Faster Algorithms for Weighted Recursive State Machines

Krishnendu Chatterjee[1], Bernhard Kragl[1],
Samarth Mishra[2], and Andreas Pavlogiannis[1]

[1] IST Austria, Klosterneuburg, Austria
[2] IIT Bombay, Mumbai, India

Abstract. Pushdown systems (PDSs) and recursive state machines (RSMs), which are linearly equivalent, are standard models for interprocedural analysis. Yet RSMs are more convenient as they (a) explicitly model function calls and returns, and (b) specify many natural parameters for algorithmic analysis, e.g., the number of entries and exits. We consider a general framework where RSM transitions are labeled from a semiring and path properties are algebraic with semiring operations, which can model, e.g., interprocedural reachability and dataflow analysis problems.

Our main contributions are new algorithms for several fundamental problems. As compared to a direct translation of RSMs to PDSs and the best-known existing bounds of PDSs, our analysis algorithm improves the complexity for finite-height semirings (that subsumes reachability and standard dataflow properties). We further consider the problem of extracting distance values from the representation structures computed by our algorithm, and give efficient algorithms that distinguish the complexity of a one-time preprocessing from the complexity of each individual query. Another advantage of our algorithm is that our improvements carry over to the concurrent setting, where we improve the best-known complexity for the context-bounded analysis of concurrent RSMs. Finally, we provide a prototype implementation that gives a significant speed-up on several benchmarks from the SLAM/SDV project.

1 Introduction

Interprocedural analysis. One of the classical algorithmic analysis problems in programming languages is the interprocedural analysis. The problem is at the heart of several key applications, ranging from alias analysis, to data dependencies (modification and reference side effect), to constant propagation, to live and use analysis [32,35,10,15,23,18,13,14,17]. In seminal works [32,35] it was shown that a large class of interprocedural dataflow analysis problems can be solved in polynomial time.

Models for interprocedural analysis. Two standard models for interprocedural analysis are *pushdown systems* (or finite automata with stacks) and *recursive state machines (RSMs)* [4,5]. An RSM is a formal model for control flow graphs of programs with recursion. We consider RSMs that consist of modules, one for each method or function that has a number of entry nodes and a number of exit nodes, and each module contains boxes that represent calls to other modules. A special case of RSMs with a single entry and a single exit node for every module (*SESE RSMs*, aka *supergraph* in [32]) has also been considered. While pushdown systems and RSMs are linearly equivalent (i.e., there is a linear translation from one model to the other and vice versa), there are two distinct advantages of RSMs. First, the model of RSMs closely resembles the problems of programming languages with explicit function calls and returns, and hence even its special cases such as SESE RSMs has been considered to model many applications. Second, the model of RSMs provides many parameters, such as the number of entry and exit nodes, and the number of modules, and better algorithms can be developed by considering that some parameters are small. Typically the SESE RSMs can model data-independent interprocedural analysis, whereas general RSMs can model data dependency as well. For most applications, the number of entries and exits of a module, usually represents the input parameters of the module.

Semiring framework. We consider a general framework to express computation properties of RSMs where the transitions of an RSM are labeled from a semiring. The labels are referred to as weights. A *computation* of an RSM executes transitions between configurations consisting of a node (representing the current control state) and a stack of boxes (representing the current calling context). To express properties of interest we need to define how to assign weights to computations, i.e., to accumulate weights *along* a computation, and how to assign weights to sets of computations, i.e., to combine weights *across* a set of computations. The weight of a given computation is the semiring product of the weights on the individual transitions of the computation, and the weight of a given set of computations is the semiring plus of the weights of the individual computations in the set. For example, (i) with the Boolean semiring (with semiring product as AND, and semiring plus as OR) we express the reachability property; (ii) with a Dataflow semiring we can express problems from dataflow analysis. One class of such problems is given by the IFDS/IDE framework [32,35] that considers the propagation of dataflow facts along distributive dataflow functions (note that the IFDS/IDE framework only considers SESE RSMs). Hence the large and important class of dataflow analysis problems that can be expressed in the IFDS/IDE framework can also be expressed in our framework. Pushdown systems with semiring weights have also been extensively considered in the literature [34,21,33,19].

Problems considered. We consider the following basic *distance problems.*

– *Configuration distance.* Given a set of *source* configurations and a *target* configuration, the *configuration distance* is the weight of the set of computations that start at some source configuration and end in the target configuration. In the *configuration distance problem* the input is a set of source configurations and the output is the configuration distance to all reachable configurations.

– *Superconfiguration distance.* We also consider a related problem of *superconfiguration distance*. A *superconfiguration* represents a sequence of modules, rather than a sequence of invocations. Intuitively, it does not consider the sequence of function calls, but only which functions were invoked. This is a coarser notion than configurations and allows for fast overapproximation. The superconfiguration distance problem is then similar to the configuration distance problem, with configurations replaced by superconfigurations.
– *Node distance.* Given a set of source configurations and a target node, the *node distance* is the weight of the set of computations that start at some source configuration and end in a configuration with the target node (with arbitrary stack). In the *node distance problem* the input is a set of source configurations and the output is the node distance to all reachable nodes.

Symbolic representation. A core ingredient for solving distance problems is the symbolic representation of sets of RSM configurations and their manipulation. Given a symbolic representation of the set of initial configurations, we provide a two step approach to solve the distance problems. In step one we compute a symbolic representation of the set of all configurations reachable from the initial configurations. Furthermore, the transitions in the representation are annotated with appropriate semiring weights to capture the various distances described above. In step two we query the computed representation for the required distances. Thus we make the important distinction between the complexity of a *one-time* preprocessing and the complexity of every *individual query*.

Concurrent RSMs. While reachability is the most basic property, the study of pushdown systems and RSMs with the semiring framework is the fundamental quantitative extension of the basic problem. An orthogonal fundamental extension is to study the reachability property in a *concurrent* setting, rather than the sequential setting. However, the reachability problem in concurrent RSMs (equivalently concurrent pushdown systems) is undecidable [31]. A very relevant problem to study in the concurrent setting is to consider context-bounded reachability, where at most k context switches are allowed. The context-bounded reachability problem is both decidable [28] and practically relevant [25,26].

Previous results. Many previous results have been established for pushdown systems, and the translation of RSMs to pushdown systems implies that similar results carry over to RSMs as well. We describe the most relevant previous results with respect to our results. For an RSM $\mathcal{R}$, let $|\mathcal{R}|$ denote its size, θ_e and θ_x the maximum number of entries and exits, respectively, and f the number of modules. The existing results for weighted pushdown systems over semirings of height H [36,34] along with the linear translation of RSMs to pushdown systems [4] gives an $O(H \cdot |\mathcal{R}| \cdot \theta_e \cdot \theta_x \cdot f)$-time algorithm for the configuration and node distance problems for RSMs. The previous results for context-bounded reachability of concurrent pushdown systems [28] applied to concurrent RSMs gives the following complexity bound: $O(|\mathcal{R}^{\|}|^5 \cdot {\theta_x^{||}}^5 \cdot n^k \cdot |G|^k)$, where $|\mathcal{R}^{\|}|$ is the size of the concurrent RSM, $\theta_x^{||}$ is the number of exit nodes, n is the number of component RSMs, G is the global part of the concurrent RSM, and k is the bound on the number of context switches.

	Sequential	**Concurrent**
Existing	$H \cdot \lvert\mathcal{R}\rvert \cdot \theta_e \cdot \theta_x \cdot f$ [36,34]	$\lvert\mathcal{R}^{\parallel}\rvert^5 \cdot \theta_x^{\lvert\rvert\ 5} \cdot n^k \cdot \lvert G\rvert^k$ [28]
Our result	$H \cdot (\lvert\mathcal{R}\rvert \cdot \theta_e + \lvert Call\rvert \cdot \theta_e \cdot \theta_x)$ [Theorem 1]	$\lvert\mathcal{R}^{\parallel}\rvert \cdot \theta_e^{\lvert\rvert} \cdot \theta_x^{\lvert\rvert} \cdot n^k \cdot \lvert G\rvert^{k+2}$ [Theorem 7]

Table 1. Asymptotic time complexity of computing configuration automata.

Semiring					RSM	
General	Boolean	Constant size	Size $\lvert D\rvert^*$		Sparse†	
Query	Query	Query	Preprocess	Query	Preprocess	Query
$n \cdot \theta_e^2$	$\lvert\mathcal{R}\rvert \cdot \theta_e \cdot \frac{n}{\log n}$	$n \cdot \frac{\theta_e^2}{\log \theta_e}$	$\lvert\mathcal{R}\rvert \cdot \theta_e^{1+\varepsilon \cdot \log \lvert D\rvert}$	$n \cdot \frac{\theta_e^2}{\varepsilon^2 \cdot \log^2 \theta_e}$	$\lvert\mathcal{R}\rvert \cdot \theta_e^{\omega-1} \cdot x$	$n \cdot \left\lceil \frac{\theta_e^2}{\log x} \right\rceil$

Table 2. Asymptotic time complexity of answering a configuration/superconfiguration distance query of size n. Preprocess time refers to additional preprocessing after the configuration automaton is constructed.

* For any fixed $\varepsilon > 0$.

† In a sparse RSM every module only calls a constant number of other modules, and the result applies only to superconfiguration distances. The parameter x has to satisfy $x = O(\text{poly}(|\mathcal{R}|))$, and ω is the smallest constant required for multiplying two square matrices of size $m \times m$ in time $O(m^\omega)$ (currently $\omega \simeq 2.372$).

Our contributions. Our main contributions are as follows:

1. *Finite-height semirings.* We present an algorithm for computing configuration and node distance problems for RSMs over semirings with finite height H with running time $O(H \cdot (|\mathcal{R}| \cdot \theta_e + |Call| \cdot \theta_e \cdot \theta_x))$, where $|Call|$ is the number of call nodes. The algorithm we present constructs the symbolic representations from which the distances can be extracted. Thus our algorithm improves the current best-known algorithms by a factor of $\Omega((|\mathcal{R}| \cdot f)/(\theta_x + |Call|))$ (Table 1) (also see Remark 3 for details).
2. *Distance queries.* Once a symbolic representation is constructed, it can be used for extracting distances. We present algorithms which given a configuration query of size n, return the distance in $O(n \cdot \theta_e^2)$ time. Furthermore, we present several improvements for the case when the semiring has a small domain. Finally, we show that when the RSM has a sparse call graph, we can obtain a range of tradeoffs between preprocessing and querying times. Our results on distance queries are summarized in Table 2.
3. *Concurrent RSMs.* For the context-bounded reachability of concurrent RSMs we present an algorithm with time bound $O(|\mathcal{R}^{\parallel}| \cdot \theta_e^{||} \cdot \theta_x^{||} \cdot n^k \cdot |G|^{k+2})$. Thus our algorithm significantly improves the current best-known algorithm (Table 1).
4. *Experimental results.* We experiment with a basic prototype implementation for our algorithms. Our implementation is an explicit (rather than symbolic) one. We compare our implementation with jMoped [1], which is a leading and mature tool for weighted pushdown systems, on several real-world benchmarks coming from the SLAM/SDV project [6,7]. We consider the basic reachability property (representative for finite-height semirings) for the sequential setting. Our experimental results show that our algorithm provides significant improvements on the benchmarks compared to jMoped.

Technical contribution. The main technical contributions are as follows:

- We show how to combine (i) the notion of *configuration automata* as a *symbolic* representation structure for sets of configurations, and (ii) entry-to-exit *summaries* to avoid redundant computations, and obtain an efficient dynamic programming algorithm for various distance problems in RSMs over finite-height semirings.
- Configuration and superconfiguration distances are extracted using graph traversal of configuration automata. When the semiring has small domain, we obtain several speedups by exploiting advances in matrix-vector multiplication. Finally, the speedup of superconfiguration distance extraction on sparse RSMs is achieved by devising a Four-Russians type of algorithm, which spends some polynomial preprocessing time in order to allow compressing the query input in blocks of logarithmic length.

All proofs are provided in Appendix A and C.

2 Preliminaries

In this section we present the necessary definitions of recursive state machines (RSMs) where every transition is labeled with a value (or weight) from an appropriate domain (semiring). Then we formally state the problems we study on weighted RSMs.

Semirings. An *idempotent semiring* is a quintuple $\langle D, \oplus, \otimes, \overline{0}, \overline{1} \rangle$, where D is a set called the *domain*, $\overline{0}$ and $\overline{1}$ are elements of D, and $\oplus$ (the *combine* operation) and $\otimes$ (the *extend* operation) are binary operators on D such that

1. $\langle D, \oplus, \overline{0} \rangle$ is an idempotent commutative monoid with neutral element $\overline{0}$,
2. $\langle D, \otimes, \overline{1} \rangle$ is a monoid with neutral element $\overline{1}$,
3. $\otimes$ distributes over $\oplus$,
4. $\overline{0}$ is an annihilator for $\otimes$, i.e., $a \otimes \overline{0} = \overline{0} \otimes a = \overline{0}$ for all $a \in D$.

An idempotent semiring has a canonical partial order $\sqsubseteq$, defined by

$$a \sqsubseteq b \iff a \oplus b = a.$$

Furthermore, this partial order is *monotonic*, i.e., for all $a, b, c \in D$

$$\begin{aligned} a \sqsubseteq b &\implies a \oplus c \sqsubseteq b \oplus c, \\ a \sqsubseteq b &\implies a \otimes c \sqsubseteq b \otimes c, \\ a \sqsubseteq b &\implies c \otimes a \sqsubseteq c \otimes b. \end{aligned}$$

The *height* H of an idempotent semiring is the length of the longest descending chain in $\sqsubseteq$. In the rest of the paper we will only write semiring to mean an idempotent finite-height semiring.

Remark 1. Instead of finite height, the more general *descending chain condition* would be sufficient for our purposes. This only requires that there are no infinite descending chains in $\sqsubseteq$, but there is not necessarily a finite height H.

Recursive State Machines (informally). Intuitively, an RSM is a collection of finite automata, called modules, such that computations consist of ordinary local transitions within a module as well as calls to other modules, and returns from other modules. For this, every module has a well-defined interface of entry and exit nodes. Calls to other modules are represented by boxes, which have call and return nodes corresponding to the respective entry and exit nodes of the called module.

Unlike pushdown automata (PDAs), there is no explicit stack manipulation in RSMs. Instead a call stack is maintained implicitly along computations as follows. When a call node of a box is reached, the control is passed to the respective entry node of the called module and the box is pushed onto the top of the stack. When an exit node of a module is reached, a box is popped off from the top of the stack and the control is passed to the corresponding return node of the box. Hence, the stack is a sequence of boxes representing the current calling context and a configuration in a computation of an RSM is a node together with a sequence of boxes.

Recursive State Machines (formally). A *recursive state machine (RSM)* over a semiring $\langle D, \oplus, \otimes, \overline{0}, \overline{1} \rangle$ is a tuple $\mathcal{R} = \langle \mathcal{M}_1, \dots, \mathcal{M}_k \rangle$, where every *module* $\mathcal{M}_i = \langle B_i, Y_i, N_i, \delta_i, w_i \rangle$ is given by

- a finite set B_i of *boxes*,
- a mapping $Y_i : B_i \mapsto \{1, \dots, k\}$,
- a finite set $N_i = \mathit{In}_i \cup \mathit{En}_i \cup \mathit{Ex}_i \cup \mathit{Call}_i \cup \mathit{Ret}_i$ of *nodes*, partitioned into
  - *internal* nodes In_i,
  - *entry* nodes En_i,
  - *exit* nodes Ex_i,
  - *call* nodes $\mathit{Call}_i = \{\langle b, e \rangle \mid b \in B_i \text{ and } e \in \mathit{En}_{Y_i(b)}\}$,
  - *return* nodes $\mathit{Ret}_i = \{\langle b, x \rangle \mid b \in B_i \text{ and } x \in \mathit{Ex}_{Y_i(b)}\}$,
- a *transition relation* $\delta_i \subseteq (\mathit{In}_i \cup \mathit{En}_i \cup \mathit{Ret}_i) \times (\mathit{In}_i \cup \mathit{Ex}_i \cup \mathit{Call}_i)$,
- a *weight function* $w_i : \delta_i \mapsto D$, with $w_i(u, x) = \overline{1}$ for every exit node $x \in \mathit{Ex}_i$.

We write B for $\bigcup_{i=1}^{k} B_i$, and similarly for N, In, En, Ex, Call, Ret, δ, w. To measure the size of an RSM we let $|\mathcal{R}| = \max(|N|, \sum_i |\delta_i|)$. A major source of complexity in analysis problems for RSMs is the number of entry and exit nodes of the modules. Throughout the paper we express complexity with respect to the *entry bound* $\theta_e = \max_{1 \le i \le k} |\mathit{En}_i|$ and the *exit bound* $\theta_x = \max_{1 \le i \le k} |\mathit{Ex}_i|$, i.e., the maximum number of entries and exits, respectively, over all modules. Note that the restriction on the weight function to assign weight $\overline{1}$ to every transition to an exit node is wlog, as any weighted RSM that does not respect this can be turned into an equivalent one that does, with only a constant factor increase in its size.

Stacks. A *stack* is a sequence of boxes $S = b_1 \dots b_r$, where the first box denotes the top of the stack; and ε is the empty stack. The *height* of S is $|S| = r$, i.e, the number of boxes it contains. For a box b and a stack S, we denote with bS the concatenation of b and S, i.e., a push of b onto the top of S.

Configurations and transitions. A *configuration of an RSM* $\mathcal{R}$ is a tuple $\langle u, S \rangle$, where $u \in \mathit{In} \cup \mathit{En} \cup \mathit{Ret}$ is an internal, entry, or return node, and S is a stack. For $S = b_1 \dots b_r$, where $b_i \in B_{j_i}$ for $1 \le i \le r$ and some j_i, we require that $Y_{j_i}(b_i) = j_{i-1}$ for $1 < i \le r$, as well as $u \in N_{Y_{j_1}(b_1)}$. This corresponds to the case where the control is inside the module of node u, which was entered via box b_1 from module $\mathcal{M}_{j_1}$, which was entered via box b_2 from module $\mathcal{M}_{j_2}$, and so on.

We define a transition relation $\Rightarrow$ over configurations and a corresponding weight function $w : \Rightarrow \mapsto D$, such that $\langle u, S\rangle \Rightarrow \langle u', S'\rangle$ with $w(\langle u, S\rangle, \langle u', S'\rangle) = v$ if and only if there exists a transition $t \in \delta_i$ in $\mathcal{R}$ with $w_i(t) = v$ and one of the following holds:

1. *Internal transition:* $u' \in In_i$, $t = \langle u, u'\rangle$, and $S' = S$.
2. *Call transition:* $u' = e \in En_{Y_i(b)}$ for some box $b \in B_i$, $t = \langle u, \langle b, e\rangle\rangle$, and $S' = bS$.
3. *Return transition:* $u' = \langle b, x\rangle \in R_i$ for some box $b \in B_i$ and exit node $x \in Ex_{Y_i(b)}$, $t = \langle u, x\rangle$, and $S = bS'$.

Note that we follow the convention that a call immediately enters the called module and a return immediately returns to the calling module. Hence, the node of a configuration can be an internal node, an entry node, or a return node, but not a call node or an exit node.

Computations. A *computation* of an RSM $\mathcal{R}$ is a sequence of configurations $\pi = c_1, \dots, c_n$, such that $c_i \Rightarrow c_{i+1}$ for every $1 \le i < n$. We say that π is a computation from c_1 to c_n, of length $|\pi| = n - 1$, and of weight $\otimes(\pi) = \bigotimes_{i=1}^{n-1} w(c_i, c_{i+1})$ (the empty extend is $\overline{1}$). We write $\pi : c \Rightarrow^* c'$ to denote that π is a computation from c to c' of any length. A computation $\pi : c \Rightarrow^* c'$ is called *non-decreasing* if the stack height of every configuration of π is at least as large as that of c (in other words, the top stack symbol of c is never popped in π). The computation π is called *same-context* if it is non-decreasing, and c and c' have the same stack height. A computation that cannot be extended by any transition is called a *halting* computation. For a set of computations Π we define its weight as $\bigoplus(\Pi) = \bigoplus_{\pi \in \Pi} \otimes(\pi)$ (the empty combine is $\overline{0}$). For a configuration c and a set of configurations R we denote by $\Pi(R, c)$ the set of all computations from any configuration in R to c. Here, and for similar purposes below, we will use the convention to write $\Pi(c, c')$ instead of $\Pi(\{c\}, c')$.

Example 1. Figure 1 shows an RSM $\mathcal{R} = \langle \mathcal{M}_1, \mathcal{M}_2\rangle$ that consists of two modules $\mathcal{M}_1$ and $\mathcal{M}_2$. The modules are mutually recursive, since box b_1 of module $\mathcal{M}_1$ calls module $\mathcal{M}_2$, and box b_2 of module $\mathcal{M}_2$ calls module $\mathcal{M}_1$. A possible computation of $\mathcal{R}$ is

$$\begin{aligned}&\langle e_1^1, \varepsilon\rangle \xRightarrow{w_1} \langle e_2, b_1\rangle \xRightarrow{w_5} \langle e_1^1, b_2 b_1\rangle \xRightarrow{w_1} \langle e_2, b_1 b_2 b_1\rangle \xRightarrow{w_6} \langle e_1^2, b_2 b_1 b_2 b_1\rangle \xRightarrow{w_2}\\ &\langle u_1, b_2 b_1 b_2 b_1\rangle \xRightarrow{w_4} \langle\langle b_2, x_1\rangle, b_1 b_2 b_1\rangle \xRightarrow{w_7} \langle\langle b_1, x_2\rangle, b_2 b_1\rangle \xRightarrow{w_3} \langle u_1, b_2 b_1\rangle \xRightarrow{w_4}\\ &\langle\langle b_2, x_1\rangle, b_1\rangle \xRightarrow{w_7} \langle\langle b_1, x_2\rangle, \varepsilon\rangle \xRightarrow{w_3} \langle u_1, \varepsilon\rangle.\end{aligned} \quad (1)$$

Distance problems. Given a set of configurations R, the set of configurations that are *reachable* from R is

$$post^*(R) = \{c \mid \exists c_0 \in R : c_0 \Rightarrow^* c\}.$$

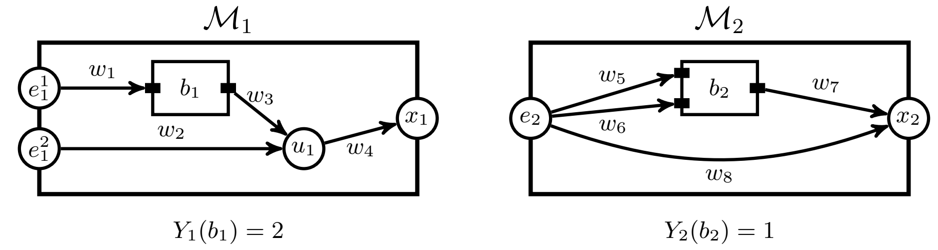

Fig. 1. Example of a weighted RSM that consists of two modules with mutual recursion.

Instead of mere reachability, we are interested in the following distance metrics that aggregate over computations from R using the semiring combine and hence are expressed as semiring values.

- *Configuration distance.* The *configuration distance* from R to c is defined as

$$d(R, c) = \bigoplus(\Pi(R, c)).$$

That is, we take the combine over the weights of all computations from a configuration in R to c. Naturally, for configurations c not reachable from R we have $d(R, c) = \overline{0}$.

- *Superconfiguration distance.* A *superstack* is a sequence of modules $\overline{S} = \mathcal{M}_1 \dots \mathcal{M}_r$. A stack $S = b_1 \dots b_r$ *refines* $\overline{S}$ if $b_i \in B_i$ for all $1 \leq i \leq r$, i.e., the i-th box of S belongs to the i-th module of $\overline{S}$. A *superconfiguration* of $\mathcal{R}$ is a tuple $\langle u, \overline{S}\rangle$. Let $[\![\langle u, \overline{S}\rangle]\!] = \{\langle u, S\rangle \mid S \text{ refines } \overline{S}\}$. The *superconfiguration distance* from R to a superconfiguration $\overline{c}$ is defined as

$$d(R, \overline{c}) = \bigoplus_{c \in [\![\overline{c}]\!]} d(R, c)$$

The superconfiguration distance is only concerned with the sequence of modules that have been used to reach the node u, rather than the concrete sequence of boxes as in the configuration distance. This is a coarser notion than configuration and allows for fast overapproximation.

- *Node and same-context distance.* The *node distance* of a node u from R is defined as

$$d(R, u) = \bigoplus_{c = \langle u, S\rangle} d(R, c)$$

where S ranges over stacks of $\mathcal{R}$. Finally, the *same-context node distance* of a node u in module $\mathcal{M}_i$ is defined as

$$d(\mathcal{M}_i, u) = \bigoplus_{e \in En_i} d(\langle e, \varepsilon\rangle, \langle u, \varepsilon\rangle).$$

Intuitively, the node distance minimizes over all possible ways (i.e., stack sequences) to reach a node, and the same-context problem considers nodes in the same module that can be reached with empty stack.

Relevance. We discuss the relevance of the model and the problems we consider in program analysis. A prime application area of our framework is the analysis of procedural programs. Computations in an RSM correspond to the interprocedurally valid paths of a program. The distance values defined above allow to obtain information at different levels of granularity, depending on the requirement for a particular analysis. MEME (multi-entry, multi-exit) RSMs naturally arise in the model checking of procedural programs, where every node represents a combination of control location and data. Checking for reachability, usually of an error state, requires only the simple Boolean semiring. On the other hand, interprocedural data flow analysis problems, like in IFDS/IDE, are usually cast on SESE (single-entry, single-exit) RSMs (the control flow graph of the program) using richer semirings. Our framework captures both of these important applications, and furthermore allows a hybrid approach of modeling program information both in the state space of the RSM as well as in the semiring.

3 Configuration Distance Algorithm

In this section we present an algorithm which takes as input an RSM $\mathcal{R}$ and a representation of a *regular set* of configurations R, and computes a representation of the set of reachable configurations $post^*(R)$ that allows the extraction of the distance metrics defined above. In Section 3.1 we introduce configuration automata as representation structures for regular sets of configurations. In Section 3.2 we present an algorithm for RSMs over finite-height semirings. The algorithm saturates the input configuration automaton with additional transitions and assigns the correct weights via a dynamic programming approach that gradually relaxes transition weights from an initial overapproximation. We exploit the monotonicity property in idempotent semirings which allows to factor the computation into subproblems, and hence corresponds to the *optimal substructure* property of dynamic programming. Although a transition might have to be processed multiple times, the finite height of the semiring prevents a transition from being relaxed indefinitely. Here we show that the final configuration automata constructed by our algorithms correctly capture configuration distances. The extraction of distance values is considered in Section 4.

3.1 Configuration Automata

In general, like R, the set $post^*(R)$ is infinite. Hence we make use of a representation of regular sets of configurations as the language accepted by configuration automata, defined below. The main feature of a regular set of configurations R is its closure under $post^*$. That is, $post^*(R)$ is also a regular set of configurations and can be represented by a configuration automaton.

Intuition. Every state in a configuration automaton corresponds to a node in the RSM. In order to represent arbitrary regular sets of configurations we must allow the replication of states with the same node. Therefore we annotate every state with a *mark* (see Remark 2 for details). Transitions are of two types:

(i) ε-transitions pointing from a node u to an entry node e and labeled with ε, denoting that a computation reaching u entered the module of u via entry e, and (ii) b-transitions pointing from an entry node e to another entry node e' and labeled with a box b, corresponding to a call transition $\langle u, \langle b, e\rangle\rangle$ in the module of e' in the RSM. Reading the labels along a path in the automaton yields a stack.

In addition to the labeling with boxes we label every transition of a configuration automaton with a semiring value. In the final configuration automata constructed by our algorithms, every run generates a configuration c and thereby captures a certain subset $\Pi \subseteq \Pi(R, c)$ of computations from the initial set of configurations R to c. The weight of the run equals the combine over the weight of the computations in Π. The combine over the weights of all runs in the automaton that generate c equals the combine over the weights of all computations from R to c, i.e., the configuration distance $d(R, c)$. Since the transitions in a configuration automaton are essentially reversed transitions of the RSM (and the extend operation is not commutative), the weight of a run is given by the extend of the transitions in reversed order.

Configuration automata. Let $\mathbb{M}$ be a countably infinite set of *marks*. A *configuration automaton for an RSM $\mathcal{R}$*, also called an *$\mathcal{R}$-automaton*, is a weighted finite automaton $\mathcal{C} = \langle Q, B, \rightarrow, I, F, \ell\rangle$, where

- $Q \subseteq (In \cup En \cup Ret) \times \mathbb{M}$ is a finite set of *states*,
- B (the boxes of $\mathcal{R}$) is the transition alphabet,
- $\rightarrow \subseteq Q \times (B \cup \{\varepsilon\}) \times Q$ is a transition relation, such that every transition has one of the following forms:
  - *b-transition:* $\langle e, m_e\rangle \xrightarrow{b} \langle e', m_{e'}\rangle$, where $b \in B_i$ for some i, $e \in En_{Y_i(b)}$, and $e' \in En_i$,
  - *ε-transition:* $\langle u, m_u\rangle \xrightarrow{\varepsilon} \langle e, m_e\rangle$, where $e \in En_i$ for some i, and either $u \in In_i \cup Ret_i$, or $u = e$,
- $I \subseteq Q$ is a set of *initial states*,
- $F \subseteq Q$ and $F \subseteq En \times \mathbb{M}$ is a set of *final states*,
- $\ell : \rightarrow \mapsto D$ is a *weight function* that assigns a semiring weight to every transition.

Remark 2 (Marks). The marks in the states of a configuration automaton are introduced to support the general setting of representing an arbitrary set of configurations, e.g., with stacks that are not even reachable in the RSM. Since every state is tied to an RSM node, the marks allow to have multiple "copies" of the same node in unrelated parts of the automaton. Furthermore, our algorithm (Section 3.2) introduces a *fresh mark* to recognize when it can safely store *entry-to-exit summaries*. For the common setting of starting the analysis from the entry nodes of a main module with empty stack, marks are not necessary and can be elided.

Runs and regular sets of configurations. A *run* of a configuration automaton $\mathcal{C}$ is a sequence $\lambda = t_1, \ldots, t_n$, such that there are states $q_1, \ldots, q_{n+1}$ and each $t_i = q_i \xrightarrow{\sigma_i} q_{i+1}$ is a transition of $\mathcal{C}$ labeled with σ_i. We say that λ is a run from q_1 to q_{n+1}, of length $|\lambda| = n$, labeled by $S = \sigma_1 \dots \sigma_n$, and of weight $\otimes(\lambda) = \bigotimes_{i=n}^{1} \ell(t_i)$ (note that the weights of the transitions are extended in reverse order). We write $\lambda : q \xrightarrow{S/v}{}^{*} q'$ to denote that λ is a run from q to q' of any length labeled by S and of weight v. We will also use the notation without v if we are not interested in the weight. The run λ is *accepting* if $q \in I$ and $q' \in F$. A configuration $\langle u, S \rangle$ is *accepted* by $\mathcal{C}$ if there is an accepting run $\lambda : \langle u, m \rangle \xrightarrow{S}{}^{*} q_f$ for some mark $m \in \mathbb{M}$, and additionally $\otimes(\lambda) \neq \overline{0}$. We say that two runs are *equivalent* if they accept the same configuration with the same weight. For technical convenience we consider that for every state $\langle e, m_e \rangle$ with entry node $e \in En$ there is an ε-self-loop $\langle e, m_e \rangle \xrightarrow{\varepsilon} \langle e, m_e \rangle$ with weight $\overline{1}$.

The set of all configurations accepted by $\mathcal{C}$ is denoted by $\mathcal{L}(\mathcal{C})$. A set of configurations R is called *regular* if there exists an $\mathcal{R}$-automaton $\mathcal{C}$ such that $\mathcal{L}(\mathcal{C}) = R$. For a configuration c let $\Lambda(c)$ be the set of all accepting runs of c and define $\mathcal{C}(c) = \bigoplus_{\lambda \in \Lambda(c)} \otimes(\lambda)$ the weight that $\mathcal{C}$ assigns to c.

We note that, despite the imposed syntactic restrictions, our definition of configuration automata is most general in the following sense.

Proposition 1. *Let R be a set of configurations such that their string representations is a regular language. Then there exists a configuration automaton $\mathcal{C}$ such that $\mathcal{L}(\mathcal{C}) = R$.*

3.2 Algorithm for Finite-Height Semirings

In the following we present algorithm `ConfDist` for computing the set $post^*(R)$ of a regular set of configurations R. The algorithm operates on an $\mathcal{R}$-automaton $\mathcal{C}$ with $\mathcal{L}(\mathcal{C}) = R$. In the end, it has constructed an $\mathcal{R}$-automaton $\mathcal{C}_{post^*}$ such that $\mathcal{L}(\mathcal{C}_{post^*}) = post^*(R)$. Moreover, the configuration distance $d(R, c)$ from R to any configuration c can be obtained from the labels of $\mathcal{C}_{post^*}$ as $\mathcal{C}_{post^*}(c)$. A computation is called *initialized*, if its first configuration is accepted by the initial configuration automation $\mathcal{C}$.

Key technical contribution. In this work we consider the configuration distance computation. Using the notion of configuration automata as a *symbolic* representation structure for regular sets of configurations, the solution of the configuration distance problem has been previously studied in the setting of (weighted) pushdown systems [36,34,9]. One of the main algorithmic ideas for the efficient RSM reachability algorithm of [4] is to expand RSM transitions and use entry-to-exit *summaries* to avoid traversing a module more than once. However, the algorithm in [4] is limited to the node reachability problem. We combine the symbolic representation of configuration automata, along with the summarization principle, to obtain an efficient algorithm for the general configuration distance problem on RSMs.

Intuitive description of `ConfDist`. The intuition behind our algorithm is very simple: it performs a forward search in the RSM. In every iteration it picks a frontier node u and extends the already discovered computations to u with the outgoing transitions from u. Depending on the type of outgoing transitions, a new node discovered and added to the frontier can be (a) an internal node by following an internal transition, (b) the entry node of another module by following a call transition, and (c) a return node corresponding to a previously discovered call by following an exit transition.

In summary, the algorithm simply follows interprocedural paths. However, the crux to achieve our complexity is to keep summaries of paths through a module. Whenever we discovered a full (interprocedural) path from an entry e to an exit x, we keep its weight as an upper bound. Now any subsequently discovered call reaching e does not need to continue the search from e, but short-circuits to x by using the stored summary.

Preprocessing. In order to ease the formal presentation of the algorithm, we consider the following preprocessing on the initial configuration automaton $\mathcal{C}$. Let $M \subseteq \mathbb{M}$ be the set of marks in the initial automaton and $\widehat{m} \in \mathbb{M} \setminus M$ a *fresh mark*.

1. For every node $u \in \mathit{In} \cup \mathit{En} \cup \mathit{Ret}$, we add a new state $\langle u, \widehat{m} \rangle$ marked with the fresh mark. Additionally, all these new states are declared initial.
2. For every initial state $\langle u, m_u \rangle \in I$ such that there is a call transition $t = \langle u, \langle b, e \rangle \rangle \in \delta_i$ in $\mathcal{R}$, for every state $\langle e', m_{e'} \rangle$ where e' is an entry node of the same module as u, we add a b-transition $\langle e, \widehat{m} \rangle \xrightarrow{b} \langle e, m_{e'} \rangle$ with weight $\overline{0}$.
3. For every state $\langle e, m_e \rangle$ with entry node $e \in \mathit{En}_i$ and every internal or return node $u \in \mathit{In}_i \cup \mathit{Ret}_i$ in the same module as e, we add an ε-transition $\langle u, \widehat{m} \rangle \xrightarrow{\varepsilon} \langle e, m_e \rangle$ with weight $\overline{0}$.

Essentially the preprocessing a priori adds to $\mathcal{C}$ all possible states and transitions, so that the algorithm only has to relax those transitions (i.e., without adding them first). Note that the preprocessing only provides for an easier presentation of our algorithm. Indeed, in practice it would be impractical to do the full preprocessing and thus our implementation adds states and transitions to the automaton on the fly.

Technical description of `ConfDist`. We present a detailed explanation of the algorithm supporting the formal description given in Algorithm 1. We require that every transition in the input configuration automaton $\mathcal{C}$ has weight $\overline{1}$, since the configurations in $\mathcal{L}(\mathcal{C})$ should not contribute any initial weight to the configuration distance. The algorithm maintains a *worklist* WL of weighted transitions either of the form $\langle u, m_u \rangle \xrightarrow{\varepsilon} \langle e, m_e \rangle$ or $\langle e, m_e \rangle \xrightarrow{b} \langle e', m_{e'} \rangle$, and a *summary function* $\mathsf{sum} : (\mathit{En} \times \mathbb{M}) \times \mathit{Ex} \mapsto D$. Initially, the worklist contains all such transitions where the source state $\langle u, m_u \rangle$ is an initial state in I, and sum is all $\overline{0}$. In every iteration a transition $t_{\mathcal{C}}$ is extracted from the worklist and processed as follows. Since every accepting run starting with $t_{\mathcal{C}}$ corresponds to a reachable configuration $\langle u, S \rangle$ (where S varies over different runs), every transition $t_{\mathcal{R}} = \langle u, u' \rangle$ in $\mathcal{R}$ gives rise to another reachable configuration. More precisely, the run corresponds to a set of computations reaching $\langle u, S \rangle$ from the initial set of configurations, and $t_{\mathcal{R}}$ allows to extend these computations by one step. The algorithm incorporates the newly discovered computations by relaxing a transition as follows, illustrated in Figure 2.

$$\boxed{t_{\mathcal{R}} = \langle u, u' \rangle}$$

$$\begin{tikzcd}
\langle u, m_u \rangle \arrow[r, "\varepsilon"] & \langle e, m_e \rangle \arrow[r, dotted, "S"] & {} \\
\langle u', \widehat{m} \rangle \arrow[ur, "\varepsilon"] & &
\end{tikzcd}$$

(a) Internal transition (Lines 8–9)

$$\boxed{t_{\mathcal{R}} = \langle u, \langle b, e' \rangle \rangle}$$

$$\begin{tikzcd}
\langle u, m_u \rangle \arrow[r, "\varepsilon"] & \langle e, m_e \rangle \arrow[r, dotted, "S"] & {} \\
\langle e', \widehat{m} \rangle \arrow[ur, "b"] & &
\end{tikzcd}$$

(b) Call transition (Lines 10–12)

$$\boxed{t_{\mathcal{R}} = \langle u, x \rangle}$$

$$\begin{tikzcd}
\langle u, m_u \rangle \arrow[r, "\varepsilon"] & \langle e, m_e \rangle \arrow[r, "b"] & \langle e', m_{e'} \rangle \arrow[r, dotted, "S'"] & {} \\
& \langle \langle b, x \rangle, \widehat{m} \rangle \arrow[ur, "\varepsilon"] & &
\end{tikzcd}$$

(c) Exit transition (Lines 13–17)

$$\boxed{\mathsf{sum}(\langle e, \widehat{m} \rangle, x)}$$

$$\begin{tikzcd}
\langle e, \widehat{m} \rangle \arrow[r, "b"] & \langle e', m_{e'} \rangle \arrow[r, dotted, "S"] & {} \\
\langle \langle b, x \rangle, \widehat{m} \rangle \arrow[ur, "\varepsilon"] & &
\end{tikzcd}$$

(d) Using summary (Lines 18–21)

Fig. 2. Relaxation steps of `ConfDist`.

1. If $t_{\mathcal{C}}$ is of the form $\langle u, m_u \rangle \xrightarrow{\varepsilon} \langle e, m_e \rangle$, then:
   (a) If u' is an internal node then the algorithm captures the internal transition $\langle u, S \rangle \Rightarrow \langle u', S \rangle$ by relaxing the transition $\langle u', \widehat{m} \rangle \xrightarrow{\varepsilon} \langle e, m_e \rangle$ using the weights $\ell(t_{\mathcal{C}})$ and $w(t_{\mathcal{R}})$.
   (b) If u' is a call node $\langle b, e' \rangle$ then the transition $\langle e', \widehat{m} \rangle \xrightarrow{b} \langle e, m_e \rangle$ is relaxed with the new weight $\ell(t_{\mathcal{C}}) \otimes w(t_{\mathcal{R}})$. Furthermore, an ε-self-loop is stored in the worklist to continue exploration from the called entry node e'.
   (c) If u' is an exit node x then the algorithm relaxes $\mathsf{sum}(\langle e, m_e \rangle, x)$ if a smaller computation to x has been discovered. Note that for $m_e = \widehat{m}$ this corresponds to valid entry-to-exit computations from e to x. If another call to e is discovered later, the summary is used to avoid traversing the module again. For $m_e \neq \widehat{m}$ the summary does not necessarily correspond to valid entry-to-exit computations (e.g., because node u was provided as an initial configuration) and is only stored to avoid redundant work. For a return transition from $\langle u, S \rangle$ the stack S has to be non-empty. The algorithm looks for all possible boxes b at the top of S by going along a b-transition from $\langle e, m_e \rangle$ to a state $\langle e', m_{e'} \rangle$. Then for any $S = bS'$, relaxing the transition $\langle \langle b, x \rangle, \widehat{m} \rangle \xrightarrow{\varepsilon} \langle e', m_{e'} \rangle$ captures the return transition $\langle u, S \rangle \Rightarrow \langle \langle b, x \rangle, S' \rangle$. Note that here we make use of the fact that the return transition itself has weight $\overline{1}$.
2. If $t_{\mathcal{C}}$ is of the form $\langle e, m_e \rangle \xrightarrow{b} \langle e', m_{e'} \rangle$, then:
   (d) for every exit node x in the module of e the summary function is used to relax the weight of the transition $\langle \langle b, x \rangle, \widehat{m} \rangle \xrightarrow{\varepsilon} \langle e', m_{e'} \rangle$ to the value $\ell(t_{\mathcal{C}}) \otimes \mathsf{sum}(\langle e, \widehat{m} \rangle, x)$.

The initial states of $\mathcal{C}_{post^*}$ are the initial states of $\mathcal{C}$ together with all states with the fresh mark added in the preprocessing. The final states of $\mathcal{C}_{post^*}$ are the unmodified final states of $\mathcal{C}$.

Algorithm 1: `ConfDist`

```
Input: RSM $\mathcal{R}$ and $\mathcal{R}$-automaton $\mathcal{C}$ with $\ell(t) = \overline{1}$ for all transitions $t$ in $\mathcal{C}$
Output: $\mathcal{R}$-automaton $\mathcal{C}_{post^*}$ with $\mathcal{C}_{post^*}(c) = d(\mathcal{L}(\mathcal{C}), c)$ for all configurations $c$
1  preprocess $\mathcal{C}$ as described in the main text
   // Initialization of worklist and summary function
2  WL := $\{t = q \xrightarrow{\varepsilon} q' \mid q \in I$ and $\ell(t) = \overline{1}\}$
3  sum$(\langle e, m_e\rangle, x) := \overline{0}$ for all states $\langle e, m_e\rangle$ and $x \in Ex$
   // Main loop
4  while WL $\neq \emptyset$ do
5      extract $t_{\mathcal{C}}$ from WL
6      if $t_{\mathcal{C}} = \langle u, m_u\rangle \xrightarrow{\varepsilon} \langle e, m_e\rangle$ then
7          let $\mathcal{M}_i$ be the module of node $u$
           // Internal transitions from $u$
8          foreach $t_{\mathcal{R}} = \langle u, u'\rangle \in \delta_i$ where $u' \in In_i$ do
9              Relax($\langle u', \widehat{m}\rangle \xrightarrow{\varepsilon} \langle e, m_e\rangle, \ell(t_{\mathcal{C}}) \otimes w_i(t_{\mathcal{R}})$)
           // Call transitions from $u$
10         foreach $t_{\mathcal{R}} = \langle u, \langle b, e'\rangle\rangle \in \delta_i$ do
11             Relax($\langle e', \widehat{m}\rangle \xrightarrow{b} \langle e, m_e\rangle, \ell(t_{\mathcal{C}}) \otimes w_i(t_{\mathcal{R}})$)
12             add $\langle e', \widehat{m}\rangle \xrightarrow{\varepsilon} \langle e', \widehat{m}\rangle$ to WL, if it was never added before
           // Exit transitions from $u$
13         foreach $t_{\mathcal{R}} = \langle u, x\rangle \in \delta_i$ where $x \in Ex_i$ do
14             if sum$(\langle e, m_e\rangle, x) \not\sqsubseteq \ell(t_{\mathcal{C}})$ then
15                 sum$(\langle e, m_e\rangle, x)$ := sum$(\langle e, m_e\rangle, x) \oplus \ell(t_{\mathcal{C}})$
16                 foreach $\langle e, m_e\rangle \xrightarrow{b/v} \langle e', m_{e'}\rangle$ do
17                     Relax($\langle\langle b, x\rangle, \widehat{m}\rangle \xrightarrow{\varepsilon} \langle e', m_{e'}\rangle, v \otimes$ sum$(\langle e, m_e\rangle, x)$)
18     else if $t_{\mathcal{C}} = \langle e, m_e\rangle \xrightarrow{b} \langle e', m_{e'}\rangle$ then
19         let $\mathcal{M}_i$ be the module of node $e$
           // Using entry-to-exit summaries
20         foreach $x \in Ex_i$ do
21             Relax($\langle\langle b, x\rangle, \widehat{m}\rangle \xrightarrow{\varepsilon} \langle e', m_{e'}\rangle, \ell(t_{\mathcal{C}}) \otimes$ sum$(\langle e, \widehat{m}\rangle, x)$)
22 Procedure Relax($t, v$)
23     if $\ell(t) \neq \ell(t) \oplus v$ then
24         $\ell(t) := \ell(t) \oplus v$
25         add $t$ to WL
```

Example 2. In Figure 3 we illustrate an execution of `ConfDist` for the reachability problem in the RSM from Figure 1. The reader can verify that every configuration in the example computation (1) is accepted by a run of the constructed automaton.

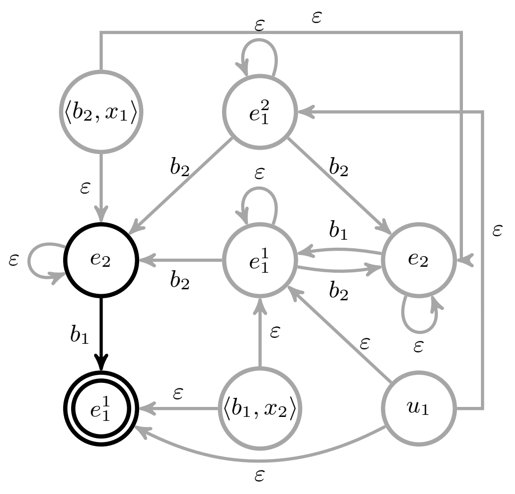

Fig. 3. The configuration automaton $\mathcal{C}_{post^*}$ constructed by `ConfDist` for the RSM in Figure 1 over the Boolean semiring $\langle\langle 0,1\rangle, \vee, \wedge, 0, 1\rangle$, expressing the reachability problem. The initial input automaton $\mathcal{C}$ is given by the black states, whereas the gray states represent the newly added states with the fresh mark $\widehat{m}$. The black/gray color gives a similar distinction for the transitions (i.e., the gray transitions have been added by the algorithm). The set of initial states of $\mathcal{C}$ is $I = \{e_1^1, e_2\}$, and the set of final states is the singleton set $F = \{e_1^1\}$. Transitions added in the preprocessing phase with value $\overline{0}$ are not shown.

Correctness. In the following we outline the correctness of the algorithm. We start with a simple observation about the shape of runs in the constructed configuration automaton.

Proposition 2. *For every accepting run λ there exists an equivalent accepting run λ' that starts with an ε-transition followed by only b-transitions. Furthermore, all but the first state contain an entry node.*

The following three lemmas capture the correctness of `ConfDist`. We start with completeness, namely that the distance computed for any configuration c is at most the actual distance from the initial set of configurations $\mathcal{L}(\mathcal{C})$ to c. The proof relies on showing that for any initialized computation $\pi : \langle u, S\rangle \Rightarrow^* \langle u', S'\rangle$ there is a run λ accepting $\langle u', S'\rangle$ such that $\otimes(\lambda) \sqsubseteq \otimes(\pi)$, and follows an induction on the length $|\pi|$.

Lemma 1 (Completeness). *For every configuration c we have $\mathcal{C}_{post^*}(c) \sqsubseteq d(\mathcal{L}(\mathcal{C}), c)$.*

We now turn our attention to soundness, namely that the distance computed for any configuration c is at least the actual distance from the initial set of configurations $\mathcal{L}(\mathcal{C})$ to c. The proof is established via a set of interdependent invariants that state that the algorithm maintains sound entry-to-exit summaries and any run in the automaton has a weight that is witnessed by a set of computations.

Lemma 2 (Soundness). *For every configuration c we have $d(\mathcal{L}(\mathcal{C}), c) \sqsubseteq \mathcal{C}_{post^*}(c)$.*

Complexity. Finally, we turn our attention to the complexity analysis of the algorithm, which is done by bounding the number of times the algorithm can perform a relaxation step. The complexity bound is based on the height of the semiring H, which implies that every transition can be relaxed at most H times. The contribution of the size of the initial automaton $\mathcal{C}$ in the complexity is captured by the number of initial marks κ.

Lemma 3 (Complexity). *Let κ be the number of distinct marks $m \in \mathbb{M}$ of the initial automaton $\mathcal{C}$. Algorithm* `ConfDist` *constructs $\mathcal{C}_{post^*}$ in time $O(H \cdot (|\mathcal{R}| \cdot \theta_e \cdot \kappa^2 + |Call| \cdot \theta_e \cdot \theta_x \cdot \kappa^3))$, and $\mathcal{C}_{post^*}$ has $O(|\mathcal{R}| \cdot \theta_e \cdot \kappa^2)$ transitions.*

We summarize the results of this section in the following theorem.

Theorem 1. *Let $\mathcal{R}$ be an RSM over a semiring of height H, and $\mathcal{C}$ an $\mathcal{R}$-automaton with κ marks. Algorithm* `ConfDist` *constructs in $O(H \cdot (|\mathcal{R}| \cdot \theta_e \cdot \kappa^2 + |Call| \cdot \theta_e \cdot \theta_x \cdot \kappa^3))$ time an $\mathcal{R}$-automaton $\mathcal{C}_{post^*}$ with $\kappa + 1$ marks, such that $d(\mathcal{L}(\mathcal{C}), c) = \mathcal{C}_{post^*}(c)$ for every configuration c.*

Remark 3 (Comparison with existing work). We now relate Theorem 1 with the existing work for computing configuration distance (often called generalized reachability in the literature) in weighted pushdown systems *(WPDS)* [36,34]. For simplicity we assume that the initial automaton is of constant size. A formal description of WPDS is omitted; the reader can refer to [4,34]. Let $\mathcal{P}$ be a WPDS where:

1. $n_{\mathcal{P}}$ is the number of states
2. n_{Δ} is the size of the transition relation
3. n_{sp} is the number of different pairs $\langle p', \gamma' \rangle$ such that there is a transition of the form $\langle p, \gamma \rangle \to \langle p', \gamma' \gamma'' \rangle$ (i.e., from some state p with γ on the top of the stack, the WPDS $\mathcal{P}$ (i) transitions to state p', (ii) swaps γ and γ'', and (iii) pushes γ' on the top of the stack).

As shown in [34], given a WPDS $\mathcal{P}$ with weights from a semiring with height H, together with a corresponding automaton $\mathcal{C}^{\mathcal{P}}$ that encodes configurations of $\mathcal{P}$, an automaton $\mathcal{C}^{\mathcal{P}}_{post^*}$ can be constructed as a solution to the configuration distance problem for $\mathcal{P}$. For ease of presentation we focus on the common case where $\mathcal{C}^{\mathcal{P}}$ has constant size (e.g., for encoding an initial configuration of $\mathcal{P}$ with empty stack). Then the time required to construct $\mathcal{C}^{\mathcal{P}}_{post^*}$ is $O(H \cdot n_{\mathcal{P}} \cdot n_{\Delta} \cdot n_{\mathsf{sp}})$ [36,34].

A direct consequence of [4, Theorem 1] is that an RSM $\mathcal{R}$ and a configuration automaton $\mathcal{C}^{\mathcal{R}}$ can be converted to an equivalent PDS $\mathcal{P}$ and configuration automaton $\mathcal{C}^{\mathcal{P}}$, and vice versa, such that the following equalities hold:

$$|\mathcal{R}| = \Theta(n_{\Delta}); \quad \theta_x = \Theta(n_{\mathcal{P}}); \quad f \cdot \theta_e = \Theta(n_{\mathsf{sp}}),$$

where f represents the number of modules. Hence, the bound we obtain by translating the input RSM to a WPDS and using the algorithm of [36,34] is $O(H \cdot |\mathcal{R}| \cdot \theta_e \cdot \theta_x \cdot f)$. Our complexity bound on Theorem 1 is better by a factor $\Omega((|\mathcal{R}| \cdot f)/(\theta_x + |Call|))$. Moreover, to verify such improvements, we have also constructed a family of dense RSMs, and apply our algorithm, and compare against the jMoped implementation of the existing algorithms, and observe a linear speed-up (see Section 6.1 for details).

The above analysis considers an explicit model, where $\mathcal{R}$ comprises two parts, a program control-flow graph $\mathcal{R}_{\text{CFG}}$ and the set of all data valuations V, where $|V| = \theta_e = \theta_x$. Hence, $|\mathcal{R}| = |\mathcal{R}_{\text{CFG}}| \cdot |V|^2$. In a symbolic model, where all the data valuations are tracked on the semiring, the input RSM is a factor $|V|^2$ smaller (i.e., the contribution of the data valuation to $|\mathcal{R}|$), and $\theta_e = \theta_x = 1$. However, now each semiring operation incurs a factor $|V|^2$ increase in time cost, and the height of the semiring increases by a factor $|V|^2$ as well, in the worst case. Hence, existing symbolic approaches for PDSs have the same worst-case time complexity as the explicit one, and our comparison applies to these as well. For further discussion on symbolic extensions of our algorithm we refer to Appendix B.

4 Distance Extraction

The algorithm presented in Section 3 takes as input a weighted RSM $\mathcal{R}$ over a semiring and a configuration automaton $\mathcal{C}$ that represents a regular set R of configurations of $\mathcal{R}$, and outputs an automaton $\mathcal{C}_{post^*}$ that encodes the distance $d(R, c)$ to every configuration c. We now discuss the algorithmic problem of extracting such distances from $\mathcal{C}_{post^*}$, and present fast algorithms for this problem. First we will consider the general case for RSMs over an arbitrary semiring. Then we present several improvements for special cases, like RSMs over a semiring with small domain, or sparse RSMs. As the correctness of the constructions is straightforward, our attention will be on the complexity.

4.1 Distances over General Semirings

Configuration distances. Given a configuration $c = \langle u, S\rangle$, $S = b_1 \dots b_{|S|}$, the task is to extract $d(R, c) = \bigoplus(\Pi(R, c))$. This is done by a dynamic-programming style algorithm, which computes iteratively for every prefix $b_1 \dots b_i$ of S and state $\langle e, m_e\rangle$ with $e \in En_j$ and $b_i \in B_j$, the weight

$$w_{\langle e,m_e\rangle} = \bigoplus\{\otimes(\lambda) \mid \lambda : \langle u, m_u\rangle \xrightarrow{b_1\dots b_i}{}^{*} \langle e, m_e\rangle\}.$$

Since there are $O(\kappa^2 \cdot \theta_e^2)$ transitions labeled with b_i, every iteration requires $O(\kappa^2 \cdot \theta_e^2)$ time, and the total time for computing $d(R, c)$ is $O(|S| \cdot \kappa^2 \cdot \theta_e^2)$.

Superconfiguration distances. Given a superconfiguration $\overline{c} = \langle u, \overline{S}\rangle$, $\overline{S} = \mathcal{M}_1 \dots \mathcal{M}_{|\overline{S}|}$, the task is to extract $d(R, \overline{c}) = \bigoplus_{c \in [\![\langle u,\overline{S}\rangle]\!]} d(R, c)$. To handle such queries, we perform a one-time preprocessing of $\mathcal{C}_{post^*}$, so that the transitions are labeled with modules instead of boxes. That is, we create an automaton $\overline{\mathcal{C}}_{post^*}$, initially identical to $\mathcal{C}_{post^*}$. Then we add a transition $t = \langle e, m_e\rangle \xrightarrow{\mathcal{M}} \langle e', m_{e'}\rangle$, with $\mathcal{M}$ being the module of e', if there exists a b-transition $\langle e, m_e\rangle \xrightarrow{b} \langle e', m_{e'}\rangle$ in $\mathcal{C}_{post^*}$. The weight function $\overline{\ell}$ of $\overline{\mathcal{C}}_{post^*}$ is such that the weight of the transition t is

$$\overline{\ell}(t) = \bigoplus_{t' : \langle e, m_e\rangle \xrightarrow{b} \langle e', m_{e'}\rangle} \ell(t')$$

where t' ranges over transitions of $\mathcal{C}_{post^*}$. This construction requires linear time in the number of b-transitions of $\mathcal{C}_{post^*}$, i.e., $O(|\mathcal{R}| \cdot \theta_e)$. It is straightforward to see that

$$\bigoplus_{\overline{\lambda}:\langle u,m_u\rangle \xrightarrow{\overline{S}}{}^* q_f} \overline{\ell}(\lambda) = \bigoplus_{\lambda:\langle u,m_u\rangle \xrightarrow{S}{}^* q_f} \ell(\lambda)$$

where $\overline{\lambda}$ and λ range over accepting runs of $\overline{\mathcal{C}}_{post^*}$ and $\mathcal{C}_{post^*}$ respectively, and S refines $\overline{S}$. Then, given a superconfiguration $\overline{c} = \langle u, \overline{S}\rangle$, the extraction of $d(R, \overline{c})$ is done similarly to the configuration distance extraction, in $O(|S| \cdot \kappa^2 \cdot \theta_e^2)$ time.

Node distances. For node distances, the task is to compute $d(R, u) = \bigoplus_{c=\langle u,S\rangle} d(R, c)$ for every node u of $\mathcal{R}$. This reduces to treating the automaton $\mathcal{C}_{post^*}$ as a graph G, and solving a traditional single-source distance problem, where the source set contains all states with old marks (i.e., old states that appear in the initial automaton $\mathcal{C}$). This requires $O(H \cdot |\mathcal{C}_{post^*}|)$ time for semirings of height H. An informal argument for these bounds is to observe that G can be itself encoded by a SESE RSM $\mathcal{R}_G$ with a single module, where the entry represents the source set of nodes with old marks. Then, running `ConfDist` for the corresponding semiring, we obtain a solution to the single-source distance problem in the aforementioned times, as established in Theorem 1. Finally, computing same-context node distances requires $O(|\mathcal{R}| \cdot \theta)$ time in total (i.e., for all nodes). Hence, regardless of the semiring, all node distances can be computed with no overhead, i.e., within the time bounds required for constructing the respective configuration automaton $\mathcal{C}_{post^*}$. The following theorem summarizes the complexity bounds that we obtain for the various distance extraction problems.

Theorem 2 (Distance extraction). *Let $\mathcal{R}$ be an RSM over a semiring of height H and $\mathcal{C}$ an $\mathcal{R}$-automaton with κ marks. After $O(H \cdot |\mathcal{R}| \cdot \theta_e \cdot \theta_x \cdot \kappa^3)$ preprocessing time*

1. *configuration and superconfiguration distance queries $\langle u, S\rangle$ are answered in $O(|S| \cdot \theta_e^2 \cdot \kappa^2)$ time;*
2. *node distance queries are answered in $O(1)$ time.*

4.2 Distances over Semirings with Small Domain

We now turn our attention to configuration and superconfiguration distance extraction for the case of semirings with small domains D. Such semirings express a range of important problems, with reachability being the most well-known (expressed on the Boolean semiring with $|D| = 2$). We harness algorithmic advancements on the matrix-vector multiplication problem and Four-Russians-style algorithms to obtain better bounds on the distance extraction problem.

Recall that given a box b, the configuration automaton $\mathcal{C}_{post^*}$ has at most $(\theta_e \cdot \kappa)^2$ transitions labeled with b . Such transitions can be represented by a matrix $A_b \in D^{(\theta_e \cdot \kappa) \times (\theta_e \cdot \kappa)}$. Additionally, for every internal node u we have one matrix $A_u \in D^{(\kappa) \times (\theta_e \cdot \kappa)}$ that captures the weights of all transitions of the form

$\langle u, m_u \rangle \xrightarrow{\varepsilon} \langle e, m_e \rangle$. Then, answering a configuration distance query $\langle u, S \rangle$ with $S = b_1, \ldots, b_{|S|}$ amounts to evaluating the expression

$$\overline{\mathbf{1}}_\kappa \cdot A_u \cdot A_{b_1} \cdots A_{b_{|S|}} \cdot \overline{\mathbf{1}}^\top_{\kappa \cdot \theta_e} \tag{2}$$

where $\overline{\mathbf{1}}_z$ is a row vector of $\overline{1}$s and size z, $\cdot^\top$ denotes the transpose, and matrix multiplication is taken over the semiring. The situation is similar in the case of superconfiguration distances, where we have one matrix $A_{\mathcal{M},\mathcal{M}'}$ for each pair of modules $\mathcal{M}$, $\mathcal{M}'$ such that $\mathcal{M}$ invokes $\mathcal{M}'$.

Evaluating equation (2) from left to right (or right to left) yields a sequence of matrix-vector multiplications. The following two theorems use the results of [24] and [38] on matrix-vector multiplications to provide a speedup on the distance extraction problem when the semiring has constant size $|D| = O(1)$.

Theorem 3 (Mailman's speedup [24]). *Let $\mathcal{R}$ be an RSM over a semiring of constant size, and $\mathcal{C}$ an $\mathcal{R}$-automaton with κ marks. After $O(|\mathcal{R}| \cdot \theta_e \cdot \theta_x \cdot \kappa^3)$ preprocessing time, configuration and superconfiguration distance queries $\langle u, S \rangle$ are answered in $O\left(|S| \cdot \frac{\theta_e^2 \cdot \kappa^2}{\log(\theta_e \cdot \kappa)}\right)$ time.*

Theorem 4 (Williams's speedup [38]). *Let $\mathcal{R}$ be an RSM over a semiring of size $|D|$, and $\mathcal{C}$ an $\mathcal{R}$-automaton with κ marks. For any fixed $\varepsilon > 0$, let $X = |\mathcal{R}| \cdot \theta_e \cdot \theta_x \cdot \kappa^3$ and $Z = |\mathcal{R}| \cdot \kappa \cdot (\theta_e \cdot \kappa)^{1+\varepsilon \log_2 |D|}$. After $O(\max(X, Z))$ preprocessing time, configuration and superconfiguration distance queries $\langle u, S \rangle$ are answered in $O\left(|S| \cdot \frac{\theta_e^2 \cdot \kappa^2}{\varepsilon^2 \cdot \log^2(\theta_e \cdot \kappa)}\right)$ time.*

Finally, using the Four-Russians technique for parsing on non-deterministic automata [27], we obtain the following speedup for the case of reachability. We note that although the alphabet is not of constant size (i.e., the number of boxes is generally non-constant) this poses no overhead, as long as comparing two boxes for equality requires constant time (which is the case in the standard RAM model).

Theorem 5 (Four-Russians speedup [27]). *Let $\mathcal{R}$ be an RSM over a binary semiring, and $\mathcal{C}$ an $\mathcal{R}$-automaton with κ marks. After $O(|\mathcal{R}| \cdot \theta_e \cdot \theta_x \cdot \kappa^3)$ preprocessing time, configuration and superconfiguration distance queries $\langle u, S \rangle$ are answered in $O\left(|\mathcal{R}| \cdot \theta_e \cdot \kappa^2 \cdot \frac{|S|}{\log(|S|)}\right)$ time.*

4.3 A Speedup for Sparse RSMs

We call an RSM $\mathcal{R}$ *sparse* if there is a constant bound r such that for all modules $\mathcal{M}_i$ we have $|\{Y_i(b) \mid b \in B_i\}| \leq r$ i.e., every module invokes at most r other modules (although $\mathcal{M}_i$ can have arbitrarily many boxes). Typical call-graphs of most programs are very sparse, e.g., typical call graphs of thousands of nodes have average degree at most eight [8,30]. Hence, an RSM modeling a typical program is expected to comprise thousands of modules, while the average module invokes a small number of other modules. Although this does not imply a constant bound on the number of invoked modules, such an assumption provides a good theoretical basis for the analysis of typical programs.

Our goal is to provide a speedup for extracting superconfiguration distances w.r.t. a sparse RSM. This is achieved by an additional polynomial-time preprocessing, which then allows to process a distance query in blocks of logarithmic size, and thus offers a speedup of the same order.

Given an RSM $\mathcal{R}$ of k modules and an integer z, there exist at most $k \cdot r^z$ valid module sequences $\mathcal{M}_1 \dots, \mathcal{M}_{z+1}$ which can appear as a substring in a module sequence $\overline{S}$ which is refined by some stack S. Recall the definition of the matrices $A_{\mathcal{M},\mathcal{M}'} \in D^{(\theta_e \cdot \kappa) \times (\theta_e \cdot \kappa)}$ from Section 4.2. For every valid sequence of $z+1$ modules $s = \mathcal{M}_1 \dots, \mathcal{M}_{z+1}$, we construct a matrix $A_s = A_{\mathcal{M}_1,\mathcal{M}_2} \cdot A_{\mathcal{M}_2,\mathcal{M}_3} \cdot \dots A_{\mathcal{M}_z,\mathcal{M}_{z+1}}$ in total time

$$k \cdot (\theta_e \cdot \kappa)^\omega \sum_{i=1}^{z} r^i = O\left(|\mathcal{R}| \cdot \theta_e^{\omega-1} \kappa^\omega \cdot r^z\right) \tag{3}$$

where $(\theta_e \cdot \kappa)^\omega = \Omega(\theta^2 \cdot \kappa^2)$ is time require to multiply two $D^{(\theta_e \cdot \kappa) \times (\theta_e \cdot \kappa)}$ matrices (currently $\omega \simeq 2.372$, due to [39]).

Observe that as long as $z = O(\log |\mathcal{R}|)$, there are polynomially many such sequences s, and thus each one can be indexed in $O(1)$ time on the standard RAM model. Then a superconfiguration distance query $\langle u, S \rangle$ can be answered by grouping S in $\lceil \frac{|S|}{z} \rceil$ blocks of size z each, and for each such block s multiply with matrix A_s.

Theorem 6 (Sparsity speedup). *Let $\mathcal{R}$ be a sparse RSM over a semiring of height H, and $\mathcal{C}$ an $\mathcal{R}$-automaton with κ marks. Let $X = H \cdot |\mathcal{R}| \cdot \theta_e \cdot \theta_x \cdot \kappa^3$, and given an integer parameter $x = O(\text{poly}\,|\mathcal{R}|)$, let $Z = |\mathcal{R}| \cdot \theta_e^{\omega-1} \kappa^\omega \cdot x$. After $O(\max(X, Z))$ preprocessing time, superconfiguration distance queries $\langle u, S \rangle$ are answered in $O\left(|S| \cdot \left\lceil \frac{\theta_e^2 \cdot \kappa^2}{\log x} \right\rceil\right)$ time.*

By varying the parameter z, Theorem 6 provides a tradeoff between preprocessing and query times. Finally, the presented method can be combined with the preprocessing on constant-size semirings of Section 4.2 which leads to a $\Theta(\log z)$ factor improvement on the query times of Theorem 3, Theorem 4, and Theorem 5.

5 Context-Bounded Reachability in Concurrent Recursive State Machines

Context bounding, i.e., limiting the number of context switches considered during state space exploration, is an effective technique for systematic analysis of concurrent programs. The context-bounded reachability problem in concurrent pushdown systems has been studied in [28]. In this section we phrase the context-bounded reachability problem over concurrent RSMs and show that the procedure of [28] using our algorithm `ConfDist` together with the results of the previous sections give a better time complexity for the problem. As the section follows closely the well-known framework of concurrent pushdown systems [28], we keep the description brief.

Concurrent RSMs. A *concurrent RSM (CRSM)* $\mathcal{R}^{\parallel}$ is a collection of RSMs $\mathcal{R}_i$ equipped with a finite set of *global states* G used for communication between the RSMs. To this end, the semantics of RSMs is lifted to $\mathcal{R}_i$-configurations of the form $\langle g, u_i, S_i \rangle$, carrying an additional global state $g \in G$. Then, a *global configuration* of $\mathcal{R}^{\parallel}$ is a tuple $\langle g, \langle u_1, S_1 \rangle, \dots, \langle u_n, S_n \rangle \rangle$, where $\langle g, u_i, S_i \rangle$ are configurations of $\mathcal{R}_i$, respectively. The semantics of $\mathcal{R}^{\parallel}$ over global configurations is the standard interleaving semantics, i.e., in each step some RSM $\mathcal{R}_i$ modifies the global state and its local configuration, while the local configuration of every other RSM remains unchanged.

Context-bounded reachability. For a positive natural number k and a fixed initial global configuration c, the *k-bounded reachability problem* asks for all global configurations c' such that there is a computation from c to c' that switches control between RSMs at most $k-1$ times.

An algorithm for context-bounded reachability. The procedure of [28] for solving the k-bounded reachability problem for *concurrent pushdown systems (CPDSs)* systematically performs *post** operations on the reachable configuration set of every constituent PDS, while capturing all possible interleavings within k context switches. The k-bounded reachability problem for CRSMs can be solved with an almost identical procedure, replacing the black-box invocations of the PDS reachability algorithm of [36] with our algorithm `ConfDist`. However, using our algorithm for each *post** operation, we obtain a complexity improvement over the method of [28].

Key complexity improvement. The key advantage of our algorithm as compared to [28] is as follows: in the algorithm of [28], in each iteration the configuration automata, used to represent the reachable configurations of each component RSM, grows by a cubic term; in contrast, replacing with our algorithm the configuration automata grows only by a linear term in each iteration. This comes from the fact that in our configuration automata every state corresponds to a node of the RSM, whereas such strong correspondence does not hold for the configuration automata of [28].

Theorem 7. *For a concurrent RSM $\mathcal{R}^{\parallel}$, and a bound k, the procedure of [28, Figure 2] using* `ConfDist` *for performing post* operations correctly solves the k-bounded reachability problem and requires $O(|\mathcal{R}^{\parallel}| \cdot \theta_e^{||} \cdot \theta_x^{||} \cdot n^k \cdot |G|^{k+2})$ time.*

Compared to Theorem 7, solving the CRSM problem by translation to a CPDS and using the algorithm of [28] gives the bound $O(|\mathcal{R}^{\parallel}|^5 \cdot {\theta_x^{||}}^5 \cdot n^k \cdot |G|^k)$. Conversely, solving the CPDS problem by translation to a CRSM and using our algorithm gives an improvement by a factor $\Omega(|\mathcal{P}^{\parallel}|^3/|G|^2)$. We refer to Appendix D for a detailed discussion.

6 Experimental Results

In this section we empirically demonstrate the algorithmic improvements achieved by our RSM-based algorithm over existing PDS-based algorithms on interprocedural program analysis problems. The main goal is to demonstrate the improvements in algorithmic ideas rather than implementation details and engineering aspects. In particular, we implemented our algorithm `ConfDist` in a prototype tool and compared its efficiency against jMoped [1], which implements the algorithms of [36,34] and is a leading tool for the analysis of weighted pushdown systems. In all cases we used an explicit representation of data valuations on the nodes of RSMs, as opposed to a symbolic semiring representation. All experiments were run on a machine with an Intel Xeon CPU and a memory limit of 80GB. We first present our result on a synthetic example to verify the algorithmic improvements on a constructed family, and then present results on real-world benchmarks.

6.1 A Family of Dense RSMs

For our first experiments we constructed a family of dense RSMs that can be scaled in size. The purpose of this experiments is to verify that (i) our algorithm indeed achieves a speedup over the algorithms of [36,34], and (ii) the speedup scales with the size of the input to ensure that improvements on real-world benchmarks are not due to implementation details, such as the used data types. Let $\mathcal{R}_n$ be a single-module RSM that consists of n entries and n exits, and a single box which makes a recursive call. The transition relation is $\delta = (En \times (Call \cup Ex)) \cup (Ret \times Ex)$, i.e., every entry node connects to every call and exit node, and every return node connects to every exit node. Hence $|\mathcal{R}_n| = n^2$. The transition weights are irrelevant, as we will focus on reachability. The initial configuration automaton $\mathcal{C}$ contains a single entry state. We considered $\mathcal{R}_n$ with n in the range from 10 to 200. For each RSM, we used the standard translation to a PDS [4], and then applied our tool and jMoped to compute a configuration automaton that represents $post^*(\mathcal{L}(\mathcal{C}))$. Figure 4 depicts the obtained speedup, which scales linearly with n. We have also experimented with other similar synthetic RSMs with different means of scaling; and

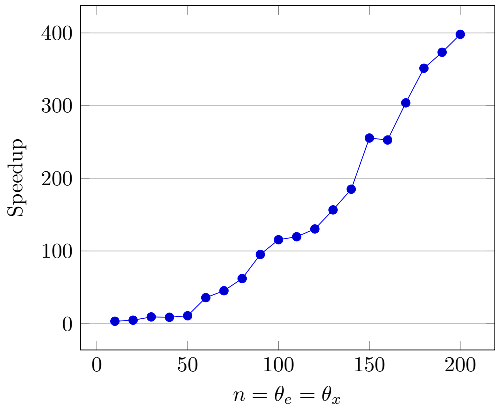

Fig. 4. Speedup of our algorithm over the algorithms of [36,34] implemented by jMoped on the RSM family $\mathcal{R}_n$.

in all cases the obtained speedups have the same qualitative behavior. This confirms the theoretical algorithmic improvements of our algorithm on the synthetic benchmarks.

6.2 Boolean Programs from SLAM/SDV

Benchmarks. For our second experiments we used the collection of Boolean programs distributed as part of the SLAM/SDV project [6,7]. These programs are the final abstractions in the verification of Windows device drivers, and thus they represent RSMs obtained from real-world programs. From the Boolean programs we obtained RSMs where every node represents a control location together with a valuation of Boolean variables, and call/entry and exit/return nodes model the parameter passing between functions. Thus, the RSMs are naturally multi-entry-multi-exit. Overall we obtained 73 RSMs, which correspond to the largest Boolean programs possible to handle explicitly.

Evaluation. To ensure a fair performance comparison, we applied two preprocessing steps to the benchmark RSMs.

– First, to ensure that both tools compute the same result without any potential unnecessary work, we restricted the state space of the RSMs to the interprocedurally reachable states.
– Second, to focus on the performance of interprocedural analysis, we eliminated all internal nodes by computing the intraprocedural transitive closure within every RSM module.

The above two transformations ensure preprocessing steps like removal of unreachable states and intraprocedural analysis is already done, and we compare the interprocedural algorithmic aspects of the algorithms. For each RSM, we used the standard translation to a PDS [4], and then applied our tool and jMoped to compute a configuration automaton that represents $post^*(\mathcal{L}(\mathcal{C}))$, where $\mathcal{C}$ is an initial configuration automaton that contains the entry states of the main module. Table 3 shows for every benchmark the number of RSM transitions (Trans.), their ratio to nodes (D), the runtime for computing the intraprocedural transitive closure (TC), the runtime of jMoped (jMop), the runtime of our tool (Ours), and the speedup our tool achieved over jMoped (SpUp).

Out tool clearly outperforms jMoped on every benchmark, with speedups from 3.94 up to 28.48. The runtimes of our tool range from 0.13 to 33.96 seconds, while the runtimes of jMoped range from 1.03 to 950.82 seconds. Thus, our experiments show that also for real-world examples our algorithm successfully exploits the structure of procedural programs preserved in RSMs. This shows the potential of our algorithm for building program analysis tools.

Note that the benchmark RSMs are quite large, with millions of nodes and transitions, which even a basic implementation of our algorithm handled quite efficiently. Moreover, in our experiments we observed that our tool uses considerably less memory than jMoped. While we set 80GB as the memory limit, the peak memory consumption of jMoped was 72GB, whereas our tool solved all benchmarks with less than 32GB memory.

#	Trans.	D	TC	jMop	Ours	SpUp
1	246,101	1.9	1.18	1.10	0.28	3.94
2	216,021	0.8	0.70	1.03	0.26	3.96
3	593,041	1.5	1.05	2.05	0.49	4.19
4	1,043,217	1.2	3.01	4.67	1.11	4.20
5	329,088	1.4	1.41	1.43	0.34	4.24
6	10,281,149	3.0	11.36	52.00	10.61	4.90
7	908,092	1.7	2.04	3.31	0.65	5.08
8	969,388	2.2	2.00	33.71	6.60	5.11
9	298,126	1.5	0.68	1.31	0.25	5.23
10	1,780,776	1.3	5.82	6.44	1.20	5.35
11	163,853	1.4	0.33	1.03	0.19	5.35
12	205,608	1.0	0.50	4.62	0.86	5.36
13	28,568,561	1.7	23.21	102.54	18.82	5.45
14	21,911,277	1.8	15.79	80.41	14.64	5.49
15	2,453,881	1.5	4.54	9.57	1.72	5.55
16	5,833,574	1.8	6.97	21.14	3.80	5.56
17	332,768	0.8	0.77	2.28	0.41	5.59
18	1,782,697	1.3	5.79	6.70	1.20	5.60
19	246,127	1.9	1.31	1.36	0.24	5.63
20	21,648,560	1.8	15.50	79.45	14.01	5.67
21	7,033,834	2.1	8.23	23.97	4.21	5.70
22	28,944,391	1.7	24.26	105.00	18.15	5.78
23	464,004	1.7	0.75	2.17	0.37	5.83
24	424,916	1.6	1.20	2.94	0.49	5.96
25	22,186,326	1.6	17.77	63.27	10.56	5.99
26	11,719,007	5.2	20.36	52.29	8.55	6.11
27	2,989,001	1.4	3.55	11.04	1.80	6.12
28	1,952,647	1.3	3.83	7.98	1.30	6.13
29	7,970,359	3.2	4.04	30.16	4.70	6.42
30	682,435	2.1	2.14	4.88	0.76	6.42
31	9,480,799	4.9	17.23	44.34	6.77	6.55
32	845,867	2.4	1.59	3.22	0.48	6.67
33	953,420	3.1	1.22	4.51	0.67	6.77
34	1,205,731	2.0	3.31	4.68	0.68	6.84
35	754,270	1.7	4.25	22.28	3.23	6.90
36	1,463,749	2.0	2.38	6.10	0.88	6.95
37	434,884	5.8	6.85	1.90	0.27	7.10
38	14,473,411	1.5	9.68	53.38	7.49	7.13
39	11,616,241	3.3	19.59	42.73	5.54	7.71
40	300,401	2.6	0.74	1.05	0.14	7.79
41	1,916,064	2.3	3.38	10.83	1.39	7.80
42	216,070	1.7	0.56	1.37	0.17	7.83
43	1,293,130	2.3	2.06	5.44	0.69	7.92
44	8,364,920	2.1	6.31	32.95	4.09	8.05
45	18,733,065	4.9	10.84	62.14	7.63	8.15
46	5,373,059	6.4	8.66	18.20	2.17	8.38
47	1,342,348	1.6	4.75	5.02	0.58	8.73
48	779,369	7.2	1.94	6.73	0.77	8.75
49	18,812,123	4.9	8.87	63.86	6.99	9.14
50	40,025,428	6.3	36.49	310.16	33.07	9.38
51	2,503,668	15.3	21.53	10.17	1.08	9.44
52	40,084,249	6.2	36.37	320.70	33.96	9.44
53	4,852,736	6.5	4.14	17.68	1.83	9.64
54	18,520,461	5.4	8.96	60.24	6.21	9.69
55	6,796,783	7.0	9.78	21.33	2.16	9.87
56	40,026,391	6.3	35.69	327.66	33.05	9.91
57	805,305	4.7	1.66	8.14	0.80	10.17
58	4,532,440	26.4	7.49	33.46	3.15	10.61
59	18,374,693	5.8	8.99	60.54	5.52	10.96
60	1,284,096	5.9	1.53	48.54	4.39	11.05
61	3,862,954	6.3	3.44	12.94	1.14	11.38
62	52,269,131	3.4	44.45	177.98	15.53	11.46
63	130,721	2.2	0.43	1.55	0.13	11.52
64	545,063	16.4	6.88	2.27	0.16	13.85
65	545,046	16.4	6.78	2.17	0.15	14.04
66	829,090	12.3	9.60	3.40	0.24	14.17
67	63,918,783	267.0	115.87	244.01	16.00	15.25
68	20,382,912	3.3	15.78	76.69	4.80	15.98
69	29,689,784	6.2	11.18	120.82	7.16	16.88
70	2,619,392	5.2	3.48	660.92	31.62	20.90
71	2,575,360	5.7	3.03	589.87	25.69	22.96
72	2,639,872	5.0	3.17	816.08	29.93	27.27
73	2,691,072	4.5	3.43	950.82	33.39	28.48

Table 3. Comparison of our tool against jMoped. Runtimes are given in seconds. The names of all benchmarks are given in Appendix E.

6.3 Discussion

In our experiments we compared the implementation of our algorithm with jMoped on sequential RSM analysis in an explicit setting. While our algorithm can be made symbolic in a straightforward way (see Appendix B), a symbolic implementation and efficiency for large symbolic domains involve significant engineering efforts. Moreover, the main goal of our work is to compare the algorithmic improvements over the existing approaches, which is best demonstrated in an explicit setting, since in the explicit setting the improvements are algorithmic rather than due to implementation details of symbolic data-structures. Our experimental results show the potential of the new algorithmic ideas, and investigating the applicability of them with a symbolic implementation is a subject of future work.

7 Related Work

Sequential setting. Pushdown systems are very well studied for interprocedural analysis [32,35,10]. While the most basic problem is reachability, the weighted pushdown systems (i.e., pushdown systems enriched with semiring) can express several basic dataflow properties, and other relevant problems in interprocedural program analysis [34,21,33,19]. Hence weighted pushdown systems have been studied in many different contexts, such as [35,32,16,12], and tools have been developed, such as Moped [2], jMoped [1], and WALi [3]. The more convenient model of RSMs was introduced and studied in [4], which on the one hand explicitly models the function calls and returns, and on the other hand specifies many natural parameters for algorithmic analysis. In this work, we improve the fundamental algorithms for RSMs over finite-height semirings, as compared to the bounds obtained by translating RSMs to pushdown systems and applying the best-known bounds for the pushdown case. Along with general RSMs, special cases of SESE RSMs have also been considered, such as RSMs with constant treewidth, and only same context queries [11] (i.e., computation of node distances between nodes of the same module). Our results apply to the general case of all RSMs and are not restricted to any special types of queries.

Concurrent setting. The problem of reachability in concurrent pushdown systems (or concurrent RSMs) is again a fundamental problem in program analysis, which allows for the interprocedural analysis in a concurrent setting. However, the problem is undecidable [31]. Motivated by practical problems, where bugs are discovered with few context switches, the context-bounded reachability problem, where there can be at most k context switches have been considered for concurrent pushdown systems [28,25,26,22,20] as well as related models of asynchronous pushdown networks [9]. We present a new algorithm for concurrent pushdown systems and concurrent RSMs which improves the existing complexity when the size of the global component is small.

8 Conclusion

In this work we consider RSMs, a fundamental model for interprocedural analysis, with path properties expressed over finite-height semirings, that can express a large class of properties for program analysis. We present algorithms that improve the previous algorithms, both in the sequential as well as in the concurrent setting. Moreover, along with our algorithm, we present new methods to extract distances from the data-structure (configuration automata) that the algorithm constructs. We present a prototype implementation for sequential RSMs in an explicit setting that provides significant improvements for real-world programs obtained from SLAM/SDV benchmarks. Our results show the potential of the new algorithmic ideas. There are several interesting directions of future work. A symbolic implementation is a direction for future work. Another direction of future work is to explore the new algorithmic ideas in the concurrent setting in practice.

Acknowledgments. This research was supported in part by the Austrian Science Fund (FWF) under grants S11402-N23, S11407-N23, P23499-N23, and Z211-N23, and by the European Research Council (ERC) under grant 279307.

References

1. jMoped 2.0. `https://www7.in.tum.de/tools/jmoped/`.
2. Moped. `http://www2.informatik.uni-stuttgart.de/fmi/szs/tools/moped/`.
3. WALi. `https://research.cs.wisc.edu/wpis/wpds/`.
4. Rajeev Alur, Michael Benedikt, Kousha Etessami, Patrice Godefroid, Thomas W. Reps, and Mihalis Yannakakis. Analysis of Recursive State Machines. *ACM Trans. Program. Lang. Syst.*, 27(4), 2005.
5. Rajeev Alur, Ahmed Bouajjani, and Javier Esparza. Model Checking Procedural Programs. In *Handbook of Model Checking*. Springer, 2016.
6. Thomas Ball, Ella Bounimova, Vladimir Levin, Rahul Kumar, and Jakob Lichtenberg. The static driver verifier research platform. In *CAV*, 2010.
7. Thomas Ball and Sriram K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN*, 2000.
8. Pamela Bhattacharya, Marios Iliofotou, Iulian Neamtiu, and Michalis Faloutsos. Graph-based analysis and prediction for software evolution. In *ICSE*, 2012.
9. Ahmed Bouajjani, Javier Esparza, Stefan Schwoon, and Jan Strejček. Reachability analysis of multithreaded software with asynchronous communication. In *FSTTCS*, 2005.
10. David Callahan, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Interprocedural constant propagation. In *CC*, 1986.
11. Krishnendu Chatterjee, Rasmus Ibsen-Jensen, Andreas Pavlogiannis, and Prateesh Goyal. Faster algorithms for algebraic path properties in recursive state machines with constant treewidth. In *POPL*, 2015.
12. Swarat Chaudhuri. Subcubic algorithms for recursive state machines. In *POPL*, 2008.
13. P. Cousot and R Cousot. Static determination of dynamic properties of recursive procedures. In *IFIP Conf. on Formal Description of Programming Concepts*, 1977.
14. Robert Giegerich, Ulrich Möncke, and Reinhard Wilhelm. Invariance of approximate semantics with respect to program transformations. In *ECI*, 1981.
15. Dan Grove and Linda Torczon. Interprocedural constant propagation: A study of jump function implementation. In *PLDI*, 1993.
16. Susan Horwitz, Thomas Reps, and Mooly Sagiv. Demand interprocedural dataflow analysis. *SIGSOFT Softw. Eng. Notes*, 1995.
17. Jens Knoop and Bernhard Steffen. The interprocedural coincidence theorem. In *CC*, 1992.
18. Jens Knoop, Bernhard Steffen, and Jürgen Vollmer. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. *ACM Trans. Program. Lang. Syst.*, 1996.
19. Akash Lal and Thomas W. Reps. Solving Multiple Dataflow Queries Using WPDSs. In *SAS*, 2008.
20. Akash Lal and Thomas W. Reps. Reducing concurrent analysis under a context bound to sequential analysis. *Formal Methods in System Design*, 35(1), 2009.
21. Akash Lal, Thomas W. Reps, and Gogul Balakrishnan. Extended weighted pushdown systems. In *CAV*, 2005.

22. Akash Lal, Tayssir Touili, Nicholas Kidd, and Thomas W. Reps. Interprocedural analysis of concurrent programs under a context bound. In *TACAS*, 2008.
23. William Landi and Barbara G. Ryder. Pointer-induced aliasing: A problem classification. In *POPL*, 1991.
24. Edo Liberty and Steven W. Zucker. The mailman algorithm: A note on matrix–vector multiplication. *Inf. Process. Lett.*, 109(3), 2009.
25. Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI*, 2007.
26. Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, 2008.
27. Gene Myers. A four russians algorithm for regular expression pattern matching. *J. ACM*, 39(2), 1992.
28. Shaz Qadeer and Jakob Rehof. Context-Bounded Model Checking of Concurrent Software. In *TACAS*, 2005.
29. Shaz Qadeer and Dinghao Wu. KISS: Keep It Simple and Sequential. In *PLDI*, 2004.
30. Yu Qu, Xiaohong Guan, Qinghua Zheng, Ting Liu, Jianliang Zhou, and Jian Li. Calling network: A new method for modeling software runtime behaviors. *ACM SIGSOFT Software Engineering Notes*, 40(1):1–8, 2015.
31. G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. Program. Lang. Syst.*, 22(2), 2000.
32. Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, 1995.
33. Thomas W. Reps, Akash Lal, and Nicholas Kidd. Program analysis using weighted pushdown systems. In *FSTTCS*, 2007.
34. Thomas W. Reps, Stefan Schwoon, Somesh Jha, and David Melski. Weighted Pushdown Systems and Their Application to Interprocedural Dataflow Analysis. *Sci. Comput. Program.*, 58(1-2), 2005.
35. Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theor. Comput. Sci.*, 1996.
36. Stefan Schwoon. *Model-Checking Pushdown Systems.* Ph.D. Thesis, Technische Universität München, 2002.
37. Dejvuth Suwimonteerabuth, Javier Esparza, and Stefan Schwoon. Symbolic Context-Bounded Analysis of Multithreaded Java Programs. In *SPIN*, 2008.
38. Ryan Williams. Matrix-Vector Multiplication in Sub-Quadratic Time (Some Preprocessing Required). In *SODA*, 2007.
39. Virginia Vassilevska Williams. Multiplying matrices faster than Coppersmith-Winograd. In *STOC*, 2012.

A Proofs of Section 3

Proposition 3. *At all times, every run in the automaton contains at most one transition switching from the fresh mark to an old mark, and no transition switching from an old mark to the fresh mark. Furthermore, every accepting run has to end in a state with an old mark.*

Lemma 1 (Completeness). *For every configuration c we have $\mathcal{C}_{post^*}(c) \sqsubseteq d(\mathcal{L}(\mathcal{C}), c)$.*

Proof. First we show that for every initialized computation $\pi : \langle u, S\rangle \Rightarrow^* \langle u', S'\rangle$ there is a run

$$\lambda : \overbrace{\langle u', m_{u'}\rangle \xrightarrow{\varepsilon} \langle e, m_e\rangle}^{t_1} \xrightarrow{S'}{}^{*} q_f$$

accepting $\langle u', S'\rangle$ such that (1) $\otimes(\lambda) \sqsubseteq \otimes(\pi)$, and (2) t_1 was added to the worklist. We proceed by induction on the length of π. Part (2) of the induction hypothesis is used to argue that t_1 will be extracted with its final weight at some point in the algorithm. We do not explicitly prove this part below since it is an obvious consequence of the steps in the algorithm we refer to in order to prove part (1).

As a base case, if $|\pi| = 0$ then $u = u'$, $S = S'$, and $\otimes(\pi) = \overline{1}$. Since $\langle u, S\rangle \in \mathcal{L}(\mathcal{C})$ there must be a $\mathcal{C}$-run λ accepting $\langle u, S\rangle$, and since all transitions in the initial automaton have weight $\overline{1}$ we have $\otimes(\lambda) = \overline{1}$.

For the induction step, if $|\pi| > 0$ there is a configuration $\langle u_1, S_1\rangle$ such that

$$\pi : \underbrace{\langle u, S\rangle \Rightarrow^* \langle u_1, S_1\rangle}_{\pi_1} \Rightarrow \langle u', S'\rangle.$$

By applying the induction hypothesis to π_1 we obtain an accepting run

$$\lambda_1 : \overbrace{\langle u_1, m_{u_1}\rangle \xrightarrow{\varepsilon} \langle e, m_e\rangle}^{t_1} \underbrace{\!\!\!\xrightarrow{S_1}{}^{*} q_f}_{\lambda_1'}$$

such that

$$\otimes(\lambda_1) = \otimes(\lambda_1') \otimes \ell(t_1) \sqsubseteq \otimes(\pi_1).$$

Let $\mathcal{M}_i$ be the module of u_1. We split cases according to the type of the last transition of π.

1. *Internal transition:* $u' \in In_i$, $\langle u_1, u'\rangle \in \delta_i$, and $S' = S_1$. We consider the iteration of the main loop where t_1 is extracted from WL with its final weight. Line 9 relaxes the transition $t = \langle u', \widehat{m}\rangle \xrightarrow{\varepsilon} \langle e, m_e\rangle$ with $\ell(t_1) \otimes w_i(u_1, u')$ and hence
$$\ell(t) \sqsubseteq \ell(t_1) \otimes w_i(u_1, u').$$

By combining t and λ_1' we obtain the accepting run

$$\lambda : \langle u', \widehat{m}\rangle \xrightarrow{\varepsilon} \langle e, m_e\rangle \xrightarrow{S'}{}^{*} q_f$$

and we derive

$$\begin{aligned}\otimes(\lambda) &= \otimes(\lambda_1') \otimes \ell(t)\\ &\sqsubseteq \otimes(\lambda_1') \otimes \ell(t_1) \otimes w_i(u_1, u')\\ &\sqsubseteq \otimes(\pi_1) \otimes w_i(u_1, u')\\ &= \otimes(\pi).\end{aligned}$$

2. *Call transition:* $u' = e' \in En_{Y_i(b)}$ for some box $b \in B_i$, $\langle u_1, \langle b, e'\rangle\rangle \in \delta_i$, and $S' = bS_1$.
   Again, we consider the iteration of the main loop where t_1 is extracted from WL with its final weight. Line 11 relaxes the transition $t = \langle e', \widehat{m}\rangle \xrightarrow{b} \langle e, m_e\rangle$ with $\ell(t_1) \otimes w_i(u_1, \langle b, e'\rangle)$ and hence

$$\ell(t) \sqsubseteq \ell(t_1) \otimes w_i(u_1, \langle b, e'\rangle).$$

   By combining t and λ_1' we obtain the accepting run

$$\lambda : \langle e', \widehat{m}\rangle \xrightarrow{\varepsilon} \langle e', \widehat{m}\rangle \xrightarrow{b} \langle e, m_e\rangle \xrightarrow{S'}{}^{*} q_f$$

   and we derive

$$\begin{aligned}\otimes(\lambda) &= \otimes(\lambda_1') \otimes \ell(t)\\ &\sqsubseteq \otimes(\lambda_1') \otimes \ell(t_1) \otimes w_i(u_1, \langle b, e'\rangle)\\ &\sqsubseteq \otimes(\pi_1) \otimes w_i(u_1, \langle b, e'\rangle)\\ &= \otimes(\pi).\end{aligned}$$

3. *Return transition:* $u' = \langle b, x\rangle \in R_i$ for some $b \in B_i$ and $x \in Ex_{Y_i(b)}$, $\langle u_1, x\rangle \in \delta_{Y_i(b)}$, and $S_1 = bS'$. First note that

$$\otimes(\pi_1) = \otimes(\pi).$$

   We consider the iteration of the main loop, where $t_{\mathcal{C}} = \langle u_3, m_{u_3}\rangle \xrightarrow{\varepsilon} \langle e, m_e\rangle$ is the transition extracted from WL such that the if-condition $\mathsf{sum}(\langle e, m_e\rangle, x) \not\sqsubseteq \ell(t_{\mathcal{C}})$ in line 14 holds for the last time. Then $\ell(t_{\mathcal{C}}) \sqsubseteq \ell(t_1)$ and since line 15 relaxes $\mathsf{sum}(\langle e, m_e\rangle, x)$ with $\ell(t_{\mathcal{C}})$ we have

$$\mathsf{sum}(\langle e, m_e\rangle, x) \sqsubseteq \ell(t_1).$$

   Now observe that we must have

$$\lambda_1' : \overbrace{\langle e, m_e\rangle \xrightarrow{b/v} \langle e_2, m_{e_2}\rangle}^{t_2} \underbrace{\langle e_2, m_{e_2}\rangle \xrightarrow{S'}{}^{*} q_f}_{\lambda_1''}.$$

We distinguish whether the transition t_2 already had weight v in the current iteration of processing $t_{\mathcal{C}}$ or not. If yes, then line 17 in the current iteration, if no, then line 21 in the later iteration where t_2 is extracted with weight v from WL, relaxes the transition $t = \langle\langle b, x\rangle, \widehat{m}\rangle \xrightarrow{\varepsilon} \langle e_2, m_{e_2}\rangle$ with $v \otimes \mathsf{sum}(\langle e, m_e\rangle, x)$ and hence

$$\ell(t) \sqsubseteq v \otimes \mathsf{sum}(\langle e, m_e\rangle, x).$$

By combining t and λ_1'' we obtain the accepting run

$$\lambda : \langle\langle b, x\rangle, \widehat{m}\rangle \xrightarrow{\varepsilon} \langle e_2, m_{e_2}\rangle \xrightarrow{S'}{}^{*} q_f$$

and we derive

$$\begin{aligned} \otimes(\lambda) &= \otimes(\lambda_1'') \otimes \ell(t) \\ &\sqsubseteq \otimes(\lambda_1'') \otimes v \otimes \mathsf{sum}(\langle e, m_e\rangle, x) \\ &= \otimes(\lambda_1') \otimes \mathsf{sum}(\langle e, m_e\rangle, x) \\ &\sqsubseteq \otimes(\lambda_1') \otimes \ell(t_1) \\ &\sqsubseteq \otimes(\pi_1) \\ &= \otimes(\pi). \end{aligned}$$

In all cases we obtain the desired run λ accepting $\langle u', S'\rangle$ with $\otimes(\lambda) \sqsubseteq \otimes(\pi)$. Now the claim of the lemma follows, as

$$\mathcal{C}_{post^*}(c) = \bigoplus_{\lambda \in \Lambda(c)} \otimes(\lambda) \sqsubseteq \bigoplus_{\pi \in \Pi(\mathcal{L}(\mathcal{C}), c)} \otimes(\pi) = d(\mathcal{L}(\mathcal{C}), c) \qquad (4)$$

were the inequality holds since, as shown above, the weight of any π is bounded from below by the weight of a λ. ☐

Lemma 4 (Soundness invariants). *Algorithm* `ConfDist` *maintains the following loop invariants:*

- I1 *The function* sum *maintains sound summaries, i.e., for every entry $e \in En_i$ and exit $x \in Ex_i$ of the same module $\mathcal{M}_i$, and every box $b \in B_j$ with $Y_j(b) = i$, there exists a set Π of computations $\pi : \langle e, b\rangle \Rightarrow^* \langle\langle b, x\rangle, \varepsilon\rangle$ such that $\bigoplus(\Pi) = \mathsf{sum}(\langle e, \widehat{m}\rangle, x)$.*
- I2 *For every run $\lambda : \langle u_2, \widehat{m}\rangle \xrightarrow{S}{}^{*} \langle u_1, \widehat{m}\rangle$, there exists a set Π of computations $\pi : \langle u_1, \varepsilon\rangle \Rightarrow^* \langle u_2, S\rangle$, such that $\bigoplus(\Pi) = \otimes(\lambda)$.*
- I3 *For every run λ accepting a configuration c there exists a set Π of initialized computations ending in c, such that $\bigoplus(\Pi) = \otimes(\lambda)$.*

Proof. Note that every accepting run has to end in a final state with an old mark since we do not add final states to the automaton. Moreover, we recall that Proposition 3 implies that every run contains at most one transition switching from the fresh mark to an old mark, and no transition switching from an old mark to the fresh mark.

Initially, the invariants hold due to the initialization steps of the algorithm.

Now we need to show that I1 is preserved by updates to the summary function in line 15, and that I2 and I3 are preserved by all possible relaxations performed in line 9, 11, 17, and 21.

Since all cases follow the similar pattern of applying the invariants to subruns and combining the obtained sets of computations suitably, we only give the details of one case for I3.

Consider the run

$$\lambda : \underbrace{\langle u_2, \widehat{m}\rangle \xrightarrow{S_2}{}^{*} \langle e', \widehat{m}\rangle}_{\lambda_2} \overbrace{\xrightarrow{b/v_t} \langle e, m_e\rangle}^{t} \underbrace{\langle e, m_e\rangle \xrightarrow{S_1}{}^{*} q_f}_{\lambda_1}$$

with $m_e \neq \widehat{m}$ and an iteration of the main loop where transition $t_{\mathcal{C}} = \langle u, m_u\rangle \xrightarrow{\varepsilon} \langle e, m_e\rangle$ is extracted from WL and transition t is relaxed in line 11 due to a call transition $t_{\mathcal{R}} = \langle u, \langle b, e'\rangle\rangle$.

We apply I3 to $t_{\mathcal{C}}, \lambda_1$ to obtain a set of initialized computations ending in $\langle u, S_1\rangle$. We extended each computation by $t_{\mathcal{R}}$ to obtain the set Π_1' of initialized computations ending in $\langle e', bS_1\rangle$. We apply I3 to t, λ_1 to obtain a set Π_1'' of initialized computations ending in $\langle e', bS_1\rangle$. We apply I2 to λ_2 to obtain a set of computations from $\langle e', \varepsilon\rangle$ to $\langle u_2, S_2\rangle$. We lift the stack of each computation by bS_1 to obtain the set Π_2 of computations from $\langle e', bS_1\rangle$ to $\langle u_2, S_2 bS_1\rangle$. Let Π be the set of computations obtained by combining every computation in $\Pi_1' \cup \Pi_1''$ with every computation in Π_2. Then Π is the desired set of initialized computations ending in $\langle u_2, S_2 bS_1\rangle$, such that $\bigoplus(\Pi) = \otimes(\lambda)$ after the relaxation of t. □

Lemma 2 (Soundness). *For every configuration c we have $d(\mathcal{L}(\mathcal{C}), c) \sqsubseteq \mathcal{C}_{post^*}(c)$.*

Proof. Conversely to the inequality of equation (4) in the proof of Lemma 1we derive

$$d(\mathcal{L}(\mathcal{C}), c) = \bigoplus_{\pi \in \Pi(\mathcal{L}(\mathcal{C}), c)} \otimes(\pi) \sqsubseteq \bigoplus_{\lambda \in \Lambda(c)} \otimes(\lambda) = \mathcal{C}_{post^*}(c) \tag{5}$$

were the inequality holds since, by invariant I3 from Lemma 4, the weight of any λ is bounded from below by the weight of a $\Pi \subseteq \Pi(\mathcal{L}(\mathcal{C}), c)$. □

Lemma 3 (Complexity). *Let κ be the number of distinct marks $m \in \mathbb{M}$ of the initial automaton $\mathcal{C}$. Algorithm* `ConfDist` *constructs $\mathcal{C}_{post^*}$ in time $O(H \cdot (|\mathcal{R}| \cdot \theta_e \cdot \kappa^2 + |Call| \cdot \theta_e \cdot \theta_x \cdot \kappa^3))$, and $\mathcal{C}_{post^*}$ has $O(|\mathcal{R}| \cdot \theta_e \cdot \kappa^2)$ transitions.*

Proof. We bound the number of times that each loop will be executed.

1. For a given node u, line 5 will be executed at most $H \cdot \theta_e \cdot (\kappa + 1)^2$ times. We denote by $\mathsf{out}_i(u)$, $\mathsf{out}_c(u)$, $\mathsf{out}_x(u)$ the number of internal, call, and exit transitions from node u in δ_i of module $\mathcal{M}_i$. For every iteration of line 4, the following upper bounds on each inner loop are straightforward:

   (a) Line 8: $\mathsf{out}_i(u)$ times.
   (b) Line 10: $\mathsf{out}_c(u)$ times.
   (c) Line 13: $\mathsf{out}_x(u)$ times.

   Hence for a given pair $\langle u, m\rangle$ the algorithm spends $O(H \cdot \theta_e \cdot (\mathsf{out}_i(u) + \mathsf{out}_c(u) + \mathsf{out}_x(u)) \cdot \kappa^2)$ time in the above loops, and summing over all u we obtain $O(H \cdot \theta_e \cdot |\mathcal{R}| \cdot \kappa^2)$ time.
2. Given a pair of a state and an exit $(\langle e, m_e\rangle, x)$, Line 14 will hold true at most H times. Summing over all possible such pairs, we obtain that Line 16 will be executed $O(H \cdot |\mathit{Call}| \cdot \theta_e \cdot \theta_x \cdot \kappa^3)$ times in total.
3. Finally, line 21 will be executed $O(H \cdot |\mathit{Call}| \cdot \theta_e \cdot \theta_x \cdot \kappa^2)$ times in total, since the total number of different edges of the form $\langle e, m_e\rangle \xrightarrow{b} \langle e', m_{e'}\rangle$ added in the worklist is bounded by the number of call nodes that were used in Line 10, times the maximum number of entries and exits in any module of the RSM.

The desired result follows. □

B Symbolic Extensions

Note that in our framework we deal with explicit RSMs, and our `ConfDist` algorithm is also explicit. However, our results carry over to symbolic extensions of RSMs, similar to symbolic PDS [5]. For the symbolic extension we describe the symbolic extension of the model and our algorithm.

– *Symbolic model.* We consider the RSM to represent the control-flow structure of a program (represented explicitly), and the semiring capturing valuations on the variables (represented symbolically). The symbolic semiring operations express the value changes along the program execution. In general the MEME RSMs can represent explicitly a combination of control-flow and some Boolean variables.
– *Symbolic algorithm.* We observe that our algorithm for computation on the semiring uses the basic semiring operations. Hence our algorithm can be straightforwardly made symbolic on the semiring, and is only explicit on the RSM structure.

C Proofs of Section 5

Theorem 7. *For a concurrent RSM $\mathcal{R}^{\|}$, and a bound k, the procedure of [28, Figure 2] using* `ConfDist` *for performing post* operations correctly solves the k-bounded reachability problem and requires $O(|\mathcal{R}^{\|}| \cdot \theta_e^{||} \cdot \theta_x^{||} \cdot n^k \cdot |G|^{k+2})$ time.*

Proof. The algorithm of [28, Figure 3] for concurrent RSMs basically calls algorithm for sequential RSMs as a black-box procedure. Using our algorithm `ConfDist` we obtain an algorithm for concurrent RSM, and the correctness follows from [28] and Theorem 1.

We now sketch the complexity analysis. By Theorem 1, every execution of the algorithm `ConfDist` increases the number of marks of the input configuration automaton by 1. In the i-th iteration of [28, Figure 3] the algorithm will perform a *post** operation on a configuration automaton of i marks, which by Theorem 1 will require $O(|\mathcal{R}| \cdot |G|^2 \cdot \theta_e \cdot \theta_x \cdot i^3)$ time. Each such iteration will spawn $n \cdot |G|$ iterations in the inner loop of [28, Figure 3], one for each component RSM (among n components) and state of the global component (among $|G|$ possible states). Then the total time is (up to constant factors)

$$|\mathcal{R}| \cdot |G|^2 \cdot \theta_e \cdot \theta_x \cdot \sum_{i=1}^{k} i^3 \cdot (n \cdot |G|)^i = O\left(|\mathcal{R}| \cdot |G|^2 \cdot \theta_e \cdot \theta_x \cdot (n \cdot |G|)^k\right)$$

The desired result follows. □

D Comparison with existing work on *k*-bounded reachability

We compare our results for CPDSs, CRSMs, and the related model of asynchronous pushdown networks (APNs).

Comparison for CPDSs. As shown in [28], the k-bounded reachability problem in a CPDS $\mathcal{P}^{\|}$ can be solved in time

$$O(|\mathcal{P}^{\|}|^5 \cdot n^k \cdot |G|^k).$$

The bisimulation relation of [4, Theorem 1] between PDSs and RSMs has a straightforward extension to CPDSs and CRSMs. In particular:

1. Given a CPDS $\mathcal{P}^{\|}$ with n components and global set G, the k-bounded reachability problem for $\mathcal{P}^{\|}$ can be reduced to the k-bounded reachability problem for a CRSM $\mathcal{R}^{\|}$ with n components and global set G. Additionally,

$$|\mathcal{R}^{\|}| = \Theta(|\mathcal{P}^{\|}|); \quad \theta_e^{\|} = \Theta(|\Gamma|) = O(|\mathcal{P}^{\|}|); \quad \theta_x^{\|} = \Theta(1)$$

2. Given a CRSM $\mathcal{R}^{\|}$ with n components and global set G, the k-bounded reachability problem for $\mathcal{R}^{\|}$ can be reduced to the k-bounded reachability problem for a CPDS $\mathcal{P}^{\|}$ with n components and global set G. Additionally,

$$|\mathcal{P}^{\|}| = \Theta(|\mathcal{R}^{\|}| \cdot \theta_x^{\|});$$

It follows that our approach can solve the k-bounded reachability problem of a CPDS $\mathcal{P}^{\|}$ in time

$$O(|\mathcal{P}^{\|}|^2 \cdot n^k \cdot |G|^{k+2}).$$

Note that typically k is very small, (e.g. $k = 2$ in [29], $k = 3$ in [37]). However, in real applications the size of G is typically smaller than $|\mathcal{P}^{\|}|$, e.g., when G encodes only the synchronization variables among threads. Our algorithm gives an improvement by a factor $\Omega(|\mathcal{P}^{\|}|^3/|G|^2)$.

Comparison for CRSMs. The naive upper bound for the k-bounded reachability problem of a CRSM $\mathcal{R}^{\|}$ obtained using a modification of the standard method of [4] to reduce it to the CPDS case, and then apply the algorithm of [28], is $O(|\mathcal{R}^{\|}|^5 \cdot {\theta_x^{||}}^{5} \cdot n^k \cdot |G|^k)$. In contrast, our bound is $O(|\mathcal{R}^{\|}| \cdot \theta_e^{||} \cdot \theta_x^{||} \cdot n^k \cdot |G|^{k+2})$.

Comparison for APNs. The problem of k-bounded reachability has also been studied in the closely related model of *asynchronous pushdown networks (APNs)* [9]. Informally, the main difference of an APN from a CPDS is that in the former case, the stacks have an additional set of local control states, different from the common global finite control G. Hence APNs are more general than CPDS. As shown in [9], the k-bounded reachability problem for an APN $\mathcal{A}^{\|}$ of n components can be solved essentially in time $O(n^k \cdot |G|^k + n \cdot |G|^{k+2} \cdot |\mathcal{A}^{\|}|^2 \cdot |P|)$, where P is the set of local control states. Since APNs are more general than CPDSs, our previous analysis implies that the algorithm of [9] can be used to solve the k-bounded reachability problem for a CRSM $\mathcal{R}^{\|}$ in time $O(n^k \cdot |G|^k + n \cdot |G|^{k+2} \cdot |\mathcal{R}^{\|}|^2 \cdot \theta_x^{||})$. This time is incomparable with what we obtain from Theorem 7. When the number of entries $\theta_e^{||}$ and the number of components n is constant, the algorithm presented in this work has a better complexity.

E Names of the Boolean programs used as benchmarks

All listed benchmarks belong to the "bebop-itp" collection.

1. src_7600_general_toaster_kmdf_filter_generic__InvalidReqAccess
2. src_7600_general_toaster_wdm_filter_devupper__PnpSurpriseRemove
3. src_7600_general_event_wdm__MarkIrpPending2
4. src_7600_storage_sfloppy__PagedCode
5. src_7600_input_kbfiltr_sys__InvalidReqAccess
6. src_7600_general_toaster_wdm_toastmon__PendedCompletedRequest
7. src_7600_general_toaster_wdm_filter_devupper__CriticalRegions
8. src_7600_network_ndis_athwifi_driver_atheros__Irql_SendRcv_Function
9. src_7600_general_event_wdm__IrpProcessingComplete
10. src_7600_general_toaster_wdm_func_featured1__WmiForward
11. src_7600_general_toaster_wdm_func_featured1__IrqlKeWaitForSingleObject
12. src_7600_network_ndis_athwifi_driver_atheros__Irql_Timer_Function
13. src_7600_general_toaster_wdm_func_featured1__IrqlIoPassive3
14. src_7600_general_toaster_wdm_func_featured2__IrqlIoApcLte
15. src_7600_general_toaster_wdm_func_featured2__WmiForward
16. src_7600_general_ioctl_kmdf_sys__InitFreeDeviceCreateType4
17. src_7600_general_toaster_wdm_func_incomplete2__IrqlReturn
18. src_7600_general_pcidrv_wdm_hw__WmiForward
19. src_7600_input_moufiltr__InvalidReqAccess
20. src_7600_general_toaster_wdm_func_featured2__IrqlIoPassive3
21. src_7600_general_toaster_kmdf_filter_sideband__ControlDeviceInitAPI
22. src_7600_general_toaster_wdm_func_featured1__IrqlIoApcLte
23. src_7600_general_toaster_wdm_func_featured2__PnpSurpriseRemove
24. src_7600_general_amcc5933_sys__KmdfIrql
25. src_7600_general_toaster_wdm_func_incomplete1__IrpProcessingComplete

26. src_7600_general_toaster_wdm_filter_devupper__PendedCompletedRequest
27. src_7600_general_toaster_wdm_func_incomplete1__TargetRelationNeedsRef
28. src_7600_general_toaster_wdm_func_incomplete2__TargetRelationNeedsRef
29. src_7600_general_toaster_wdm_func_featured1__PnpSurpriseRemove
30. src_7600_bth_bthecho_bthcli_sys__RequestFormattedValid
31. src_7600_general_toaster_wdm_filter_buslower__PendedCompletedRequest
32. src_7600_input_hiddigi_wacompen__MarkIrpPending
33. src_7600_general_toaster_wdm_bus__PnpSurpriseRemove
34. src_7600_general_toaster_kmdf_bus_static__PdoInitFreeDeviceCreateType4
35. src_7600_network_ndis_athwifi_driver_atheros__Irql_IrqlSetting_Function
36. src_7600_serial_serenum__IrqlIoApcLte
37. src_7600_hid_hidusbfx2_sys__SyncReqSend2
38. src_7600_general_toaster_wdm_toastmon__IrpProcessingComplete
39. src_7600_general_toaster_kmdf_filter_sideband__ControlDeviceDeleted
40. src_7600_general_cancel_startio__MarkIrpPending2
41. src_7600_general_toaster_wdm_func_featured2__IrqlKeSetEvent
42. src_7600_general_toaster_wdm_func_incomplete2__IrqlKeSetEvent
43. src_7600_serial_serenum__IrqlIoPassive3
44. src_7600_general_toaster_wdm_filter_devlower__IrpProcessingComplete
45. src_7600_general_toaster_wdm_func_featured1__IrqlExAllocatePool
46. src_7600_general_toaster_wdm_func_featured2__CriticalRegions
47. src_7600_general_toaster_wdm_bus__MarkIrpPending2
48. src_7600_storage_class_cdrom__WdfSpinlockRelease
49. src_7600_general_toaster_wdm_func_featured2__IrqlKeWaitForSingleObject
50. src_7600_network_ndis_xframeii_sys_ndis6__MandatoryOid
51. src_7600_general_toaster_kmdf_filter_sideband__DeviceInitAllocate
52. src_7600_network_ndis_xframeii_sys_ndis6__NdisStallExecution_Delay
53. src_7600_bth_bthecho_bthsrv_sys__InvalidReqAccessLocal
54. src_7600_general_toaster_wdm_func_featured2__IrqlKeApcLte2
55. src_7600_general_toaster_wdm_bus__IrqlIoPassive3
56. src_7600_network_ndis_xframeii_sys_ndis6__SpinlockRelease
57. src_7600_storage_filters_diskperf__PnpIrpCompletion
58. src_7600_storage_filters_diskperf__TargetRelationNeedsRef
59. src_7600_general_toaster_wdm_func_featured2__IrqlZwPassive
60. src_7600_hid_hidusbfx2_hidmapper__ForwardedAtBadIrql
61. src_7600_general_pcidrv_wdm_hw__DoubleCompletion
62. src_7600_serial_serenum__MarkIrpPending2
63. src_7600_general_toaster_wdm_func_incomplete2__MarkIrpPending2
64. src_7600_input_moufiltr__SyncReqSend2
65. src_7600_general_toaster_kmdf_filter_generic__SyncReqSend2
66. src_7600_input_kbfiltr_sys__SyncReqSend2
67. src_7600_input_hiddigi_wacompen__SpinLock
68. src_7600_general_toaster_wdm_filter_devupper__IrpProcessingComplete
69. src_7600_general_pcidrv_wdm_hw__IrqlIoPassive1
70. src_7600_general_toaster_wdm_func_incomplete1__ForwardedAtBadIrql

71. src_7600_general_toaster_wdm_filter_devlower__ForwardedAtBadIrql
72. src_7600_general_toaster_wdm_toastmon__ForwardedAtBadIrql
73. src_7600_general_toaster_wdm_filter_devupper__ForwardedAtBadIrql