composed transformer functions from [25], summary functions from

[88], summary edges from [81], and summary micro-functions from [84]. In all of these cases, information is tabulated that summarizes the net effect of following all possible paths from certain kinds of sources to certain kinds of targets. The path information is pre-computed and added to a structure that is used for answering queries.

One difference between our work and the aforementioned work is that in all of the latter the paths summarized are same-level valid paths (paths in which pushes and pops match as in a language of balanced parentheses), whereas the *pop*-function summarizes paths that result in the net loss of a stack symbol. In this respect, the *pop*-function is more like the "unbalanced-by-1" summarization information used in the simulation technique for testing membership of a string in the language accepted by a 2NDPDA (2-way non-deterministic PDA) [1]. Note that the "unbalanced-by-1" nature of the *pop*-function is what makes it useful in an automaton construction (i.e., the popped symbol corresponds to a letter consumed by the automaton).

Composing transducers. There is a large body of work on weighted automata and weighted transducers in the speech-recognition community [64, 65]. However, the weights in their applications usually satisfy many more properties than those of a semiring, including the existence of an inverse and commutativity of extend. We refrain from making such assumptions.

Tensor products have been used previously in program analysis for combining abstractions [71]. We use them in a different context and for a different purpose. In particular, previous work has used them for combining abstractions that are performed in *lock-step*; in contrast, we use them to stitch together the data state *before* a context switch with the data state *after* a context switch.

Chapter 7

Reducing Concurrent Analysis Under a Context Bound to Sequential Analysis

This chapter presents a second result towards our goal of automatically extending analyses for sequential programs to analyses for concurrent programs under a bound on the number of context switches. Chapter 6 showed that the existence of a tensor-product operation automatically enabled precise CBA of various program models (namely, all those that could be encoded using a weighted pushdown system). In this chapter, we present a more direct way of obtaining algorithms for CBA that does not require tensor products.

Let us recall the existing results on CBA. The decidability of CBA, when each program thread is abstracted as a pushdown system (PDS) was shown in [77]. This result was extended to PDSs with bounded heaps in [11]. Our work, which was described in Chapter 6, extended the result to weighted PDSs (WPDSs). All of this work required devising new algorithms. Moreover, each of the algorithms have certain disadvantages towards realizing a practical implementation.

In the sequential setting, model checkers, such as those described in [6, 85, 37], use symbolic techniques in the form of binary decision diagrams (BDDs) for scalability. With the CBA algorithms of [77, 11], it is not clear if symbolic techniques can be applied. Those algorithms require the enumeration of all reachable states of the shared memory at a context switch. This can potentially be very expensive. However, those algorithms have the nice property that they only consider those states that actually arise during valid (abstract) executions of the model. (We call this *lazy* exploration of the state space.)

The results presented in Chapter 6 extend the algorithm of [77] to use symbolic techniques. However, the disadvantage there is that it requires computing auxiliary information, in the form of a transducer, for exploring the reachable state space. (We call this *eager* exploration of the state space.) The transducer summarizes the effect of executing a thread from any control location to any other control location, and hence may consider many more program behaviors than can actually occur in a valid execution of the program (whence the term "eager").

This contrast between lazy and eager approaches can also be illustrated by considering interprocedural analysis of sequential programs: for a procedure, it is possible to construct a summary for the procedure that describes the effect of executing it for any possible inputs to the procedure (eager computation of the summary). It is also possible to construct the summary lazily (also called partial transfer functions [69]) by only describing the effect of executing the procedure for input states under which it is called during the analysis of the program. The former (eager) approach has been successfully applied to Boolean programs [6], but the latter (lazy) approach is often desirable in the presence of more complex abstractions, especially those that contain pointers (based on the intuition that only a few aliasing scenarios occur during abstract execution). Interprocedural analysis frameworks, like the Sharir and Pnueli tabulation algorithm [88], the Reps-Horwitz-Sagiv graph reachability approach [81], and others [84] are also lazy. The option of switching between eager and lazy exploration exists in some model checkers [6, 50].

Contributions

The work presented in this chapter makes three main contributions. First, we show how to reduce a concurrent program to a sequential one that simulates all its executions for a given number of context switches. This has the following advantages:

- It allows one to obtain algorithms for CBA using different program abstractions. We specialize the reduction to Boolean programs (Section 7.2), PDSs (Section 7.3), symbolic PDSs (Section 7.4), and WPDSs (Section 7.5). The reduction for Boolean programs shows that the use of PDS-based technology, which seemed crucial in previous work, is not necessary: standard interprocedural algorithms [81, 88, 52] can also be used for CBA. Moreover, it allows one to carry over symbolic techniques designed for sequential programs to CBA.

- Our reduction provides a way to harness existing abstraction techniques to obtain new algorithms for CBA. The reduction introduces symbolic constants and `assume` statements. Thus, any sequential analysis that can deal with these two features can be extended to handle concurrent programs as well (under a context bound).

  Symbolic constants are only associated with the shared data in the program. When only a finite amount of data is shared between the threads of a program (e.g., there are only a finite number of locks), *any* sequential analysis, even of programs with pointers or integers, can be extended to perform CBA of concurrent programs. When the shared data is not finite, our reduction still applies; for instance, numeric analyses, such as polyhedral analysis [27], can be applied to CBA of concurrent programs.

- For the case in which a PDS is used to model each thread, we obtain better asymptotic complexity than previous algorithms, just by using the standard PDS algorithms (Section 7.3).

- The reduction shows how to obtain algorithms that scale linearly with the number of threads (whereas previous algorithms scaled exponentially).

Second, we show how to obtain a lazy symbolic algorithm for CBA on Boolean programs (Section 7.6). This combines the best of previous algorithms: the algorithms of [77, 11] are lazy but not symbolic, and the algorithm presented in Chapter 6 is symbolic but not lazy.

Third, we implemented both eager and lazy algorithms for CBA on Boolean programs. We report the scalability of these algorithms on programs obtained from various sources and also show that most bugs can be found in a few context switches (Section 7.7).

The rest of this chapter is organized as follows: Section 7.1 gives a general reduction from concurrent to sequential programs; Section 7.2 specializes the reduction to Boolean programs; Section 7.3 specializes the reduction to PDSs; Section 7.4 specializes the reduction to symbolic PDSs; Section 7.5 specializes the reduction to WPDSs; Section 7.6 gives a lazy symbolic algorithm for CBA on Boolean programs; Section 7.7 reports experiments performed using both eager and lazy versions of the algorithms presented in this chapter; Section 7.8 discusses related work. Proofs can be found in Section 7.9.

7.1 A General Reduction

This section gives a general reduction from concurrent programs to sequential programs under a given context bound. This reduction transforms the non-determinism in control, which arises because of concurrency, to non-determinism on data. (The motivation is that the latter problem is understood much better than the former one.)

The execution of a concurrent program proceeds in a sequence of *execution contexts*, defined as the time between consecutive context switches during which only a single thread has control. We do not consider dynamic creation of threads, and assume that a concurrent program is given as a fixed set of threads, with one thread identified as the starting thread.

Suppose that a program has two threads, T_1 and T_2, and that the context-switch bound is $2K - 1$. Then any execution of the program under this bound will have up to $2K$ execution contexts, with control alternating between the two threads, informally written as $T_1; T_2; T_1; \cdots$. Each thread has control for at most K execution contexts. Consider three consecutive execution contexts $T_1; T_2; T_1$. When T_1 finishes executing the first of these, it gets swapped out and its local state, say l, is stored. Then T_2 gets to run, and when it is swapped out, T_1 has to resume execution from l (along with the global store produced by T_2).

The requirement of resuming from the same local state is one difficulty that makes analysis of concurrent programs hard—during the analysis of T_2, the local state of T_1 has to be remembered (even though it is unchanging). This forces one to consider the cross product of the local states of the threads, which causes exponential blowup when the local state space is finite, and undecidability when the local state includes a stack. An advantage of introducing a context bound is the reduced complexity with respect to the size $|L|$ of the local state space: the algorithms of [77, 11] scale as $\mathcal{O}(|L|^5)$; and the one from Chapter 6 scales as $\mathcal{O}(|L|^K)$. Our algorithm, for PDSs (Section 7.3), is $\mathcal{O}(|L|)$. (Strictly speaking, in each of these, $|L|$ is the size of the local transition system.)

The key observation is the following: for analyzing $T_1; T_2; T_1$, we modify the threads so that we only have to analyze $T_1; T_1; T_2$, which eliminates the requirement of having to drag along the local state of T_1 during the analysis of T_2. For this, we *assume* the effect that T_2 might have on the shared memory, apply it while T_1 is executing, and then *check* our assumption after analyzing T_2.

Consider the general case when each of the two threads have K execution contexts. We refer to the state of shared memory as the *global state*. First, we guess $K-1$ (arbitrary) global states, say $s_1, s_2, \cdots, s_{K-1}$. We run T_1 so that it starts executing from the initial state s_0 of the shared memory. At a non-deterministically chosen time, we record the current global state s'_1, change it to s_1, and resume execution of T_1. Again, at a non-deterministically chosen time, we record the current global state s'_2, change it to s_2, and resume execution of T_1. This continues $K-1$ times. Implicitly, this implies that we assumed that the execution of T_2 will change the global state from s'_i to s_i in its i^{th} execution context. Next, we repeat this for T_2: we start executing T_2 from s'_1. At a non-deterministically chosen time, we record the global state s''_1, we change it to s'_2 and repeat $K-1$ times. Finally, we verify our assumption: we check that $s''_i = s_{i+1}$ for all i between 1 and $K-1$. If these checks pass, we have the guarantee that T_2 can reach state s if and only if the concurrent program can have the global state s after K execution contexts per thread.

The fact that we do not alternate between T_1 and T_2 implies the linear scalability with respect to $|L|$. Because the above process has to be repeated for all valid guesses, our approach scales as $\mathcal{O}(|G|^K)$, where G is the global state space. In general, the exponential complexity with respect to K may not be avoidable because the problem is NP-complete when the input has K written in unary (Thm. 6.6.1). However, symbolic techniques can be used for a practical implementation.

We show how to reduce the above assume-guarantee process into one of analyzing a sequential program. We add more variables to the program, initialized with symbolic constants, to represent our guesses. The switch from one global state to another is made by switching the set of variables being accessed by the program. We verify the guesses by inserting `assume` statements at the end.

7.1.1 The reduction

Consider a concurrent program P with two threads T_1 and T_2 that only has scalar variables (i.e., no pointers, arrays, or heap).[1] We assume that the threads share their global variables, i.e., they have the same set of global variables. Let $\textsc{Var}_G$ be the set of global variables of P. Let $2K-1$ be the bound on the number of context switches.

The result of our reduction is a sequential program P^s. It has three parts, performed in sequence: the first part T_1^s is a reduction of T_1; the second part T_2^s is a reduction of T_2; and the third part, `Checker`, consists of multiple `assume` statements to verify that a correct interleaving was performed. Let L_i be the label preceding the i^{th} part. P^s has the form shown in the first column of Fig. 7.1.

The global variables of P^s are K copies of $\textsc{Var}_G$. If $\textsc{Var}_G = \{\mathtt{x}_1, \cdots, \mathtt{x}_n\}$, then let $\textsc{Var}_G^i = \{\mathtt{x}_1^i, \cdots, \mathtt{x}_n^i\}$. The initial values of $\textsc{Var}_G^i$ are a set of symbolic constants that represent the i^{th} guess s_i. P^s has an additional global variable `k`, which will take values between 1 and $K+1$. It tracks the current execution context of a thread: at any time P^s

[1] Such models are often used in model checking and numeric program analysis.

Program P^s	$\texttt{st} \in T_i$	Checker

$L_1:$ T_1^s;
$L_2:$ T_2^s;
$L_3:$ Checker;
	if $\texttt{k} = 1$ **then**
	$\tau(\texttt{st}, 1)$;
	else if $\texttt{k} = 2$ **then**
	$\tau(\texttt{st}, 2)$;
	$\cdots$
	else if $\texttt{k} = K$ **then**
	$\tau(\texttt{st}, K)$;
	end if
	if $\texttt{k} \leq K$ and $*$ **then**
	k ++;
	end if
	if $\texttt{k} = K + 1$ **then**
	$\texttt{k} = 1$;
	goto L_{i+1}
	end if
		for $i = 1$ to $K - 1$ **do**
		for $j = 1$ to n **do**
		assume $(\texttt{x}_j^i = v_j^{i+1})$;
		end for
		end for

Figure 7.1 The reduction for general concurrent programs under a context bound $2K - 1$. In the second column, $*$ stands for a nondeterministic Boolean value.

can only read and write to variables in $\text{VAR}_G^{\texttt{k}}$. The local variables of T_i^s are the same as those of T_i.

Let $\tau(\texttt{x}, i) = \texttt{x}^i$. If st is a program statement in P, let $\tau(\texttt{st}, i)$ be the statement in which each global variable x is replaced with $\tau(\texttt{x}, i)$, and the local variables remain unchanged. The reduction constructs T_i^s from T_i by replacing each statement st by what is shown in the second column of Fig. 7.1. The third column shows Checker. Variables VAR_G^1 are initialized to the same values as VAR_G in P. Variable $\texttt{x}_j^i$, when $i \neq 1$, is initialized to the symbolic constant v_j^i (which is later referenced inside Checker), and k is initialized to 1.

Because local variables are not replicated, a thread resumes execution from the same local state it was in when it was swapped out at a context switch.

The `Checker` enforces a correct interleaving of the threads. It checks that the values of global variables when T_1 starts its $i+1^{\text{st}}$ execution context are the same as the values produced by T_2 when T_2 finished executing its i^{th} execution context. (Because the execution of T_2^s happens after T_1^s, each execution context of T_2^s is guaranteed to use the global state produced by the corresponding execution context of T_1^s.)

The reduction ensures the following property: when P^s finishes execution, the variables $\textsc{Var}_G^K$ can have a valuation s if and only if the variables $\textsc{Var}_G$ in P can have the same valuation after $2K-1$ context switches.

Symbolic constants

One way to deal with symbolic constants is to consider all possible values for them (eager computation). We show instances of this strategy for Boolean programs (Section 7.2) and for PDSs (Section 7.3). Another way is to lazily consider the set of values they may actually take during the (abstract) execution of the concurrent program, i.e., only consider those values that pass the `Checker`. We show an instance of this strategy for Boolean programs (Section 7.6).

7.1.2 Multiple threads

If there are n threads, $n > 2$, then a precise reasoning for K context switches would require one to consider all possible thread schedulings, e.g., $(T_1;T_2;T_1;T_3)$, $(T_1;T_3;T_2;T_3)$, etc. There are $\mathcal{O}((n-1)^K)$ such schedulings. Previous analyses [77, 11] enumerate explicitly all these schedulings, and thus have $\mathcal{O}((n-1)^K)$ complexity even in the best case. We avoid this exponential factor as follows: we only consider the round-robin thread schedule $T_1;T_2;\cdots T_n;T_1;T_2;\cdots$ for CBA, and bound the length of this schedule instead of bounding the number of context switches. Because a thread is allowed to perform no steps during its execution context, CBA still considers other schedules. For example, when $n=3$, the

schedule $T_1; T_2; T_1; T_3$ will be considered while analyzing a round-robin schedule of length 6 (in the round-robin schedule, T_3 does nothing in its first execution context, and T_2 does nothing in its second execution context).

Setting the bound on the length of the round-robin schedule to nK allows CBA to consider all thread schedulings with K context switches (as well as some schedulings with more than K context switches). Under such a bound, a schedule has K execution contexts per thread.

The reduction for multiple threads proceeds in a similar way to the reduction for two threads. The global variables are copied K times. Each thread T_i is transformed to T_i^s, as shown in Fig. 7.1, and P^s calls the T_i^s in sequence, followed by `Checker`. `Checker` remains the same (it only has to check that the state after the execution of T_n^s agrees with the symbolic constants).

The advantages of this approach are as follows: (*i*) we avoid an explicit enumeration of $\mathcal{O}((n-1)^K)$ thread schedules, thus, allowing our analysis to be more efficient in the common case; (*ii*) we explore more of the program behavior with a round-robin bound of nK than with a context-switch bound of K; and (*iii*) the cost of analyzing the round-robin schedule of length nK is about the same (in fact, better) than what previous analyses take for exploring one schedule with a context bound of K (see Section 7.3). These advantages allow our analysis to scale much better in the presence of multiple threads than previous analyses. Our implementation tends to scale linearly with respect to the number of threads (Section 7.7).

In the rest of this chapter, we only consider two threads because the extension to multiple threads is straightforward for round-robin scheduling.

7.1.3 Ability of the reduction to harness different analyses for CBA

The reduction introduces `assume` statements and symbolic constants. Any sequential analysis that can deal with these two features can be extended to handle concurrent programs as well (under a context bound).

Any abstraction prepared to interpret program conditions can also handle `assume` statements. Certain analysis, such as affine-relation analysis (ARA) over integers cannot make use of the reduction: the presence of `assume` statements makes the ARA problem undecidable [67]. The reduction presented in Section 7.5 avoids introducing `assume` statements.

It is harder to make a general claim about whether most sequential analyses can handle symbolic constants. A variable initialized with a symbolic constant can be treated safely as an uninitialized variable; thus, any analysis that considers all possible values for an uninitialized variable can, in some sense, accommodate symbolic constants.

Another place where symbolic constants are used in sequential analyses is to construct summaries for recursive procedures. Eager computation of a procedure summary is similar to analyzing the procedure while assuming symbolic values for the parameters of the procedure.

It is easy to see that our reduction applies to concurrent programs that only share finite-state data. In this case, the symbolic constants can only take on a finite number of values. Thus, any sequential analysis can be extended for CBA merely by enumerating all their values (or considering them lazily using techniques similar to the ones presented in Section 7.6). This implies that sequential analyses of programs with pointers, arrays, and/or integers can be extended to perform CBA of such programs when only finite-state data (e.g., a finite number of locks) is shared between the threads.

The reduction also applies when the shared data is not finite-state, although in this case the values of symbolic constants cannot be enumerated. For instance, the reduction can take a concurrent numeric program (defined as one having multiple threads, each manipulating some number of potentially unbounded integers), and produce a sequential numeric program.

Then most numeric analyses, such as polyhedral analysis [27], can be applied to the program. Such analyses are typically able to handle symbolic constants.

7.2 The Reduction for Boolean Programs

For ease of exposition, we assume that all procedures of a Boolean program have the same number of local variables. Furthermore, the global variables can have any value when program execution starts, and similarly for the local variables when a procedure is invoked. Let G be the set of valuations of the global variables, and L be the set of valuations of the local variables. A program *data-state* is an element of $G \times L$. Each program statement `st` of the Boolean program can be associated with a relation $[\![\texttt{st}]\!] \subseteq (G \times L) \times (G \times L)$ such that $(g_0, l_0, g_1, l_1) \in [\![\texttt{st}]\!]$ when the execution of `st` on the state (g_0, l_0) can lead to the state (g_1, l_1).

7.2.1 Analysis of sequential Boolean programs

In this section, we recall analyses for sequential Boolean programs. The goal of analyzing Boolean programs is to compute the set of data-states that can reach a program node. This is done using the rules shown in Fig. 7.2 [6]. These rules follow standard interprocedural analyses [81, 88]. Let entry(`f`) denote the entry node of procedure `f`, *proc*(n) denote the procedure that contains node n, ep(n) denote entry(*proc*(n)); let exitnode(n) denote a predicate on nodes that is true when n is the exit node of its procedure. Let Pr be the set of procedures of the program, which includes a distinguished procedure `main`. The rules of Fig. 7.2 compute three types of relations: $H_n(g_0, l_0, g_1, l_1)$ denotes the fact that if (g_0, l_0) is the data state at entry(n), then the data state (g_1, l_1) can reach node n; $S_{\texttt{f}}$ is the summary relation for procedure `f`, which captures the net transformation that an invocation of the procedure can have on the global state; R_n is the set of data states that can reach node n. All relations are initialized to be empty.

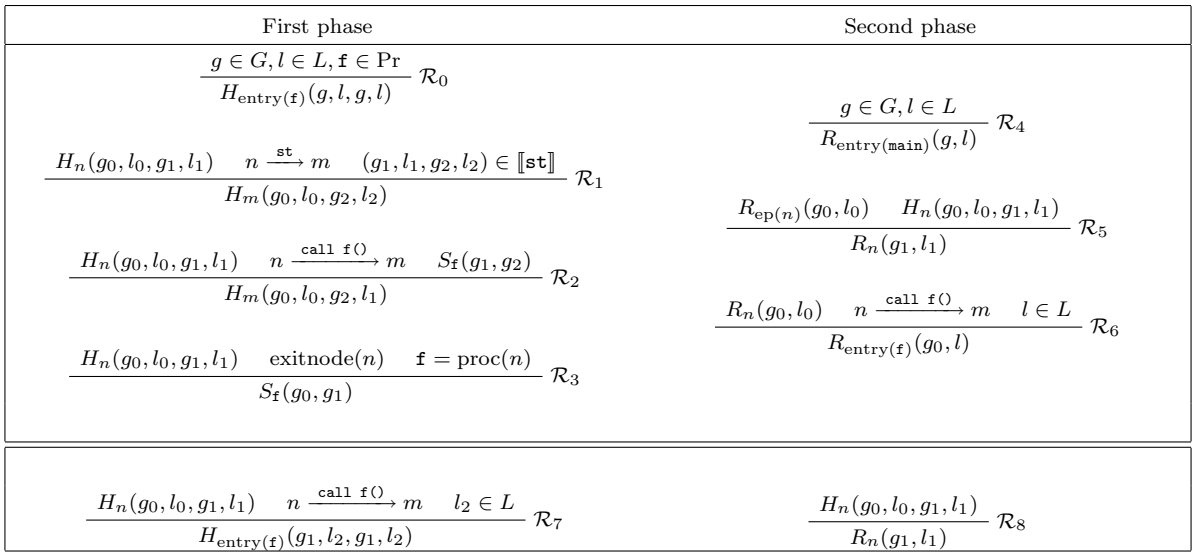


Figure 7.2 Rules for the analysis of Boolean programs.

Eager analysis. Rules $\mathcal{R}_0$ to $\mathcal{R}_6$ describe an eager analysis. The analysis proceeds in two phases. In the first phase, the rules $\mathcal{R}_0$ to $\mathcal{R}_3$ are used to saturate the relations H and S. In the next phase, this information is used to build the relation R using rules $\mathcal{R}_4$ to $\mathcal{R}_6$.

Lazy analysis. Let rule $\mathcal{R}'_0$ be the same as $\mathcal{R}_0$ but restricted to just the `main` procedure. Then the rules $\mathcal{R}'_0, \mathcal{R}_1, \mathcal{R}_2, \mathcal{R}_3, \mathcal{R}_7, \mathcal{R}_8$ describe a lazy analysis. The rule $\mathcal{R}_7$ restricts the analysis of a procedure to only those states it is called in. As a result, the second phase gets simplified and consists of only the rule $\mathcal{R}_8$.

Practical implementations [6, 50] use BDDs to encode each of the relations H, S, and R and the rule applications are changed into BDD operations. For example, rule $\mathcal{R}_1$ is simply the relational composition of relations H_n and $[\![\mathtt{st}]\!]$, which can be implemented efficiently using BDDs.

7.2.2 Context-bounded analysis of concurrent Boolean programs

Concurrent Boolean programs were defined in Section 6.1. We can apply the reduction presented in Section 7.1 on a concurrent Boolean program to obtain a sequential Boolean program by making the following changes to the reduction: (*i*) the variable `k` is modeled

using a vector of $\log(K)$ Boolean variables, and the increment operation is implemented using a simple Boolean circuit on these variables; (*ii*) the **if** conditions are modeled using `assume` statements; and (*iii*) the symbolic constants are modeled using additional (uninitialized) global variables that are not modified in the program. Running any sequential analysis algorithm, and projecting out the values of the K^{th} set of global variables from R_n gives the precise set of reachable global states at node n in the concurrent program.

The worst-case complexity of analyzing a Boolean program P is bounded by $\mathcal{O}(|P||G|^3|L|^2)$, where $|P|$ is the number of program statements. Thus, using our approach, a concurrent Boolean program P_c with m threads, and K execution contexts per thread (with round-robin scheduling), can be analyzed in time $\mathcal{O}(K|P_c|(K|G|^K)^3|L|^2|G|^K)$: the size of the sequential program obtained from P_c is $K|P_c|$; it has the same number of local variables, and its global variables have $K|G|^K$ number of valuations. Additionally, the symbolic constants can take $|G|^K$ number of valuations, adding an extra multiplicative factor of $|G|^K$. The analysis scales linearly with the number of threads ($|P_c|$ is $\mathcal{O}(m)$).

This reduction actually applies to any model that works with finite-state data, which includes Boolean programs with references [8, 75]. In such models, the heap is assumed to be bounded in size. The heap is included in the global state of the program, hence, our reduction would create multiple copies of the heap, initialized with symbolic values. Our experiments (Section 7.7) used such models.

Such a process of duplicating the heap can be expensive when the number of heap configurations that actually arise in the concurrent program is very small compared to the total number of heap configurations possible. The lazy version of our algorithm (Section 7.6) addresses this issue.

7.3 The Reduction for PDSs

The motivation for presenting the reduction for PDSs is that it allows one to apply the numerous algorithms developed for PDSs to concurrent programs under a context bound.

For each $\langle p, \gamma\rangle \hookrightarrow \langle p', u\rangle \in (\Delta_1 \cup \Delta_2)$ and for all $p_i \in P, k \in \{1, \cdots, K\}$:		
$\langle (k, p_1, \cdots, p_{k-1}, p, p_{k+1}, \cdots, p_K), \gamma\rangle$	$\hookrightarrow$	$\langle (k, p_1, \cdots, p_{k-1}, p', p_{k+1}, \cdots, p_K), u\rangle$

For each $\gamma \in \Gamma_j$ and for all $p_i \in P, k \in \{1, \cdots, K\}$:		
$\langle (k, p_1, \cdots, p_K), \gamma\rangle$	$\hookrightarrow$	$\langle (k+1, p_1, \cdots, p_K), \gamma\rangle$
$\langle (K+1, p_1, \cdots, p_K), \gamma\rangle$	$\hookrightarrow$	$\langle (1, p_1, \cdots, p_K), e_{j+1}\ \gamma\rangle$

Figure 7.3 PDS rules for $\mathcal{P}_s$.

For instance, one can use backward analysis of PDSs to get a backward analysis on the concurrent program, or even compute error projections (Chapter 5) of concurrent programs.

Concurrent PDSs were defined in Section 6.1. In this section, we only consider concurrent PDSs with two threads. Let the concurrent PDS be $(\mathcal{P}_1, \mathcal{P}_2)$, where $\mathcal{P}_i = (P, \Gamma_i, \Delta_i)$. Let $\Rightarrow_i$ be the transition system of $\mathcal{P}_i$ and let $\Rightarrow_i^c$ be its extension to configurations of the concurrent PDS (also defined in Section 6.1). Let the context-switch bound be $2K - 1$, so that each thread gets K chances to execute.

We will reduce $(\mathcal{P}_1, \mathcal{P}_2)$ to a single PDS $\mathcal{P}_s = (P_s, \Gamma_s, \Delta_s)$. Let P_s be the set of all $K + 1$ tuples whose first component is a number between 1 and K, and the rest are from the set P, i.e., $P_s = \{1, \cdots, K\} \times P \times P \times \cdots \times P$. This set relates to the reduction from Section 7.1 as follows: an element $(k, p_1, \cdots, p_K) \in P_s$ represents that the value of the variable `k` is k; and p_i encodes a valuation of the variables $\textsc{Var}_G^i$. When $\mathcal{P}_s$ is in such a state, its rules only modify p_k.

Let $e_i \in \Gamma_i$ be the starting node of the i^{th} thread. Let Γ_s be the disjoint union of Γ_1, Γ_2 and an additional symbol $\{e_3\}$. $\mathcal{P}_s$ does not have an explicit checking phase. The rules Δ_s are defined in Fig. 7.3.

We deviate slightly from the reduction presented in Section 7.1 by changing the **goto** statement, which passes control from the first thread to the second, into a procedure call. This ensures that the stack of the first thread is left intact when control is passed to the next thread. Furthermore, we assume that the PDSs cannot empty their stacks, i.e., it is

not possible that $\langle p, e_1\rangle \Rightarrow^*_{\mathcal{P}_1} \langle p', \varepsilon\rangle$ or $\langle p, e_2\rangle \Rightarrow^*_{\mathcal{P}_2} \langle p', \varepsilon\rangle$ for all $p, p' \in P$ (in other words, the `main` procedure should not return). This can be enforced by introducing new symbols e'_i, e''_i in $\mathcal{P}_i$ such that e'_i calls e_i, pushing e''_i on the stack, and ensuring that no rule can fire on e''_i.

Theorem 7.3.1. *Starting execution of the concurrent PDS $(\mathcal{P}_1, \mathcal{P}_2)$ from the state $\langle p, e_1, e_2\rangle$ can lead to the state $\langle p', c_1, c_2\rangle$ under the transition system $((\Rightarrow^c_1)^*; (\Rightarrow^c_2)^*)^K$ if and only if there exist states $p_2, \cdots, p_K \in P$ such that $\langle (1, p, p_2, \cdots, p_K), e_1\rangle \Rightarrow_{\mathcal{P}_s} \langle (1, p_2, p_3, \cdots, p_K, p'), e_3\ c_2\ c_1\rangle$.*

Note that the checking phase is implicit in the statement of Thm. 7.3.1. (One can also make the PDS $\mathcal{P}_s$ have an explicit checking phase, starting at node e_3.) A proof is given in Section 7.9.

Complexity. Using our reduction, one can find the set of all reachable configurations of the concurrent PDS $(\mathcal{P}_1, \mathcal{P}_2)$ in time $\mathcal{O}(K^2|P|^{2K}|Proc||\Delta_1 + \Delta_2|)$, where $|Proc|$ is the number of procedures in the concurrent PDS[2] (see Section 7.9). Using backward reachability algorithms, one can verify if a given configuration in reachable in time $\mathcal{O}(K^3|P|^{2K}|\Delta_1 + \Delta_2|)$. Both these complexities are asymptotically better than those of previous algorithms for PDSs [77], including the one presented in Chapter 6. Note that the complexity for backward reachability is linear in the program size $|\Delta_1 + \Delta_2|$.

A similar reduction works for multiple threads as well (under round-robin scheduling). Moreover, the complexity of finding all reachable states under a bound of nK with n threads, using a standard PDS reachability algorithm, is $\mathcal{O}(K^3|P|^{4K}|Proc||\Delta|)$, where $|\Delta| = \Sigma_{i=1}^n|\Delta_i|$ is the total number of rules in the concurrent PDS.

This reduction produces a large number of rules ($\mathcal{O}(|P|^K|\Delta|)$) in the resultant PDS, but we can leverage work on *symbolic* PDSs (SPDSs) [85] to obtain a symbolic implementation.

[2]The number of procedures of a PDS is defined as the number of symbols appearing as the first of the two stack symbols on the right-hand side of a call rule.

7.4 The Reduction for Symbolic PDSs

A *symbolic pushdown system* (SPDS) is a triple $(\mathcal{P}, G, val)$, where $\mathcal{P} = (\{p\}, \Gamma, \Delta)$ is a single-state PDS, G is a finite set, and $val : \Delta \to (G \times G)$ assigns a binary relation on G to each PDS rule. val is extended to a sequence of rules as follows: $val([r_1, \cdots, r_n]) = val(r_1); val(r_2); \cdots ; val(r_n)$. For a rule sequence $\sigma \in \Delta^*$ and PDS configurations c_1 and c_2, we say $c_1 \Rightarrow^\sigma c_2$ if applying those rules on c_1 results in c_2. The reachability question is extended to computing the join-over-all-paths (JOP) value between two sets of configurations:

$$\mathrm{JOP}(C_1, C_2) = \bigcup\{val(\sigma) \mid c_1 \Rightarrow^\sigma c_2, c_1 \in C_1, c_2 \in C_2\}$$

PDSs and SPDSs have equivalent theoretical power; each can be converted to the other. SPDSs are used for efficiently analyzing PDSs. For a PDS $\mathcal{P} = (P, \Gamma, \Delta)$, one constructs an SPDS as follows: it consists of a PDS $(\{p\}, \Gamma, \Delta')$ and $G = P$. The rules Δ' and their assigned relations are defined as follows: for each $\gamma \in \Gamma, u \in \Gamma^*$, include rule $\langle p, \gamma\rangle \hookrightarrow \langle p, u\rangle$ with the relation $\{(p_1, p_2) \mid \langle p_1, \gamma\rangle \hookrightarrow \langle p_2, u\rangle \in \Delta\}$, if the relation is non-empty. The SPDS captures all state changes in the relations associated with the rules. Under this conversion: $\langle p_1, u_1\rangle \Rightarrow_{\mathcal{P}} \langle p_2, u_2\rangle$ if and only if $(p_1, p_2) \in \mathrm{JOP}(\{\langle p, u_1\rangle\}, \{\langle p, u_2\rangle\})$.

The advantage of using SPDSs is that the relations can be encoded using BDDs, and operations such as relational composition and union can be performed efficiently using BDD operations. This allows scalability to large data-state spaces [85]. (SPDSs can also encode part of the local state in the relations, but we do not consider that issue in this section.)

The reverse construction can be used to encode an SPDS as a PDS: given an SPDS $((\{p\}, \Gamma, \Delta), G, val)$, construct a PDS $\mathcal{P} = (G, \Gamma, \Delta')$ with rules: $\{\langle g_1, \gamma\rangle \hookrightarrow \langle g_2, u\rangle \mid r = \langle p, \gamma\rangle \hookrightarrow \langle p, u\rangle, r \in \Delta, (g_1, g_2) \in val(r)\}$. Then $(g_1, g_2) \in \mathrm{JOP}(\{\langle p, u_1\rangle\}, \{\langle p, u_2\rangle\})$ if and only if $\langle g_1, u_1\rangle \Rightarrow^* \langle g_2, u_2\rangle$.

Context-bounded analysis of concurrent SPDSs

A concurrent SPDS with two threads consists of two SPDSs $\mathcal{S}_1 = ((\{p\}, \Gamma_1, \Delta_1), G, val_1)$ and $\mathcal{S}_2 = ((\{p\}, \Gamma_1, \Delta_1), G, val_1)$ with the same set G. The transition relation $\Rightarrow_c = (\Rightarrow_1^*; \Rightarrow_2^*)^K$, which describes all paths in the concurrent PDS for $2K - 1$ context switches, is defined

in the same manner as for PDS, using the transition relations of the two PDSs. Let $\langle p, e_1, e_2 \rangle$ be the starting configuration of the concurrent SPDS. The problem of interest is to compute the following relation for a given set of configurations C:

$$R_C = \text{JOP}(\langle p, e_1, e_2 \rangle, C) = \bigcup \{ val(\sigma) \mid \langle p, e_1, e_2 \rangle \Rightarrow_c^\sigma c, c \in C \}.$$

A concurrent SPDS can be reduced to a single SPDS using the constructions presented earlier: (i) convert the SPDSs $\mathcal{S}_i$ to PDSs $\mathcal{P}_i$; (ii) convert the concurrent PDS system $(\mathcal{P}_1, \mathcal{P}_2)$ to a single PDS $\mathcal{P}_s$; and (iii) convert the PDS $\mathcal{P}_s$ to an SPDS $\mathcal{S}_s$. The rules of $\mathcal{S}_s$ will have binary relations on the set G^K (K-fold Cartesian product of G). Recall that the rules of $\mathcal{P}_s$ change the global state in only one component. Thus, the BDDs that represent the relations of rules in $\mathcal{S}_s$ would only be $\log(K)$ times larger than the BDDs for relations in $\mathcal{S}_1$ and $\mathcal{S}_2$ (the identity relation on n elements can be represented with a BDD of size $\log(n)$ [85]).

Let $C' = \{\langle p, e_3\ u_2\ u_1 \rangle \mid \langle p, u_1, u_2 \rangle \in C\}$. On $\mathcal{S}_s$, one can solve for the value $R = \text{JOP}(\langle p, e_1 \rangle, C')$. Then $R_C = \{(g, g') \mid ((g, g_2, \cdots, g_K), (g_2, \cdots, g_K, g')) \in R\}$ (note the similarity to Thm. 7.3.1).

7.5 The Reduction for WPDSs

A concurrent WPDS is defined as a set of WPDSs, one for each thread. The problem of CBA for concurrent WPDSs was defined in Section 6.1.

In this section, we only consider concurrent WPDSs with two threads. Moreover, we restrict each WPDS to have a single control state in the underlying PDS. A WPDS that does not satisfy this restriction can be converted to one that does satisfy it by appropriately changing the weight domain (similar to the conversion from a PDS to a symbolic PDS).

Let the concurrent WPDS be $(\mathcal{W}_1, \mathcal{W}_2)$, where $\mathcal{W}_i = (\mathcal{P}_i, \mathcal{S}, f_i)$, and $\mathcal{P}_i = (\{p\}, \Gamma_i, \Delta_i)$. Let K be the number of execution contexts per thread. We will reduce $(\mathcal{W}_1, \mathcal{W}_2)$ to a single WPDS $\mathcal{W}_s$ over a different weight domain using the tensor-product operation. Let $\mathcal{S}^K$ be the K^{th}-STP (Defn. 6.5.1) of $\mathcal{S}$. For weight domains, the tensor-product operation serves the same role as the duplication of shared variables that was used in Section 7.1 to keep

For each rule $\langle p, \gamma\rangle \hookrightarrow \langle p, u\rangle \in (\Delta_1 \cup \Delta_2)$ with weight w and for all $k \in \mathbb{N}_K$:
$\langle k, \gamma\rangle \quad \hookrightarrow \quad \langle k, u\rangle$ with weight $\mathrm{ec}(k, w)$

For each $\gamma \in \Gamma_j$ and for all $k \in \mathbb{N}_K$:
$\langle k, \gamma\rangle \quad \hookrightarrow \quad \langle k+1, \gamma\rangle$ with weight $\overline{1}$
$\langle K+1, \gamma\rangle \quad \hookrightarrow \quad \langle 1, e_{j+1}\ \gamma\rangle$ with weight $\overline{1}$

Figure 7.4 WPDS rules for $\mathcal{W}_s$.

track of the shared state at each context switch. The *DeTensor* operation will serve the role of the `Checker`.

Let $\mathbb{N}_K = \{1, 2, \cdots, K\}$. The WPDS $\mathcal{W}_s$ is defined as $(\mathcal{P}_s, \mathcal{S}^K, f_s)$, where $\mathcal{P}_s = (\mathbb{N}_K, \Gamma_s, \Delta_s)$. (The control states of $\mathcal{P}_s$ will keep track of the current execution context.) Define a function $\mathrm{ec} : \mathbb{N}_K \times \mathcal{S} \to \mathcal{S}^K$ as follows:

$$\mathrm{ec}(i, w) = \odot(\underbrace{\overline{1}, \cdots, \overline{1}}_{i-1}, w, \underbrace{\overline{1}, \cdots, \overline{1}}_{K-i})$$

$\mathrm{ec}(i, w)$ takes the tensor of K weights, where w appears in the i^{th} position.

Let $e_i \in \Gamma_i$ be the start node of the i^{th} thread. Let Γ_s be the disjoint union of Γ_1, Γ_2 and an additional symbol $\{e_3\}$. $\mathcal{W}_s$ does not have an explicit checking phase. The rules Δ_s are defined in Fig. 7.4.

As in Section 7.3, we assume that the PDSs $\mathcal{P}_1$ and $\mathcal{P}_2$ cannot empty their stacks, i.e., it is not possible that $\langle p, e_1\rangle \Rightarrow^*_{\mathcal{P}_1} \langle p, \varepsilon\rangle$ or $\langle p, e_2\rangle \Rightarrow^*_{\mathcal{P}_2} \langle p, \varepsilon\rangle$.

Theorem 7.5.1. *In the concurrent WPDS $(\mathcal{W}_1, \mathcal{W}_2)$, the net effect of all paths that go from $\langle p, e_1, e_2\rangle$ to $\langle p, c_1, c_2\rangle$ with K execution contexts per thread is exactly DeTensor*$(\mathrm{IJOP}_{\mathcal{W}_s}(\langle 1, e_1\rangle, \langle 1, e_3\ c_2\ c_1\rangle))$*, where the DeTensor operation is for the K-STP $\mathcal{S}^K$.*

7.6 Lazy CBA of Concurrent Boolean Programs

In the reduction presented in Section 7.2, the analysis of the generated sequential program had to assume all possible values for the symbolic constants. The lazy analysis has the property that at any time, if the analysis considers the K-tuple $(g_1, \cdots, g_K)$ of valuations of the symbolic constants, then there is at least one valid execution of the concurrent program in which the global state is g_i at the end of the i^{th} execution context of the first thread, for all $1 \leq i \leq K$.

The idea is to iteratively build up the effect that each thread can have on the global state in its K execution contexts. Note that T_1^s (or T_2^s) does not need to know the values of VAR_G^i when $i >$ `k`. Hence, the analysis proceeds by making no assumptions on the values of VAR_G^i when $i >$ `k`. When `k` is incremented to $k+1$ in the analysis of T_1^s, it consults a table E^2 that stores the effect that T_2^s can have in its first k execution contexts. Using that table, it figures out a valuation of VAR_G^{k+1} to continue the analysis of T_1^s, and stores the effect that T_1^s can have in its first k execution contexts in table E^1. These tables are built iteratively. More precisely, if the analysis can deduce that T_1^s, when started in state $(1, g_1, \cdots, g_k)$, can reach the state $(k, g'_1, \cdots, g'_k)$, and T_2^s, when started in state $(1, g'_1, \cdots, g'_k)$ can reach $(k, g_2, g_3, \cdots, g_k, g_{k+1})$, then an increment of `k` in T_1^s produces the global state $s = (k+1, g'_1, \cdots, g'_k, g_{k+1})$. Moreover, s can be reached when T_1^s is started in state $(1, g_1, \cdots, g_{k+1})$ because T_1^s could not have touched VAR_G^{k+1} before the increment that changed `k` to $k+1$. The algorithm is shown in Fig. 7.5. The entities used in it have the following meanings:

- Let $\bar{G} = \cup_{i=1}^{K} G^i$, where G is the set of global states. An element from the set $\bar{G}$ is written as $\bar{g}$. Let L be the set of local states.

- The relation H_n^j is related to program node n of the j^{th} thread. It is a subset of $\bar{G} \times \{1, \cdots, K\} \times \bar{G} \times L \times \{1, \cdots, K\} \times \bar{G} \times L$. If $H_n^j(\bar{g_0}, k_1, \bar{g_1}, l_1, k_2, \bar{g_2}, l_2)$ holds, then each of the $\bar{g_i}$ are an element of G^{k_2} (i.e., a k_2-tuple of global states), and the thread T_j is in its $k_2{}^{\text{th}}$ execution context. Moreover, if the valuation of VAR_G^i, $1 \leq i \leq k_2$, was $\bar{g_0}$

when T_j^s (the reduction of T_j) started executing, and if the node $\mathrm{ep}(n)$, the entry node of the procedure containing n, could be reached in data state $(k_1, \bar{g_1}, l_1)$, then n can be reached in data state $(k_2, \bar{g_2}, l_2)$, and the variables $\textsc{Var}_G^i$, $i > k_2$ are not touched (hence, there is no need to know their values). Note that this means that $\mathrm{ep}(n)$ was reached in the $k_1{}^{\text{th}}$ execution context of T_j and n was reached in the $k_2{}^{\text{th}}$ execution context.

- The relation $S_{\mathtt{f}}$ captures the summary of procedure `f`.
- The relations E^j store the *effect* of executing a thread. If $E^j(k, \bar{g_0}, \bar{g_1})$ holds, then $\bar{g_0}, \bar{g_1} \in G^k$, and the execution of thread T_j^s, starting from $\bar{g_0}$ can lead to $\bar{g_1}$, without touching variables in $\textsc{Var}_G^i$, $i > k$.
- The function $\mathtt{check}(k, (g_1, \cdots, g_k), (g'_1, \cdots, g'_k))$ returns g'_k if $g_{i+1} = g'_i$ for $1 \leq i \leq k-1$, and is undefined otherwise. This function checks for the correct transfer of the global state from T_2 to T_1 at a context switch.
- Let $[(g_1, \cdots, g_i), (g_{i+1}, \cdots g_j)] = (g_1, \cdots, g_j)$. We sometimes write g to mean (g), i.e., $[(g_1, \cdots, g_i), g] = (g_1, \cdots, g_i, g)$.

Understanding the rules. The rules $\mathcal{R}'_1, \mathcal{R}'_2, \mathcal{R}'_3$, and $\mathcal{R}'_7$ describe intra-thread computation, and are similar to the corresponding unprimed rules in Fig. 7.2. The rule $\mathcal{R}_{10}$ initializes the variables for the first execution context of T_1. The rule $\mathcal{R}_{12}$ initializes the variables for the first execution context of T_2. The rules $\mathcal{R}_8$ and $\mathcal{R}_9$ ensure proper hand-off of the global state from one thread to another. These two are the only rules that change the value of k. For example, consider rule $\mathcal{R}_8$. It ensures that the global state at the end of the $k_2{}^{\text{th}}$ execution context of T_2 is passed to the $(k_2+1)^{\text{th}}$ execution context of T_1, using the function `check`. The value g returned by this function represents a reachable valuation of the global variables when T_1 starts its $(k_2+1)^{\text{th}}$ execution context.

The following theorem shows that the relations E^1 and E^2 are built lazily, i.e., they only contain relevant information. A proof is given in Section 7.9.

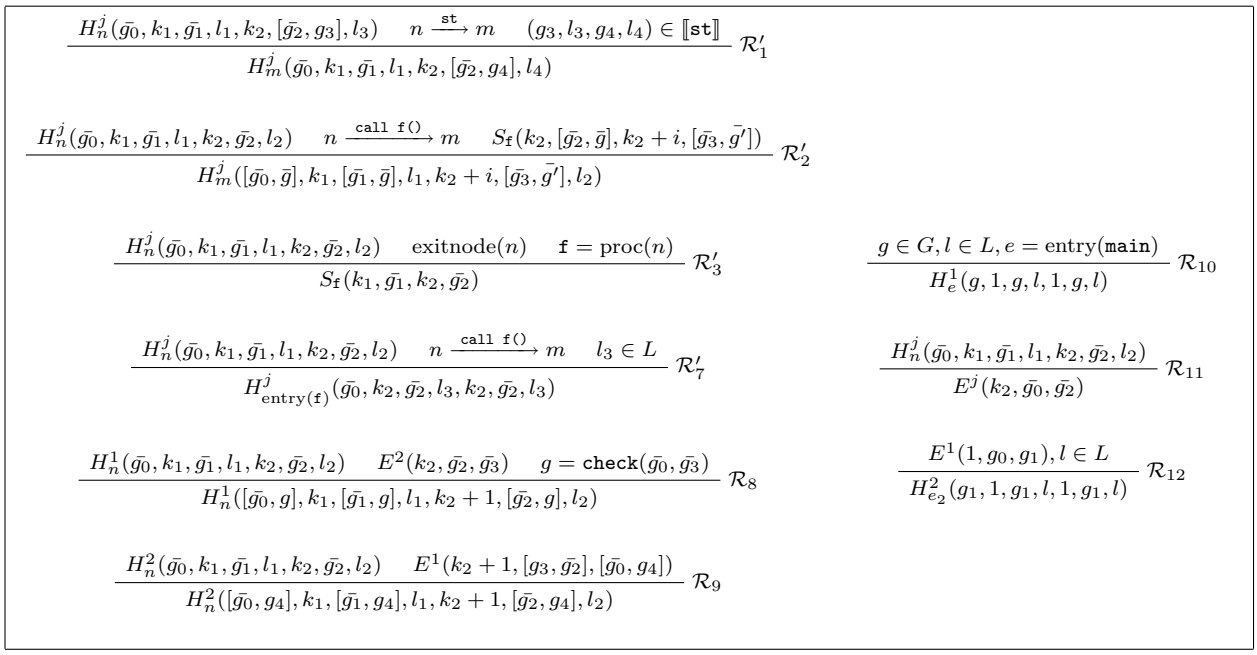


Figure 7.5 Rules for lazy analysis of concurrent Boolean programs with two threads.

Theorem 7.6.1. *After running the algorithm described in Fig. 7.5,* $E^1(k,(g_1,\cdots,g_k),(g'_1,\cdots,g'_k))$ *and* $E^2(k,(g'_1,\cdots,g'_k),(g_2,\cdots,g_k,g))$ *hold if and only if there is an execution of the concurrent program with* $2k-1$ *context switches that starts in state* g_1 *and ends in state* g*, and the global state is* g_i *at the start of the* i^{th} *execution context of* T_1 *and* g'_i *at the start of the* i^{th} *execution context of* T_2*. The set of reachable global states of the program in* $2K-1$ *context switches are all* $g \in G$ *such that* $E^2(K,\bar{g_1},[\bar{g_2},g])$ *holds.*

Multiple threads. In the presence of multiple threads, we fix round-robin scheduling, and impose a bound K on the number of execution contexts per thread.

The analysis rules remain similar to the ones for two threads, with E^i relations summarizing the behavior of the i^{th} thread. The only difference is the following: in the presence of two threads, for a thread, say T_1, one only needs to consult E^2 to find the global state for the next execution context (rule $\mathcal{R}_8$). In the presence of r threads, $r > 2$, for a thread T_i, one needs to consult each of $E^{i+1}, E^{i+2}, \cdots, E^r, E^1, \cdots, E^{i-1}$, in order. For this, we

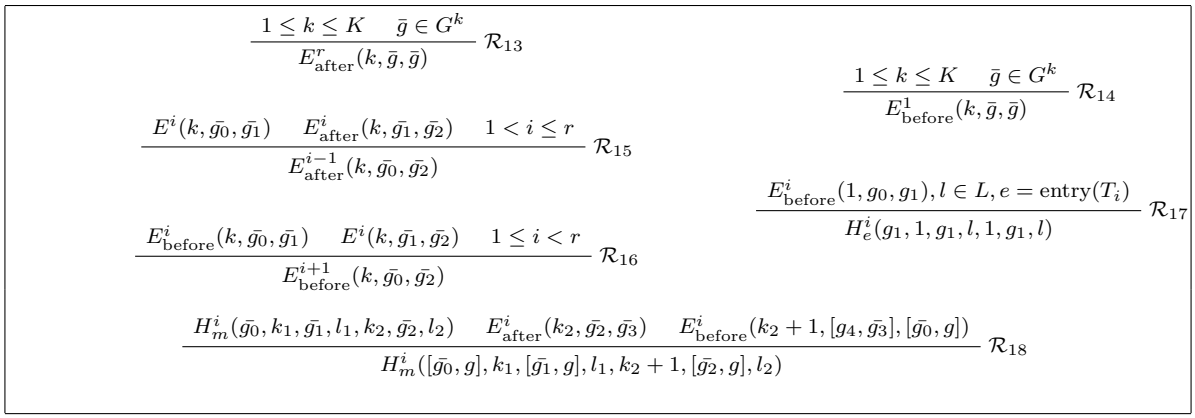


Figure 7.6 Rules for lazy analysis of concurrent Boolean programs with r threads.

build relations E^i_{after} and E^i_{before} that summarize the effect of $T_{i+1}, \cdots, T_r$ and $T_1, \cdots, T_{i-1}$, respectively.

The analysis rules for multiple threads include $\mathcal{R}'_1, \mathcal{R}'_2, \mathcal{R}'_3, \mathcal{R}'_7$, and $\mathcal{R}_{11}$ from Fig. 7.5. The rest of the rules are shown in Fig. 7.6. Rules $\mathcal{R}_{13}$ and $\mathcal{R}_{14}$ initialize E^r_{after} and E^1_{before} to the identity relation, respectively. Rules $\mathcal{R}_{15}$ and $\mathcal{R}_{16}$ compute these relations compositionally. Rule $\mathcal{R}_{17}$ generalizes rules $\mathcal{R}_{10}$ and $\mathcal{R}_{12}$ of Fig. 7.5. Rule $\mathcal{R}_{18}$ generalizes rules $\mathcal{R}_8$ and $\mathcal{R}_9$ of Fig. 7.5. Note that the use of `check` in $\mathcal{R}_8$ is made implicitly in $\mathcal{R}_{18}$. For instance, consider the case when $i = 1$ in $\mathcal{R}_{18}$. Then E^1_{before} is the identity relation on the global-state vectors. Thus, $[g_4, \bar{g_3}] = [\bar{g_0}, g]$, which implies that $g = \texttt{check}(\bar{g_0}, \bar{g_3})$.

7.7 Experiments

We implemented both the lazy and eager analyses for concurrent Boolean programs by extending the model checker MOPED [50]. These implementations find the set of all reachable states of the shared memory after a given number of context switches. We could have implemented the eager version using a source-to-source transformation; however, we took a different approach because it allows us to switch easily between the lazy and eager versions. Both versions are based on the rules shown in Fig. 7.5.

In the lazy version, the rules are applied in the following order: (*i*) The H relations are saturated for execution context k; (*ii*) Then the E relations are computed for k; (*iii*) then rules $\mathcal{R}_8$ and $\mathcal{R}_9$ are used to initialize the H relations for execution context $k+1$ and the process is repeated. In this way, the first step can be performed using the standard (sequential) reachability algorithm of MOPED. Thm. 7.6.1 allows us to find the reachable states directly from the E relations.

The eager version is implemented in a similar fashion, except that it uses a fixed set of E relations that include all possible global state changes. Once the H relations are computed, as described above, then the E relations are reinitialized using rule $\mathcal{R}_{11}$. Next, the following rule, which encodes the `Checker` phase, computes the set of reachable states (assuming that K is the given bound on the number of execution contexts).

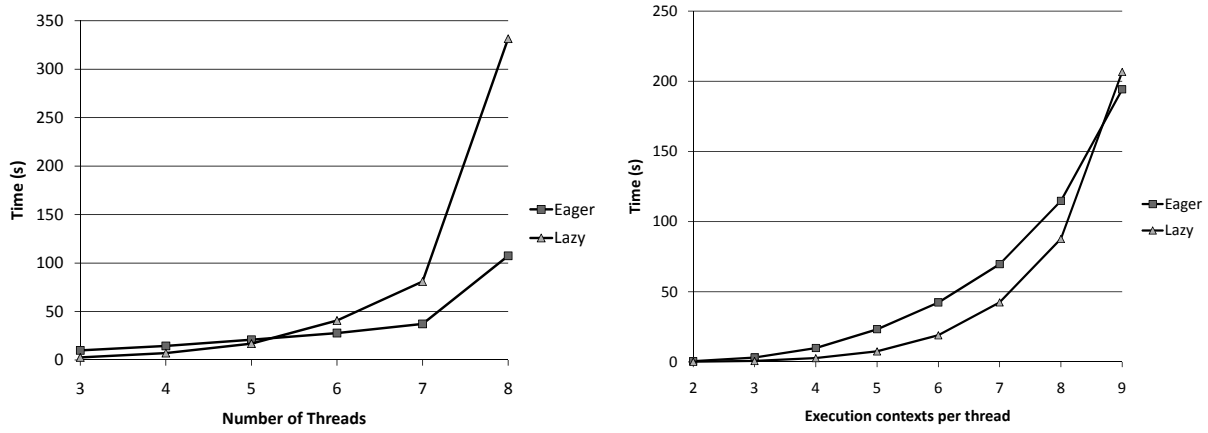
$$\frac{E^1(K, \bar{g_0}, \bar{g_1}) \quad E^2(K, \bar{g_1}, \bar{g_2}) \quad g = \texttt{check}(\bar{g_0}, \bar{g_2})}{\mathit{Reachable}(g)}\ \texttt{Checker}$$

Our implementation supports any number of threads. It uses round-robin scheduling with a bound on the number of execution context per thread, as described in Section 7.1.2.

All of our experiments, discussed below, were performed on a 2.4GHz machine with 3.4GB RAM running Linux version 2.6.18-92.1.17.el5.

BlueTooth driver model. First, we report the results for a model of the BlueTooth driver, which has been used in several past studies [78, 19, 89]. The driver model can have multiple threads, where each thread requests the addition or the removal of devices from the system, and checks to see if a user-defined assertion can fail. We used this model to test the scalability of our tool with respect to the number of threads, as well as the number of execution contexts per thread. The results are shown in Fig. 7.7. The model has 8 shared global variables, at most 7 local variables per procedure, 5 procedures but no recursion, and 37 program statements.

It is interesting to note that the eager analysis is faster than the lazy analysis in some cases (when there are a large number of threads or execution contexts). The running times for symbolic techniques need not be proportional to the number of states explored: even



(a) (b)

Figure 7.7 Experiments with the BlueTooth driver model. Each thread tries to either start or stop the device. (a) Running time when the number of execution contexts per thread is fixed at 4. (b) Running time when the number of threads is fixed at 3.

when the eager analysis explores more behaviors than the lazy version, its running time is shorter because it is able to exploit more symmetry in the search space, and the resulting BDDs are small.

The graph in Fig. 7.7(b) shows the exponential dependence of the running time on the number of execution contexts. The graph in Fig. 7.7(a) shows the expected linear dependence of the running time on the number of threads, until the number of threads is 8. We believe that the sharp increase is due to BDDs getting large enough so that operations on them do not entirely fit inside the BDD-cache.

Binary search tree. We also measured the performance of our techniques on a model of a concurrent binary search tree, which was also used in [89]. (Because our model was hand-coded, and the model used in [89] was automatically extracted from Java code, our results are not directly comparable.) This model has a finite heap, and a thread either tries to insert a value, or search for it in the tree. The model has 72 shared global variables, at most 52 local variables per procedure, 15 procedures, and 155 program statements. The model uses recursion.

Threads		Execution contexts per thread				
Inserters	Searchers	2	3	4	5	6
1	1	6.1	21.6	84.5	314.8	1054.8
2	1	11.9	46.8	211.9	832.0	2995.6
2	2	14.1	64.4	298.0	1255.4	4432.1

Figure 7.8 Lazy context-bounded analysis of the binary search tree model. The table reports the running time for various configurations in seconds.

The eager version of the algorithm timed out on this model for most settings. This may be because the analysis has to consider symbolic operations on the heap, which results in huge BDDs. The results for the lazy version are reported in Fig. 7.8. They show trends similar to the BlueTooth driver model: a linear increase in running time according to the number of threads, and an exponential increase in running time according to the number of execution contexts per thread.

BEEM benchmark suite. The third set of experiments consisted of common concurrent algorithms, for which finite, non-recursive models were obtained from the BEEM benchmark suite [73]. We hand-translated some of the SPIN models into the input language of MOPED. These models do not exploit the full capabilities of our tool because they all have a single procedure. We use these models for a more comprehensive evaluation of our tool. All examples that we picked use a large number of threads. As before, the eager version timed out for most settings, and we report the results for the lazy version.

The benchmark suite also has buggy versions of each of the test examples. The bugs were introduced by perturbing the constants in the correct version by ± 1 or by changing comparison operators (e.g., $>$ to $\geq$, or vice versa). Interestingly, the bugs were found within a budget of 2 or 3 execution contexts per thread. (Note that this may still involve multiple context switches.)

The results are reported in Fig. 7.9. To put the numbers in perspective, we also give the time required by SPIN to enumerate all reachable states of the program. These are finite-state models, meant for explicit-state model checkers; however, SPIN ran out of memory on three of the eight examples.

The CBA techniques presented in this chapter, unlike explicit-state model checkers, do not look for repetition of states: if a state has been reached within k context switches, then it need not be considered again if it shows up after $k + i$ context switches. In general, for recursive programs, this is hard to check because the number of states that can arise after a context switch may be infinite. However, it would still be interesting to explore techniques that can rule out some repetitions.

Predicate-abstraction based Boolean programs. Our fourth set of experiments use the concurrent Boolean programs generated using the predicate abstraction performed by DDVERIFY [96]. We use this set of experiments to validate two hypotheses: first, most bugs manifest themselves in a few context switches; second, our tool, based on CBA, remains competitive with current verification tools when it is given a reasonable bound on the number of context switches.

First, we briefly describe how DDVERIFY operates. When given C source code, DDVERIFY performs predicate abstraction to produce an abstract model of the original program. This model is written out as a concurrent Boolean program and fed to the model checker BOPPO, or in the input language of SMV and fed to SMV. DDVERIFY uses SMV by default because it performs better then BOPPO on concurrent models [96]. If the model checker is able to prove the correctness of all assertions in the model, the entire process succeeds (no bugs). If the model checker returns a counterexample, then it is checked concretely, and if it is spurious, then the abstraction is refined to create a new abstract model and this repeats. DDVERIFY checks for a number of different properties on the source code separately. (The abstract models produced by DDVERIFY have a single procedure and very few local variables. Thus, these experiments do not exploit the full capabilities of CBA as well.)

Name	Inst	#gvars	#lvars	#Threads	#EC	Time (s)	SPIN (s)
Anderson N=6,ERROR=0	pos	11	4	6	2	52.46	OOM
Anderson N=6,ERROR=1	neg	11	4	6	2	54.90	OOM
Bakery N=4,MAX=7	pos	17	7	4	2	5.87	28.5
Bakery N=4,MAX=5	neg	17	7	4	2	13.88	44.2
Peterson N=4	pos	25	7	4	3	5.46	3.05
Peterson N=4,ERROR=1	neg	25	7	4	3	25.72	OOM
Msmie N=5,S=10,M=10	pos	23	1	20	2	47.94	31.0
Msmie N=5,S=10,M=10	neg	13	1	13	2	1.29	1.04

Figure 7.9 Experiments on finite-state models obtained from the BEEM benchmark suite. The names, along with the given parameter values uniquely identify the program in the test suite. The columns, in order, report: the name; buggy (neg) or correct (pos) version; number of shared variables; number of local variables per thread; number of threads; execution context budget per thread; running time of our tool in seconds; and the time needed by SPIN to enumerate the entire state space. "OOM" stands for Out-Of-Memory.

We now describe the experimental setup. We chose 6 drivers among the ones provided with the distribution of DDVERIFY. For each driver, we chose some properties at random and let DDVERIFY run normally using SMV as its model checker, but we saved the Boolean programs that it produced at each iteration. For each driver and each property, we collected the SMV files and the Boolean programs produced during the last iteration. The experiments were conducted on these files. We gave our tool a budget of 2 threads and 4 execution contexts

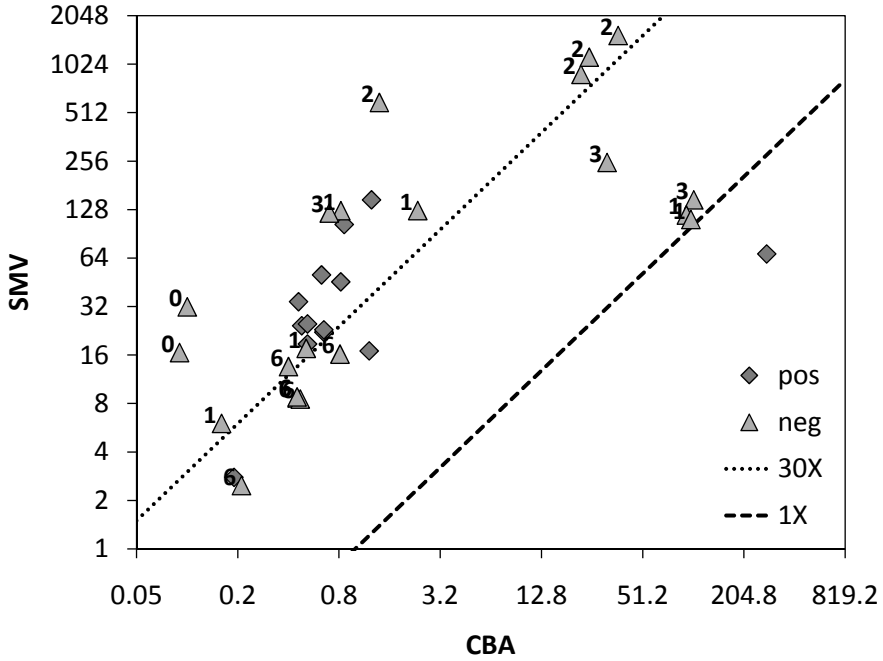


Figure 7.10 Scatter plot of the running times of our tool (CBA) against SMV on the files obtained from DDVERIFY. Different dots are used for the cases when the files had a bug (neg) and when they did not have a bug (pos). For the "neg" dots, the number of context switches before a bug was found is shown alongside the dot. The median speedup was about 30×. Lines indicating 1× and 30× speedups are also shown as dashed and dotted lines, respectively.

per thread. A scatter plot of the running times is shown in Fig. 7.10 and the aggregate times for proving all chosen properties for a given driver are reported in Fig. 7.11.

Two things should be noted from the results. First, whenever a model was buggy, our tool could find it within the budget given to it. This validates our hypothesis that bugs manifest in few context switches. Second, our tool was much faster than SMV on these benchmarks, with speedups of up to 120×; our tool was slower on only one example. As shown by the dotted line in Fig. 7.10, the median speedup was about 30×.

7.8 Related Work

A reduction from concurrent programs to sequential programs was given in [78] for the case of two threads and two context switches (it has a restricted extension to multiple threads

Name	Inst	Time (s)	SMV (s)	Speedup	# CS
applicom	pos	1.3	147.1	117.7	
	neg	33.0	147.1	15.1	[1, 3]
generic_nvram	pos	2.3	88.1	38.3	
gpio	pos	280.6	92.8	0.33	
	neg	106.8	299.6	2.8	[1, 6]
machzwd	pos	0.9	103.4	120.2	
	neg	85.6	4214.7	49.2	[0, 6]
nwbutton	pos	0.19	2.8	14.6	
	neg	0.88	26.1	29.7	[1, 6]
toshiba	pos	3.8	197.1	52.4	
	neg	192.8	243.43	1.3	[1, 6]

Figure 7.11 Experiments on concurrent Boolean programs obtained from DDVERIFY. The columns, in order, report: the name of the driver; buggy (neg) or correct (pos) version, as determined by SMV; running time of our tool in seconds; the running time of SMV; speedup of our tool against SMV; and the range of the number of context switches after which a bug was found. Each row summarizes the time needed for checking multiple properties.

as well). In such a case, the only thread interleaving is $T_1; T_2; T_1$. The context switch from T_1 to T_2 is simulated by a procedure call. Then T_2 is executed on the program stack of T_1, and at the next context switch, the stack of T_2 is popped off to resume execution in T_1. Because the stack of T_2 is destroyed, the analysis cannot return to T_2 (hence the context bound of 2). Their algorithm cannot be generalized to an arbitrary context bound.

A symbolic algorithm for context-bounded analysis was presented recently by Suwimonteerabuth et al. [89]. An earlier algorithm by Qadeer and Rehof [77] required enumeration of all reachable global states at a context switch. Suwimonteerabuth et al. identify places where such an enumeration is not required, essentially by finding different abstract states that the program model cannot distinguish. This enables symbolic computation to some

extent. However, in the worst case, the algorithm still requires enumeration of all reachable states.

Analysis of message-passing concurrent systems, as opposed to ones having shared memory, has been considered in [19]. They bound the number of messages that can be communicated, similar to bounding the number of contexts.

There has been a large body of work on verification of concurrent programs. Some recent work is [38, 76]. However, CBA is different because it allows for precise analysis of complicated program models, including recursion. As future work, it would be interesting to explore CBA with the abstractions used in the aforementioned work.

7.9 Proofs

7.9.1 Proof of Thm. 7.3.1

($\Leftarrow$) First, we show that a path of the concurrent program can be simulated by a path in the sequential program. (In this proof, we will deviate from the notation of the theorem to make the proof more clear.) Let $c_0 = e_1$, and $d_0 = e_2$. If the configuration $\langle p_0, c_0, d_0\rangle$ can lead to $\langle p_{2K}, c_K, d_K\rangle$ under the transition system $(\Rightarrow_1^*; \Rightarrow_2^*)^K$, then we show that there exist states $p_2, p_4, \cdots, p_{2K-2} \in P$ such that $\langle (1, p_0, p_2, \cdots, p_{2K-2}), c_0\rangle \Rightarrow_{\mathcal{P}_s} \langle (1, p_2, p_4, \cdots, p_{2K}), e_3\ d_K\ c_K\rangle$.

If a sequence of rules σ take a configuration c to a configuration c' under the transition system $\Rightarrow$, then we say $c \Rightarrow^\sigma c'$. For a rule $r \in \Delta_i$, $r = \langle p, \gamma\rangle \hookrightarrow \langle p', u\rangle$, let $r^s[k, p_1, \cdots, p_{k-1}, p_{k+1}, \cdots p_K] \in \Delta_s$ be the rule $\langle (k, p_1, \cdots, p_{k-1}, p', p_{k+1}, \cdots, p_K), \gamma\rangle \hookrightarrow \langle (k, p_1, \cdots, p_{k-1}, p', p_{k+1}, \cdots, p_K), u\rangle$. We extend this notation to rule sequences as well, and drop the p_i, when they are clear from the configuration the rules are applied on. Let $r_{\text{inc}}[k]$ stand for a rule of $\mathcal{P}_s$ that increments the value of k (note that it can fire with anything on the top of the stack). Let $r_{1\to 2}$ stand for the rules that call from the first PDS to the second, and $r_{2\to 3}$ stand for the rules that call e_3.

A path in $(\Rightarrow_1^*; \Rightarrow_2^*)^K$ can be broken down at each switch from $\Rightarrow_1$ to $\Rightarrow_2$, and from $\Rightarrow_2$ to $\Rightarrow_1$. Hence, there must exist c_i, d_i, $1 \le i \le K-1$; p_j, $1 \le j \le 2K-1$; and σ_h,

	$\langle p_0, c_0, d_0\rangle$
$\Rightarrow_1^{\sigma_1}$	$\langle p_1, c_1, d_0\rangle$
$\Rightarrow_2^{\sigma_2}$	$\langle p_2, c_1, d_1\rangle$
$\Rightarrow_1^{\sigma_3}$	$\langle p_3, c_2, d_1\rangle$
$\Rightarrow_2^{\sigma_4}$	$\langle p_4, c_2, d_2\rangle$
$\cdots$	
$\Rightarrow_1^{\sigma_{2K-1}}$	$\langle p_{2K-1}, c_K, d_{K-1}\rangle$
$\Rightarrow_2^{\sigma_{2K}}$	$\langle p_{2K}, c_K, d_K\rangle$
(a)	

	$\langle(1, p_0, p_2, \cdots, p_{2K-2}), c_0\rangle$
$\Rightarrow^{\sigma_1^s[1]}$	$\langle(1, p_1, p_2, p_4, \cdots, p_{2K-2}), c_1\rangle$
$\Rightarrow^{r_{\text{inc}}[1]}$	$\langle(2, p_1, p_2, p_4, \cdots, p_{2K-2}), c_1\rangle$
$\Rightarrow^{\sigma_3^s[2]}$	$\langle(2, p_1, p_3, p_4, \cdots, p_{2K-2}), c_2\rangle$
$\Rightarrow^{r_{\text{inc}}[2]}$	$\langle(3, p_1, p_3, p_4, \cdots, p_{2K-2}), c_2\rangle$
$\cdots$	
$\Rightarrow^{r_{\text{inc}}[K-1]}$	$\langle(K, p_1, p_3, p_5, \cdots, p_{2K-3}, p_{2K-2}), c_{K-1}\rangle$
$\Rightarrow^{\sigma_{2K-1}^s[K]}$	$\langle(K, p_1, p_3, p_5, \cdots, p_{2K-3}, p_{2K-1}), c_K\rangle$
$\Rightarrow^{r_{\text{inc}}[K]}$	$\langle(K+1, p_1, p_3, p_5, \cdots, p_{2K-3}, p_{2K-1}), c_K\rangle$
$\Rightarrow^{r_{1\to 2}}$	$\langle(1, p_1, p_3, p_5, \cdots, p_{2K-1}), d_0\ c_K\rangle$
$\Rightarrow^{\sigma_2^s[1]}$	$\langle(1, p_2, p_3, p_5, \cdots, p_{2K-1}), d_1\ c_K\rangle$
$\Rightarrow^{r_{\text{inc}}[1]}$	$\langle(2, p_2, p_3, p_5, \cdots, p_{2K-1}), d_1\ c_K\rangle$
$\cdots$	
$\Rightarrow^{r_{\text{inc}}[K-1]}$	$\langle(K, p_2, p_4, p_6, \cdots, p_{2K-2}, p_{2K-1}), d_{K-1}\ c_K\rangle$
$\Rightarrow^{\sigma_{2K}^s[K]}$	$\langle(K, p_2, p_4, p_6, \cdots, p_{2K-2}, p_{2K}), d_K\ c_K\rangle$
$\Rightarrow^{r_{\text{inc}}[K]}$	$\langle(K+1, p_2, p_4, p_6, \cdots, p_{2K-2}, p_{2K}), d_K\ c_K\rangle$
$\Rightarrow^{r_{2\to 3}}$	$\langle(1, p_2, p_4, p_6, \cdots, p_{2K-2}, p_{2K}), e_3\ d_K\ c_K\rangle$
(b)	

Figure 7.12 Simulation of a concurrent PDS run by a single PDS. For clarity, we write $\Rightarrow$ to mean $\Rightarrow_{\mathcal{P}_s}$ in (b).

$1 \leq h \leq 2K$, such that a path in the concurrent program can be broken down as shown in Fig. 7.12(a). Then the path shown in Fig. 7.12(b) is a valid run of $\mathcal{P}_s$ that establishes the required property.

$(\Rightarrow)$ For the reverse direction, a path σ in $\Rightarrow_{\mathcal{P}_s}$, from $\langle(1, p_0, p_2, \cdots, p_{2K-2}), c_0\rangle$ to $\langle(1, p_2, p_4, \cdots, p_{2K}), e_3\ d_K\ c_K\rangle$ can be broken down as $\sigma = \sigma_A\ r_{1\to 2}\ \sigma_B\ r_{2\to 3}$. (This is because

one must use the rules $r_{1\to 2}$ and $r_{2\to 3}$, in order, to push e_3 on the stack, after which no rules can fire.) Hence we must have the following (for some states $p_1, p_3, \cdots, p_{2K-1}$):

$$\begin{aligned}
\langle (1, p_0, p_2, \cdots, p_{2K-2}), c_0 \rangle &\Rightarrow^{\sigma_A}_{\mathcal{P}_s} \langle (K+1, p_1, p_3, \cdots, p_{2K-1}), c_K \rangle \\
&\Rightarrow^{r_{1\to 2}}_{\mathcal{P}_s} \langle (1, p_1, p_3, \cdots, p_{2K-1}), d_0\ c_K \rangle \\
&\Rightarrow^{\sigma_B}_{\mathcal{P}_s} \langle (K+1, p_2, p_4, \cdots, p_{2K}), d_K\ c_K \rangle \\
&\Rightarrow^{r_{2\to 3}}_{\mathcal{P}_s} \langle (1, p_2, p_4, \cdots, p_{2K}), e_3\ d_K\ c_K \rangle
\end{aligned}$$

Because σ_A changes the value of `k` from 1 to $K+1$, it must have $K+1$ uses of r_{inc}. Hence, it can be written as: $\sigma_A = \sigma^s_1[1]\ r_{\text{inc}}[1]\ \sigma^s_3[2]\ r_{\text{inc}}[2] \cdots r_{\text{inc}}[K-1]\ \sigma^s_{2K-1}[K]\ r_{\text{inc}}[K]$. Because only $\sigma^s[i]$ can change the i^{th} state component, we must have the following:

$$\begin{aligned}
\langle (1, p_0, p_2, \cdots, p_{2K-2}), c_0 \rangle &\Rightarrow^{\sigma^s_1[1]}_{\mathcal{P}_s} \langle (1, p_1, p_2, \cdots, p_{2K-2}), c_1 \rangle \\
&\Rightarrow^{r_{\text{inc}}[1]}_{\mathcal{P}_s} \langle (2, p_1, p_2, \cdots, p_{2K-2}), c_1 \rangle \\
&\cdots \\
&\Rightarrow^{\sigma^s_{2K-1}[K]}_{\mathcal{P}_s} \langle (K, p_1, p_3, \cdots, p_{2K-1}), c_K \rangle \\
&\Rightarrow^{r_{\text{inc}}[K]}_{\mathcal{P}_s} \langle (K+1, p_1, p_3, \cdots, p_{2K-1}), c_K \rangle
\end{aligned}$$

Similarly, $\sigma_B = \sigma^s_2[1]\ r_{\text{inc}}[1]\ \sigma^s_4[2]\ r_{\text{inc}}[2] \cdots r_{\text{inc}}[K-1]\ \sigma^s_{2K}[K]\ r_{\text{inc}}[K]$. The reader can verify that the rule sequence $\sigma_1\ \sigma_2 \cdots \sigma_{2K-1}\ \sigma_{2K}$ describes a path in $(\Rightarrow^*_1; \Rightarrow^*_2)^K$ and takes the configuration $\langle p_0, c_0, d_0 \rangle$ to $\langle p_{2K}, c_K, d_K \rangle$.

7.9.2 Complexity argument for Thm. 7.3.1

A PDS can have infinite number of configurations. Hence, sets of configurations are represented using automata [85]. We do not go into the details of such automata, but only present the running-time complexity arguments. Given an automata $\mathcal{A}$, and a PDS $(P_{\text{in}}, \Gamma_{\text{in}}, \Delta_{\text{in}})$, the set of configurations forward reachable from those represented by $\mathcal{A}$ can be calculated in time $\mathcal{O}(|P_{\text{in}}||\Delta_{\text{in}}|(|Q| + |P_{\text{in}}||Proc_{\text{in}}|) + |P_{\text{in}}||\ \rightarrow_{\mathcal{A}}\ |)$, where Q is the set of states of $\mathcal{A}$, and $\rightarrow_{\mathcal{A}}$ is the set of its transitions [85]. We call the algorithm from [85] *poststar*, and its output, which is also an automaton, *poststar*($\mathcal{A}$).

For the PDS $\mathcal{P}_s$, obtained from a concurrent PDS with n threads $(\mathcal{P}_1, \mathcal{P}_2, \cdots, \mathcal{P}_n)$, $|P_s| = K|P|^K, |\Delta_s| = K|P|^{K-1}|\Delta|, |Proc_s| = |Proc|$, where $\Delta = \cup_{i=1}^{n}\Delta_i$ and $|Proc| = \sum_{i=1}^{n}|Proc_i|$. To obtain the set of forward reachable configurations from $\langle p, e_1, e_2, \cdots, e_n\rangle$, we will solve *poststar*($\mathcal{A}$) for each $\mathcal{A}$ that represents the singleton set of configurations $\{\langle(1, p, p_2, \cdots, p_K), e_1\rangle\}$, i.e., $|P|^{K-1}$ separate calls to *poststar*. In the result, we can project out all configurations that do not have $(1, p_2, \cdots, p_K, p')$ as their state, for some p'. Directly using the above complexity result, we get a total running time of $\mathcal{O}(K^3|P|^{4K}|\Delta||Proc|)$. For the case of two threads, we use a more sophisticated argument to calculate the running time.

When asking for the set of reachable configurations of $\mathcal{P}_s$, we are only interested in some particular configurations: when starting from $\langle(1, p, p_2, \cdots, p_K), e_1\rangle$, we only want configurations of the form $\langle(1, p_2, \cdots, p_K, p'), u\rangle$. Hence, when we run *poststar*, starting from the above configuration, we remove some rules from Δ_s: we remove all rules with left-hand side $\langle(k, p'_2, p'_3, \cdots, p'_K, p'), \gamma\rangle$ if $\gamma \in \Gamma_2$ and $p_i \neq p'_i$ for some i between 1 and $k-1$, both inclusive. We statically know that removing such rules would not affect the result.

Further, we make two observations about the algorithm from [85]: (*i*) if an automaton $\mathcal{A}$ is split into two automata $\mathcal{A}_1$ and $\mathcal{A}_2$, such that the union of the transitions (represented configurations) of $\mathcal{A}_1$ and $\mathcal{A}_2$ equals the set of transitions (represented configurations) of $\mathcal{A}$, then the running time of *poststar*($\mathcal{A}$) is strictly smaller than than the sum of the running times of *poststar*($\mathcal{A}_1$) and *poststar*($\mathcal{A}_2$). (*ii*) splitting the set of PDS rules Δ into two (Δ_1 and Δ_2) such that no rule in Δ_1 can fire after a rule of Δ_2 is applied, then the running time of $poststar_{\Delta_2}(poststar_{\Delta_1}(\mathcal{A}))$ is the same as the running time of $poststar_{\Delta}(\mathcal{A})$, where the *poststar* algorithm is subscripted with the set of rules it operates on. Using these two observations, we show that running *poststar* using $\mathcal{P}_s$ takes less time than the above-mentioned complexity.

Let $\Delta^i \subseteq \Delta_s$ be the set of rules that operate when the first component of the state (the value of `k`) is i, and $\Delta_{\text{call}} \subseteq \Delta_s$ be the set of rules that call to e_2 (from Γ_1) or e_3. We know that any path in $\mathcal{P}_s$ can be decomposed into a rule sequence from $\mathcal{S} = \Delta^{1^*}\ \Delta^{2^*} \cdots \Delta^{K^*}\ \Delta_{\text{call}}\ \Delta^{1^*}\ \Delta^{2^*} \cdots \Delta^{K^*}\ \Delta_{\text{call}}$. Using observation (*ii*) above, we break the running of *poststar* on Δ_s into a series operating on each of the above sets, in order.

Iter	Num	$\mid \rightarrow \mid$	Time	Split	$\mid Q \mid$
1	1	1	$\mid P \mid^2 \mid \Delta \mid\mid Proc \mid$	$\mid P \mid$	$\mid P \mid\mid Proc \mid$
2	$\mid P \mid$	$\mid P \mid\mid \Delta \mid\mid Proc \mid$	$2\mid P \mid^2 \mid \Delta \mid\mid Proc \mid$	$\mid P \mid$	$2\mid P \mid\mid Proc \mid$
i	$\mid P \mid^{i-1}$	$(i-1)\mid P \mid\mid \Delta \mid\mid Proc \mid$	$2(i-1)\mid P \mid^2 \mid \Delta \mid\mid Proc \mid$	$\mid P \mid$	$i\mid P \mid\mid Proc \mid$
K	$\mid P \mid^{K-1}$	$(K-1)\mid P \mid\mid \Delta \mid\mid Proc \mid$	$2(K-1)\mid P \mid^2 \mid \Delta \mid\mid Proc \mid$	$\mid P \mid$	$K\mid P \mid\mid Proc \mid$

Table 7.1 Running times for different stages of *poststar* on $\mathcal{P}_s$.

Next, after running *poststar* on one of Δ^{i^*}, we split the resultant automaton $\mathcal{A}$ into as many automata as the number of states in the configurations of $\mathcal{A}$, e.g., if $\mathcal{A}$ represents the set $\{\langle \bar{p}_1, c_1 \rangle, \langle \bar{p}_2, c_2 \rangle, \langle \bar{p}_2, c_3 \rangle\}$, then we split it into two automata representing the sets $\{\langle \bar{p}_1, c_1 \rangle\}$ and $\{\langle \bar{p}_2, c_2 \rangle, \langle \bar{p}_2, c_3 \rangle\}$, respectively. Observation (i) shows that this splitting only increases the running time.

Tab. 7.1 shows the running time for performing *poststar* on the first K of the Δ^{i^*} from $\mathcal{S}$. The column "Iter" shows which Δ^i is being processed. The column "Num" is the number of *poststar* that have to be run using Δ^i. The column "$\mid \rightarrow \mid$" shows the upper bound on the number of transitions in the automaton *poststar* is run on. The column "Time" is the running time of *poststar* on such automata. The column "Split" is an upper bound on the the number of automata the result is split into, and the last column in the number of states in each of the resultant automata. For example, there are $\mid P \mid^{i-1}$ number of invocations to *poststar* with rule set Δ^i, each on an automata with at most $(i-1)\mid P \mid\mid \Delta \mid\mid Proc \mid$ transitions, taking time $2(i-1)\mid P \mid^2 \mid \Delta \mid\mid Proc \mid$. Each result is split into $\mid P \mid$ different automata, each with at most $i\mid P \mid\mid Proc \mid$ states. The reader can inductively verify the correctness of the table.

Thus, this requires a total running time of $\mathcal{O}(K\mid P \mid^{K+1} \mid \Delta \mid\mid Proc \mid)$. Next, we use the rules in Δ_{call} and repeat the above process for the last K of the sequence $\mathcal{S}$. However, in this case, no splitting is necessary, because we know the desired target state, and have already removed some rules from Δ_s. For example, if the initial state chosen was $(1, p, p_2, \cdots, p_K)$, and after performing the computation of Tab. 7.1, we obtain an automaton $\mathcal{A}$ that has the single state

$(1, p'_1, \cdots .p'_K)$ for all configurations represented by it. After processing $\mathcal{A}$ with Δ^1 suppose the result is $\mathcal{A}'$. There is no need to split $\mathcal{A}'$ because of the rules removed from Δ^2. The rules of Δ^2 would only fire on configurations that have the state $(2, p_2, p'_2, p'_3, \cdots, p'_K)$. Thus, splitting is not necessary, and the time required to process each of the $|P|^{K-1}$ automata obtained from Tab. 7.1 using Δ^i is $2(K + i - 1)|P|^2|\Delta||Proc|$. Hence, the time required to process the entire $\mathcal{S}$ is $\mathcal{O}(K^2|P|^{K+1}|\Delta||Proc|)$. Because we have to repeat for $|P|^{K-1}$ initial states, the running time of *poststar* on $\mathcal{P}_s$ with two threads can be bounded by $\mathcal{O}(K^2|P|^{2K}|\Delta||Proc|)$.

Backward analysis from a set of configurations represented by an automaton $\mathcal{A}$ with $|Q|$ states can be performed in time $\mathcal{O}(K|P|^{2K}(K|P|^K + |Q|)^2|\Delta|)$ for multiple threads, and $\mathcal{O}(K|P|^{2K}(K|P| + |Q|)^2|\Delta|)$ for two threads.

7.9.3 Proof of Thm. 7.6.1

For proving Thm. 7.6.1, we will make use of the fact that our reduction to a (sequential) Boolean program is correct. Let T_1^s be the reduction of the first thread, and T_2^s be the reduction of the second thread. First, we show that given an execution ρ of T_1^s, and certain facts about E^2 (which summarizes the effect of the second thread), ρ can be simulated by the subset of rules from Fig. 7.5 that apply to the first thread. Formally, suppose that ρ is the execution shown in Fig. 7.13 (where n_0 is the entry point of the thread).

The execution ρ is broken at the points where the value of `k` is incremented. Note that this execution implies that in the concurrent program the global state, when T_1 begins its i^{th} execution context, is g_i, and when T_2 begins its i^{th} execution context, it is g'_i. Further, suppose that the following facts hold: $E^2(i, (g'_1, g'_2, \cdots, g'_i), (g_2, g_3, \cdots, g_i))$ for $1 \leq i \leq k-1$. Given these, we will show that rules for the first thread can be used to establish that $H^1_{n_k}((g_1, \cdots, g_k), k_1, \bar{g}, l, k, (g'_1, \cdots, g'_k), l')$ holds, for some $k_1, \bar{g}, l$ and l'.

Corresponding to the execution ρ, there would be a sequence of deductions, using the rules from Fig. 7.2 on T_1^s that derives the state at n_k. These rules simply perform an interprocedural analysis on T_1^s (the symbolic constants can take any value when program

$$
\begin{array}{c}
n_0 : (1, g_1, g_2, \cdots, g_k, g_{k+1}, \cdots, g_K, l_0) \\
\Downarrow \\
n_1 : (1, g_1', g_2, \cdots, g_k, g_{k+1}, \cdots, g_K, l_1) \\
\Downarrow \\
n_1 : (2, g_1', g_2, \cdots, g_k, g_{k+1}, \cdots, g_K, l_1) \\
\Downarrow \\
n_2 : (2, g_1', g_2', \cdots, g_k, g_{k+1}, \cdots, g_K, l_2) \\
\Downarrow \\
\vdots \\
\Downarrow \\
n_{k-1} : (k-1, g_1', g_2', \cdots, g_{k-1}', g_k, g_{k+1}, \cdots, g_K, l_{k-1}) \\
\Downarrow \\
n_k : (k, g_1', g_2', \cdots, g_k', g_{k+1}, \cdots, g_K, l_k)
\end{array}
$$

Figure 7.13 An execution in T_1^s.

execution starts). We formalize the notation of using these rules on T_1^s. Let the rules operate on the relations H^s and S^s. These relations are of the form: $H_n^s([k_1, \bar{g_1}], l_1, [k_2, \bar{g_2}], l_2)$, which semantically means that if the data state at ep(n) was $([k_1, \bar{g_1}], l_1)$, then the data state at n can be $([k_2, \bar{g_2}], l_2)$.; and the summary relation would be $S_{\mathtt{f}}^s([k_1, \bar{g_1}], [k_2, \bar{g_2}])$. For a statement `st` in T_1, its translation in T_1^s encodes the transformer:

$$\{((k, g_1, \cdots, g_k, \cdots, g_K), l, (k, g_1, \cdots, g_k', \cdots, g_K), l') \mid (g_k, l, g_k', l') \in \mathtt{st}, 1 \leq k \leq K\}$$

Additionally, one has a self-loop edge associated with a transformer that increments the value of `k`: $\{([k, \bar{g}], l, [k+1, \bar{g}], l) \mid 1 \leq k \leq K\}$. Given a proof tree π for ρ, we build a proof tree π' using rules of Fig. 7.5 by induction on the bottom-most rule of π.

When $k = 1$ in ρ, the conversion is straightforward: just replace a rule $\mathcal{R}$ in π with the primed rule $\mathcal{R}'$ from Fig. 7.5. An example is shown in Fig. 7.14 for a program path

$\pi_1 =$

$$\dfrac{\dfrac{[1,[g_0,\bar{g}]] \in G^s, l_0 \in L}{H^s_{n_0}([1,[g_0,\bar{g}]],l_0,[1,[g_0,\bar{g}]],l_0)}\,\mathcal{R}_0 \quad n_0 \xrightarrow{\texttt{st}_1} n_1 \quad (g_0,l_0,g_1,l_1) \in [\![\texttt{st}_1]\!]}{H^s_{n_1}([1,[g_0,\bar{g}]],l_0,[1,[g_1,\bar{g}]],l_1)}\,\mathcal{R}_1$$

$\pi_2 =$

$$\dfrac{\dfrac{\dfrac{\pi_1}{H^s_{n_1}([1,[g_0,\bar{g}]],l_0,[1,[g_1,\bar{g}]],l_1)} \quad n_1 \xrightarrow{\texttt{call f}} n_2}{H^s_{n_3}([1,[g_1,\bar{g}]],l_2,[1,[g_1,\bar{g}]],l_2)}\,\mathcal{R}_7 \quad n_3 \xrightarrow{\texttt{st}_2} n_4 \quad (g_1,l_2,g_2,l_3) \in [\![\texttt{st}_2]\!]}{H^s_{n_3}([1,[g_1,\bar{g}]],l_2,[1,[g_2,\bar{g}]],l_3)}\,\mathcal{R}_1$$

$\pi =$

$$\dfrac{\dfrac{\pi_1}{H^s_{n_1}([1,[g_0,\bar{g}]],l_0,[1,[g_1,\bar{g}]],l_1)} \quad n_1 \xrightarrow{\texttt{call f}} n_2 \quad \dfrac{\pi_2}{S_{\texttt{f}}([1,[g_1,\bar{g}]],[1,[g_2,\bar{g}]])}}{H^s_{n_2}([1,[g_0,\bar{g}]],l_0,[1,[g_2,\bar{g}]],l_1)}\,\mathcal{R}_2$$

$\pi'_1 =$

$$\dfrac{\dfrac{g_0 \in G, l_0 \in L}{H^1_{n_0}(g_0,1,g_0,l_0,1,g_0,l_0)}\,\mathcal{R}_{10} \quad n_0 \xrightarrow{\texttt{st}_1} n_1 \quad (g_0,l_0,g_1,l_1) \in [\![\texttt{st}_1]\!]}{H^1_{n_1}(g_0,1,g_0,l_0,1,g_1,l_1)}\,\mathcal{R}'_1$$

$\pi'_2 =$

$$\dfrac{\dfrac{\dfrac{\pi'_1}{H^1_{n_1}(g_0,1,g_0,l_0,1,g_1,l_1)} \quad n_1 \xrightarrow{\texttt{call f}} n_2}{H^1_{n_3}(g_0,1,g_1,l_2,1,g_1,l_2)}\,\mathcal{R}'_7 \quad n_3 \xrightarrow{\texttt{st}_2} n_4 \quad (g_1,l_2,g_2,l_3) \in [\![\texttt{st}_2]\!]}{H^1_{n_3}(g_0,1,g_1,l_2,1,g_2,l_3)}\,\mathcal{R}'_1$$

$\pi' =$

$$\dfrac{\dfrac{\pi'_1}{H^1_{n_1}(g_0,1,g_0,l_0,1,g_1,l_1)} \quad n_1 \xrightarrow{\texttt{call f}} n_2 \quad \dfrac{\pi'_2}{S_{\texttt{f}}(1,g_1,1,g_2)}}{H^1_{n_2}(g_0,1,g_0,l_0,1,g_2,l_1)}\,\mathcal{R}'_2$$

Figure 7.14 An example of converting from proof π to proof π'. For brevity, we use `st` to mean a statement in the thread T_1 (and not its translated version in T^s_1).

$n_0 \xrightarrow{\texttt{st}_1} n_1 \xrightarrow{\texttt{call f}} n_2$, where the call to `f` takes the path $n_3 \xrightarrow{\texttt{st}_2} n_4$. Let $(g_1,\cdots,g_{k+i})|_k = (g_1,\cdots,g_k)$.

The induction hypothesis is as follows: given ρ, as shown in Fig. 7.13, if there is a proof tree π that derives $H^s_{n_k}([k_1,\bar{g}],l,[k,(g'_1,\cdots,g'_k,g_{k+1},\cdots,g_K)],l')$ then one can derive

$H^1_{n_k}((g_1, \cdots, g_k), k_1, \bar{g}|_k, l, k, (g'_1, \cdots, g'_k), l')$. Note that in this case, the last $(K - k_1)$ components of $\bar{g}$ must be $(g_{k_1+1}, \cdots, g_K)$ because T^s_1 could not have modified them. We have already proved the base case above. Fix $\bar{g}_{\text{init}} = (g_1, \cdots, g_k)$ and $\bar{g}_{\text{final}} = (g'_1, \cdots, g'_k, g_{k+1}, \cdots, g_K)$.

The bottom-most rule of π can be $\mathcal{R}_1, \mathcal{R}_2$ or $\mathcal{R}_7$. For the rule $\mathcal{R}_1$, one can either use a statement transformer, or increment the value of `k`. All these cases, and the way to obtain π' are shown in Fig. 7.15.

One can prove a similar result for T^s_2. Note that $H^1_{n_k}(\bar{g}_{\text{init}}, k_1, \bar{g}|_k, l, k, \bar{g}_{\text{final}}|_k, l')$ implies $E^1(k, \bar{g}_{\text{init}}, \bar{g}_{\text{final}}|_k)$. Thus, these results are sufficient to prove one side of the theorem: given an execution of the concurrent program, we can obtain executions of T^s_1 and T^s_2, and then use the above results together to show that the rules in Fig. 7.5 can simulate the execution of the concurrent program.

Going the other way is similar. A deduction on H^1 can be converted into an interprocedural path of T^s_1. The rule $\mathcal{R}_8$ corresponds to incrementing the value of `k`, and must be used a bounded number of times in a derivation of H^1 fact. The E^2 assumptions used in a derivation have to be of the form $E^2(1, g'_1, g_2), E^2(2, (g'_1, g'_2), (g_2, g_3)), \cdots, E^2(i, (g'_1, \cdots, g'_i), (g_2, \cdots, g_{i+1}))$. This is because the second component of H^1 is only extended, but never modified, and once `k` is incremented, the first `k` components cannot be modified either. Now, we can use the conversions of Fig. 7.15 in the opposite direction to prove the reverse direction of the theorem.

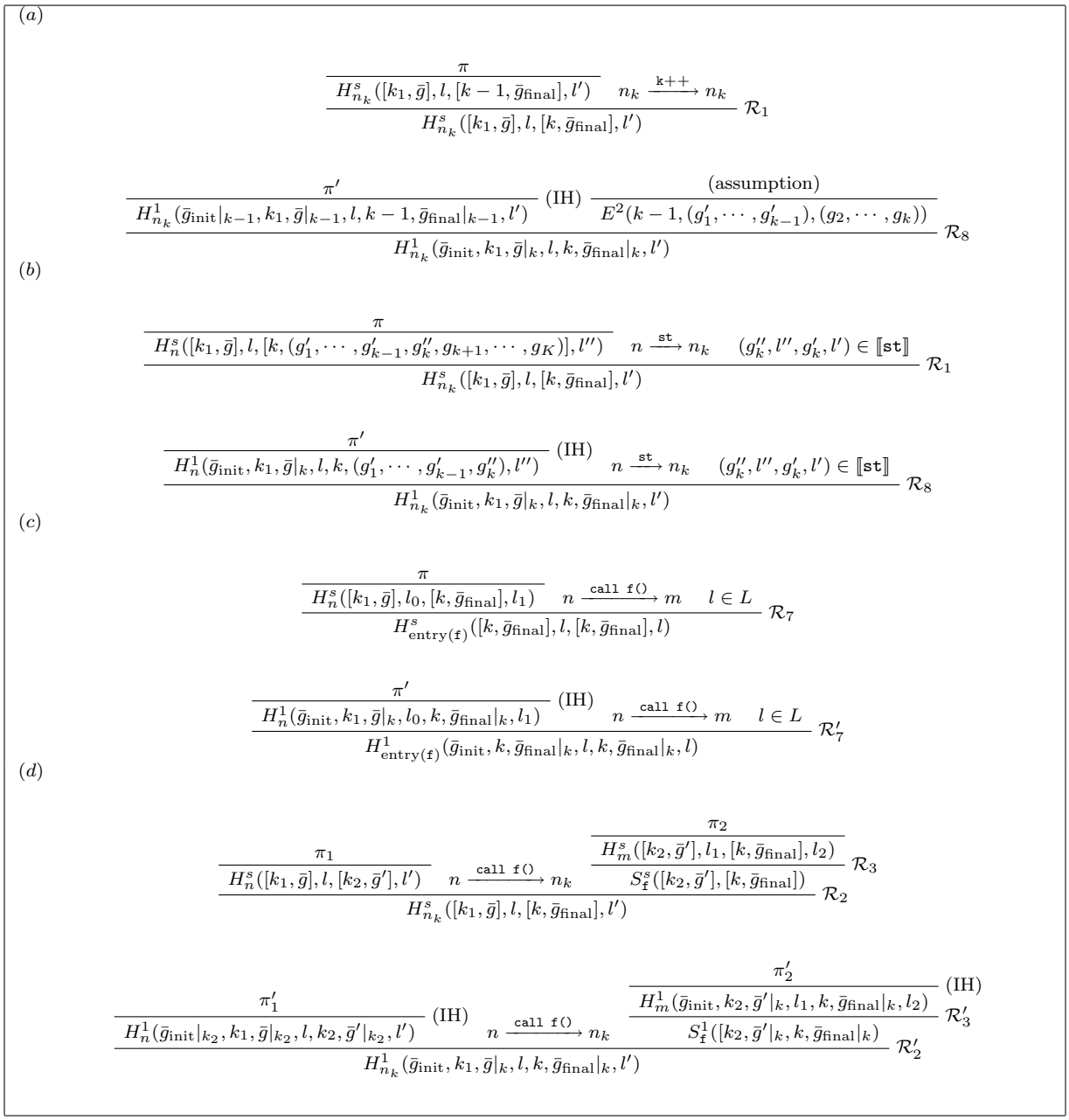


Figure 7.15 Simulation of run ρ using rules in Fig. 7.5. In case (a), $g_k = \texttt{check}(\bar{g}_{\text{init}}|_{k-1}, (g_2, \cdots, g_k))$ and $g'_k = g_k$ (because ρ does not edit these set of variables). In case (d), exitnode(m) holds, $\texttt{f} = proc(m)$, $k_1 \leq k_2 \leq k$, the $k_2 + 1$ to k components of $\bar{g}'$ are $(g_{k_2+1}, \cdots, g_k)$ because it arises when $\texttt{k} = k_2$, and the $k_1 + 1$ to k components of $\bar{g}$ are $(g_{k_1+1}, \cdots, g_k)$ for the same reason.

Chapter 8

Conclusions

A program-verification technique aims to gain certain knowledge about a program's behavior to determine whether some program execution can be faulty, or whether no faulty executions are possible. Such techniques are becoming increasingly important as software gets larger, more complex, and often hard to reason about manually. This dissertation gives several techniques that can reason about two important aspects of a program: procedures (and procedure calls) and concurrency.

We follow the common design of program-verification tools in which verification is split into two phases: an abstraction phase, which produces an abstract model, and an analysis phase, which precisely reasons about the abstract model. The contributions of this dissertation are to give expressive abstract models that can easily encode programs with procedures and concurrency, and efficient analysis algorithms for these models. Thus, to solve a new verification problem, one only needs to encode the problem using one of our abstract models, and then analyze the model using one of our algorithms.

Analysis of Sequential Programs

In Chapter 3, we defined Extended Weighted Pushdown Systems (EWPDSs). We demonstrated the power of EWPDSs by showing that several problems can be solved using EWPDSs, including Boolean program verification, affine-relation analysis, and single-level alias analysis. We gave efficient algorithms for analyzing EWPDSs. One of the advantages of using EWPDSs is that it supports stack-qualified queries. In our previous work [56], we showed

the importance of using stack-qualified queries in the context of debugging: the stack trace at the point of a program crash is an important clue about what the program execution did before failing. Obtaining information from the program model that is specific to the stack trace requires a stack-qualified query.

In Chapter 4, we gave an algorithm, called FWPDS, for faster analysis of WPDSs and EWPDSs. Because FWPDS applies to abstract models, it improves the running time of any application based on these abstract models. We observed 1.8× to 3.6× speedups in three different applications that used EWPDSs without requiring any fine-tuning for an application. These applications were: (*i*) the debugging application mentioned above, which searches for a particular path in the control-flow graph of a program; (*ii*) an analysis that finds a set of affine relations in x86 programs; and (*iii*) an assertion checker for Boolean programs. In Chapter 5, we showed how to answer more expressive queries on EWPDSs, which compute what we call error projections. An error projection is the set of all nodes that lie on an error trace in the abstract model. Computing an error projection can help speed up abstraction-refinement-based techniques.

All of the techniques mentioned above, namely EWPDSs, FWPDSs, and error projections, are implemented as a library and available for download as part of the WALi package [47]. We have also addressed the problem of speeding up multiple (E)WPDS queries [57], and that is included with WALi as well.

Analysis of Concurrent Programs

The above work is on interprocedural analysis of sequential programs. In Chapters 6 and 7, we presented techniques for the analysis of multi-procedure concurrent programs. Because such analyses are undecidable, even for simple abstractions, we explored the area of context-bounded analysis (CBA), where the number of context-switches between different threads is bounded. We show that given an interprocedural analysis for sequential programs, one can automatically extend it to perform CBA of concurrent programs, under certain conditions.

In Chapter 6, we showed that when each thread of a concurrent program is modeled using a WPDS, and a tensor-product operation exists for the weights, CBA of the program can be carried out effectively. The algorithm for CBA has two key steps. First, each thread is analyzed separately to build a weighted transducer that captures the effect of executing the thread without interruption from other threads. In particular, a thread T is converted into a weighted transducer τ_T that represents the following relation:

$$\{(s_1, s_2) \mid \text{ execution of } T \text{ starting from state } s_1 \text{ can lead to state } s_2 \}.$$

We also showed how to construct such a transducer for a WPDS. This provides a strong characterization of the behaviors of a WPDS. Second, the transducers from each of the threads are composed as many times as the context bound K, resulting in the net effect of executing the concurrent program for K context switches. From this, one can find the set of all reachable states within K context switches and verify properties of the program under that bound. The importance of this result is that one has to do little work to obtain an algorithm for CBA: one has to show that each thread can be (soundly) modeled using a WPDS (which one would have to do even for sequential analysis) and that a tensor operation exists for the weights.

A topic left for future work is to extend these results to EWPDSs as well. The difficulty lies in finding a tensor-like operation for merge functions.

In Chapter 7, we gave a practical algorithm for CBA of concurrent programs. We showed that given a concurrent program P and a context bound K, one can create a sequential program P_K such that the analysis of P_K is sufficient for CBA of P under the bound K. This reduction is a source-to-source transformation, and requires no assumptions nor extra work on the part of the user, except for the identification of thread-local data.

We implemented the technique on Boolean programs to create the first known implementation of CBA. Using this tool, we conducted a study on concurrent Linux drivers and showed that most bugs could be found (1) in a few context switches and (2) much faster than previous approaches.

One interesting aspect of the reduction from the concurrent program P to the sequential program P_K is how the program execution changes. An execution of P is split into pieces (one for each execution context), and then rearranged (so that the pieces for each thread are put together). Extra checks are put in place to ensure that no spurious behaviors are introduced. The technique of allowing actions to happen in a different order, but at the same time using constraints to ensure that the semantics is preserved, allowed us to reduce the complexity of CBA and make it linear in the size of the local state space. This technique may be useful in contexts other than CBA as well.

Follow-up Work. Because our reduction is a source-to-source transformation, one can apply several different techniques, developed for sequential programs, to concurrent programs. Our implementation of CBA, which applies to Boolean programs, uses a BDD-based solver. In a follow-up work by others, our reduction was extended and applied to C programs [55]. They used an SMT-based solver to do partial verification of the sequential program produced as a result of the reduction.

As future work, it would be interesting to study further extensions of our reduction. A key factor that affects scalability is the size of shared memory because the shared state has to be recorded at each context switch. In languages like C, the shared memory cannot be determined statically. Thus, it would be useful to design a technique that identifies the shared memory on-the-fly as the program is analyzed.

In follow-up work by yet another group, our reduction was extended to a "lazy" reduction [92]. The sequential program P_K produced by their reduction has the property that the analysis of P_K permits a lazy analysis similar to one we presented in Section 7.6.

Techniques for Weighted Systems

One of the themes of the work presented in the dissertation—and one of the areas in which it makes a contribution that extends beyond program verification—is the development of techniques and algorithms for manipulating weighted automata and weighted transducers. Both WPDSs, and EWPDSs are based on Pushdown Systems (PDSs), which provide a

convenient abstraction for the program's runtime stack. Modeling the stack is important for programs with procedures because it allows precise reasoning about the call-return semantics of procedure calls.

Algorithms for PDSs are based on automata-theoretic techniques. The set of all reachable states of a PDS can be captured using a finite-state machine. This allows one to leverage the vast existing knowledge about finite-state machines to design various analyses of PDSs. However, the situation changes when dealing with WPDSs and EWPDSs. The set of all reachable states of such models can only be captured using weighted automata. Thus, one no longer has the same rich collection of techniques available as one has for unweighted automata.

This dissertation presented several new algorithms for weighted automata. For instance, in Chapter 5, we gave a method to intersect two weighted automata under the restriction that one automaton is a forward-weighted automaton and the other is a backward-weighted automaton. The algorithm allowed us to intersect the set of forward-reachable states from the start of the program with the backward-reachable states from an error point in the program to compute an error projection. In Chapter 6, we further extended this result to show how to intersect any two weighted automata, provided that a tensor-product operation exists for weights. The result was then generalized to composition of weighted transducers, which provided an algorithm for CBA.

These results on weighted automata are general and form the building blocks of some of the verification techniques described in the dissertation. They may be useful for solving other verification or program-analysis problems. Moreover, these results are of interest in their own right, and should be applicable to problems outside the areas of verification and program analysis.

LIST OF REFERENCES

[1] A. Aho, J. Hopcroft, and J. Ullman. Time and tape complexity of pushdown automaton languages. *Information and Control*, 13(3):186–206, 1968.

[2] G. Balakrishnan. *WYSINWYX: What You See Is Not What You eXecute*. PhD thesis, Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI, Aug. 2007. Tech. Rep. 1603.

[3] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *CC*, 2004.

[4] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI*, 2001.

[5] T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: localizing errors in counterexample traces. In *POPL*, 2003.

[6] T. Ball and S. Rajamani. Bebop: A symbolic model checker for Boolean programs. In *SPIN*, 2000.

[7] T. Ball and S. K. Rajamani. Boolean programs: A model and process for software analysis. Technical Report MSR-TR-2000-14, Microsoft Research, 2000.

[8] F. Berger, S. Schwoon, and D. Suwimonteerabuth. jMoped, 2005. `http://www.informatik.uni-stuttgart.de/fmi/szs/tools/moped/jmoped/`.

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

[10] A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. In *POPL*, 2003.

[11] A. Bouajjani, S. Fratani, and S. Qadeer. Context-bounded analysis of multithreaded programs with dynamic linked structures. In *CAV*, 2007.

[12] A. Bouajjani, M. Müller-Olm, and T. Touili. Regular symbolic analysis of dynamic networks of pushdown systems. In *CONCUR*, 2005.

[13] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Int. Conf. on Formal Methods in Prog. and their Appl.*, 1993.

[14] R. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(6):677–691, Aug. 1986.

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

[16] O. Burkart and B. Steffen. Model checking for context-free processes. In *CONCUR*, pages 123–137, 1992.

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

[18] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *ICSE*, 2003.

[19] S. Chaki, E. M. Clarke, N. Kidd, T. W. Reps, and T. Touili. Verifying concurrent message-passing C programs with recursive calls. In *TACAS*, 2006.

[20] S. Chaki, A. Groce, and O. Strichman. Explaining abstract counterexamples. In *FSE*, 2004.

[21] H. Chen and D. Wagner. MOPS: An infrastructure for examining security properties of software. In *Conf. on Computer and Communications Security*, November 2002.

[22] C. L. Conway, K. S. Namjoshi, D. Dams, and S. A. Edwards. Incremental algorithms for inter-procedural analysis of safety properties. In *CAV*, 2005.

[23] P. Cousot. Méthodes itératives de construction et d'approximation de point fixes d'opérateurs monotones sur un treillis, analyse sémantique des programmes. Thèse ès sciences mathématiques, Univ. of Grenoble, 1978.

[24] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[25] P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In *Formal Descriptions of Programming Concepts*, pages 237–277, 1978.

[26] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, 1979.

[27] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, 1978.

[28] D. R. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI*, pages 1–16, 2000.

[29] U. Erlingsson. *The inlined reference monitor approach to security policy enforcement.* PhD thesis, Cornell University, Ithaca, NY, USA, 2004.

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

[31] J. Esparza, A. Kučera, and S. Schwoon. Model-checking LTL with regular valuations for pushdown systems. In *TACAS*, 2001.

[32] J. Esparza and S. Schwoon. A BDD-based model checker for recursive programs. In *CAV*, 2001.

[33] A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. *Electronic Notes in Theoretical Comp. Sci.*, 9, 1997.

[34] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[35] GrammaTech, Inc. CodeSurfer Path Inspector, 2005. `http://www.grammatech.com/products/codesurfer/overview_pi.html`.

[36] S. Gulwani and G. C. Necula. Precise interprocedural analysis using random interpretation. In *POPL*, 2005.

[37] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, 2002.

[38] T. A. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. In *PLDI*, 2004.

[39] H. S. Hong, I. Lee, and O. Sokolsky. Abstract slicing: A new approach to program slicing based on abstract interpretation and model checking. In *SCAM*, pages 25–34, 2005.

[40] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[41] B. Jeannet and W. Serwe. Abstracting call-stacks for interprocedural verification of imperative programs. In *AMAST*, 2004.

[42] T. Jensen, D. L. Métayer, and T. Thorn. Verification of control flow based security properties. In *IEEE Symposium on Security and Privacy*, pages 89–103, 1999.

[43] V. Kahlon and A. Gupta. On the analysis of interacting pushdown systems. In *POPL*, 2007.

[44] J. Kam and J. Ullman. Monotone data flow analysis frameworks. *Acta Inf.*, 7(3):305–318, 1977.

[45] M. Karr. Affine relationships among variables of a program. *Acta Inf.*, 6:133–151, 1976.

[46] K. W. Kennedy. Node listings applied to data flow analysis. In *POPL*, 1975.

[47] N. Kidd, A. Lal, and T. Reps. WALi: The Weighted Automaton Library, Feb. 2009. `http://www.cs.wisc.edu/wpis/wpds`.

[48] N. Kidd, A. Lal, and T. W. Reps. Language strength reduction. In *SAS*, pages 283–298, 2008.

[49] N. Kidd, T. Reps, D. Melski, and A. Lal. WPDS++: A C++ library for weighted pushdown systems, 2005. `http://www.cs.wisc.edu/wpis/wpds`.

[50] S. Kiefer, S. Schwoon, and D. Suwimonteerabuth. Moped. `http://www.fmi.uni-stuttgart.de/szs/tools/moped/`.

[51] G. Kildall. A unified approach to global program optimization. In *POPL*, pages 194–206, 1973.

[52] J. Knoop and B. Steffen. The interprocedural coincidence theorem. In *CC*, 1992.

[53] J. Kodumal and A. Aiken. Banshee: A scalable constraint-based analysis toolkit. In *SAS*, pages 218–234, 2005.

[54] T. Kremenek, K. Ashcraft, J. Yang, and D. R. Engler. Correlation exploitation in error ranking. In *SIGSOFT FSE*, pages 83–93, 2004.

[55] S. K. Lahiri, S. Qadeer, and Z. Rakamaric. Static and precise detection of concurrency errors in systems code using SMT solvers. In *CAV*, 2009.

[56] A. Lal, J. Lim, M. Polishchuk, and B. Liblit. Path optimization in programs and its application to debugging. In *ESOP*, 2006.

[57] A. Lal and T. Reps. Solving multiple reachability queries on WPDSs. In *SAS*, 2007.

[58] A. Lal and T. Reps. Reducing concurrent analysis under a context bound to sequential analysis. In *CAV*, pages 37–51, 2008.

[59] A. Lal and T. Reps. Reducing concurrent analysis under a context bound to sequential analysis. In *FMSD*, 2009.

[60] A. Lal, T. Reps, and G. Balakrishnan. Extended weighted pushdown systems. In *CAV*, 2005.

[61] W. Landi and B. Ryder. Pointer-induced aliasing: A problem classification. In *POPL*, 1991.

[62] D. Massé. Combining forward and backward analyses of temporal properties. In *PADO*, 2001.

[63] Y. Matsunaga, P. C. McGeer, and R. K. Brayton. On computing the transitive closure of a state transition relation. In *Design Automation Conference (DAC)*, pages 260–265, 1993.

[64] M. Mohri, F. Pereira, and M. Riley. Weighted automata in text and speech processing. In *ECAI*, 1996.

[65] M. Mohri, F. Pereira, and M. Riley. The design principles of a weighted finite-state transducer library. In *TCS*, 2000.

[66] M. Müller-Olm and H. Seidl. On optimal slicing of parallel programs. In *STOC*, 2001.

[67] M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. In *POPL*, 2004.

[68] M. Müller-Olm and H. Seidl. Analysis of modular arithmetic. In *ESOP*, 2005.

[69] B. Murphy and M. Lam. Program analysis with partial transfer functions. In *PEPM*, 2000.

[70] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI*, 2007.

[71] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[72] G. Patin, M. Sighireanu, and T. Touili. Spade: Verification of multithreaded dynamic and recursive programs. In *CAV*, 2007.

[73] R. Pelánek. BEEM: Benchmarks for explicit model checkers. In *SPIN*, 2007.

[74] E. Post. A variant of a recursively unsolvable problem. *Bulletin of the American Mathematical Society*, 52, 1946.

[75] S. Qadeer and S. Rajamani. Deciding assertions in programs with references. Technical Report MSR-TR-2005-08, Microsoft Research, Redmond, Jan. 2005.

[76] S. Qadeer, S. K. Rajamani, and J. Rehof. Summarizing procedures in concurrent programs. In *POPL*, 2004.

[77] S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *TACAS*, 2005.

[78] S. Qadeer and D. Wu. KISS: Keep it simple and sequential. In *PLDI*, 2004.

[79] G. Ramalingam. Data flow frequency analysis. In *PLDI*, pages 267–277, 1996.

[80] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. In *TOPLAS*, 2000.

[81] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, 1995.

[82] T. Reps, S. Schwoon, and S. Jha. Weighted pushdown systems and their application to interprocedural dataflow analysis. In *SAS*, 2003.

[83] T. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. In *SCP*, volume 58, 2005.

[84] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *TCS*, 167, 1996.

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

[86] S. Schwoon. WPDS: A library for weighted pushdown systems, 2003. `http://www.fmi.uni-stuttgart.de/szs/tools/wpds/`.

[87] S. Schwoon, S. Jha, T. Reps, and S. Stubblebine. On generalized authorization problems. In *Comp. Sec. Found. Workshop*, Wash., DC, 2003. IEEE Comp. Soc.

[88] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.

[89] D. Suwimonteerabuth, J. Esparza, and S. Schwoon. Symbolic context-bounded analysis of multithreaded Java programs. In *SPIN*, pages 270–287, 2008.

[90] R. E. Tarjan. Fast algorithms for solving path problems. *J. ACM*, 28(3):594–614, 1981.

[91] R. E. Tarjan. A unified approach to path problems. *J. ACM*, 28(3):577–593, 1981.

[92] S. L. Torre, P. Madhusudan, and G. Parlato. Reducing context-bounded concurrent reachability to sequential reachability. In *CAV*, 2009.

[93] M. Weiser. Program slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984.

[94] J. Whaley, D. Avots, M. Carbin, and M. S. Lam. Using datalog with binary decision diagrams for program analysis. In *APLAS*, pages 97–118, 2005.

[95] Wikipedia. Kronecker product, July 2009. `http://en.wikipedia.org/wiki/Kronecker_product`.

[96] T. Witkowski, N. Blanc, D. Kroening, and G. Weissenbacher. Model checking concurrent linux device drivers. In *ASE*, 2007.