Computer
Sciences
Department

Interprocedural Analysis of Concurrent Programs Under a Context Bound

Akash Lal
Tayssir Touili
Nicholas Kidd
Thomas Reps

Technical Report #1598

July 2007

UNIVERSITY OF
WISCONSIN
MADISON

Interprocedural Analysis of Concurrent Programs Under a Context Bound

Akash Lal
University of Wisconsin
akash@cs.wisc.edu

Tayssir Touili
LIAFA, CNRS & University of Paris 7
touili@liafa.jussieu.fr

Nicholas Kidd Thomas Reps
University of Wisconsin
{kidd, reps}@cs.wisc.edu

Abstract

Analysis of recursive programs in the presence of concurrency and shared memory is undecidable. A common approach is to remove the recursive nature of the program while dealing with concurrency. A different approach is to bound the number of context switches, which has been shown to be very useful for program analysis. In previous work, Qadeer and Rehof [36] showed that context-bounded analysis is decidable for recursive programs under a finite-state abstraction of program data. In this paper, we generalize their result to infinite-state abstractions, and also provide a new symbolic algorithm for the finite case.

1. Introduction

In this paper, we consider analysis of concurrent programs with shared-memory and interleaving semantics. Analysis of such programs is generally considered hard because of the large number of interleavings the analysis has to consider. Especially hard is the analysis of recursive programs, because their control state space is infinite. In fact, the analysis of recursive programs, even with a finite-state abstraction of data (such as in Boolean programs), is undecidable in the presence of concurrency and shared memory.

An analysis for recursive programs has to accurately model the procedure call and return semantics, i.e., it should only consider program executions in which a procedure return is matched with the most recent call. Such *interprocedural* analyses have proven to be very useful for sequential programs [3, 39–41]. Considering the desirability of interprocedural analysis, its undecidability in the presence of concurrency is unfortunate. 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.

A different way to sidestep the undecidability issue 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 very useful for program analysis because many bugs can be found in a few context switches [32, 36, 37]. We call an analysis of a recursive and concurrent program under a context bound, a *context-bounded analysis* (CBA).

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 [32].

In previous work, Qadeer and Rehof [36] showed that CBA is decidable for recursive programs under a finite-state abstraction of program data. In this paper, we generalize their result to infinite-state abstractions, and also provide a new symbolic algorithm for the finite 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) [20, 22, 43]. 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, we require the existence of a *tensor product* (defined in §6). 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 [3], as well as infinite-state abstractions, such as affine-relation analysis (ARA) [29]. Note that without a context bound, reasoning about concurrent programs under these abstractions is undecidable [28, 38].

For a precise CBA, one needs to start off with a precise interprocedural analysis. *Weighted pushdown systems* (WPDSs) [25, 40] are a general model for interprocedural analysis. They are a generalization of pushdown systems (PDSs). PDSs can model recursive programs [42], and WPDSs add a general “black-box” abstraction for program data (through *weights*) to PDSs. WPDSs also generalize other frameworks for interprocedural analysis such as the Sharir-Pnueli functional approach [43] and the Reps-Horwitz-Sagiv summary-based approach [41]. 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.

1.1 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 very useful for program analysis. KISS, a verification tool that analyzes programs for up to two context switches, was able to find a number of bugs in drivers [37]. Another study, using explicit-state model checking, also confirms that many bugs can be found in a few context switches [32]. Moreover, it shows state-space coverage graphs that indicate that many program behaviors are captured in the first few context switches, with fewer behaviors being added with additional context switches. 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 very useful for analyzing concurrent programs. For example, consider the program snippet in Fig. 1. Here, multiple threads share the circular buffer q in a producer (enq) consumer (deq) fashion. Using CBA with ARA with modular arithmetic, one can prove (under a given context bound) that (hd - tl - cnt) % SIZE = 0 provided SIZE is a prime power [30]. 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 increase the precision of other analysis.

```
Elem q[SIZE];              void enq(Elem e) {         Elem deq() {
int hd = cnt = tl = 0;       while(true) {              while(true) {
                             atomic {                   atomic {
                               if( cnt < SIZE) {          if(cnt > 0) {
                                 q[tl] = e;                 Elem e = q[hd];
                                 tl = (tl+1)%SIZE;          hd = (hd+1)%SIZE;
                                 cnt ++;                    cnt--;
                                 break;                     return e;
                               }                          }
                           }}}                        }}}
```

Figure 1. A concurrent program that manages a circular queue.

The context bound 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".

1.2 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. A CBA has to consider the possibility of a context switch occurring at any one of these states.

The Qadeer-Rehof (QR) algorithm used (unweighted) PDSs (which can encode recursive programs with a finite data abstraction) to encode program threads. An influential result by Büchi [9] showed that the set of reachable states of a PDS can be represented using an automaton. 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 over all reachable data states at a context switch.

Our first step is to develop a new algorithm for the case of unweighted PDSs. Our motivation is to have an algorithm that is more likely to generalize to handle other abstractions. The new algorithm (§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 [10] showed that such transducers can be constructed for PDSs, a result more general than that of Büchi's. Next, these transducers are composed to describe the behavior of the entire program with multiple threads. One transducer composition is performed for each context switch.

We then generalize this algorithm for WPDSs (§5 and §6). 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 (§6). Tensor products have been used previously in program analysis for combining abstractions [33]. 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 nontrivial 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 this paper 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 [36]. 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 [40], 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 (§5), generalizing previous result for PDSs by Caucal [10]. The use of weighted transducers (instead of Büchi's result, or its generalization to weighted systems [40]) appears to be a necessary step for CBA with infinite-state data abstractions.
- We give precise algorithms for composing weighted transducers (§6), when tensor products exist for the weights. This generalizes previous work on manipulating weighted automata and transducers [26, 27]. We also show a class of abstractions that satisfies this property.
- We discuss implementation issues for realizing CBA in §7. 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 paper is organized as follows. In §2, we discuss previous work on CBA under a finite-state data abstraction. In §3, we present our algorithm based on transducers. In §4, we give background on WPDSs. In §5, we give an efficient construction for transducers for WPDSs. In §6, we show how weighted transducers can be composed. In §7, we discuss implementation issues for CBA. In §8, we discuss related work.

2. Context Bounded Model Checking

In this section, we consider CBA under a finite-state data abstraction, which we call context-bounded model checking or CBMC. Here, each thread of a concurrent program is modeled using a PDS.

First we define *Boolean programs*, a popular program abstraction used in model checking [3]. They serve as a program-modeling framework that provides finite data and unbounded control. We show how PDSs can encode them. We then formally define the CBMC problem, and discuss the QR algorithm.

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.

2.1 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. We assume that procedures do not have parameters (they can be passed through the global variables). The variables in scope inside a procedure are the global variables and its set of local variables.

A procedure is described by its *control-flow graph* (CFG), which has a designated entry and a designated exit node. Nodes of the graph are program control locations, and each edge is labeled with a statement. A statement can be an assignment, reading from and writing to variables in scope; or an assume statement (for conditions); or a procedure call. An example is shown in Fig. $4(a)$.

Let G be the set of global states of the program, consisting of valuations of global variables. Let L be the set of local states of the program, consisting of the program counter, a valuation of local variables, and the program stack (consisting of return addresses and a valuation of the local variables for each unfinished call).

2.2 Pushdown Systems

The semantics of a Boolean program can be given nicely in terms of PDSs.

DEFINITION 1. *A **pushdown system** is a triple $\mathcal{P} = (P, \Gamma, \Delta)$, where P is a finite set of states or control locations, Γ is a finite set of stack symbols, and $\Delta \subseteq P \times \Gamma \times P \times \Gamma^*$ is a finite set of rules. A **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 $u \in \Gamma^*$. These rules define a transition relation $\Rightarrow$ on configurations of $\mathcal{P}$ as follows: If $r = \langle p, \gamma \rangle \hookrightarrow \langle p', u \rangle$ then $\langle p, \gamma u' \rangle \Rightarrow \langle p', uu' \rangle$ for all $u' \in \Gamma^*$. The reflexive transitive closure of $\Rightarrow$ is denoted by $\Rightarrow^*$. For a set of configurations C, we define $pre^*(C) = \{c' \mid \exists c \in C : c' \Rightarrow^* c\}$ and $post^*(C) = \{c' \mid \exists c \in C : c \Rightarrow^* c'\}$, which are just backward and forward reachability under the transition relation $\Rightarrow$.*

Without loss of generality, we restrict the pushdown rules to have at most two stack symbols on the right-hand side [42].

Encoding Boolean programs. The standard approach for modeling the program control flow with a pushdown system is as follows: P contains a single state $\{p\}$, Γ corresponds to the program locations, and Δ corresponds to edges of the CFG (see Fig. 2).

For encoding Boolean programs with PDSs, the state alphabet P is expanded to encode the values of global variables, and the stack alphabet is expanded to encode the values of local variables [42]. Let G be the set of valuations of global variables, Val_i be the set of valuations of local variables, and N_i be the set of control locations of the i^{th} procedure. The effect of executing an assignment or assume statement `st`, denoted as $[\![\texttt{st}]\!]$, is a binary relation on $G \times \mathrm{Val}_i$ that describes how values of variables in scope can change. We set P to be G, and Γ to be the union of $N_i \times \mathrm{Val}_i$ over all procedures (note that the set of local states L equals Γ^*). Rules for the i^{th} procedure are constructed as follows: (i) a CFG edge $u \rightarrow v$ with a statement `st` is encoded as a set of rules $\langle g, (u, l) \rangle \hookrightarrow \langle g', (v, l') \rangle$ such that $((g, l), (g', l')) \in [\![\texttt{st}]\!]$; (ii) a procedure return at node u is a set of rules $\langle g, (u, l) \rangle \hookrightarrow \langle g, \varepsilon \rangle$ for each $(g, l) \in G \times \mathrm{Val}_i$; (iii) a call edge $c \rightarrow r$ that calls procedure f, with entry node f_{enter}, is a set of rules $\langle g, (c, l) \rangle \hookrightarrow \langle g, (f_{\text{enter}}, l_0)\ (r, l) \rangle$, for all $(g, l) \in G \times \mathrm{Val}_i$ and $l_0 \in \mathrm{Val}_f$.

Rule	Control flow modeled
$\langle p, u \rangle \hookrightarrow \langle p, v \rangle$	CFG edge $u \rightarrow v$, which is not a call
$\langle p, c \rangle \hookrightarrow \langle p, f_{\text{enter}}\ r \rangle$	CFG edge $c \rightarrow r$, which calls procedure f beginning at node f_{enter}
$\langle p, f_{\text{exit}} \rangle \hookrightarrow \langle p, \varepsilon \rangle$	Return from procedure f at f_{exit}

Figure 2. The encoding of control flow as PDS rules.

Under such an encoding of Boolean programs as PDSs, a configuration $\langle p, \gamma_1 \gamma_2 \cdots \gamma_n \rangle$ is an element of $G \times L$ that describes the instantaneous state of a program. The state p encodes the values of global variables; γ_1 encodes the current program location 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 PDS transition relation ($\Rightarrow$), which is essentially a transition relation on $G \times L$, represents the semantics of the Boolean program.

The problem of interest, for sequential programs, is to find the set of all reachable configurations, starting from a given set of configurations. This can then be used, for example, for assertion checking (i.e., determining if a given assertion can ever fail) or to find the set of all data values that may arise at a program point (for dataflow analysis). Because the number of configurations of a pushdown system is unbounded, it is useful to use finite automata to describe regular sets of configurations.

DEFINITION 2. *If $\mathcal{P} = (P, \Gamma, \Delta)$ is a pushdown system then a $\mathcal{P}$-**automaton** is a finite automaton $(Q, \Gamma, \rightarrow, P, F)$, where $Q \supseteq P$ is a finite set of states, $\rightarrow \subseteq Q \times \Gamma \times Q$ is the transition relation, P is the set of initial states, and F is the set of final states. 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 $pre^*(C)$ (the forward and the backward reachable sets of configurations, respectively) are also regular sets of configurations [5, 9]. 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 [5, 13, 14].

2.3 Concurrent Boolean Programs and CBMC

A *concurrent Boolean program* is a set of Boolean programs (one for each thread) where the global variables are shared between the threads. Thus, any of the threads can modify the global variables, but they have their own copy of the local variables. Synchronization is easily implementable using global variables as locks. Analysis of such models is undecidable in general [38], i.e., it is not possible to verify if a given configuration is reachable or not. However, Qadeer and Rehof have shown that CBMC is decidable. Let n be the number of threads and let $t_1, t_2, \cdots, t_n$ denote the threads. We do not consider dynamic creation of threads in our model.[1]

Let G be the set of global states (valuations of global variables) and L_i be the set of local states of t_i (as described before). Then the state space of the entire program consists of the global state

[1] Dynamic creation up to n threads can be encoded in the model [36]. Moreover, for CBA that considers k context switches, n can be bounded by k because other threads would never get a chance to run.

paired with local states of each of the threads, i.e., the set of states is $G \times L_1 \times \cdots \times L_n$. A concurrent program can be represented by n PDSs, one for each thread, where the PDSs share the same set of global states G.

Let the transition relation of thread t_i be $\Rightarrow_{t_i}$, which is a binary relation on $G \times L_i$ as described in the previous section. If $(g, l_i) \Rightarrow_{t_i} (g', l_i')$, the transition $(g, l_1, \cdots, l_i, \cdots, l_n) \Rightarrow_{t_i}^c (g', l_1 \cdots, l_i', \cdots, l_n)$ is a valid transition for the concurrent program.

The execution of a concurrent program proceeds in 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. The CBMC problem is to find the set of reachable states in the transition relation of the concurrent program under a bound on the number of context switches. Formally, let k be the bound on context switches. Then there are $k+1$ execution contexts. Let $\Rightarrow_1^c$ be $(\cup_{i=1}^n (\Rightarrow_{t_i}^c)^*)$, the transition relation that describes the effect of one execution context. Then we wish to find the reachable states in the transition relation given by $(\Rightarrow_1^c)^{k+1}$. The reachable states could be used, for example, to find out the data values when t_1 is at node n_1 and the rest of the threads can be anywhere, or when t_1 is at n_1 and t_2 is at node n_2 and so on. 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.

2.4 The Qadeer-Rehof Algorithm for CBMC

The Qadeer-Rehof (QR) algorithm 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 in terms of explicit state spaces. Its implementation using PDSs is described in [36].

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.
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. 3 in the form of a tree.

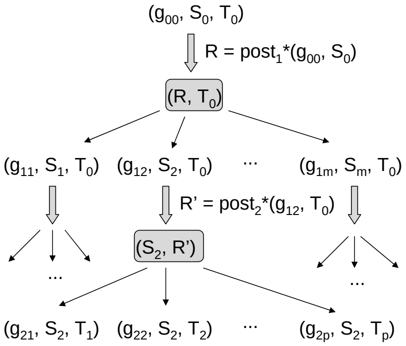


Figure 3. The computation of the QR algorithm, for two threads, shown schematically in the form of a tree. The shaded boxes are just temporary placeholders and are not inserted into the worklist. The thick arrows correspond to Step 3 and other arrows correspond to Step 4. The set of tuples at level i of the tree correspond to all states reached in i context switches.

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. The sets S_i are kept in symbolic form using automata (Defn. 2). The *poststar* algorithm works on these representations, mapping automata (capturing starting configurations) to automata (capturing reachable configurations).

3. 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. 3) to pass the global state to all other threads. This is also true for the automaton-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 §5 and §6).

The QR algorithm makes several calls to the PDS-based algorithm *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. Both of these sets can be infinite, and the summary must have some symbolic representation. We construct the summary using a finite-state transducer (an automaton with input and output tapes).

DEFINITION 3. *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. This result was first given by Caucal [10], 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 [5, 13, 40, 42] for an efficient construction. Since such transducers are of general importance, we give a complexity result. The following theorem is derived from Thm. 2 given in §5.

THEOREM 1. *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 1. *Given transducers τ_1 and τ_2 with input and output alphabet Σ, one can construct a transducer $(\tau_1; \tau_2)$ such that $\mathcal{L}(\tau_1; \tau_2) = \mathcal{L}(\tau_1); \mathcal{L}(\tau_2)$. 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 [18]. 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 $\Rightarrow_{t_i}^*$. By extending τ_{t_i} to perform the identity transformation on stack symbols of threads other than t_i (using transitions of the form $p \xrightarrow{\gamma/\gamma} q$), we obtain a transducer $\tau_{t_i}^c$ for $(\Rightarrow_{t_i}^c)^*$. Next, a union of these transducers gives τ_1^c, which represents $\Rightarrow_1^c$. Performing the composition of τ_1^c k times with itself gives us a transducer τ that represents $(\Rightarrow_1^c)^{k+1}$. 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).

Roadmap for the Remainder of the Paper

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 problem, 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 (§4). Extending our algorithm to WPDSs presents two challenges: one is the construction of a weighted transducer for a WPDS, and the other is their composition. These issues are addressed in §5 and §6, respectively.

4. Weighted Pushdown Systems

A weighted pushdown system is obtained by augmenting a PDS with a weight domain that is a *bounded idempotent semiring* [6, 40]. Such semirings are powerful enough to encode finite-state data abstractions, such as the one required for bitvector dataflow analysis, Boolean program verification, or the IFDS framework of Reps-Horwitz-Sagiv [39], as well as infinite-state data abstractions, such as linear-constant propagation and affine-relation analysis [29]. We recall some of this here, but details on using WPDSs for interprocedural analysis can be found in [40].

Weights encode the effect that each statement (or PDS rule) has on the data state of the program. They can be thought of as abstract transformers, that specify how the abstract state changes when a statement is executed. WPDSs compute over these weights. Computing over transformers, instead of the underlying abstract states, is customary for interprocedural analysis [12, 22, 39] where procedure summaries need to be calculated as transformations on abstract states.

DEFINITION 4. *A **bounded idempotent semiring** is a tuple $(D, \oplus, \otimes, \overline{0}, \overline{1})$, where D is a set whose elements are called **weights**, $\overline{0}, \overline{1} \in D$, 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 $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$ 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 $\forall a, b \in D,\ a \sqsubseteq b$ iff $a \oplus b = a$, there are no infinite descending chains.*

DEFINITION 5. *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 rule of $\mathcal{P}$.*

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 $v(\sigma) \overset{\text{def}}{=} f(r_1) \otimes \ldots \otimes f(r_k)$. Moreover, for any two configurations c and c' of $\mathcal{P}$, we use $path(c, c')$ to denote the set of all rule sequences that transform c into c'. If $\sigma \in path(c, c')$, then we say $c \Rightarrow^\sigma c'$. Reachability problems on PDSs are generalized to WPDSs as follows:

DEFINITION 6. *Let $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$ be a weighted pushdown system, where $\mathcal{P} = (P, \Gamma, \Delta)$, and let $S, T \subseteq P \times \Gamma^*$ be regular sets of configurations. Then the **meet-over-all-paths** value MOP(S, T) is defined as $\bigoplus\{v(\sigma) \mid s \Rightarrow^\sigma t, s \in S, t \in T\}$.*

We call bounded idempotent semirings *weight domains*.

A PDS is simply a WPDS with the *Boolean weight domain* $(\{\overline{0}, \overline{1}\}, \oplus, \otimes, \overline{0}, \overline{1})$ and weight assignment $f(r) = \overline{1}$ for all rules $r \in \Delta$. In this case, MOP$(S, T) = \overline{1}$ iff there is a path from a configuration in S to a configuration in T, i.e., $post^*(S) \cap T$ and $S \cap pre^*(T)$ are non-empty sets.

One way of modeling programs as WPDSs is as follows: the PDS models the control flow of the program, as in Fig. 2. The weight domain models abstract transformers for an abstraction of the program's data. The next two sections describe two data abstractions that can be encoded using weight domains. For simplicity, we only show the treatment for global variables, and do not consider local variables. Local variables (under an infinite-state abstraction) pose an extra complication for WPDSs [25], and their treatment can be found in App. B. Finite-state abstraction of local variables can always be encoded in the stack alphabet, as for PDSs.

4.1 Finite-State Abstractions

An important weight domain for WPDSs is the set of all binary relations on a finite set.

DEFINITION 7. *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.*

Instantiating G to be the set of global states of a Boolean program, we obtain a weight domain for Boolean programs. The weight associated with a rule is its effect on the global state, which, as described earlier, is a binary relation on G. (Methods for handling local variables can be found in [25, 42].) An example is shown in Fig. 4(b). The Boolean program has two variables ranging over the set V, so $G = V \times V$, with the first component being the value of $\mathtt{x}$. Weights are shown using a shorthand notation, e.g., $((v_1, v_2), (v_1, v_1))$ represents the set $\{((v_1, v_2), (v_1, v_1)) \mid v_1, v_2 \in V\}$.

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 start node and T be the set $\{\langle p, n\, u\rangle \mid u \in \Gamma^*\}$. Let $w = \mathrm{MOP}(S, T)$. 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 that reach node n. For example, in Fig. 4, the MOP weight to node n_6 is the weight w_6 shown in the figure. Its range shows that $\mathtt{x} = \mathtt{y} = 3$ or $\mathtt{x} = \mathtt{y} = 7$.

Because T can be any regular set, one can also make stack-qualified queries [40]. 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 $T = \{\langle p, n\, m\, u\rangle \mid u \in \Gamma^*\}$

A WPDS with a weight domain that has a finite set of weights, such as the one described above, can be encoded as a PDS. However, it is often useful to use weights because they can be symbolically encoded. Tools such as MOPED, BEBOP, and BLAST use BDDs to encode sets of data values, allowing them to scale to a large number of variables. Using PDSs for Boolean program verification, without any symbolic encoding, would not be feasible.

4.2 Infinite-State Abstractions

An infinite-state abstraction is one in which the number of abstract states (or weights) is infinite. We begin with some simple examples of infinite-weight domains, and then discuss the one used for affine-relation analysis.

DEFINITION 8. *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 MOP weight between two configurations is the length of the shortest path (shortest rule sequence) between them. Another infinite-weight domain, based on the minpath semiring, is given in [24] and was shown to be useful for debugging programs.

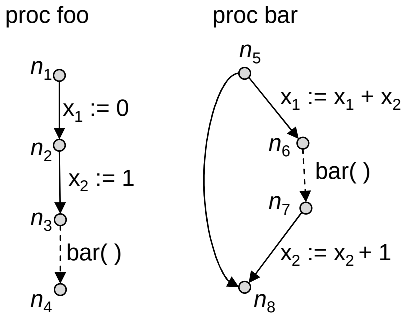


Figure 5. An affine program that starts execution at node $\mathtt{n}_1$. There are two global variables $\mathtt{x}_1$ and $\mathtt{x}_2$.

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 exhibiting some property).

DEFINITION 9. *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 10. *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.*

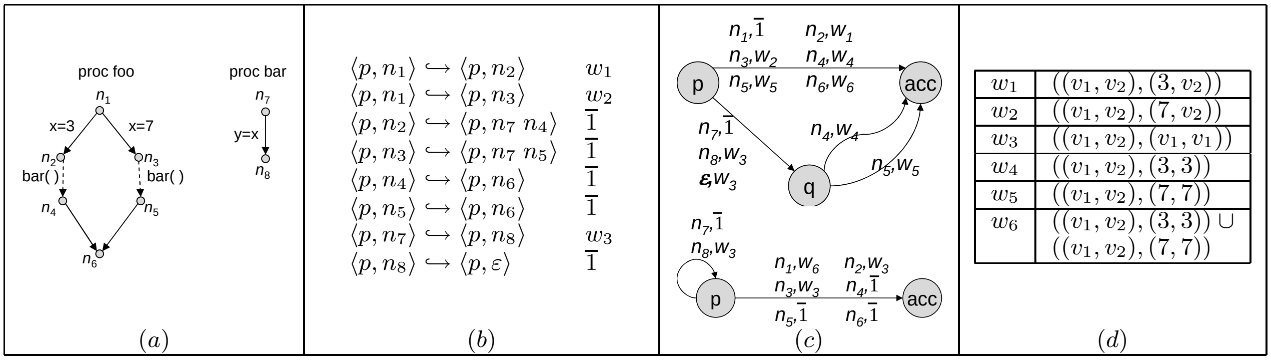
If G is the set of global states of a Boolean program, then the relational weight domain on $(G, \mathcal{M})$ can be used for finding the shortest trace: for each rule, if $R \subseteq G \times G$ is the effect of executing the rule on the global state of the Boolean program, then associate the following weight with the rule: $(\lambda(g_1, g_2).\text{if}((g_1, g_2) \in R)$ then 1 else $\infty)$. Then, if $w = \mathrm{MOP}(C_1, C_2)$, 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). Such a weight domain is a small extension over the pure relational domain for a Boolean program. However, the QR algorithm cannot handle this abstraction, whereas the algorithm we gave in §3 can be generalized to handle it (as shown in §5 and §6).

4.2.1 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. 5. For this program, ARA would, for example, infer that $\mathtt{x}_2 = \mathtt{x}_1 + 1$ at program node $\mathtt{n}_4$.

ARA for single-procedure programs was first given by Karr [21]. It took almost 30 years to develop an analysis for multi-procedure programs [29]. Using the results of this paper, we can extend ARA to deal with (context-bounded) concurrency. The advantage of our framework is that we get a CBA automatically from an interprocedural analysis.

ARA generalizes other analyses, including copy-constant propagation, linear-constant propagation [41], and induction-variable analysis [21]. We have used ARA (for sequential programs) on



$\langle p, n_1\rangle \hookrightarrow \langle p, n_2\rangle$	w_1
$\langle p, n_1\rangle \hookrightarrow \langle p, n_3\rangle$	w_2
$\langle p, n_2\rangle \hookrightarrow \langle p, n_7\ n_4\rangle$	$\overline{1}$
$\langle p, n_3\rangle \hookrightarrow \langle p, n_7\ n_5\rangle$	$\overline{1}$
$\langle p, n_4\rangle \hookrightarrow \langle p, n_6\rangle$	$\overline{1}$
$\langle p, n_5\rangle \hookrightarrow \langle p, n_6\rangle$	$\overline{1}$
$\langle p, n_7\rangle \hookrightarrow \langle p, n_8\rangle$	w_3
$\langle p, n_8\rangle \hookrightarrow \langle p, \varepsilon\rangle$	$\overline{1}$

w_1	$((v_1, v_2), (3, v_2))$
w_2	$((v_1, v_2), (7, v_2))$
w_3	$((v_1, v_2), (v_1, v_1))$
w_4	$((v_1, v_2), (3, 3))$
w_5	$((v_1, v_2), (7, 7))$
w_6	$((v_1, v_2), (3, 3)) \cup$ $((v_1, v_2), (7, 7))$

Figure 4. (a) A Boolean program with two procedures and two global variables x and y over a finite domain $V = \{0, 1, \cdots, 7\}$. (b) A WPDS that models the Boolean program. (c) 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*. (d) An index of the weights used in this figure. The unbound variables v_i are universally quantified over V.

machine code to find induction-variable relationships between machine registers [2]. These help in increasing the precision of an abstract-interpretation based pointer analysis for machine code.

The Analysis

Interprocedural ARA can be performed precisely on affine programs, and has been the focus of several papers [16, 21, 29, 30]. Affine programs are similar to Boolean programs, but with integer-valued variables. Again, we restrict our attention to global variables, and defer treatment of local variables to App. B. All branch conditions in affine programs are non-deterministic (ARA cannot interpret conditions). If $\{\mathtt{x}_1, \mathtt{x}_2, \cdots, \mathtt{x}_n\}$ is the set of global variables of the program, then all assignments 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 a linear function of the variables.)

ARA Weight Domain

We briefly describe the weight domain based on the matrix-formulation of ARA from [29]. An affine relation $a_0 + \sum_{i=1}^n a_i \mathtt{x}_i = 0$ is represented using column vector of size $n+1$ as $\vec{a} = (a_0, a_1, \cdots, a_n)^t$. 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 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 of their linear combinations. 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. Therefore, the set of affine relations that hold at a program node form a vector space. This implies that a (possibly infinite) set of affine relations can be represented by its linearly independent basis (which is always a finite set).

For reasoning about affine programs, each statement is abstracted by a set of matrices of size $(n+1) \times (n+1)$ (this abstraction turns out to be precise, i.e., it is able to find all affine relationships in the affine program). This set is the weakest-precondition transformer on affine relations for that 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 $(m_1\vec{a}), (m_2\vec{a}), \cdots, (m_r\vec{a})$ held before the execution of the statement.

Under such an abstraction of program statements, one can define the extend operation, which is transformer composition, as elementwise matrix multiplication, 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 unboundedly. 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 its basis to get a finite and a bounded representation: a vector space over $(n+1) \times (n+1)$ sized matrices cannot have more that $(n+1)^2$ number of matrices in its 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 can be represented using its basis. 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 the 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, MOP(S, T) is a weight that is the net weakest-precondition transformer between S and T. Suppose this weight has basis $\{m_1, \cdots, m_r\}$. The affine relation representing 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 $m_1\vec{a} = m_2\vec{a} = \cdots = m_r\vec{a} = \vec{0}$. The set of all affine relations that hold at T can be found as the intersection of null spaces of the matrices $m_1, m_2, \cdots, m_r$.

Extensions to ARA

ARA can also be performed for modular arithmetic to precisely model machine arithmetic (which is modulo 2 to the power of the word size) [30]. Our result for CBA holds for both integer arithmetic and modular arithmetic, but we only focus on the former in this paper.

ARA in the presence of branch conditions is undecidable in general. However, there are approximation techniques [31], which we can make use of by giving up the distributivity property in place of monotonicity (see §4.3). The approximation techniques are safe for interprocedural analysis, and also for CBA, as carried out using the algorithms from this paper.

4.3 Solving for the MOP Value

There are two algorithms for solving for MOP values, called *prestar* and *poststar* (by analogy with the algorithms for PDSs). They take as input an automaton that accepts the set of initial configurations. As output, they produce a *weighted automaton*:

DEFINITION 11. *Given a weighted pushdown system $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$, 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. The automaton is said to accept a configuration $c = \langle p, u\rangle$ with weight $w = \mathcal{A}(c)$ if w is the combine of weights of all accepting paths for u starting from state p in $\mathcal{A}$. 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 $\mathcal{A}_{pre^*}(c) = \text{MOP}(\{c\}, \mathcal{L}(\mathcal{A}))$, whereas *poststar*$(\mathcal{A})$ produces a backward weighted automaton $\mathcal{A}_{post^*}$ as output, such that $\mathcal{A}_{post^*}(c) = \text{MOP}(\mathcal{L}(\mathcal{A}), \{c\})$ [40]. An example is shown in Fig. 4(c). One thing to note here is how the *poststar* automaton works. The procedure `bar` is analyzed independently of its calling context (not knowing the exact value of `x`), resulting in transitions between p and q. Its calling context, having the input values, is represented using the transitions coming out of state q. This is how, for instance, the automaton can tell that `x` $=$ `y` $= 3$ at node n_8 when `bar` is called from the (n_3, n_4) edge.

Using standard automata-theoretic techniques, one can also compute $\mathcal{A}_w(C)$ for (forward or backward) weighted automaton $\mathcal{A}_w$ and a regular set of configurations C, where $\mathcal{A}_w(C) = \bigoplus\{\mathcal{A}_w(c) \mid c \in C\}$. This allows one to solve for the meet-over-all-paths value MOP(S, T) for configuration sets S and T, as *poststar*$(S)(T)$ = *prestar*$(T)(S)$.

We now provide some intuition into why one needs both forwards and backwards automata. Consider the automata in Fig. 4(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 (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 (since it is a backward analysis), which occurs after the transformation inside `bar`. Thus, it is a forwards automaton.

The following lemma states the complexity for solving *poststar* by the algorithm of Reps et al. [40]. We use the notation $O_s(.)$ to denote the time bound in terms of semiring operations.

LEMMA 2. *[40] 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.*

The *height* of a weight domain is defined to be the length of the longest descending chain in the domain. In this paper, we assume the height to be bounded for ease of discussing complexity results, but WPDSs, and the algorithms in this paper, can also be used in certain cases when the height is unbounded (as long as there are no infinite descending chains, as is the case for $\mathcal{M}$ in Defn. 8).

Approximate analysis

Among the properties imposed by a weight domain, one important property is distributivity (Defn. 4, item 2). This is a common requirement for a precise analysis, also considered in various *coincidence theorems* for dataflow analysis [20, 22, 43]. 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 MOP computation may not be precise, but it will be *safe* under the partial order $\sqsubseteq$. The same applies for the results in this paper. When distributivity holds, our CBA is precise, otherwise, if only monotonicity holds, it will be a safe approximation.

4.4 CBA Problem Definition

The transition relation of a WPDS is a weighted relation (Defn. 9) 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 MOP$(\{c_1\}, \{c_2\})$.

The CBA problem is defined as in §2.3, except that all relations are weighted. This means that each thread is modeled as a WPDS. The threads share PDS states of the WPDSs, as well as the weights (the former can be eliminated, because PDS states can be pushed inside the weights).

This problem definition allows one to precisely model concurrent Boolean programs (with variations such as finding the shortest trace), as well as concurrent affine programs, where both are defined as having multiple threads and shared global variables.

Given the weighted relation $(\Rightarrow^c_1)^{k+1}$ as R, 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 §5 and §6 allow us to solve for this value when S and T are regular sets.

For example, consider two copies of the program in Fig. 4(a) running in parallel. Let the control locations 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 $R(S, T)$ would imply that the valuations $(3, 3), (3, 7), (7, 3)$, and $(7, 7)$ are possible.

5. 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 section (Defn. 12).

$$\begin{array}{lll}\langle p, \gamma_1\,\gamma_2\,\gamma_3\cdots\gamma_n\rangle & \Rightarrow^* & \langle p_1, \gamma_2\,\gamma_3\cdots\gamma_{k+1}\,\gamma_{k+2}\cdots\gamma_n\rangle \\ & \Rightarrow^* & \langle p_2, \gamma_3\cdots\gamma_{k+1}\,\gamma_{k+2}\cdots\gamma_n\rangle \\ & \Rightarrow^* & \cdots \\ & \Rightarrow^* & \langle p_k, \gamma_{k+1}\,\gamma_{k+2}\cdots\gamma_n\rangle \\ & \Rightarrow^* & \langle p_{k+1}, u_1\,u_2\cdots u_j\,\gamma_{k+2}\cdots\gamma_n\rangle\end{array}$$

Figure 6. A path in the PDS's transition relation; $u_i \in \Gamma, j \geq 1$.

Our solution is based on the following observation about paths in a PDS's transition relation. Suppose that $\langle p, \gamma_1\gamma_2\cdots\gamma_n\rangle$ is a configuration of a PDS $\mathcal{P}$. Then any path in the transition relation $\Rightarrow^*$ described by $\mathcal{P}$, starting from this configuration, can be written as shown in Fig. 6. The figure shows that the path starts initially by *popping off* some stack symbols (k symbols in the figure, $k < n$) in possibly multiple steps, after which it does not touch the rest of the stack ($\gamma_{k+1}\cdots\gamma_n$), except for the top symbol (γ_{k+1}). It is also possible for the path to pop off all stack symbols ($k = n$) and stop because no PDS rule can fire on an empty stack. To make this observation more formal, we decompose a path into phases as follows:

1. **Pop-phase.** The path pops off the top stack symbol without looking at the rest of the stack, i.e., it follows a sequence of rules that takes $\langle p, \gamma u\rangle$ to $\langle p', u\rangle$, for any $u \in \Gamma^*$.

2. **Growth-phase.** The path only looks at the top of the stack, and possibly rewrites it, but does not pop it off, i.e., it follows a sequence of rules (possibly empty) that takes a configuration $\langle p, \gamma u\rangle$ to $\langle p', u'u\rangle$ with $u' \in \Gamma^+$, for any $u \in \Gamma^*$.

Each path in the PDS's transition relation has zero or more pop-phases followed by a single (optional) growth-phase. We construct the transducer for a WPDS by essentially *pre-computing* each of these phases. First, we define two procedures:

1. *pop* : $P \times \Gamma \times P \rightarrow D$ is defined as follows:
$$\textit{pop}(p, \gamma, p') = \bigoplus\{v(\sigma) \mid \langle p, \gamma\rangle \Rightarrow^{\sigma} \langle p', \varepsilon\rangle\}$$
2. *grow* : $P \times \Gamma \rightarrow ((P \times \Gamma^+) \rightarrow D)$ is defined as follows:
$$\textit{grow}(p, \gamma)(p', u) = \bigoplus\{v(\sigma) \mid \langle p, \gamma\rangle \Rightarrow^{\sigma} \langle p', u\rangle\}$$

Note that $\textit{grow}(p, \gamma) = \textit{post}^*(\langle p, \gamma\rangle)$. The following Lemmas give efficient algorithms for computing the above procedures. Proofs are given in App. A.

LEMMA 3. *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 $\mathcal{A}_{\textit{pop}}$ be the forward weighted-automaton obtained by running prestar on $\mathcal{A}$. Then $\textit{pop}(p, \gamma, p')$ is the weight on the transition (p, γ, p') in $\mathcal{A}_{\textit{pop}}$. We can generate $\mathcal{A}_{\textit{pop}}$ in time $O_s(|P|^2|\Delta|H)$, and it has at most $|P|$ states.*

LEMMA 4. *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 $\mathcal{A}_{\textit{grow}}$ be the backward weighted-automaton obtained from running poststar on $\mathcal{A}$ (it does not need to know the final states). If we restrict the final states in $\mathcal{A}_{\textit{grow}}$ 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 $\mathcal{A}_{p,\gamma} = \textit{poststar}(\langle p, \gamma\rangle) = \textit{grow}(p, \gamma)$. We can compute $\mathcal{A}_{\textit{grow}}$ in time $O_s(|P||\Delta|(|P||\Gamma| + |\Delta|)H)$, and it has at most $|P||\Gamma| + |\Delta|$ states.*

The advantage of the construction presented in Lemma 4 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 (Lemma 2), $\mathcal{A}_{\textit{grow}}$ has many fewer states than those in all the $\mathcal{A}_{p,\gamma}$ put together.

Fig. 7(b) and (c) show the $\mathcal{A}_{\textit{grow}}$ and $\mathcal{A}_{\textit{pop}}$ automata for a simple WPDS constructed over the minpath semiring (Defn. 8).

The idea behind our approach is to use $\mathcal{A}_{\textit{pop}}$ to simulate the first phase where the PDS pops off stack symbols. With reference to Fig. 6, 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 in $\mathcal{A}_{\textit{pop}}$ with γ_{k+1} being the next symbol in the input, it passes control to $\mathcal{A}_{p_k,\gamma_{k+1}}$ 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 $\mathcal{A}_{\textit{grow}}$, but with final states restricted to $\{q_{p,\gamma} \mid \gamma \in \Gamma\}$, and unreachable states appropriately pruned (see Fig. 7(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 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 $\mathcal{A}_{\textit{grow}}$.

Note that $\mathcal{A}_{\textit{pop}}$ is a forward-weighted automaton, whereas $\mathcal{A}_{\textit{grow}}$ 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. This seems necessary, because going back to Fig. 6, 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 12. *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 impose a restriction on the transitions crossing 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 $q \in I$, 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.*

Computing the weight of a path requires more care. 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 paper, 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, as in the case of the Boolean semiring used for encoding PDSs as WPDSs, the partitioning is unnecessary.

Let $St(\mathcal{A})$ denote the set of states of an automaton $\mathcal{A}$. Because each of $\mathcal{A}_{\textit{pop}}$ and $\mathcal{A}_p$ have P as a subset of their set of states, we distinguish them by referring to a state $q \in St(\mathcal{A}_{\textit{pop}})$ by $q_{\textit{pop}}$ 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 $\{q_i, q_f\} \cup$

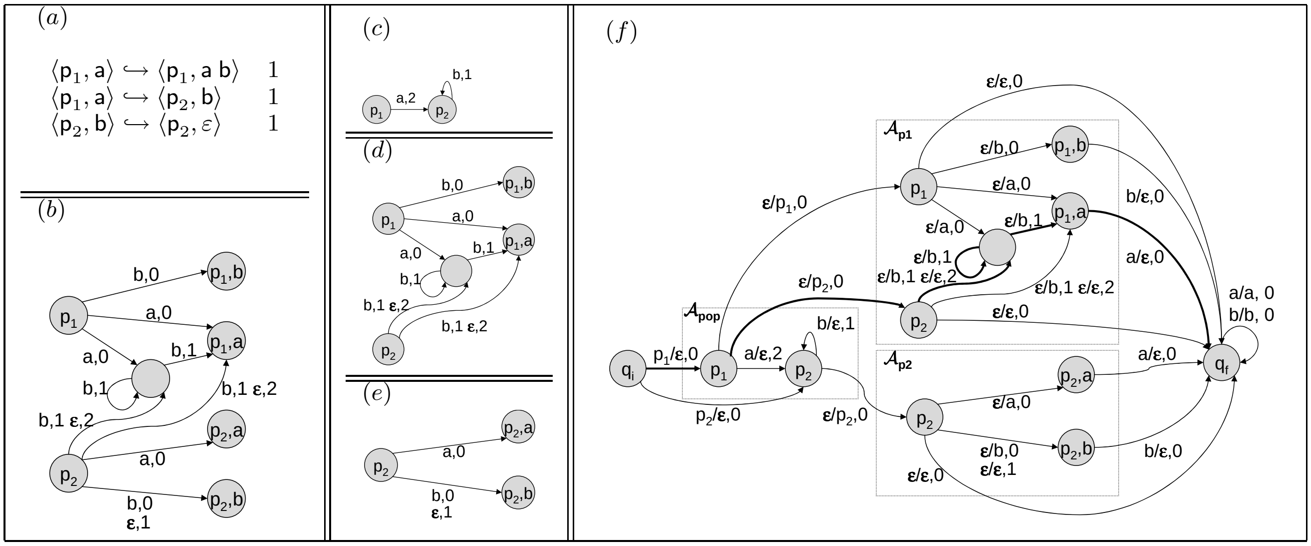


Figure 7. Weighted transducer construction: (a) A simple WPDS with the minpath semiring. (b) The $\mathcal{A}_{grow}$ automaton. Edges are labeled with their stack symbol and weight. (c) The $\mathcal{A}_{pop}$ automaton. (d) The $\mathcal{A}_{p_1}$ automaton obtained from $\mathcal{A}_{grow}$. (e) The $\mathcal{A}_{p_2}$ automaton obtained from $\mathcal{A}_{grow}$. The unnamed state in (c) and (d) is an extra state added by the *poststar* algorithm used in Lemma 4. (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.

$St(\mathcal{A}_{pop}) \cup (\bigcup_{p\in P} St(\mathcal{A}_p))$, 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. 7):

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 2. *When the transducer $\tau_{\mathcal{W}}$, as constructed above, is given input $(p\ u)$, $p \in P, u \in \Gamma^*$, then the combine over the values of all paths in $\tau_{\mathcal{W}}$ that output the string $(p'\ u')$ is precisely MOP$(\{\langle p, u\rangle\}, \{\langle p', u'\rangle\})$. 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.*

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. A proof of Thm. 2 is given in App. A.

6. Composing Weighted Transducers

Composition of unweighted transducers is straightforward, but this is not the case with weighted transducers. The requirement here is to take two weighted transducers and create another one whose (weighted) language is a relational composition of the (weighted) language of the two transducers (see Lemma 1 and Defn. 9). In particular, the composition is to be performed by extend ($\otimes$) which is the requirement that presents difficulties.

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.

6.1 The Sequential Product of Two Weighted Automata

Given forward-weighted automata $\mathcal{A}_1$ and $\mathcal{A}_2$, we wish to construct another weighted automaton $\mathcal{A}_3$ such that for any configuration c, $\mathcal{A}_3(c) = \mathcal{A}_1(c) \otimes \mathcal{A}_2(c)$. Assume that configurations consist of just the stack (and $|P| = 1$, which fixes the starting states of the automata). This is a special case of transducer composition: when a transducer only has transitions of the form (γ/γ), it is essentially an automaton, and composition of such transducers reduces to the above problem. For the Boolean weight domain, this reduces to unweighted automaton intersection (with words accepted with weight $\overline{0}$ being considered as words not accepted by the automaton).

Because the extend operation intuitively corresponds to concatenation of paths, a first attempt at solving this problem is to con-

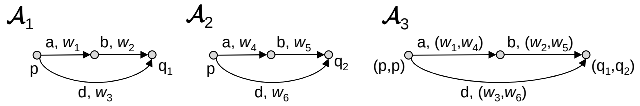


Figure 8. Forward-weighted automata. Their final states are q_1, q_2 and (q_1, q_2), respectively.

catenate the two automata by connecting the final states of $\mathcal{A}_1$ to the initial state of $\mathcal{A}_2$ via an epsilon transition with weight $\overline{1}$. However, this concatenated structure is not satisfactory if interpreted as an automaton: reading off the desired weight for configuration c requires quantification over all paths that accept the word (cc). For a (regular) set of configurations C, one would need quantification over the language $\{(cc) \mid c \in C\}$. This language is not even context-free. We overcome this by transferring some of the complexity into the weights, and retaining the regular structure of the automaton.

To take the sequential product of weighted automata, we start with the algorithm for intersecting unweighted automata (see §3). This is done by matching corresponding transitions in the two automata to produce a transition in the new automaton. We would like to do the same with weighted transitions. For this, given weights of the matching transitions, we want to compute a weight to put on the transition in the new automaton. Consider the automata shown in Fig. 8. Intersecting $\mathcal{A}_1$ and $\mathcal{A}_2$ without weights produces $\mathcal{A}_3$ (ignore the weights for now). The weight with which we want $\mathcal{A}_3$ to accept $(\texttt{a b})$ is $\mathcal{A}_1(\texttt{a b}) \otimes \mathcal{A}_2(\texttt{a b}) = w_1 \otimes w_2 \otimes w_4 \otimes w_5$.

One way of achieving this is to pair the weights while intersecting (as shown for $\mathcal{A}_3$ in Fig. 8). 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 $\mathcal{A}_3(\texttt{a b}) = (w_1, w_4) \otimes_p (w_2, w_5) = (w_1 \otimes w_2, w_4 \otimes w_5)$. Then by taking an extend of the two components, we get the desired answer. Essentially, the paired weights keep track of the weight from the first automaton in the first component, and the weight from the second automaton in the second component. Taking extend of the components in the paired weight of a path, produces the desired result for the (ordinary) weight of the path.

Because the number of accepting paths in an automaton may be infinite, one also needs a combine ($\oplus_p$) on paired weights. Defining it componentwise is not precise. For example, if $C = \{c_1, c_2\}$ is a set that contains two configurations, then we want the value of $\mathcal{A}_3(C)$ to 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 $\mathcal{A}_3(C) = \mathcal{A}_3(c_1) \oplus_p \mathcal{A}_3(c_2) = (\mathcal{A}_1(c_1) \oplus \mathcal{A}_1(c_2), \mathcal{A}_2(c_1) \oplus \mathcal{A}_2(c_2))$ and the extend of the components gives four terms $(\mathcal{A}_1(c_1) \otimes \mathcal{A}_2(c_1)) \oplus (\mathcal{A}_1(c_2) \otimes \mathcal{A}_2(c_2)) \oplus (\mathcal{A}_1(c_1) \otimes \mathcal{A}_2(c_2)) \oplus (\mathcal{A}_1(c_2) \otimes \mathcal{A}_2(c_1))$, which includes cross terms like $\mathcal{A}_1(c_1) \otimes \mathcal{A}_2(c_2)$. (Componentwise combine does give a safe approximation.)

We now show that, under certain circumstances, it is possible to use a different weight domain instead of weight-pairs, to precisely compute the desired value for the sequential product of weighted automata. The following defines the required weight domain.

DEFINITION 13. *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 weight-tuple by taking their extend. The third condition is the key. It distinguishes the tensor product from a simple tupling operation. It enforces that the *DeTensor* operation distribute over the combine of the tensored domain. A componentwise combine on tuples does not satisfy this condition.

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, add the transition $((p_1, p_2), \gamma, (q_1, q_2))$ with weight $(w_1 \odot w_2)$. Then the value of $\mathcal{A}_3(c)$ for a configuration c, or the value $\mathcal{A}_3(C)$ for a set of configurations can be computed as before, but then followed by the *DeTensor* operation.

The proof follows from the definitions. Let *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). 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^i_u is an accepting path for u in $\mathcal{A}_i$ ($i = 1, 2$), then we can uniquely determine an accepting path $\langle \sigma^1_u, \sigma^2_u \rangle$ for u in $\mathcal{A}_3$, and vice versa. These properties can be used to prove that if *accPath*$(\mathcal{A}_3, \langle \sigma^1_u, \sigma^2_u \rangle, w)$ holds, then $w = w_1 \odot w_2$ such that *accPath*$(\mathcal{A}_i, \sigma^i_u, w_i)$ hold for $i = 1, 2$. This gives us:

$$\begin{aligned}
&\textit{DeTensor}(\mathcal{A}_3(C)) \\
&= \textit{DeTensor}(\oplus_t \{w \mid \textit{accPath}(\mathcal{A}_3, \sigma_c, w), c \in C\}) \\
&= \oplus\{\textit{DeTensor}(w) \mid \textit{accPath}(\mathcal{A}_3, \sigma_c, w), c \in C\} \\
&= \oplus\{\textit{DeTensor}(w_1 \odot w_2) \mid \textit{accPath}(\mathcal{A}_i, \sigma^i_c, w_i), c \in C, \\
&\qquad\qquad\qquad i = 1, 2, \sigma_c = \langle \sigma^1_c, \sigma^2_c \rangle\} \\
&= \oplus\{w_1 \otimes w_2 \mid \textit{accPath}(\mathcal{A}_i, \sigma^i_c, w_i), c \in C, i = 1, 2\} \\
&= \oplus\{\mathcal{A}_1(c) \otimes \mathcal{A}_2(c) \mid c \in C\}
\end{aligned}$$

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 the weight domains presented in this paper (§4.1 and §4.2).

6.2 Sequentializable Tensor Product

Let $\mathcal{S} = (D, \oplus, \otimes, \overline{0}, \overline{1})$ whose STP we wish to construct. The first thing to note is that if the extend operation is commutative (in this case, we say $\mathcal{S}$ is commutative), $\mathcal{S}$ is its own STP for all n. The tensor operation can be defined as the extend of all its arguments, and *DeTensor* operation as 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. In this case, when weighted transitions are matched during the intersection operation, the weight on the new transition can be the extend of the weights on the matching transitions.

Recall that PDSs are the special case of WPDSs with the Boolean weight domain. The above result shows that our algorithm for WPDSs, when dealing with the Boolean weight domain, reduces exactly to the one we gave for PDSs (§3).

For commutative domains, it is easy to construct their STP, but they are not very useful for encoding abstractions for CBA. Under a commutative extend, interference from other threads can have no effect on the execution of a thread. This is unreasonable to assume for models of programs (i.e., for CBA). 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 14. *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 if $\mathcal{S}_c$ 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. 7) 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. 10) 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* [44] 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 $C(i, j) = A(i \operatorname{div} n_3, j \operatorname{div} n_4) \otimes B(i \mod n_3, j \mod n_4)$, where matrix indices start from zero. It is much easier to understand this definition pictorially (writing $A(i, j)$ as a_{ij}):

$$C = \begin{pmatrix} a_{11}B & \cdots & a_{1n_2}B \\ \vdots & \ddots & \vdots \\ a_{n_11}B & \cdots & a_{n_1n_2}B \end{pmatrix}$$

The Kronecker product is an associative operation, and also written as the tensor product $\odot$. 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 has all pairwise products of elements from the original matrices. One can come up with *projection* matrices p_i (with just $\overline{1}$ and $\overline{0}$ entries) such that $p_i \otimes m \otimes p_j$ selects the (i, j) entry of m (zeros out other entries). Using these matrices in conjunction with *permutation* matrices, one can compute the product of two matrices from their Kronecker product: there are fixed matrices e_i, e_j and an expression $\theta_m = \bigoplus_{i,j}(e_i \otimes m \otimes e_j)$, such that $\theta_{m_1 \odot m_2} = m_1 \otimes m_2$. This can be generalized to multiple matrices to get 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 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 (this is where we need commutativity of the underlying semiring) and the expression θ_m. 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 $(\Rightarrow_1^c)^k$ can be used to construct one for $(\Rightarrow_1^c)^{k+1}$.

The weight domain for ARA (§4.2.1) is not quite a matrix weight domain, but it is similar. The weights are 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 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,

$$\begin{array}{lcl} \beta(w_1 \odot w_2) & = & \beta(\beta(w_1) \odot \beta(w_2)) \\ \beta(\mathit{DeTensor}(w_1)) & = & \beta(\mathit{DeTensor}(\beta(w_1))) \end{array}$$

These properties follow quite trivially from the linearity of Kronecker product and the *DeTensor* operator (both distribute over addition).

6.3 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 two more operations on weights.

DEFINITION 15. *Let $\mathcal{S} = (D, \oplus, \otimes, \overline{0}, \overline{1})$ be a weight domain. Then a* ***transpose*** *operation on this domain is defined as $(.)^T : D \rightarrow D$ 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: $(w_1^T)^T = w_1$. A n* ***transposable STP*** *(TSTP) on $\mathcal{S}$ is defined as an n-STP along with another de-tensor operation: TDeTensor $: D^n \rightarrow D$ 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, just as for the sequential product of automata.

THEOREM 4. *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 $\mathcal{L}(\tau_1), \cdots, \mathcal{L}(\tau_n)$.*

7. Implementability of CBA

This paper develops novel machinery that shows how precise CBA can be carried out for various abstractions, including infinite-state abstractions. Our 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 (a proof is in App. A).

THEOREM 5. *The language $\{\langle M, 0^k, c_1, c_2\rangle \mid M$ is a set of PDSs with shared state, c_1 and c_2 are configurations of M, and $c_1(\Rightarrow_1^c)^{k+1}c_2\}$ is NP-complete.*

However, the analysis of sequential Boolean programs is also NP-complete (in the size of the Boolean program; the above result is in terms of the size of the PDS) but tools [4, 17, 42] 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 sharing inside 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 them (and doubling the number of BDD variables), which is a linear-time operation. Composing n transducers would produce BDDs with n 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 [26] or exploiting the symmetric structure of compositions (the same transducer is composed each time). We plan to explore these issues in future work. Moreover, different abstractions can be used for increasing the precision of CBA.

8. Related Work

Some of the related work has already been covered in §1 and §2. In this section we discuss some of the more technically related work.

A CBA of bounded-heap-manipulating Boolean programs is given in [7]. It encodes such Boolean programs using PDSs, and then uses the QR algorithm. Reachability analysis of concurrent recursive programs has also been considered in [6, 11, 34]. These works tackle the problem by computing overapproximations of the execution paths of the program, whereas here we compute underapproximations (bounded context) of the reachable configurations. Analysis under restricted communication policies (in contrast to shared memory) has also been considered [8, 19].

Constructing transducers. As mentioned in the introduction, a transducer construction for solving reachability in PDSs was given earlier by Caucal [10]. 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 §5, 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). Construction of a transducer for WPDSs has not been considered before. This was crucial for developing an algorithm for a general CBA.

The *pop*-function used in §5 represents summary information about paths, and is similar to the use of composed transformer functions from [12], summary functions from [43], summary edges from [39], and summary micro-functions from [41]. 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 afore mentioned 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 nondeterministic 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 [26, 27]. 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.

The sequential product of weighted automata on semirings was also considered in [23]. However, it was presented for the special case of taking one product of a forwards automaton with a backwards one. It cannot take the product of three or more automata. The techniques in this paper are for taking the product any number of times (provided STPs exist).

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 and T. Reps. Analyzing memory accesses in x86 executables. In *CC*, 2004.

[3] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI*, 2001.

[4] T. Ball and S. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN*, 2000.

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

[6] A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. In *POPL*, 2003.

[7] A. Bouajjani, S. Fratani, and S. Qadeer. Context-bounded analysis of multithreaded programs with dynamic linked structures. In *CAV*, 2007.

[8] A. Bouajjani, M. Müller-Olm, and T. Touili. Regular symbolic analysis of dynamic networks of pushdown systems. In *CONCUR*, 2005.

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

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

[11] 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.

[12] P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In *Formal Descriptions of Programming Concepts*. 1978.

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

[14] A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. *Electronic Notes in Theoretical Computer Science*, 9, 1997.

[15] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[16] S. Gulwani and G. C. Necula. Precise interprocedural analysis using random interpretation. In *POPL*, 2005.

[17] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, 2002.

[18] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[19] V. Kahlon and A. Gupta. On the analysis of interacting pushdown systems. In *POPL*, 2007.

[20] J. Kam and J. Ullman. Monotone data flow analysis frameworks. *Acta Inf.*, 7(3):305–318, 1977.

[21] M. Karr. Affine relationships among variables of a program. *Acta Inf.*, 6:133–151, 1976.

[22] J. Knoop and B. Steffen. The interprocedural coincidence theorem. In *CC*, 1992.

[23] A. Lal, N. Kidd, T. Reps, and T. Touili. Abstract error projection. In *SAS*, 2007.

[24] A. Lal, J. Lim, M. Polishchuk, and B. Liblit. Path optimization in programs and its application to debugging. In *ESOP*, 2006.

[25] A. Lal, T. Reps, and G. Balakrishnan. Extended weighted pushdown systems. In *CAV*, 2005.

[26] M. Mohri, F. Pereira, and M. Riley. Weighted automata in text and speech processing. In *ECAI*, 1996.

[27] M. Mohri, F. Pereira, and M. Riley. The design principles of a weighted finite-state transducer library. In *TCS*, 2000.

[28] M. Müller-Olm and H. Seidl. On optimal slicing of parallel programs. In *STOC*, 2001.

[29] M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. In *POPL*, 2004.

[30] M. Müller-Olm and H. Seidl. Analysis of modular arithmetic. In *ESOP*, 2005.

[31] M. Müller-Olm and H. Seidl. A generic framework for interprocedural analysis of numerical properties. In *SAS*, 2005.

[32] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI*, 2007.

[33] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[34] G. Patin, M. Sighireanu, and T. Touili. Spade: Verification of multithreaded dynamic and recursive programs. In *CAV*, 2007.

[35] E. Post. A variant of a recursively unsolvable problem. *Bulletin of the American Mathematical Society*, 52, 1946.

[36] S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *TACAS*, 2005.

[37] S. Qadeer and D. Wu. KISS: Keep it simple and sequential. In *PLDI*, 2004.

[38] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. In *TOPLAS*, 2000.

[39] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, 1995.

[40] T. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. In *SCP*, volume 58, 2005.

[41] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *TCS*, 167, 1996.

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

[43] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[44] Wikipedia. Kronecker product. `http://en.wikipedia.org/wiki/Kronecker_product`.

A. Proofs

Lemma 3. A formal proof for this lemma would follow from a characterization of the rule sequences that each automaton transition represents, based on the *abstract grammar* formulation of *prestar* [40]. We give a slightly informal, but intuitive, proof here. We use the fact that the saturation-based implementation of *prestar* is correct [40].

The lemma runs *prestar* on the empty automaton (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, $\mathcal{A}^p_\beta(\langle p', \gamma\ \beta \rangle) = \mathit{pop}(p', \gamma, p)$ 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 results follows by repeating the argument for all $p \in P$.

Lemma 4. The proof is similar to the one given for Lemma 3. Let $\beta \notin \Gamma$ be a new stack symbol. Let $\mathcal{A}^{p,\gamma}_\beta$ 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. $\mathcal{A}^{p,\gamma}_\beta$ represents the configuration set $\{\langle p, \gamma\ \beta \rangle\}$. The automaton returned by *poststar*$(\mathcal{A}^{p,\gamma}_\beta)$ would then represent the configuration set *grow*(p, γ) with β appended at the end of the stack. The proof follows from the fact that running *poststar* on $\mathcal{A}^{p,\gamma}_\beta$ is the same as running it on $\mathcal{A}$ (for all p and γ) with the exception of the extra β-transition.

$$\begin{array}{llll}
\langle p, \gamma_1\gamma_2\cdots\gamma_n\rangle & \Rightarrow^* & \langle p_1, \gamma_2\cdots\gamma_n\rangle & w_1 \\
 & \Rightarrow^* & \langle p_2, \gamma_3\cdots\gamma_n\rangle & w_2 \\
 & \Rightarrow^* & \cdots & \\
 & \Rightarrow^* & \langle p_k, \gamma_{k+1}\cdots\gamma_n\rangle & w_k \\
 & \Rightarrow^* & \langle p_{k+1}, u\gamma_{k+2}\cdots\gamma_n\rangle & w_{k+1}
\end{array}$$

Figure 9. A path in the PDS's transition relation with corresponding weights of each step.

Theorem 2. The proof is based on the observation made in Fig. 6. Suppose 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. 9.

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) 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 $w^i \sqsubseteq w_i$ (where $a \sqsubseteq b$ iff $a \oplus b = a$). This follows from Lemma 3. 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 $q_{p_k,\gamma_{k+1}}$ (Lemma 4). Moreover, the combine of weights of all such paths in the transducer is $\sqsubseteq w_{k+1}$. 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 $(p\ \gamma_1\gamma_2\cdots\gamma_n)$ and output $(p_{k+1}\ u\gamma_{k+2}\cdots\gamma_n)$ as required, and the combine of weights of all such paths is $\sqsubseteq$ the weight of the path shown in Fig. 9 $(w_1 \otimes w_2 \otimes \cdots \otimes w_{k+1})$. Note that there is a corresponding path in the transducer (that uses transitions inserted in Step 6) when the path shown in Fig. 9 has no growth phase.

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. 9. Using Lemmas 3 and 4, the weight of such a path in the transducer would be $\sqsupseteq$ the combine of weights of all paths between the configurations in the PDS's transition relation.

Theorem 5. [SKETCH] The proof follows from two earlier pieces of work. Ramalingam [38] showed that reachability in multi-threaded programs with synchronization primitives is undecidable by giving a reduction from the Post's correspondence problem (PCP) [35]. We also know that bounded-PCP is NP-complete [15, Problem SR11]. It is easy to see that shared memory with a bounded number of context switches can simulate a similar number of synchronization steps. Thus, Ramalingam's reduction can be used to give a reduction from bounded-PCP to CBMC.

B. Local Variables

Local variables pose a complication for WPDSs. An extension of WPDSs to handle abstractions with local variables (for sequential programs) is given in [25]. We will only summarize the essential details of that paper here. Also, because it is hard to characterize abstractions with local variables using matrix domains, we only focus on CBA of Boolean programs (with variations such as finding the shortest trace) and affine programs.

B.1 Boolean Programs

Sequential analysis

For encoding Boolean programs with local variables, assume, without loss of generality, that each procedure has the same number of local variables. Let G be the set of valuations of the global variables and L be the set of valuations of local variables. Weights, abstracting program statements, are now binary relations on $G \times L$. The weight domain is a relational weight domain on the set $G \times L$ but with an extra *merge* function defined on weights. Because different weights can talk about local variables from different procedures, one cannot take relational composition of weights from different procedures. The *merge* 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:

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

Once the summary of a procedure is calculated as a weight w involving local variables of the procedure, the *merge* function is applied to it, and the result $\textit{merge}(w)$ 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. The property required of *merge* is that it should distribute over combine, i.e., $\textit{merge}(w_1 \oplus w_2) = \textit{merge}(w_1) \oplus \textit{merge}(w_2)$. More details can be found in [25].

For encoding Boolean programs with other abstractions, such as finding the shortest trace, one can use the relational weight domain on $(G \times L, \mathcal{S})$, where $\mathcal{S}$ is a weight domain such as the minpath semiring (transparent to the presence or absence of local variables). The *merge* function on weights from this domain can be defined as follows:

$$\begin{array}{ll}\textit{merge}(w) = \lambda(g_1, l_1, g_2, l_2). & \text{if } (l_1 \neq l_2) \text{ then } \overline{0}_{\mathcal{S}} \\ & \text{else } \bigoplus_{l \in L} w(g_1, l_1, g_2, l)\end{array}$$

Context-Bounded Analysis

For CBA, the two main steps are transducer construction and their composition. The transducer construction does not change, except for the fact that the *prestar* query (Lemma 3) and the *poststar* query (Lemma 4) are carried out using the algorithms from [25]. Transducer composition requires more care.

First, reconsider the composition algorithm from §6. To compose transducers τ_1 and τ_2, one carries out the following: the transition $(q_1, \gamma_1/\gamma_2, q_2)$ with weight w_1 in τ_1 is matched with transition $(q_1', \gamma_2/\gamma_3, q_2')$ with weight w_2 in τ_2 to produce the transition $((q_1, q_1'), \gamma_1/\gamma_3, (q_2, q_2'))$ with weight $w_1 \odot w_2$. The tensor operation on binary relations on a set G results in a binary relation on the set $G \times G$ as follows:

$$w_1 \odot w_2 = \{((g_1, g_3), (g_2, g_4)) \mid (g_1, g_2) \in w_1, (g_3, g_4) \in w_2\}$$

If the values of local variables were encoded in the stack symbols (as for PDSs), then the matching of the stack symbol γ_2 in the two transitions essentially matches the local-variable valuation from τ_1 with that from τ_2 and quantifies it out (γ_2 does not appear on the transition produced for the composed transducer).

When local variables are encoded in the weights, the tensor operation is changed as follows:

$$\begin{array}{ll}
w_1 \odot w_2 = & \\
\{((g_1, l_1, g_3, l_4), (g_2, l_1, g_4, l_4)) \mid & (g_1, l_1, g_2, l_2) \in w_1, \\
 & (g_3, l_3, g_4, l_4) \in w_2, \\
 & l_2 = l_3\}
\end{array}$$

The third condition $l_2 = l_3$ performs matching on the local variables, which are then quantified out, and replaced with an identity transformation on the local variables. This can be seen as extending the *merge* function on simple weights to ones in the tensored domain.

For the relational weight domain on $(G \times L, \mathcal{S})$, the tensor operation is as follows:

$$\begin{array}{ll} w_1 \odot w_2 = & \lambda((g_1, l_1, g_3, l_3), (g_2, l_2, g_4, l_4)). \\ & \text{if } (l_1 \neq l_2 \text{ or } l_3 \neq l_4) \text{ then } \overline{0}_{\mathcal{S}} \\ & \text{else } \bigoplus_{l \in L} (w_1(g_1, l_1, g_2, l) \odot_{\mathcal{S}} w_2(g_3, l, g_4, l_4)) \end{array}$$

B.2 Affine Programs

Context-bounded analysis of affine programs with local variables follows much on the same line as for Boolean programs. The details for sequential analysis of such programs can be found in [29].

Sequential Analysis

If an affine program has n global variables and no local variables, then the matrices have size $(n+1) \times (n+1)$ (as explained in §4.2.1). Assume, without loss of generality, that each procedure has l local variables. Then the matrices have size $(n+l+1) \times (n+l+1)$. In such a case, a matrix can be divided into four quadrants, as shown below, along with their sizes.

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

A matrix encodes a transformation on variables, i.e., a map on variable valuations. For example, the matrix for $\mathtt{x}_1 := \mathtt{x}_2 + 1$ would encode the map that takes the valuation $\overline{\mathtt{x}}$ before the execution of the statement and maps it to $\overline{\mathtt{x}}[\mathtt{x}_1 \mapsto \overline{\mathtt{x}}(2) + 1]$. The four quadrants of a matrix describe four pieces of this transformation: the first quadrant encodes the contribution of old values of global variables to new values of global variables; the second quadrant encodes the contribution of old globals to new locals; the third quadrant encodes the contribution of old locals to new globals; and the fourth quadrant encodes the contribution of old locals to new locals.

The *merge* function should quantify out the local variable map, and replace it with an identity map. For the quantification, the first, second, and third quadrants are zeroed out, and the identity map is installed by changing the fourth quadrant to the identity matrix. If M is a set of matrices, *merge*(M) is defined as the application of the following operation on all matrices of M [29]:

$$\begin{array}{|c|c|} \hline m_1 & m_2 \\ \hline m_3 & m_4 \\ \hline \end{array} \quad \mapsto \quad \begin{array}{|c|c|} \hline m_1 & 0 \\ \hline 0 & gI_{l \times l} \\ \hline \end{array}$$

Here g is the topmost-leftmost element of m_1. It is used to make the above operation linear (which, in turn, makes *merge* distribute over combine of sets of matrices). The matrix $I_{l \times l}$ is the identity matrix of size $l \times l$.

Context-Bounded Analysis

As for Boolean programs, the *merge* function is incorporated into the tensor operation:

$$\begin{array}{|c|c|} \hline m_1 & m_2 \\ \hline m_3 & m_4 \\ \hline \end{array} \odot \begin{array}{|c|c|} \hline m_5 & m_6 \\ \hline m_7 & m_8 \\ \hline \end{array} = \begin{array}{|c|c|} \hline m_1 \odot m_5 & m_2 \odot m_7 \\ \hline 0 & g_1 g_5 I_{l^2 \times l^2} \\ \hline \end{array}$$

Here g_i is the topmost-leftmost element of matrix m_i. The main idea is that m_2 and m_7 get stitched together, which corresponds to putting together the values of local variables before a context switch to the values after the thread gets control back.