Computer Sciences Department

Advanced Querying for Property Checking

Nicholas Kidd
Akash Lal
Thomas Reps

Technical Report #1621

November 2007

UNIVERSITY OF WISCONSIN MADISON

Advanced Querying for Property Checking *

Nicholas Kidd[1], Akash Lal[1**], and Thomas Reps[1,2]

[1] University of Wisconsin; Madison, WI; USA. {kidd, akash, reps}@cs.wisc.edu
[2] GrammaTech, Inc.; Ithaca, NY; USA.

Abstract. Extended weighted pushdown systems (EWPDSs) are an extension of pushdown systems that incorporate infinite-state data abstractions. Nested-word automata (NWAs) are able to recognize languages that exhibit context-free properties, while retaining many of the decidability properties of finite automata. We study property checking of programs where the program model is an EWPDS and the property is specified by an NWA. We show how to combine an NWA A with an EWPDS $\mathcal{E}$ to create an EWPDS $\mathcal{E}_A$ such that reachability analysis on $\mathcal{E}_A$ checks property A on program $\mathcal{E}$. This construction allows us to retain the capability of running advanced queries on programs modeled as EWPDSs, such as the ability to (i) find all program nodes that lie on an error path (via error projections); and (ii) answer context-bounded reachability queries for concurrent programs with infinite-state abstractions (via context-bounded model checking).

1 Introduction

The goal of property checking is to verify that all possible executions of a program adhere to a property specification. Because the precise answer to this problem is undecidable in general, software model checkers instead attempt to determine if a model of the program satisfies the property specification. This technique is sound so long as the set of behaviors of the model is an over-approximation of the set of actual behaviors of the program.

Different families of models can be used. One such model is a Boolean program. A Boolean program is a program whose variables all have the Boolean datatype—there are no pointers or heap allocated storage. Both BLAST and SLAM use predicate abstraction [1] to model a `C` program as a Boolean program. A Boolean program is typically interpreted as a pushdown system (PDS). PDSs are a natural choice because they are able to accurately model the program's runtime stack, and because the set of reachable program configurations of a PDS can be computed in polynomial time and space.

Another modeling formalism is affine programs [2]. Affine programs are similar to Boolean programs, but with integer-valued variables. The common use of the affine-program model is to perform affine-relation analysis, where an affine relation is a linear-equality constraint between integer-valued variables. The goal

* Supported by NSF under grants CCF-0540955 and CCF-0524051.
** Supported by a Microsoft Research Fellowship.

of affine-relation analysis is to find all of the affine relations that hold in the program. Because affine-relation analysis requires an infinite-state abstraction, PDSs cannot be used to represent an affine program. Weighted pushdown systems (WPDSs) [3,4] address this issue by adding a *weight* to each rule of the PDS transition system, where weights can describe an infinite set of data points. Extended WPDSs (EWPDSs) further augment WPDSs by providing a generic framework that permits program models to track the values of local variables.

The ability to reason about local variables—or distinguish between global and local state—is also of interest for property checking. Software model checkers have traditionally required that the property specification be defined as a finite automaton. This requirement implies that only a limited amount of global and local state can be distinguished. In the presence of recursion, where the stack is unbounded, properties specified by a finite automaton are not able to make any such distinction.

Alur and Madhusudan address the expressibility problem via nested-word automata (NWAs) [5]. Like EWPDSs, the NWA formalism provides a generic framework that enables distinguishing between global and local state.

We study property checking where the program is modeled as an EWPDS and the property is specified by an NWA. The result of our study is the synthesis of several recent threads of research by Alur et al. [5,6] and ourselves [7,8,9]. Compared to standard approaches to property checking, we show how to simultaneously check properties

1. stated in a more expressive specification language
2. on program models that support more powerful abstractions
3. while furnishing a broader range of diagnostic information when property violations are detected.

This is achieved in polynomial time and space for (possibly recursive) sequential programs, and can be used in a semi-decision procedure for (possibly recursive) concurrent shared-memory programs [9]. Heretofore it was only known how to achieve items 2 and 3 simultaneously.

The key is combining an NWA A with an EWPDS $\mathcal{E}$ to form an EWPDS $\mathcal{E}_A$ such that reachability analysis on $\mathcal{E}_A$ is able to check property A on $\mathcal{E}$. Besides more powerful abstractions, an important benefit of modeling the combination as an EWPDS is that we get for free the ability to answer advanced diagnosis queries, such as:

A. "What abstract state can the program be in when it violates the property specification?"
B. "What are the set of program nodes that lie on an error path, and what abstract states can arise at those nodes on an error path?"

Query A is the direct result of a reachability query on $\mathcal{E}_A$. Query B is answered by computing the *error projection* [7] of $\mathcal{E}_A$.

Briefly, error projections provide (for certain families of abstractions) a way to achieve simultaneous forwards and backwards analysis. With respect to property

checking, error projections divide the program statements (or locations) into two sets: those that potentially lie on an error path to a specified error configuration, and those that definitely do not. In addition to this, *abstract error projections* [7] provide the ability to compute the abstract memory configuration of all error paths from program entry to a program statement, and hence answer Query B.

This paper makes the following contributions:

- We present two constructions for combining an NWA with a PDS, and one for combining an NWA with an EWPDS.
- We formally define the nested-word language of a PDS and an EWPDS.
- For each construction, we present a safety query that can be used to verify that the combined NWA plus (EW)PDS adheres to the property specified by the NWA. In addition, we show how to solve this query using EWPDS reachability.
- Because the constructions yield a PDS or EWPDS, they automatically provide a means to determine the error projection of a program with respect to a property defined by an NWA. Similarly, the constructions automatically provide for context-bounded model checking of concurrent programs, where the property specification is in the form of an NWA.

The remainder of the paper is organized as follows: §2 presents a property checking example that is expressible as an NWA but not an NFA. §3 and §4 present background material on NWAs and PDSs, respectively. §5 defines the notion of a nested-word language for a PDS. §6 presents a construction that leverages PDS reachability to perform property checking. §7 presents definitions and notations for EWPDSs. §8 presents two constructions that leverage EWPDS reachability to perform property checking. §9 discusses related work.

2 Verification Example

Suppose that Eve is developing an interactive graphical software application, and that there are certain interactive requirements that the program must adhere to (e.g., a minimum frame rate of 30 fps must be maintained). Furthermore, let us assume that Eve knows that when certain timing-sensitive functions are on the stack,[3] a subsequent call to a potentially blocking function (e.g., one that performs I/O) should not be performed. Because of the size and complexity of the code base, Eve has decided to employ a software model checker to verify that the "blocking" discipline is followed. That is, for a given "sensitive" function f, Eve asks the model checker to verify that when f is on the stack, the program does not invoke a (potentially) blocking function b.

The property that Eve desires must distinguish between program states when f is on the stack and when it is not. A first attempt would be to model the property as a state machine that consists of the states YES, NO, and ERR; which signify that f is on the stack, not on the stack, and that b was called when

[3] A function that is on the stack is one that has been invoked but not yet completed.

f was on the stack, respectively. This technique works as long as the function f is not recursive. When f is recursive, it is not possible to distinguish between returning to a recursive context and returning to a non-recursive context. To make this distinction, one requires the ability to inspect the state of the caller to properly define the property of interest. This can be accomplished by using a nested-word automaton.

3 Nested Words

A nested word nw is a pair (w, v), where w is a word $a_1 \dots a_k$ over a finite alphabet and v, the *nesting relation*, is a binary relation over the length of the word w. Formally, v is a subset of $\{1, 2, \dots, k\} \times (\{1, 2, \dots, k\} \cup \{\infty\})$ [5]. The nesting relation denotes a set of *properly nested* hierarchical edges of the nested word nw. For a valid nesting relation, $v(i, j)$ implies $i < j$, and if both $v(i, j)$ and $v(i', j')$ hold and $i < i'$, then either $j < i'$ or $j' < j$. For a *jump edge* $v(i, j)$, Alur et al. refer to i as the *call predecessor* for the return position j, and j as the *return successor* for the call position i.

Nested words are a natural model for describing a trace of program execution. The nesting relation v defines the matched calls and returns that arise during the trace. One can view a program as a nested-word generator, and the set of all program traces (i.e., the set of generated nested words) defines the *nested-word language* (NWL) of the program.

A *regular* NWL is an NWL that can be modeled by a *nested-word automaton* (NWA) [5]. An NWA A is a tuple $(Q, \Sigma, q_0, \delta, F)$, where Q is a finite set of states, Σ is a finite alphabet, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is a set of final states, and δ is a transition relation. The transition relation δ consists of three components, $(\delta_c, \delta_i, \delta_r)$, where:

- $\delta_c \subseteq Q \times \Sigma \times Q$ defines the transition relation for call positions.
- $\delta_i \subseteq Q \times \Sigma \times Q$ defines the transition relation for internal positions.
- $\delta_r \subseteq Q \times Q \times \Sigma \times Q$ defines the transition relation for return positions.

Starting from q_0, an NWA A reads a nested word $nw = (w, v)$ from left to right, and performs transitions (possibly non-deterministically) according to the input symbol and the nesting relation. That is, if A is in state q when reading input symbol σ at position i in w, then if i is an internal or call position, A makes a transition to q' using $(q, \sigma, q') \in \delta_i$ or $(q, \sigma, q') \in \delta_c$, respectively. Otherwise, i is a return position. Let k be the call predecessor of i, and q_c be the state A was in just before the transition it made on the k^{th} symbol; then A uses $(q, q_c, \sigma, q') \in \delta_r$ to make a transition to q'. If, after reading nw, A is in a state $q \in F$, then A accepts nw [5].

We use $L(A)$ to denote the nested-word language that A accepts, and $L(A, q)$ to denote the nested-word language such that for each nested word $nw \in L(A, q)$, A is left in state q after reading nw. We extend this notion to sets of states in the obvious way. Thus, $L(A) = L(A, F)$.

Verification Example Revisited: Returning to our example, Eve would like to verify that the potentially blocking function b is never invoked when the timing-sensitive function f is on the stack. Let us assume that the property specification can refer to the entry point of a function. For example, functions f and b have entry points f_{enter} and b_{enter}, respectively. Additionally, we use $\boxtimes$ to signify a program state where b cannot be called (i.e., f is on the stack), and $\boxdot$ to signify a program state where b can be called (i.e., f is *not* on the stack). Eve can specify the desired propery via the NWA $A = (Q, \Sigma, q_0, \delta, F)$, where $Q = \{\boxtimes, \boxdot, err\}$, Σ is the control locations of the program, $q_0 = \boxdot$, $F = \{err\}$, and δ is defined in Tab. 1 (note that we use $q \in Q$ and $\sigma \in (\Sigma - \{f_{\text{enter}}, b_{\text{enter}})\}$ as wildcards).

δ_c	δ_r	δ_i
$(q, f_{\text{enter}}, \boxtimes)$	$(q, \boxtimes, \sigma, \boxtimes)$	(q, σ, q)
$(\boxtimes, b_{\text{enter}}, err)$	$(q, \boxdot, \sigma, \boxdot)$	
$(\boxdot, b_{\text{enter}}, \boxdot)$		
(q, σ, q)		

Table 1. NWA for "blocking" discipline.

4 Pushdown Systems

This section presents definitions and notation for pushdown systems and reachability queries on pushdown systems. Readers familiar with this material are encouraged to skip to §5.

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

Without loss of generality, we restrict the pushdown rules to have at most two stack symbols on the right-hand side [10]. A rule $r = \langle p, \gamma \rangle \hookrightarrow \langle p', u \rangle$, $u \in \Gamma^*$, is called a *push*, *step*, or *pop* rule if $|u| = 2$, $|u| = 1$, or $|u| = 0$, respectively. We use $\Delta_i \subseteq \Delta$ to denote the set of all rules with i stack symbols on the right-hand side.

A *run* of $\mathcal{P}$ from a configuration c is a rule sequence $\rho = [r_1, \ldots, r_j]$ that transforms c into a configuration c'. We denote the transformation by $c \Rightarrow^\rho c'$, and the set of all runs of $\mathcal{P}$ from c by $Runs(\mathcal{P}, c)$, or simply $Runs(\mathcal{P})$ if c is implied from the context.

A PDS naturally models a program's control flow. The standard approach is as follows: P contains a single state p, Γ corresponds to the nodes of the program's inter-procedural control flow graph (ICFG), and Δ corresponds to

Rule	Control flow modeled
$\langle p, n_1\rangle \hookrightarrow \langle p, n_2\rangle$	Intraprocedural edge $n_2 \to n_2$
$\langle p, n_c\rangle \hookrightarrow \langle p, e_f\ r_c\rangle$	Call to f from n_c that returns to r_c
$\langle p, x_f\rangle \hookrightarrow \langle p, \varepsilon\rangle$	Return from f at exit node x_f

Fig. 1. The encoding of an ICFG's edges as PDS rules.

edges of the program's ICFG (see Fig. 1). Let us denote the entry point of a program's main function by n_{main}. The set of all valid paths of the program is defined by: $\mathit{Runs}(\mathcal{P}, \langle p, n_{\text{main}}\rangle)$.

PDS reachability is a fundamental primitive that is useful for performing property checking on program models. The goal is to find the set of all reachable configurations C' when starting from a given set of configurations C. Because the number of configurations of a PDS is unbounded, it is useful to use finite automata to describe regular sets of configurations.

Definition 2. *If $\mathcal{P} = (P, \Gamma, \Delta)$ is a PDS then a* **$\mathcal{P}$-automaton** *is a finite automaton $(Q, \Gamma, \to, P, F)$, where $Q \supseteq P$ is a finite set of states, $\to \subseteq Q \times \Gamma \times Q$ is the transition relation, P is the set of initial states, and F is the set of final states. We say that a configuration $\langle p, u\rangle$ is accepted by a $\mathcal{P}$-automaton if the automaton can accept u when it is started in the state p (written as $p \xrightarrow{u}{}^{*} q$, where $q \in F$). A set of configurations is called* ***regular*** *if some $\mathcal{P}$-automaton accepts it. Without loss of generality, $\mathcal{P}$-automata are restricted to not have any transitions leading to an initial state.*

An important result is that for a regular set of configurations C, both $\mathit{post}^*(C)$ and $\mathit{pre}^*(C)$ (the forward and the backward reachable sets of configurations, respectively) are also regular sets of configurations [11,12]. The algorithms for computing post^* and pre^*, called *poststar* and *prestar*, respectively, take a $\mathcal{P}$-automaton $\mathcal{A}$ as input, and if C is the set of configurations accepted by $\mathcal{A}$, they produce $\mathcal{P}$-automata $\mathcal{A}_{\mathit{post}^*}$ and $\mathcal{A}_{\mathit{pre}^*}$ that accept the sets of configurations $\mathit{post}^*(C)$ and $\mathit{pre}^*(C)$, respectively [11,13,14].

5 The Nested-Word Language of a PDS

Our goal is to perform property checking where the program is modeled by a PDS and the property is specified by an NWA. To achieve our desired result, we must be able to formally reason about the nested-word language (NWL) of a PDS. Intuitively, for a PDS $\mathcal{P} = (P, \Gamma, \Delta)$, if (w, v) is a nested word in the NWL for $\mathcal{P}$, w consists of the sequence of left-hand-side stack symbols $\gamma_1 \ldots \gamma_j$ for a run $[r_1, \ldots, r_j]$ of $\mathcal{P}$ from some configuration. (To simplify the notation, and because for property checking one is usually interested in runs that start from a distinguished main function and empty stack, we assume that associated with each PDS is an initial configuration, and all runs of a PDS begin from this configuration.)

For reasons that will be clarified in the next section, we slightly augment our definition of a PDS so that the stack alphabet Γ is composed of two sets, Γ^α and Γ^β, such that $\Gamma^\alpha \cap \Gamma^\beta = \emptyset$. The two sets distinguish between "actual" and "bookkeeping" stack symbols. If left unspecified, $\Gamma^\beta = \emptyset$. Additionally, a bookkeeping symbol can only appear on the left-hand side of a step rule.

For a nested word $nw = (w, v)$ and rule $r \in \Delta$, we define the function $post[r]((w, v))$ as follows:

$$post[r]((w,v)) = \begin{cases} (w\gamma, v) & \text{if } r = \langle p,\gamma\rangle \hookrightarrow \langle p',\gamma'\rangle, \gamma \in \Gamma^\alpha \\ (w, v) & \text{if } r = \langle p,\gamma\rangle \hookrightarrow \langle p',\gamma'\rangle, \gamma \in \Gamma^\beta \\ (w\gamma, (v - \{\langle i, \infty\rangle\}) \cup \{\langle i, |w\gamma|\rangle\}) & \text{if } r = \langle p,\gamma\rangle \hookrightarrow \langle p',\epsilon\rangle, \\ & \quad i = max(\{j \mid \langle j,\infty\rangle \in v\}) \\ (w\gamma, v \cup \{\langle |w\gamma|, \infty\rangle\}) & \text{if } r = \langle p,\gamma\rangle \hookrightarrow \langle p',\gamma'\ \gamma''\rangle \\ (w, v) & \text{if } r = \epsilon \end{cases}$$

Using $post[r]$, we define the function $NWofRun([r_1 \dots r_j])$[4] as follows:

$$\begin{aligned} NWofRun([]) &= (\epsilon, \emptyset) \\ NWofRun([r_1, \dots, r_j]) &= post[r_j](NWofRun([r_1, \dots, r_{j-1}])) \end{aligned}$$

Definition 3. *The* **effective length** *of a run $\rho = [r_1, \dots, r_j]$ is equal to the length of the word component w of the nested word $nw = (w, v) = NWofRun(\rho)$. For a run ρ, we denote its effective length by $EffLen(\rho)$.*

Note that for a run $\rho = [r_1, \dots, r_j]$, $EffLen(\rho)$ is equal to the number of rules in ρ whose left-hand-side stack symbol is in Γ^α. This follows from the definition of *NWofRun*.

Definition 4. *For a PDS $\mathcal{P}$, the NWL $L(\mathcal{P}) = \{NWofRun(\rho) \mid \rho \in Runs(\mathcal{P})\}$.*

The set of all nested words that drive $\mathcal{P}$ to some state $p \in P$ is denoted by $L(\mathcal{P}, p)$. We extend this notion to sets of states in the obvious way. Thus, $L(\mathcal{P}) = L(\mathcal{P}, P)$.

6 Property Checking via PDS Reachability

We now present our first construction, *Explicit NWA plus PDS*, which allows PDS reachability to be used to perform property checking. The construction can be viewed as a recasting of Alur and Madhusudan's formulation of model checking using NWAs [5] in our vocabulary. It is presented because it provides insight into the more interesting construction *Symbolic NWA plus PDS* presented in §8.

[4] $post[r](nw)$ is not always defined because of *max*; and thus neither is *NWofRun*. However, for a run of a PDS from the initial configuration, both will always be defined.

Construction 1 *[Explicit NWA plus PDS].* A PDS $\mathcal{P}$ and an NWA A are combined to form a new PDS $\mathcal{P}_A = (P_A, \Gamma_A, \Delta_A)$, where $P_A \subseteq (P \times Q) \cup (P \times (Q \times \Gamma))$, $\Gamma_A = \Gamma_A^\alpha \cup \Gamma_A^\beta$, $\Gamma_A^\alpha = \Gamma$, $\Gamma_A^\beta = \Gamma \times Q$, and Δ_A is defined by the constructor $\kappa : \Delta \times \delta \to 2^{\Delta_A}$ as follows:

1. For $r = \langle p, n_1 \rangle \hookrightarrow \langle p', n_2 \rangle \in \Delta_1$ and $t = (q, n_1, q') \in \delta_i$, $\kappa(r, t) = \langle (p, q), n_1 \rangle \hookrightarrow \langle (p', q'), n_2 \rangle$.
2. For $r = \langle p, n_c \rangle \hookrightarrow \langle p', e\ r_c \rangle \in \Delta_2$ and $t = (q_c, n_c, q) \in \delta_c$, $\kappa(r, t) = \langle (p, q_c), n_c \rangle \hookrightarrow \langle (p', q), e\ (r_c, q_c) \rangle$, where $(r_c, q_c) \in \Gamma^\beta$.
3. For $r = \langle p, x \rangle \hookrightarrow \langle p', \epsilon \rangle \in \Delta_0$ and $t = (q_r, q_c, x, q) \in \delta_r$, $\kappa(r, t) = \langle (p, q_r), x \rangle \hookrightarrow \langle (p', q_r^x), \epsilon \rangle$. In addition, $\kappa(r, t)$ constructs a set of rules of the form $\langle (p', q_r^x), (r_c, q_c) \rangle \hookrightarrow \langle (p', q), r_c \rangle$ for each return point r_c that occurs in Δ_2.

Let $\langle p_0, n_{\text{main}} \rangle$ be the initial configuration for $\mathcal{P}$; then the initial configuration for $\mathcal{P}_A$ is $\langle (p_0, q_0), n_{\text{main}} \rangle$. Notice the initial state q_0 of A is paired with the state of the initial configuration of $\mathcal{P}$.

It is worth explaining some parts of the construction. First, for composed call rules (item 2), the return stack symbol (r_c, q_c), which is a bookkeeping symbol, encodes both the return point of the PDS rule, r_c, as well as the NWA state of the caller, q_c. This is required so that when modeling δ_r, the state that held at the call is available (see item 3). This is why we split the stack alphabet Γ into Γ^α and Γ^β; i.e., to have the ability to distinguish between the stack alphabet of $\mathcal{P}$ and the "bookkeeping" symbols introduced to model A.

Second, notice that combining a pop rule $r \in \Delta_0$ with a return transition $t \in \delta_r$ generates a set of PDS rules (item 3). The generated-pop rule pops the top of the stack and records, via the control location (p', q_r^x), that when making the pop, A was in state q_r and the top of stack was x. The generated-step rules transition $\mathcal{P}_A$ to the correct return point of $\mathcal{P}$. In doing so, they use the state (p', q_r^x), as well as the return transition t to ensure that the NWA is properly modeled. In some sense, one can view the pop rule as modeling a multi-state return function of the callee. The caller then uses the (appropriate) step rule to coalesce the multiple return points into the single return point encoded in $\mathcal{P}$.

Theorem 1. *An NWA A combined with a PDS $\mathcal{P}$ results in a new PDS $\mathcal{P}_A$ such that $L(\mathcal{P}_A, (p', q')) = L(\mathcal{P}, p') \cap L(A, q')$, for any $p' \in P$ and $q' \in Q$.*

Proof. The proof is organized as follows:

1. $L(\mathcal{P}_A, (p', q')) \subseteq L(\mathcal{P}, p')$ by Lem. 3;
2. $L(\mathcal{P}_A, (p', q')) \subseteq L(A, q')$ by Lem. 4; and
3. $L(\mathcal{P}_A, (p', q')) \supseteq L(\mathcal{P}, p') \cap L(A, q')$ by Lem. 5.

□

Corollary 1. $L(\mathcal{P}_A, P \times Q) = L(\mathcal{P}, P) \cap L(A, Q)$

Proof.

$$\begin{aligned} L(\mathcal{P}, P) \cap L(A, Q) &= \bigcup_{p \in P} L(\mathcal{P}, p) \cap \bigcup_{q \in Q} L(A, q) \\ &= \bigcup_{p \in P, q \in Q} L(\mathcal{P}, p) \cap L(A, q) \\ &= L(\mathcal{P}_A, P \times Q) \end{aligned}$$

Notice that the last step above follows from Thm. 1. □

When discussing the nested-word languages for $\mathcal{P}$ and $\mathcal{P}_A$, we will always be starting from configuration $\langle p_0, n_{\text{main}} \rangle$ and $\langle (p_0, q_0), n_{\text{main}} \rangle$, respectively. We first present the proofs for Lemmas 1 and 2. Lemmas 1 and 2 are helper lemmas that aid in proving Lemmas 3, 4, and 5.

Lemma 1. *For a run $[r_1, \ldots, r_j]$ of $\mathcal{P}_A$ that generates a nested word $nw \in L(\mathcal{P}_A, (p', q'))$, the rule r_j is either a step rule or a push rule from Δ_A, but not a pop rule.*

Proof. For each nested word nw in $L(\mathcal{P}_A, (p', q'))$, there exists a run ρ of $\mathcal{P}_A$ such that $nw = NWofRun(\rho)$. From the definition of $L(\mathcal{P}_A, (p', q'))$, starting from the configuration $\langle p_0, n_{\text{main}} \rangle$ and making a transition for each rule $r \in \rho$, the result is a configuration of the form $\langle (p', q'), u \rangle$,$u \in \Gamma^*, p' \in P, q' \in Q$. By definition, all pop rules in Δ_A cause a configuration of the form $\langle (p, q_r), x\, u \rangle$ to make a transition to a configuration of the form $\langle (p, q_r^x), u \rangle$. Because the state q_r^x is not in Q, r_j must be either a step or push rule. □

Lemma 2. *For a nested word $nw = (w, v) \in L(\mathcal{P}, p') \cap L(A, q')$, the length j of the run $\rho = [r_1, \ldots, r_j]$ of $\mathcal{P}$ that generates nw is equal to $|w|$.*

Proof. From the definition of $post[r]$, a rule appends its left-hand-side stack symbol γ only if $\gamma \in \Gamma^\alpha$. For $\mathcal{P}$, $\Gamma^\beta = \emptyset$, and $NWofRun([r_1, \ldots, r_j])$ will generate a nested word $nw = (w, v)$ such that $|w| = j$. □

Lemma 3. $L(\mathcal{P}_A, (p', q')) \subseteq L(\mathcal{P}, p')$.

Proof. We must show that $\mathcal{P}_A$ is a subset of $\mathcal{P}$; i.e., that *Construction 1* did not introduce behaviors or runs that were not originally a part of $\mathcal{P}$. The intuition on which the proof is based is that *Construction 1* produces a PDS whose set of runs is a restriction of the set of runs of $\mathcal{P}$. Thus, we prove Lem. 3 by providing a function that maps each run ρ_A of $\mathcal{P}_A$ to a run ρ of $\mathcal{P}$ such that $NWofRun(\rho_A) = NWofRun(\rho)$. The proof makes use of the deconstructor $\kappa_\Delta^{-1} : \Delta_A \to \Delta$, defined as follows:

$$\kappa_\Delta^{-1}(r) = \begin{cases} \langle p, n_1 \rangle \hookrightarrow \langle p', n_2 \rangle & \text{if } r = \langle (p, q), n_1 \rangle \hookrightarrow \langle (p', q'), n_2 \rangle \\ \langle p, n_c \rangle \hookrightarrow \langle p', e\ r_c \rangle & \text{if } r = \langle (p, q_c), n_c \rangle \hookrightarrow \langle (p', q), e\ (r_c, q_c) \rangle \\ \langle p, x \rangle \hookrightarrow \langle p', \epsilon \rangle & \text{if } r = \langle (p, q_r), x \rangle \hookrightarrow \langle (p', q_r^x), \epsilon \rangle \\ \epsilon & \text{if } r = \langle (p, q_r^x), (r_c, q_c) \rangle \hookrightarrow \langle (p', q), r_c \rangle \end{cases}$$

We extend the function $\kappa_{\Delta}^{-1}(r)$ to work on a run as follows:

$$\begin{aligned}\kappa_{\Delta}^{-1}([]) &= []\\ \kappa_{\Delta}^{-1}([r_1,\ldots,r_j]) &= \kappa_{\Delta}^{-1}(r_1) :: \kappa_{\Delta}^{-1}([r_2,\ldots,r_j])\end{aligned}$$

For a rule $r \in \Delta_A$ and nested word $nw = (w, v)$, $post[r](nw) = post[\kappa_{\Delta}^{-1}(r)](nw)$. We show this by a case analysis on the form of the rule r.

1. $r = \langle (p,q), n_1\rangle \hookrightarrow \langle (p',q'), n_2\rangle$, and $n_1 \in \Gamma_A^{\alpha}$. By definition, $\kappa_{\Delta}^{-1}(r) = \langle p, n_1\rangle \hookrightarrow \langle p', n_2\rangle$. Both $post[r](nw)$ and $post[\kappa_{\Delta}^{-1}(r)](nw)$ append the symbol n_1 to the nested word nw, and neither affect the nesting relation.
2. $r = \langle (p,q), n_1\rangle \hookrightarrow \langle (p',q'), n_2\rangle$, and $n_1 \in \Gamma_A^{\beta}$. Because $n_1 \in \Gamma_A^{\beta}$, $post[r](nw) = nw$. By definition, $\kappa_{\Delta}^{-1}(r) = \epsilon$, and $post[\kappa_{\Delta}^{-1}(r)](nw) = nw$.
3. $r = \langle (p,q_c), n_c\rangle \hookrightarrow \langle (p',q'), e\ (r_c,q_c)\rangle$. By definition, $\kappa_{\Delta}^{-1}(r) = \langle p, n_c\rangle \hookrightarrow \langle p', e\ r_c\rangle$. Both $post[r](nw)$ and $post[\kappa_{\Delta}^{-1}(r)](nw)$ append n_c to w and add $\langle(|wn_c|), \infty\rangle$ to v.
4. $r = \langle (p,q_r), x\rangle \hookrightarrow \langle (p',q_r^x), \epsilon\rangle$. By definition, $\kappa_{\Delta}^{-1}(r) = \langle p, x\rangle \hookrightarrow \langle p', \epsilon\rangle$. Let $i = max(\{k \mid \langle k, \infty\rangle \in v\})$. Both $post[r](nw)$ and $post[\kappa_{\Delta}^{-1}(r_j)](nw)$ append x to w, remove $\langle i, \infty\rangle$ from v, and add $\langle i, |wx|\rangle$ to v.

Combining the above case analysis with the definition of *NWofRun*, we have shown that

$$NWofRun([r_1,\ldots,r_j]) = NWofRun([\kappa_{\Delta}^{-1}(r_1),\ldots,\kappa_{\Delta}^{-1}(r_j)])$$

We follow this by showing that $[\kappa_{\Delta}^{-1}(r_1),\ldots,\kappa_{\Delta}^{-1}(r_j)]$ is a run of $\mathcal{P}$. We show this via an inductive argument on the effective length of the run.

Base case: There are two types of runs that have an effective length of zero.

1. $j = 0$. By definition, $[] \in Runs(\mathcal{P})$.
2. Each rule of the run is such that its left-hand-side stack symbol $\gamma \in \Gamma_A^{\beta}$. By the definition of *Construction 1*, each such rule must be preceded by a pop rule, and the left-hand-side stack symbol of every pop rule in Δ_A is a member of Γ_A^{α}. Thus, this case cannot arise.

Inductive step: Let k be the effective length of the run. Let $[r_1,\ldots,r_{i-1}]$ be a prefix of the run such that $EffLen([r_1,\ldots,r_{i-1}]) = k-1$, and $NWofRun([r_1,\ldots,r_{i-1}]) \in L(\mathcal{P}_A,(p,q))$. By the inductive hypothesis we can assume that $[\kappa_{\Delta}^{-1}(r_1),\ldots,\kappa_{\Delta}^{-1}(r_{i-1})]$ is a run of $\mathcal{P}$ and that $NWofRun([\kappa_{\Delta}^{-1}(r_1),\ldots,\kappa_{\Delta}^{-1}(r_{i-1})] \in L(\mathcal{P},p)$. Note that r_{i-1} cannot be a pop rule by Lem. 1. We perform a case analysis on the suffix $[r_i,\ldots,r_j]$ of the run to prove Lem. 3. In each case, we assume that the prefix $[r_1,\ldots,r_{i-1}]$ transforms the configuration $\langle (p_0,q_0), n_{\text{main}}\rangle$ to a configuration $\langle (p,q), \gamma\, u\rangle$ and the prefix $[\kappa_{\Delta}^{-1}(r_1),\ldots,\kappa_{\Delta}^{-1}(r_{i-1})]$ transforms the configuration $\langle p_0, n_{\text{main}}\rangle$ to the configuration $\langle p, \gamma u'\rangle$. Note that u and u' are related. Namely, for each stack symbol $(r_c,q_c) \in u$, u' has the corresponding stack symbol r_c.

1. The suffix $[r_i, \ldots, r_j]$ has length zero. This case invalidates the inductive assumptions because it does not increase the effective length of the run by 1.
2. The suffix $[r_i, \ldots, r_j]$ has length one. In this case, $r_i = r_j$ and there are four possible forms that the rule r_j can have.
   (a) $r_j = \langle (p, q), \gamma \rangle \hookrightarrow \langle (p', q'), \gamma' \rangle$, and $\gamma \in \Gamma_A^\alpha$. The configuration $\langle (p, q), \gamma\, u \rangle$ is transformed to the configuration $\langle (p', q'), \gamma'\, u \rangle$. By definition, $\kappa_\Delta^{-1}(r_j) = \langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \rangle$, and the configuration $\langle p, \gamma\, u' \rangle$ is transformed to the configuration $\langle p', \gamma'\, u' \rangle$.
   (b) $r_j = \langle (p, q_c), \gamma \rangle \hookrightarrow \langle (p', q'), e\ (r_c, q_c) \rangle$. The configuration $\langle (p, q), \gamma\, u \rangle$ is transformed to the configuration $\langle (p', q'), e\, (r_c, q_c)\, u \rangle$. By definition, $\kappa_\Delta^{-1}(r_j) = \langle p, \gamma \rangle \hookrightarrow \langle p', e\ r_c \rangle$, and the configuration $\langle p, \gamma\, u' \rangle$ is transformed to the configuration $\langle p', e\, r_c\, u' \rangle$.
   (c) $r_j = \langle (p, q), \gamma \rangle \hookrightarrow \langle (p, q'), \gamma' \rangle$, and $\gamma' \in \Gamma_A^\beta$. This case is not valid because the effective length of the run $[r_1, \ldots, r_j]$ is $k - 1$.
   (d) $r_j = \langle (p, q_r), \gamma \rangle \hookrightarrow \langle (p', q_r^x), \epsilon \rangle$. This case is not valid because a run cannot end with a pop rule by Lem. 1.
3. The suffix $[r_i, \ldots, r_j]$ has length two. In this case, one of the rules must have its left-hand-side stack symbol in Γ_A^α and the other in Γ_A^β. The only rules whose left-hand-side stack symbols are in Γ_A^β are of the form $\langle (p', q_r^x), (r_c, q_c) \rangle \hookrightarrow \langle (p', q'), r_c \rangle$. For a run of $\mathcal{P}_A$, a rule of this form must be immediately preceded by a pop rule. This follows from *Construction 1*. Because r_{i-1} is not a pop rule, we only need to consider the following case:

$$[r_i, r_j] = [\langle (p, q), x \rangle \hookrightarrow \langle (p', q_r^x), \epsilon \rangle, \langle (p', q_r^x), (r_c, q_c) \rangle \hookrightarrow \langle (p', q'), r_c \rangle]$$

   For the suffix to be valid, the configuration $\langle (p, q), \gamma\, u \rangle$ must be of the form $\langle (p, q), x\, (r_c, q_c)\, u'' \rangle$. In this case, the suffix transforms the configuration into $\langle (p', q'), r_c\, u'' \rangle$. From the inductive hypothesis, the configuration $\langle p, \gamma\, u' \rangle$ of $\mathcal{P}$ must be of the form $\langle p, x\, r_c\, u''' \rangle$. Additionally, we have the following:

$$[\kappa_\Delta^{-1}(r_i), \kappa_\Delta^{-1}(r_j)] = [\langle p, x \rangle \hookrightarrow \langle p', \epsilon \rangle, \epsilon]$$

   Applying the suffix to the configuration $\langle p, x\, r_c\, u''' \rangle$ results in the configuration $\langle p, r_c\, u''' \rangle$.
4. The suffix $[r_i, \ldots, r_j]$ has length equal to 3 or more. This case cannot occur because the suffix would have to contain at least 2 rules that do not change the effective length of the run. From the definition of *Construction 1*, each such rule must be immediately preceded by a pop rule. Thus, the effective length would increase by more than 1.

□

Lemma 4. $L(\mathcal{P}_A, (p', q')) \subseteq L(A, q')$.

Proof. The proof of Lem. 4 is similar to the proof of Lem. 3. Specifically, we provide a function such that for a run $[r_1, \ldots, r_j]$ of $\mathcal{P}_A$, the function defines

a run $[t_1, \ldots, t_j]$ of A, where a run of A is a sequence of transitions that A can use to read a nested word NW. The proof makes use of the deconstructor $\kappa_\delta^{-1} : \Delta_A \to \delta$, defined as follows:

$$\kappa_\delta^{-1}(r) = \begin{cases} (q, n_1, q') & \text{if } r = \langle (p,q), n_1 \rangle \hookrightarrow \langle (p', q'), n_2 \rangle \\ (q_c, n_c, q') & \text{if } r = \langle (p,q_c), n_c \rangle \hookrightarrow \langle (p', q'), e\ (r_c, q_c) \rangle \\ \epsilon & \text{if } r = \langle (p,q_r), x \rangle \hookrightarrow \langle (p', q_r^x), \epsilon \rangle \\ (q_r, q_c, x, q') & \text{if } r = \langle (p', q_r^x), (r_c, q_c) \rangle \hookrightarrow \langle (p', q'), r_c \rangle \end{cases}$$

We extend the function $\kappa_\delta^{-1}(r)$ to work on a run as follows:

$$\begin{aligned} \kappa_\delta^{-1}([]) &= [] \\ \kappa_\delta^{-1}([r_1, \ldots, r_j]) &= \kappa_\delta^{-1}(r_1) :: \kappa_\delta^{-1}([r_2, \ldots, r_j]) \end{aligned}$$

Let $nw \in L(\mathcal{P}_A, (p', q'))$ be the nested word (w, v). By the definition of $L(\mathcal{P}_A, (p', q'))$, there exists a run $[r_1, \ldots, r_j]$ of $\mathcal{P}_A$ that generates nw. We show that A can read nw and end in state q' using the run $\kappa_\delta^{-1}([r_1, \ldots, r_j])$. The proof is by induction on the effective length of the run. The only cases that need to be considered are exactly the cases enumerated in the proof of Lem. 3. Hence, we omit the invalid cases instead of restating why they are invalid.

Base case: The run is empty. In this case, $nw = (\epsilon, \emptyset)$ and $nw \in L(\mathcal{P}_A, (p, q_0))$. The corresponding run of A, $\kappa_\delta^{-1}([])$, is also empty and $nw \in L(A, q_0)$.

Inductive step: Let k be the effective length of the run. We assume that Lem. 4 holds for the prefix $[r_1, \ldots, r_{i-1}]$ of the run whose effective length is $k - 1$. We perform a case analysis on the suffix $[r_i, \ldots, r_j]$ of the run to prove Lem. 4. In each case, we assume that the prefix transforms the configuration $\langle (p_0, q_0), n_{\text{main}} \rangle$ to some configuration $\langle (p, q), \gamma u \rangle$.

1. The suffix $[r_i, \ldots, r_j]$ has length one. In this case, $r_i = r_j$ and there are two possible forms that the rule r_j can have such that it is a valid prefix and suffix of the run.
   (a) $r_j = \langle (p,q), \gamma \rangle \hookrightarrow \langle (p', q'), \gamma' \rangle$, $\gamma \in \Gamma_A^\alpha$. A can make a transition from state q to state q' when reading input symbol γ via $\kappa_\delta^{-1}(r_j)$.
   (b) $r_j = \langle (p,q), \gamma \rangle \hookrightarrow \langle (p', q'), \gamma' \gamma'' \rangle$. A can make a transition from state q to state q' when reading input symbol γ via $\kappa_\delta^{-1}(r_j)$.
2. The suffix $[r_i, \ldots, r_j]$ has length two and is of the form:

   $$[r_i, r_j] = [\langle (p', q), x \rangle \hookrightarrow \langle (p', q_r^x), \epsilon \rangle, \langle (p', q_r^x), (r_c, q_c) \rangle \hookrightarrow \langle (p', q'), r_c \rangle]$$

   Because r_i is a pop rule and nw is a nested word, there must have been a rule r_c in the run such that r_c is a push rule. Furthermore, because r_j fired, it must be the case that r_c is of the form $\langle (p_c, q_c), n_c \rangle \hookrightarrow \langle (p_e, q_e), e\ (r_c, q_c) \rangle$. From this, we know that when A made its corresponding transition $\kappa_\delta^{-1}(r_c)$ (item 1b), it was in a state q_c. Thus, in this case, A can move to state q' via the transition $\kappa_\delta^{-1}([r_i, r_j]) = (q_r, q_c, x, q')$.

□

Lemma 5. $L(\mathcal{P}_A, (p', q')) \supseteq L(\mathcal{P}, p') \cap L(A, q')$

Proof. Let $nw \in L(\mathcal{P}, p') \cap L(A, q')$ be the nested word (w, v). By the definition of $L(\mathcal{P}, p')$, there exists a run $[r_1, \ldots, r_j]$ of $\mathcal{P}$ from $\langle p_0, n_{\text{main}} \rangle$ such that $NWofRun([r_1, \ldots, r_j]) = nw$. From Lem. 2, we know that the length j of the run is equal to $|w|$. From the definition of $L(A, q')$, there exists a run $[t_1, \ldots, t_m]$ of A that reads nw, and leaves A in state $q' \in Q$. By the definition of NWAs, the number of transitions $m = |w|$ (each transition reads exactly one character from the input string). Thus, $j = m = |w|$. We show that for nw, there exists a run of $\mathcal{P}_A$ that simulates both $\mathcal{P}$ and A. The proof is via induction on the length j of the runs of $\mathcal{P}$ and A.

Base case: If $j = 0$, then $nw = (\epsilon, \emptyset)$, and $nw = NWofRun([])$. By definition, $nw \in L(\mathcal{P}_A, (p_0, q_0))$.

Inductive step: We assume that $\mathcal{P}_A$ has successfully simulated the first $j-1$ rules of the run of $\mathcal{P}$ and the first $j-1$ transitions of the run of A. We show that $\mathcal{P}_A$ can simulate runs $[r_1, \ldots, r_{j-1}, r_j]$ and $[t_1, \ldots, t_{j-1}, t_j]$ of $\mathcal{P}$ and A, respectively. We prove the inductive step via a case analysis on the rule r_j and transition t_j.

1. $r_j \in \Delta_1$ and $t_j \in \delta_i$. $\mathcal{P}_A$ simulates the steps of $\mathcal{P}$ and A by the rule r_j and transition t_j, respectively, via the rule $\kappa(r_j, t_j) \in \Delta_A$.
2. $r_j \in \Delta_2$ and $t_j \in \delta_c$. $\mathcal{P}_A$ simulates the steps of $\mathcal{P}$ and A by r_j and t_j, respectively, via the rule $\kappa(r_j, t_j) \in \Delta_A$.
3. $r_j \in \Delta_0$ and $t_j \in \delta_r$. Let $r_j = \langle p, x \rangle \hookrightarrow \langle p', \epsilon \rangle$ and $t_j = (q_r, q_c, x, q')$. Because $\mathcal{P}$ and A are able to make a transition on r_j and t_j, respectively, and from our assumption that $\mathcal{P}_A$ has simulated $\mathcal{P}$ and A through $j-1$ steps, one of the rules r_i, $0 \leq i < j$, is a push rule of the form $\langle p, n_c \rangle \hookrightarrow \langle p', e\ r_c \rangle$. Additionally, t_i must be a transition from δ_c of the form (q_c, n_c, q). From item 2, $\mathcal{P}_A$ simulates these instructions via the rule $\kappa(r_i, t_i) = \langle (p, q_c), n_c \rangle \hookrightarrow \langle (p', q), e\,(r_c, q_c) \rangle$. Combining these facts with the result of $\kappa(r_j, t_j)$, we prove that $\mathcal{P}_A$ simulates $\mathcal{P}$ and A for this case. In particular, by definition $\kappa(r_j, t_j)$ is a set of rules such that the set contains at least two rules, one of the form $\langle (p, q_r), x \rangle \hookrightarrow \langle (p', q_r^x), \epsilon \rangle$ and another of the form $\langle (p', q_r^x), (r_c, q_c) \rangle \hookrightarrow \langle (p', q'), r_c \rangle$. The former is a direct result of $\kappa(r_j, t_j)$. The latter exists because t_j exists and r_c is one of the return points from the original PDS $\mathcal{P}$. Thus, via the application of the two rules, $\mathcal{P}_A$ simulates $\mathcal{P}$ and A.

□

Safety Query Given $\mathcal{P}_A$, to ensure that $\mathcal{P}$ adheres to the property specified by A, one checks the following:

$$\emptyset = \bigcup_{p \in P} L(\mathcal{P}_A, (p, q_{err})) \tag{1}$$

The following *post** query checks whether the safety property holds: $post^*(\mathcal{P}_A, C) \cap C' = \emptyset$, where $C = \{\langle (p_0, q_0), n_{\text{main}} \rangle\}$ and $C' = \{\langle (p, q_{err}), S \rangle \mid p \in P, S \in \Gamma_A^*\}$.

The *Explicit-NWA-plus-PDS* construction suffers from two major drawbacks. First, the number of control locations of $\mathcal{P}_A$ is $|P| \times |Q| \times |\Gamma|$. This is undesirable because reachability queries on PDSs are quadratic in the number of control locations [10]. Second, each function is analyzed for each state of the NWA in which it can be called. Explicit modeling of the state in this way can cause a simple *post** query to become infeasible. We will address both of these problems by applying symbolic techniques to encode the set of control locations.

Roadmap. The focus of the remainder of this paper is the two constructions *Symbolic NWA plus PDS* and *Symbolic NWA plus EWPDS*. Each construction makes use of extended weighted pushdown systems (EWPDSs), defined in §7, which are a generalization of PDSs that adds the capability to reason about infinite-state data abstractions through *weights*. The first construction uses weights to encode the transition relation of A. The second construction extends program models to also include weights. We begin by presenting definitions and notations for EWPDSs. §8 presents the constructions *Symbolic NWA plus PDS* and *Symbolic NWA plus EWPDS*.

7 Weighted Pushdown Systems

A weighted pushdown system (WPDS) is obtained by augmenting a PDS with a weight domain that is a *bounded idempotent semiring* [3,4]. We refer to semiring elements as weights. They encode the effect that each statement (or PDS rule) has on the data state of the program. They can be thought of as abstract transformers that specify how the abstract state changes when a statement is executed.

Definition 5. *A* **bounded idempotent semiring** *(or* **weight domain***) is a tuple* $(D, \oplus, \otimes, \overline{0}, \overline{1})$*, where* D *is a set whose elements are called* **weights***,* $\overline{0}, \overline{1} \in D$*, and* $\oplus$ *(the combine operation) and* $\otimes$ *(the extend operation) are binary operators on* D *such that*

1. $(D, \oplus)$ *is a commutative monoid with* $\overline{0}$ *as its neutral element, and where* $\oplus$ *is idempotent.* $(D, \otimes)$ *is a monoid with the neutral element* $\overline{1}$*.*
2. $\otimes$ *distributes over* $\oplus$*, i.e., for all* $a, b, c \in D$ *we have* $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$ *and* $(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$ *.*
3. $\overline{0}$ *is an annihilator with respect to* $\otimes$*, i.e., for all* $a \in D$*,* $a \otimes \overline{0} = \overline{0} = \overline{0} \otimes a$*.*
4. *In the partial order* $\sqsubseteq$ *defined by* $\forall a, b \in D,\ a \sqsubseteq b$ *iff* $a \oplus b = a$*, there are no infinite descending chains.*

Definition 6. *A* **weighted pushdown system** *is a triple* $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$*, where* $\mathcal{P} = (P, \Gamma, \Delta)$ *is a PDS,* $\mathcal{S} = (D, \oplus, \otimes, \overline{0}, \overline{1})$ *is a bounded idempotent semiring, and* $f : \Delta \to D$ *is a map that assigns a weight to each rule of* $\mathcal{P}$*.*

We use the following approach for modeling a program as a WPDS: the PDS models the control flow of the program, as in Fig. 1. The weight domain models abstract transformers for an abstraction of the program's data. Our first use of weights will be to encode the property automaton's transition relation δ. However, to be able to accurately model the NWA's *jump edges*, the weight domain requires the ability to distinguish between global and local states. This functionality is provided by *extended weighted pushdown systems* (EWPDSs) [8].

EWPDSs lift WPDSs to handle local states in much the same way that Knoop and Steffen lifted conventional dataflow-analysis algorithms to handle local variables [15]: at a call site at which procedure P calls procedure Q, the local variables of P are modeled as if the current incarnations of P's locals are stored in locations that are inaccessible to Q and to procedures transitively called by Q—consequently, the contents of P's locals cannot be affected by the call to Q; one uses special merging functions to combine the locals of the caller with the value returned by Q to create the state after Q returns.[5]

For a semiring $\mathcal{S}$ on domain D, a *merging function* is defined as follows:

Definition 7. *A function $m : D \times D \to D$ is a* **merging function** *with respect to a bounded idempotent semiring $(D, \oplus, \otimes, \overline{0}, \overline{1})$ if it satisfies the following properties.*

1. **Strictness.** *For all $a \in D$, $m(\overline{0}, a) = m(a, \overline{0}) = \overline{0}$.*
2. **Distributivity.** *The function distributes over $\oplus$. For all $a, b, c \in D$,*

$$m(a \oplus b, c) = m(a, c) \oplus m(b, c) \text{ and } m(a, b \oplus c) = m(a, b) \oplus m(a, c)$$

Definition 8. *Let $(\mathcal{P}, \mathcal{S}, f)$ be a weighted pushdown system; let $\mathcal{M}$ be the set of all merging functions on semiring $\mathcal{S}$, and let Δ_2 denote the set of push rules of $\mathcal{P}$. An* **extended weighted pushdown system** *is a quadruple $\mathcal{E} = (\mathcal{P}, \mathcal{S}, f, g)$ where $g : \Delta_2 \to \mathcal{M}$ assigns a merging function to each rule in Δ_2.*

Note that a push rule has both a weight and a merging function associated with it. Merging functions are used to fuse the local state of the calling procedure as it existed just before the call with the effects on the global state produced by the called procedure.

Like for PDSs, a run of an EWPDS $\mathcal{E}$ from a configuration c is a rule sequence $\rho = [r_1, \ldots, r_j]$ that transforms c into a configuration c'. We denote the transformation by $c \Rightarrow^{\rho} c'$, and the set of all runs from c by $Runs(\mathcal{E}, c)$. Using f and g, we can associate a value to ρ, denoted by $v(\rho)$. To do so, we define several helper functions. The function $v[r]((z, S))$ takes a weight and a weight/rule-stack, and returns a weight and weight/rule-stack as follows:

$$v[r](z, S) = \begin{cases} (z \otimes f(r), S) & \text{if } r = \langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \rangle \\ (\overline{1}, (z, r)||S) & \text{if } r = \langle p, \gamma \rangle \hookrightarrow \langle p', \gamma'\gamma'' \rangle \\ (g(r_c)(z_c, f(r_c) \otimes z \otimes f(r)), S') & \text{if } r = \langle p, \gamma \rangle \hookrightarrow \langle p', \epsilon \rangle \\ & \quad \text{and } S = (z_c, r_c)||S' \\ (z \otimes f(r), S) & \text{if } r = \langle p, \gamma \rangle \hookrightarrow \langle p', \epsilon \rangle \text{ and } S = \emptyset \end{cases}$$

[5] Note that this model agrees with programming languages like Java, where it is not possible to have pointers to local variables (i.e., pointers into the stack).

The function $\mathit{build}(\rho)$ that maps a run to a weight and weight/rule-stack as follows:

$$\begin{aligned}\mathit{build}([]) &= (\overline{1}, \emptyset)\\ \mathit{build}([r_1, \ldots, r_j]) &= v[r_j](\mathit{build}([r_1, \ldots, r_{j-1}]))\end{aligned}$$

The function $\mathit{flatten}(z, S)$ that "flattens" a weight and weight/rule-stack by using the semiring extend ($\otimes$) operation:

$$\begin{aligned}\mathit{flatten}(z, \emptyset) &= z\\ \mathit{flatten}(z, (z_c, r_c)||S') &= \mathit{flatten}(z_c \otimes f(r_c) \otimes z, S')\end{aligned}$$

Given these definitions, $v(\rho) = \mathit{flatten}(\mathit{build}(\rho))$.

We next prove properties of the functions *build*, *flatten*, and v for a run $\rho = [r_1, \ldots, r_j]$ that are important for proving Thms. 2 and 3.

Lemma 6. $\mathit{flatten}(z \otimes z', S) = \mathit{flatten}(z, S) \otimes z'$.

Proof. The proof is by induction of the size of S.

Base case: If $S = \emptyset$, then

$$\mathit{flatten}(z \otimes z', \emptyset) = z \otimes z' = \mathit{flatten}(z, \emptyset) \otimes z'$$

Inductive step: Let $S = (z_c, r_c)||S'$, and assume that $\mathit{flatten}(z \otimes z', S') = \mathit{flatten}(z, S') \otimes z'$.

$$\begin{aligned}\mathit{flatten}(z \otimes z', (z_c, r_c)||S') &= \mathit{flatten}((z_c \otimes f(r_c)) \otimes (z \otimes z'), S')\\ &= \mathit{flatten}((z_c \otimes f(r_c) \otimes z) \otimes z', S')\\ &= \mathit{flatten}((z_c \otimes f(r_c) \otimes z), S') \otimes z'\end{aligned}$$

□

Lemma 7. *For a run $[r_1, \ldots, r_j]$, if r_j is a step rule, then $v([r_1, \ldots, r_j]) = v([r_1, \ldots, r_{j-1}]) \otimes f(r_j)$.*

Proof. Let $(z', S') = \mathit{build}([r_1, \ldots, r_{j-1}])$. From Lem. 6 and the definitions of $v[r]$, *build*, and *flatten*, the following holds:

$$\begin{aligned}v([r_1, \ldots, r_{j-1}, r_j]) &= \mathit{flatten}(\mathit{build}([r_1, \ldots, r_{j-1}, r_j]))\\ &= \mathit{flatten}(v[r_j](\mathit{build}([r_1, \ldots, r_{j-1}])))\\ &= \mathit{flatten}(v[r_j](z', S'))\\ &= \mathit{flatten}(z' \otimes f(r_j), S')\\ &= \mathit{flatten}(z', S') \otimes f(r_j)\\ &= \mathit{flatten}(\mathit{build}([r_1, \ldots, r_{j-1}])) \otimes f(r_j)\\ &= v([r_1, \ldots, r_{j-1}]) \otimes f(r_j)\end{aligned}$$

□

Lemma 8. *For a run* $[r_1, \ldots, r_j]$*, if* r_j *is a call rule, then* $v([r_1, \ldots, r_j]) = v([r_1, \ldots, r_{j-1}]) \otimes f(r_j)$*.*

Proof. Let $(z', S') = \mathit{build}([r_1, \ldots, r_{j-1}])$. From Lem. 6 and the definitions of $v[r]$, *build*, and v, the following holds:

$$\begin{aligned} v([r_1, \ldots, r_{j-1}, r_j]) &= \mathit{flatten}(\mathit{build}([r_1, \ldots, r_{j-1}, r_j])) \\ &= \mathit{flatten}(v[r_j](\mathit{build}([r_1, \ldots, r_{j-1}]))) \\ &= \mathit{flatten}(v[r_j](z', S')) \\ &= \mathit{flatten}(\overline{1}, (z', r_j)||S') \\ &= \mathit{flatten}(z' \otimes f(r_j) \otimes \overline{1}, S') \\ &= \mathit{flatten}(z' \otimes f(r_j), S') \\ &= \mathit{flatten}(z', S') \otimes f(r_j) \\ &= \mathit{flatten}(\mathit{build}([r_1, \ldots, r_{j-1}])) \otimes f(r_j) \\ &= v([r_1, \ldots, r_{j-1}]) \otimes f(r_j) \end{aligned}$$

□

Reachability problems on PDSs generalize to EWPDSs as follows:

Definition 9. *Let* $\mathcal{E} = (\mathcal{P}, \mathcal{S}, f, g)$ *be an EWPDS, where* $\mathcal{P} = (P, \Gamma, \Delta)$*. For any two configurations* c *and* c' *of* $\mathcal{P}$*, let* $path(c, c')$ *denote the set of all rule sequences that transform* c *into* c'*. Let* $S, T \subseteq P \times \Gamma^*$ *be regular sets of configurations. If* $\rho \in path(c, c')$*, then we say* $c \Rightarrow^\rho c'$*. The* ***meet-over-all-valid-paths*** *value* $\mathrm{MOVP}(S, T)$ *is defined as* $\bigoplus\{v(\rho) \mid s \Rightarrow^\rho t, s \in S, t \in T\}$*.*

A polynomial-time algorithm for computing MOVP is given in [8].

Definition 10. *For an EWPDS* $\mathcal{E}$*, the NWL* $L(\mathcal{E})$ *is defined as:*

$$L(\mathcal{E}) = \{\mathit{NWofRun}(\rho) \,|\, \rho \in \mathit{Runs}(\mathcal{E}) \ \wedge \ v(\rho) \neq \overline{0} \ \}$$

Given that the NWL of an EWPDS $\mathcal{E}$ involves a calculation on the weighted valuation of a run ρ of $\mathcal{E}$, we can further restrict $L(\mathcal{E})$ via the notion of φ-*acceptance*.

Definition 11. *For an EWPDS* $\mathcal{E}$*, we define the* φ***-accepted*** *NWL for an EWPDS* $\mathcal{E}$ *and function* $\varphi : D \to \mathbb{B}$ *as:*

$$L^\varphi(\mathcal{E}) = \{\mathit{NWofRun}(\rho) \mid \rho \in \mathit{Runs}(\mathcal{E}) \ \wedge \ v(\rho) \neq \overline{0} \ \wedge \ \varphi(v(\rho))\}$$

Note that φ-acceptance can only restrict the NWL of an EWPDS. That is, for any EWPDS $\mathcal{E}$ and any function $\varphi : D \to \mathbb{B}$, $L^\varphi(\mathcal{E}) \subseteq L(\mathcal{E})$.

8 Property Checking via EWPDS Reachability

We next present the constructions *Symbolic NWA plus PDS* and *Symbolic NWA plus EWPDS*. Each construction results in an EWPDS, and property checking is performed via a reachability query on the resulting EWPDS.

8.1 Symbolic NWA Plus PDS

Recall the two drawbacks of the *Explicit NWA plus PDS* construction: (i) the enlarged state space, and (ii) the cost of analyzing a function for each state in which it may be called. The solution to these problems is to apply symbolic techniques. The idea is as follows: instead of combining the control locations of the NWA A (i.e., Q) with the control locations of the PDS $\mathcal{P}$ (i.e., P), we encode the transition relation δ of A using a relational weight domain over Q.

Definition 12. *If G is a finite set, then the* ***relational weight domain*** *on G is defined as $(2^{G\times G}, \cup, ;, \emptyset, id)$: weights are binary relations on G, combine is union, extend is relational composition (";"), $\overline{0}$ is the empty relation, and $\overline{1}$ is the identity relation on G. If R is a relation on G and $R(s_1, s_2)$ holds, then we write $s_1 \rightarrow s_2 \in R$.*

By encoding δ using a relational weight domain, we can use binary decision diagrams [16] to represent the weights that arise. This addresses the desire to economize on the number of PDS states, and also allows each function to be analyzed symbolically.

Before presenting this construction, we need to introduce some notation. First, we define $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$. The relational weight domain that we will use will be over the finite set $Q \times \Sigma_\epsilon$. We denote an element of this domain as (q_i^σ, q_j^σ), but omit σ when $\sigma = \epsilon$.

Second, we define the restriction of δ_i to σ, denoted by $\delta_{i|\sigma}$, as the relation with $(q, q') \in \delta_{i|\sigma}$ iff $(q, \sigma, q') \in \delta_i$. Note that $\delta_{i|\sigma}$ can be embedded into $(Q \times \Sigma_\epsilon) \times (Q \times \Sigma_\epsilon)$ using only states in which $q \in Q$ is paired with ϵ (i.e., q^ϵ). Henceforth, we abuse notation and use $\delta_{i|\sigma}$ to mean the version that is embedded in $(Q \times \Sigma_\epsilon) \times (Q \times \Sigma_\epsilon)$. We define $\delta_{c|\sigma}$ similarly.

Third, we define the function *expand*(σ), which takes as input a symbol $\sigma \in \Sigma$ and generates the relation $\{(q^\epsilon, q^\sigma) \mid q \in Q\}$. *Expand*$(\sigma)$ serves a purpose similar to pop rule that is generated in *Construction 1*, item 3.

Fourth, we define $\hat{\delta}$ so that $(q_r^\sigma, q_c, q) \in \hat{\delta}$ iff $(q_r, q_c, \sigma, q) \in \delta_r$. Notice that $\hat{\delta}$ simply merges the input symbol σ used in δ_r with the return state. This is used to allow the weight domain to model the second set of PDS rules generated by *Construction 1*, item 3.

Construction 2 *[Symbolic NWA plus PDS].* The combination of a PDS $\mathcal{P}$ and an NWA A is modeled by an EWPDS $\mathcal{E} = (\mathcal{P}, \mathcal{S}, f, g)$, where $\mathcal{S}$ is a relational weight domain on $(Q \times \Sigma)$, and f and g are defined as follows:

1. For step rule $r = \langle p, n_1\rangle \hookrightarrow \langle p', n_2\rangle \in \Delta$ and $(q, n_1, q') \in \delta_i$, $f(r) = \{s_1 \rightarrow s_2 \mid \delta_{i|n_1}(s_1, s_2)\}$.
2. For push rule $r = \langle p, n_c\rangle \hookrightarrow \langle p', e\ r_c\rangle \in \Delta$ and $(q_c, n_c, q) \in \delta_c$, $f(r) = \{s_1 \rightarrow s_2 \mid \delta_{c|n_c}(s_1, s_2)\}$ and

$$g(r)(w_c, w_x) = \left\{ s_1 \rightarrow s_2 \mid \exists a, b : \begin{pmatrix} s_1 \rightarrow a & \in w_c \\ \wedge\ a \rightarrow b & \in (f(r) \otimes w_x) \\ \wedge\ \hat{\delta}(b, a, s_2) & \end{pmatrix} \right\} \quad (2)$$

3. For pop rule $r = \langle p, x\rangle \hookrightarrow \langle p', \epsilon\rangle \in \Delta$ and $(q_r, q_c, x, q) \in \delta_r$, $f(r) = \{s_1 \rightarrow s_2 \mid \mathit{expand}(x)(s_1, s_2)\}$.

To perform property checking on $\mathcal{P}$ with respect to A, we would like to restrict the language $L(\mathcal{E})$ so that it only inludes the nested words of runs where the weighted valuation of the run contains a tuple (q_0, q) for some $q \in Q$. This is accomplished by the restricted language $L^{\varphi}(\mathcal{E})$, where φ-acceptance of a run ρ is defined as $\varphi(v(\rho)) = \exists q \in Q : q_0 \rightarrow q \in v(\rho)$.

Theorem 2. *An NWA A combined with a PDS $\mathcal{P}$ results in an EWPDS $\mathcal{E}$ such that $L^{\varphi}(\mathcal{E}) = L(\mathcal{P}) \cap (A, Q)$*

Proof. The proof is organized as follows:

1. $L^{\varphi}(\mathcal{E}) \subseteq L(\mathcal{P})$ by Lem. 9 ;
2. $L^{\varphi}(\mathcal{E}) \subseteq L(A, Q)$ by Lem. 10; and
3. $L^{\varphi}(\mathcal{E}) \supseteq L(A, Q) \cap L(\mathcal{P})$ by Lem. 11.

□

Lemma 9. $L^{\varphi}(\mathcal{E}) \subseteq L(\mathcal{P})$.

Proof. We prove that $L(\mathcal{E}) \subseteq L(\mathcal{P})$. The PDS component of $\mathcal{E}$ is $\mathcal{P}$, and every run of $\mathcal{E}$ is a run of $\mathcal{P}$. Thus, by definition, $L(\mathcal{E})$ can only be a restriction on $L(\mathcal{P})$ due to the non-zero test on $v(\rho)$ for a run ρ of $\mathcal{E}$. By Defn. 11, $L^{\varphi}(\mathcal{E}) \subseteq L(\mathcal{E})$, thus proving Lem. 9. □

Lemma 10. $L^{\varphi}(\mathcal{E}) \subseteq L(A, Q)$.

Proof. Let $nw = (w, v)$ be a nested word in $L(\mathcal{E})$, and let $[r_1, \ldots, r_j]$ be a run of $\mathcal{E}$ such that $nw = \mathit{NWofRun}([r_1, \ldots, r_j])$. Given the weighted valuation $y = v([r_1, \ldots, r_j])$ of the run, we prove that for each $(q_0, q) \in y$, $nw \in L(A, q)$, which implies Lem. 10. The proof is by induction on the length j of the run.

Base case: If j is equal to 0, then $v([]) = \overline{1}$ by the definition of v. For a relational weight domain, the weight $\overline{1}$ is the identity relation, which is the set $\{q \rightarrow q \mid q \in Q\}$. Therefore, the only tuple with q_0 on the left-hand side in y is $q_0 \rightarrow q_0$. Also, because $j = 0$, we know that $nw = (\epsilon, \emptyset)$, which by the definition of $L(A, q)$ is a member of $L(A, q_0)$.

Inductive step: We assume that for length $j-1$, $nw' = \mathit{NWofRun}([r_1, \ldots, r_{j-1}])$, $y' = v([r_1, \ldots, r_{j-1}])$, and for each $q_0 \rightarrow q' \in y'$, $nw' \in L(A, q')$. We now consider the possible forms of rule r_j.

1. $r_j = \langle p, n_1\rangle \hookrightarrow \langle p', n_2\rangle$. From Lem. 7, $y = y' \otimes f(r_j)$. From *Construction 1*, $f(r_j) = \delta_{i|n_1}$. By the definition of $\delta_{i|n_1}$, for each $q' \rightarrow q \in f(r_j)$, the NWA A can make a transition from state q' to state q when reading input symbol n_1. By the definition of $y' \otimes f(r_j)$, for each $q_0 \rightarrow q' \in y'$ and $q' \rightarrow q \in f(r_j)$, the weight y contains the tuple $q_0 \rightarrow q$. Thus, $nw \in L(A, q)$

2. $r_j = \langle p, n_c \rangle \hookrightarrow \langle p', e\ r_c \rangle$. From Lem. 8, $y = y' \otimes f(r_j)$. From *Construction 1*, $f(r_j) = \delta_{c|n_c}$. By the definition of $\delta_{c|n_c}$, for each $q' \to q \in f(r_j)$, the NWA A can make a transition from state q' to state q when reading input symbol n_c. By the definition of $y' \otimes f(r_j)$, for each $q_0 \to q' \in y'$ and $q' \to q \in f(r_j)$, the weight y contains the tuple $q_0 \to q$. Thus $nw \in L(A, q)$.
3. $r_j = \langle p, x \rangle \hookrightarrow \langle p', \epsilon \rangle$. Because nw is a nested word and by the definition of *NWofRun* (i.e., *NWofRun* is undefined for the case where a pop rule does not have a matching push rule), there must exist a matching push rule r_c for r_j in the run. Therefore, $\mathit{build}([r_1, \ldots, r_{j-1}])$ must return a pair of the form $(y_x, (y_c, r_c)||S')$. From our assumptions, the following must hold:

$$\begin{aligned} y' &= v([r_1, \ldots, r_{j-1}]) \\ &= \mathit{flatten}(\mathit{build}([r_1, \ldots, r_{j-1}])) \\ &= \mathit{flatten}(y_x, (y_c, r_c)||S') \\ &= \mathit{flatten}((y_c \otimes f(r_c) \otimes y_x), S') \\ &= \mathit{flatten}(\{q'' \to q' \mid \exists a : y_c(q'', a) \wedge (f(r_c) \otimes y_x)(a, q')\}, S') \end{aligned}$$

The last line replaces the weight equation with its corresponding relational equation. A similar breakdown for y is as follows:

$$\begin{aligned} y &= v([r_1, \ldots, r_{j-1}, r_j]) \\ &= \mathit{flatten}(\mathit{build}([r_1, \ldots, r_{j-1}, r_j])) \\ &= \mathit{flatten}(v[r_j](\mathit{build}([r_1, \ldots, r_{j-1}]))) \\ &= \mathit{flatten}(v[r_j]((y_x, (y_c, r_c)||S'))) \\ &= \mathit{flatten}(g(r_c)(y_c, f(r_c) \otimes y_x \otimes f(r_j)), S') \\ &= \mathit{flatten}(\{g'' \to q \mid \exists a, b : \underline{y_c(q'', a) \wedge (f(r_c) \otimes y_x)(a, q')} \wedge f(r_j)(q', b) \wedge \hat{\delta}(b, a, q)\}, S') \end{aligned}$$

The last line in the equation above replaces the weighted equation with its corresponding relational equation. The underlined section highlights the relationship between $v([r_1, \ldots, r_{j-1}])$ and $v([r_1, \ldots, r_{j-1}, r_j])$. Notice the additional use of $\hat{\delta}(b, a, q)$ and $f(r_j)$. By the definition of $\mathcal{E}$, we know that $f(r_j) = \mathit{expand}(x)$, where x is the left-hand-side stack symbol of r_j. Additionally, we know that when the modeling of A was in a state q', then the state b must be equal to q'^x. Thus, the two derivations prove that if $q_0 \to q' \in y'$ and $nw' \in L(A, q')$, then $q_0 \to q \in y$ and $nw \in L(A, q)$ for pop rule r_j.

□

Lemma 11. $L^{\varphi}(\mathcal{E}) \supseteq L(A, Q) \cap L(\mathcal{P})$

Proof. Let $nw = (w, v)$ be a nested word in $L(A, Q) \cap L(\mathcal{P})$. By the definition of $L(\mathcal{P})$, there must exist a run $[r_1, \ldots, r_j]$ of $\mathcal{P}$ such that $\mathit{NWofRun}([r_1, \ldots, r_j]) = nw$. Additionally, by Lem. 2, $j = |w|$. Likewise, by the definition of $L(A, Q)$, there must exist a run $[t_1, \ldots, t_j]$ of A. The proof is a simulation proof and is performed by induction on the length j.

Base case: If $j = 0$, then $nw = (\epsilon, \emptyset)$, and $nw \in L^{\varphi}(\mathcal{E})$.

Inductive step: We assume that $\mathcal{E}$ has successfully simulated the initial $j-1$ steps of the runs of both $\mathcal{P}$ and A. Let $nw' = \mathit{NWofRun}([r_1, \ldots, r_{j-1}]) \in L^{\varphi}(\mathcal{E})$ and $y' = v([r_1, \ldots, r_{j-1}]) \neq \overline{0}$, where the valuation is computed using f and g of $\mathcal{E}$ as defined by *Construction 2*. Additionally, let q' be the state that A ends up in after making the transitions $[t_1, \ldots, t_{j-1}]$. From our assumptions, $q_0 \to q' \in y'$. The remainder of the proof is a case analysis on r_j and t_j. (We note that in each case, $\mathcal{E}$ can simulate $\mathcal{P}$ because $\mathcal{P}$ is the PDS component of $\mathcal{E}$.)

1. $r_j = \langle p, n_1 \rangle \hookrightarrow \langle p', n_2 \rangle$ and $t_j = (q', n_1, q)$. From Lem. 7, $y = v([r_1, \ldots, r_j]) = y' \otimes f(r_j)$. By *Construction 2*, $f(r_j) = \delta_{i|n_1}$. By the definition of $\delta_{i|n_1}$, $q' \to q \in f(r_j)$. Thus, $q_0 \to q \in y$ and $\mathcal{E}$ simulates A's move on t_j.
2. $r_j = \langle p, n_c \rangle \hookrightarrow \langle p', e\, r_c \rangle$ and $t_j = (q', n_c, q)$. From Lem. 8, $y = v([r_1, \ldots, r_j]) = y' \otimes f(r_j)$. By *Construction 2*, $f(r_j) = \delta_{c|n_c}$. By the definition of $\delta_{c|n_c}$, $q' \to q \in f(r_j)$. Thus, $q_0 \to q \in y$ and $\mathcal{E}$ simulates A's move on t_j.
3. $r_j = \langle p, x \rangle \hookrightarrow \langle p', \epsilon \rangle$ and $t_j = (q', q_c, x, q)$. Let $y = v([r_1, \ldots, r_j])$. Because nw is a nested word and by the definition of *NWofRun* (i.e., *NWofRun* is undefined for the case where a pop rule does not have a matching call rule), there must exist a matching push rule r_c for r_j in the run. Thus,

$$\begin{aligned} \mathit{flatten}(\mathit{build}([r_1, \ldots, r_{j-1}])) &= \mathit{flatten}(y_x, (y_c, r_c)||S) \\ &= \mathit{flatten}(y_c \otimes f(r_c) \otimes y_x, S) \\ &= \mathit{flatten}(\overline{1} \otimes (y_c \otimes f(r_c) \otimes y_x), S) \\ &= \mathit{flatten}(\overline{1}, S) \otimes (y_c \otimes f(r_c) \otimes y_x) \end{aligned} \tag{3}$$

From the definition of $v[r_j]$, we know that

$$\begin{aligned} \mathit{flatten}(\mathit{build}([r_1, \ldots, r_j])) &= \mathit{flatten}(v[r_j](\mathit{build}([r_1, \ldots, r_{j-1}]))) \\ &= \mathit{flatten}(v[r_j](y_x, (y_c, r_c)||S)) \\ &= \mathit{flatten}(g(r_c)(y_c, (f(r_c) \otimes y_x \otimes f(r_j))), S) \\ &= \mathit{flatten}(\overline{1} \otimes (g(r_c)(y_c, (f(r_c) \otimes y_x \otimes f(r_j)))), S) \\ &= \mathit{flatten}(\overline{1}, S) \otimes g(r_c)(y_c, (f(r_c) \otimes y_x \otimes f(r_j))) \end{aligned} \tag{4}$$

Notice that in Eqns. (3) and (4), the left-hand side of the extend is the same (i.e., $\mathit{flatten}(\overline{1}, S)$). Hence it contributes the same value in both cases. Let $y_{j-1} = y_c \otimes f(r_c) \otimes y_x$ and $y_j = g(r_c)(y_c, (f(r_c) \otimes y_x \otimes f(r_j)))$. We show that $q'' \to q' \in y_{j-1} \implies q'' \to q \in y_j$.

$$\begin{aligned} y_{j-1} &= y_c \otimes f(r_c) \otimes y_x \\ &= \{q'' \to q' \mid \exists a : y_c(q'', a) \wedge (f(r_c) \otimes y_x)(a, q')\} \end{aligned}$$

$$\begin{aligned} y_j &= g(r_c)(y_c, (f(r_c) \otimes y_x \otimes \mathit{expand}(x))) \\ &= \left\{ q'' \to q \mid \exists a, b, c : \underline{y_c(q'', a) \wedge (f(r_c) \otimes y_x)(a, b)} \wedge f(r_j)(b, c) \wedge \hat{\delta}(c, a, q) \right\} \end{aligned}$$

Notice that y_{j-1} is the underlined computation in the derivation of y_j. Because $f(r_j) = \mathit{expand}(x)$, it must be the case that for each $q'' \to q' \in y_{j-1}$, $q'' \to q'^x \in (y_{j-1} \otimes f(r_j))$. (This is precisely what is computed by the conjunction $y_{j-1}(q'', b) \wedge f(r_j)(b, c)$.) Additionally, t_j, along with the definition of $\hat{\delta}$, imples that $\hat{\delta}(q'^x, q_c, q)$ holds. Thus, we have proved that $q'' \to q' \in y_{j-1} \implies q'' \to q \in y_j$. From this and the fact that both equations begin with $\mathit{flatten}(\overline{1}, S)$, it follows that for each $q_0 \to q' \in y'$, $q_0 \to q \in y$.

□

Safety Query To verify that $\mathcal{P}$ adheres to the property specified by A, one checks that the following holds for $\mathcal{E}$:

$$L^{\varphi_{err}}(\mathcal{E}) = \emptyset, \text{where } \varphi_{err}(y) = q_0 \to q_{err} \in y \tag{5}$$

The idea is that the nested word for a run is only in the language if it drives the property automaton A to the error state q_{err}. Eqn. (5) can be answered via a reachability query on the gererated EWPDS $\mathcal{E}$. That is, Eqn. (5) can be computed by the following: $\{(q_0, q_{err})\} \notin \mathrm{MOVP}(\{\langle p_0, n_{\mathrm{main}}\rangle\}, \mathcal{U})$, where $\mathcal{U} = \{\langle p, S\rangle \mid p \in P, S \in \Gamma^*\}$.

8.2 Symbolic NWA Plus EWPDS

We now present our final construction, *Symbolic NWA plus EWPDS*. This construction extends *Symbolic NWA plus PDS* by modeling the program as an EWPDS instead of a PDS. This allows one to answer new queries with respect to property checking. For example, it is natural to ask, "What abstract state can the program be in when the property automaton can be in error state q_{err}?"

We define *Symbolic NWA plus EWPDS* by extending the relations used by *Symbolic NWA plus PDS* to be weighted relations.

Definition 13. *A **weighted relation** on a set S, weighted with semiring $(D, \oplus, \otimes, \overline{0}, \overline{1})$, is a function from $(S \times S)$ to D. The composition of two weighted relations R_1 and R_2 is defined as $(R_1; R_2)(s_1, s_3) = \oplus\{w_1 \otimes w_2 \mid \exists s_2 \in S : w_1 = R_1(s_1, s_2), w_2 = R_2(s_2, s_3)\}$. The union of the two weighted relations is defined as $(R_1 \cup R_2)(s_1, s_2) = R_1(s_1, s_2) \oplus R_2(s_1, s_2)$. The identity relation is the function that maps each pair (s, s) to $\overline{1}$ and others to $\overline{0}$. The reflexive transitive closure is defined in terms of these operations, as before. If R is a weighted relation and $R(s_1, s_2) = z$, then we write $s_1 \xrightarrow{z} s_2 \in R$.*

Definition 14. *If $\mathcal{S}$ is a weight domain with set of weights D and G is a finite set, then the relational weight domain on $(G, \mathcal{S})$ is defined as $(2^{G \times G \to D}, \cup, ;, \emptyset, id)$: weights are weighted relations on G and the operations are the corresponding ones for weighted relations.*

This weight domain can be symbolically encoded using techniques such as algebraic decision diagrams [17].

Construction 3 *[Symbolic NWA plus EWPDS].* The combination of an EWPDS $\mathcal{E} = (\mathcal{P}, \mathcal{S}, f, g)$ and an NWA $A = (Q, \Sigma, \delta)$ is modeled by an EWPDS $\mathcal{E}_A$ that has the same underlying PDS as $\mathcal{E}$, but with a new weight domain and new assignments of weights and merge functions to rules: $\mathcal{E}_A = (\mathcal{P}_A, \mathcal{S}_A, f_A, g_A)$, where $\mathcal{P}_A = \mathcal{P}$, $\mathcal{S}_A = (D_A, \oplus_A, \otimes_A, \overline{0}_A, \overline{1}_A)$ is a weighted relation on the set $Q \times \Sigma$ and semiring $\mathcal{S}$, and f_A and g_A are defined as follows ($\overline{marked}$ items signify extensions to *Construction 2*):

1. For rule $r = \langle p, n_1\rangle \hookrightarrow \langle p', n_2\rangle \in \Delta$ and $(q, n_1, q') \in \delta_i$, $f_A(r) = \{s_1 \xrightarrow{\overline{f(r)}} s_2 \mid \delta_{i|n_1}(s_1, s_2)\}$.
2. For push rule $r = \langle p, n_c\rangle \hookrightarrow \langle p', e\ r_c\rangle \in \Delta$ and $(q_c, n_c, q) \in \delta_c$, $f_A(r) = \{s_1 \xrightarrow{f(r)} s_2 \mid \delta_{c|n_c}(s_1, s_2)\}$ and

$$g_A(r)(w_c, w_x) = \left\{ s_1 \xrightarrow{\overline{z}} s_2 \mid \exists a, b : \begin{pmatrix} s_1 \xrightarrow{\overline{z_1}} a & \in w_c \\ \wedge\ a \xrightarrow{\overline{z_2}} b & \in (f_{\overline{A}}(r) \otimes w_x) \\ \wedge\ \hat{\delta}(b, a, s_2) & \end{pmatrix}, \overline{z = g(r)(z_1, z_2)} \right\} \quad (6)$$

3. For pop rule $r = \langle p, x\rangle \hookrightarrow \langle p', \epsilon\rangle \in \Delta$ and $(q_r, q_c, x, q) \in \delta_r$, $f_A(r) = \{s_1 \xrightarrow{f(r)} s_2 \mid expand(x)(s_1, s_2)\}$.

As for *Construction 2*, we are again interested in a restriction of $L(\mathcal{E}_A)$, and again we rely on φ-acceptance. That is, our target language is $L^{\varphi}(\mathcal{E}_A)$, where for a run ρ of $\mathcal{E}_A$ such that $z = v(\rho)$, $\varphi(z) = \exists q \in Q : q_0 \xrightarrow{y} q \in z$, and $y \neq \overline{0}$.

Theorem 3. *An NWA A combined with an EWPDS $\mathcal{E}$ results in an EWPDS $\mathcal{E}_A$ such that $L^{\varphi}(\mathcal{E}_A) = L(A, Q) \cap L(\mathcal{E})$.*

Proof. The proof is organized as follows:

1. $L^{\varphi}(\mathcal{E}_A) \subseteq L(\mathcal{E})$ by Lem. 13;
2. $L^{\varphi}(\mathcal{E}_A) \subseteq L(A, Q)$ by Lem. 14; and
3. $L^{\varphi}(\mathcal{E}_A) \supseteq L(A, Q) \cap L(\mathcal{E})$ by Lem. 15.

□

For a run ρ, we use the notation $v_{\mathcal{E}}(\rho)$, $\mathit{flatten}_{\mathcal{E}}$,$\mathit{build}_{\mathcal{E}}$,and $v_{\mathcal{E}}[r]$ to signify the weighted valuation of ρ using $\mathcal{S}$,f, and g as defined by $\mathcal{E}$; and we use the notation $v_{\mathcal{E}_A}(\rho)$, $\mathit{flatten}_{\mathcal{E}_A}$,$\mathit{build}_{\mathcal{E}_A}$,and $v_{\mathcal{E}_A}[r]$ to signify the weighted valuation of ρ using $\mathcal{S}_A$,f_A,and g_A as defined by $\mathcal{E}_A$. Additionally, we use $\overline{0}$, $\overline{1}$, and y (and its variants) to denote weights from $\mathcal{S}$; and we use $\overline{0}_A$, $\overline{1}_A$, and z (and its variants) to denote weights from $\mathcal{S}_A$.

Lemma 12. *Let $\rho = [r_1, \ldots, r_j]$ be a run of $\mathcal{E}$ and $\mathcal{E}_A$ such that: $nw = NWofRun(\rho)$, $nw \in L(\mathcal{E})$, $nw \in L(\mathcal{E}_A)$, $y = v_{\mathcal{E}}(\rho)$, and $z = v_{\mathcal{E}_A}(\rho)$. If $y \neq \overline{0}$, then for all $q \xrightarrow{y'} q' \in z$ such that $y' \neq \overline{0}$, $y' = y$. Otherwise if $y = \overline{0}$, then $z = \overline{0}_A$.*

Proof. The proof is by induction on the length j of the run. We first handle the case where $y \neq \overline{0}$.

Base case: If $j = 0$, then $v_{\mathcal{E}}([]) = \overline{1}$ and $v_{\mathcal{E}_A}([]) = \overline{1}_A$. By definition, $\overline{1}_A = \{q \xrightarrow{\overline{1}} q \mid q \in Q\}$.

Inductive step: We assume that Lem. 12 holds for $\rho' = [r_1, \ldots, r_{j-1}]$, and prove that it holds for ρ by a case analysis of r_j. By our assumption, we have the following:

- $v_{\mathcal{E}}(\rho') = y'$.
- $v_{\mathcal{E}_A}(\rho') = z'$, and for all $q \xrightarrow{y''} q' \in z'$ where $y'' \neq \overline{0}$, $y'' = y'$.

Let $v_{\mathcal{E}}(\rho) = y$ and $v_{\mathcal{E}_A}(\rho) = z$.

1. $r_j = \langle p, n_1 \rangle \hookrightarrow \langle p', n_2 \rangle$. By Lem. 7, $y = y' \otimes f(r_j)$. By definition, $f_A(r_j) = \{q' \xrightarrow{f(r_j)} q'' \mid q' \to q'' \in \delta_{i|n_1}\}$. From Lem. 7, the following holds:

$$\begin{aligned}
v_{\mathcal{E}_A}(\rho) &= v_{\mathcal{E}_A}([r_1, \ldots, r_{j-1}, r_j]) \\
&= v_{\mathcal{E}_A}([r_1, \ldots, r_{j-1}]) \otimes f_A(r_j) \\
&= v_{\mathcal{E}_A}(\rho') \otimes f_A(r_j) \\
&= z' \otimes f_A(r_j) \\
&= z' \otimes \{q' \xrightarrow{f(r_j)} q'' \mid q' \to q'' \in \delta_{i|n_1}\} \\
&= \{q \xrightarrow{y' \otimes f(r_j)} q'' \mid q \xrightarrow{y'} q' \in z' \wedge q' \xrightarrow{f(r_j)} q'' \in f_A(r_j)\} \\
&= \{q \xrightarrow{y} q'' \mid q \xrightarrow{y'} q' \in z' \wedge q' \xrightarrow{f(r_j)} q'' \in f_A(r_j)\}
\end{aligned}$$

2. $r_j = \langle p, n_c \rangle \hookrightarrow \langle p', e\ r_c \rangle$. The same argument as above applies here, simply replace Lem. 7 with Lem. 8 and $f_A(r_j) = \{q' \xrightarrow{f(r_j)} q'' \mid q' \to q'' \in \delta_{c|n_c}\}$.
3. $r_j = \langle p, x \rangle \hookrightarrow \langle p', \epsilon \rangle$. Because nw is a nested word and by the definition of *NWofRun* (i.e., *NWofRun* is undefined for the case where a pop rule does not have a matching push rule), there must exist a matching push rule r_c for r_j in the run. Thus, with respect to $v_{\mathcal{E}}$, the following holds:

$$\begin{aligned}
v_{\mathcal{E}}([r_1, \ldots, r_{j-1}]) &= \mathit{flatten}_{\mathcal{E}}(\mathit{build}_{\mathcal{E}}([r_1, \ldots, r_{j-1}])) \\
&= \mathit{flatten}_{\mathcal{E}}(y_x, (y_c, r_c)||S) \\
&= \mathit{flatten}_{\mathcal{E}}(y_c \otimes f(r_c) \otimes y_x, S) \\
&= \mathit{flatten}_{\mathcal{E}}(\overline{1} \otimes (y_c \otimes f(r_c) \otimes y_x), S) \\
&= \mathit{flatten}_{\mathcal{E}}(\overline{1}, S) \otimes (y_c \otimes f(r_c) \otimes y_x) \qquad (7) \\
v_{\mathcal{E}}([r_1, \ldots, r_j]) &= \mathit{flatten}_{\mathcal{E}}(\mathit{build}_{\mathcal{E}}([r_1, \ldots, r_j])) \\
&= \mathit{flatten}_{\mathcal{E}}(v_{\mathcal{E}}[r_j](\mathit{build}_{\mathcal{E}}([r_1, \ldots, r_{j-1}]))) \\
&= \mathit{flatten}_{\mathcal{E}}(v_{\mathcal{E}}[r_j](y_x, (y_c, r_c)||S)) \\
&= \mathit{flatten}_{\mathcal{E}}(g(r_c)(y_c, f(r_c) \otimes y_x \otimes f(r_j)), S) \\
&= \mathit{flatten}_{\mathcal{E}}(\overline{1} \otimes (g(r_c)(y_c, f(r_c) \otimes y_x \otimes f(r_j))), S) \\
&= \mathit{flatten}_{\mathcal{E}}(\overline{1}, S) \otimes g(r_c)(y_c, f(r_c) \otimes y_x \otimes f(r_j)) \qquad (8)
\end{aligned}$$

Similar for $v_{\mathcal{E}_A}$, the following holds:

$$\begin{aligned}
v_{\mathcal{E}_A}([r_1,\ldots,r_{j-1}]) &= \mathit{flatten}_{\mathcal{E}_A}(\mathit{build}_{\mathcal{E}_A}([r_1,\ldots,r_{j-1}])) \\
&= \mathit{flatten}_{\mathcal{E}_A}(z_x,(z_c,r_c)||S) \\
&= \mathit{flatten}_{\mathcal{E}_A}(z_c \otimes f_A(r_c) \otimes z_x, S) \\
&= \mathit{flatten}_{\mathcal{E}_A}(\overline{1}_A \otimes (z_c \otimes f_A(r_c) \otimes z_x), S) \\
&= \mathit{flatten}_{\mathcal{E}_A}(\overline{1}_A, S) \otimes (z_c \otimes f_A(r_c) \otimes z_x) \qquad (9) \\
v_{\mathcal{E}_A}([r_1,\ldots,r_j]) &= \mathit{flatten}_{\mathcal{E}_A}(\mathit{build}_{\mathcal{E}_A}([r_1,\ldots,r_j])) \\
&= \mathit{flatten}_{\mathcal{E}_A}(v_{\mathcal{E}_A}[r_j](\mathit{build}_{\mathcal{E}_A}([r_1,\ldots,r_{j-1}]))) \\
&= \mathit{flatten}_{\mathcal{E}_A}(v_{\mathcal{E}_A}[r_j](z_x,(z_c,r_c)||S)) \\
&= \mathit{flatten}_{\mathcal{E}_A}(g_A(r_c)(z_c, f_A(r_c) \otimes z_x \otimes f_A(r_j)), S) \\
&= \mathit{flatten}_{\mathcal{E}_A}(\overline{1}_A \otimes (g_A(r_c)(z_c, f_A(r_c) \otimes z_x \otimes f_A(r_j))), S) \\
&= \mathit{flatten}_{\mathcal{E}_A}(\overline{1}_A, S) \otimes g_A(r_c)(z_c, f_A(r_c) \otimes z_x \otimes f_A(r_j)) \\
& \qquad (10)
\end{aligned}$$

Because both Eqns. (7) and (8) contain $\mathit{flatten}_{\mathcal{E}}(\overline{1}, S)$ on the left-hand side of the extend ($\otimes$), it contributes the same value in both cases. We denote the right-hand sides of the extend of Eqns. (7) and (8) by y_{j-1} and y_j, respectively. Likewise, both Eqns. (9) and (10) contain $\mathit{flatten}_{\mathcal{E}_A}(\overline{1}_A, S)$ on the left-hand side of the extend, and thus it contributes the same value in both cases. Similarly, we denote the right-hand side of the extend of Eqns. (9) and (10) by z_{j-1} and z_j, respectively. From our assumptions, we have the following:

$$z_{j-1} = \left\{ q \xrightarrow{y_{j-1}} q' \;\middle|\; \exists a,b : \begin{pmatrix} q \xrightarrow{y_c} a & \in z_c \\ \wedge\ a \xrightarrow{f(r_c)} b & \in f_A(r_c) \\ \wedge\ b \xrightarrow{y_x} q' & \in z_x \end{pmatrix}, y_{j-1} = y_c \otimes f(r_c) \otimes y_x \right\}$$

Notice that the right-hand side of the extend in Eqn. (7) annotates the tuples in z_{j-1}. From the definition of g_A (Eqn. (6)), the following holds:

$$z_j = \left\{ q \xrightarrow{y_j} q' \;\middle|\; \exists a,b,c,d : \begin{pmatrix} q \xrightarrow{y_c} a & \in z_c \\ \wedge\ a \xrightarrow{f(r_c)} b & \in f_A(r_c) \\ \wedge\ b \xrightarrow{y_x} c & \in z_x \\ \wedge\ c \xrightarrow{f(r_j)} d & \in f_A(r_j) \\ \wedge\ (d,a,q') & \in \hat{\delta} \end{pmatrix}, y_j = g(r_c)(y_c, f(r_c) \otimes y_x \otimes f(r_j)) \right\}$$

Notice that the right-hand side of Eqn. (8) annotates the tuples in z_j. Thus, we have proved the inductive step.

We next handle the case where $y = \overline{0}$. This case, namely that $z = \overline{0}_A$, follows from the above argument. That is, for each $q \xrightarrow{y} q' \in z$, $y = \overline{0}$ and thus $z = \overline{0}_A$. □

Lemma 13. $L^{\varphi}(\mathcal{E}_A) \subseteq L(\mathcal{E})$.

Proof. We prove Lem. 13 by showing that $L(\mathcal{E}_A) \subseteq L(\mathcal{E})$. Because both $\mathcal{E}_A$ and $\mathcal{E}$ have the same underlying PDS $\mathcal{P}$, a run of $\mathcal{E}_A$ is a run of $\mathcal{E}$. Specifically, for a run $\rho = [r_1, \ldots, r_j]$, $\rho \in \mathit{Runs}(\mathcal{E}_A)$ and $\rho \in \mathit{Runs}(\mathcal{E})$. The NWL for an EWPDS is defined in terms of runs and the weighted valuation for a run. Thus, we need only show that $v_{\mathcal{E}}(\rho) = \overline{0} \implies v_{\mathcal{E}_A}(\rho) = \overline{0}_A$, which follows from Lem. 12. From Defn. 11, $L^{\varphi}(\mathcal{E}_A) \subseteq L(\mathcal{E}_A)$, which proves Lem. 13. □

Lemma 14. $L^{\varphi}(\mathcal{E}_A) \subseteq L(A, Q)$.

Proof. For a nested word $nw \in L(\mathcal{E}_A)$, let $[r_1, \ldots, r_j]$ be a run that generates nw, and let $z = v_{\mathcal{E}_A}([r_1, \ldots, r_j])$ be the weighted valuation of the run. We show that for each $q_0 \xrightarrow{y} q \in z$ such that $y \neq \overline{0}$, $nw \in L(A, q)$. Lem. 14 follows from this.

Lem. 12 proves that for a run $\rho = [r_1, \ldots, r_j]$ of $\mathcal{E}_A$, the weighted valuation $z = v_{\mathcal{E}_A}(\rho)$ of the run conceptually consists of two parts. The first part is the relational part of the weighted relation, which models the NWA A. The second part is the weighted part of weighted relations, which models $\mathcal{E}$. We take advantage of this fact by observing that if we mask off the second part, then the weight domain resembles an ordinary relational weight domain like that used in *Construction 2*. In fact, relations are a degenerate form of weighted relations, where the weight domain is the Boolean semiring $(\{\overline{1}_{\mathbb{B}}, \overline{0}_{\mathbb{B}}\}, \oplus_{\mathbb{B}}, \otimes_{\mathbb{B}}, \overline{0}_{\mathbb{B}}, \overline{1}_{\mathbb{B}})$. From this observation, we follow the approach depicted in Fig. 2. We first define an operation called *reduce*, which performs the masking referred to above. We use this operation to show that the two paths that join at point (i) in Fig. 2 produce the same output, $\mathcal{E}_R$. Second, we show that the path from $\mathcal{E}_A \to \mathcal{E}_R \to A$ denotes a chain of language inclusions.

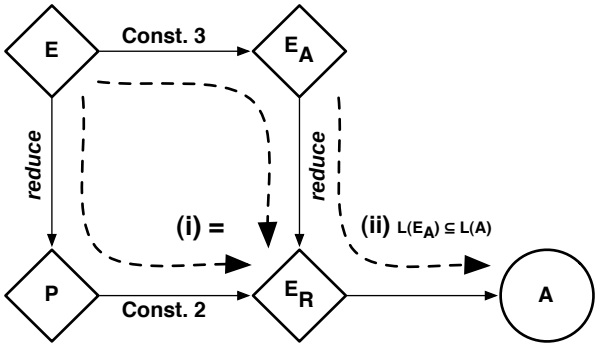

Fig. 2. Outline for the proof of Lem. 14. The paths that join at (i) both produce $\mathcal{E}_R$, and hence they commute. The path denoted by (ii) signifies a chain of language inclusions.

For any nontrivial semiring $\mathcal{S}$ (i.e., one in which $\overline{0} \neq \overline{1}$), we can define a function $\alpha_{\mathbb{B}} : \mathcal{S} \to \mathcal{S}_{\mathbb{B}}$ that maps each non-zero weight of $\mathcal{S}$ to $\overline{1}_{\mathbb{B}}$ and the zero

element of $\mathcal{S}$ to $\overline{0}_{\mathbb{B}}$. Note that $\alpha_{\mathbb{B}}(\mathcal{S})$ is an abstraction of $\mathcal{S}$.

$\otimes_{\mathcal{S}}$	$\overline{0}_{\mathcal{S}}$	$\neg\overline{0}_{\mathcal{S}}$
$\overline{0}_{\mathcal{S}}$	$\overline{0}_{\mathcal{S}}$	$\overline{0}_{\mathcal{S}}$
$\neg\overline{0}_{\mathcal{S}}$	$\overline{0}_{\mathcal{S}}$	$\{\neg\overline{0}_{\mathcal{S}}, \overline{0}_{\mathcal{S}}\}$

$\otimes_{\mathbb{B}}$	$\overline{0}_{\mathbb{B}}$	$\overline{1}_{\mathbb{B}}$
$\overline{0}_{\mathbb{B}}$	$\overline{0}_{\mathbb{B}}$	$\overline{0}_{\mathbb{B}}$
$\overline{1}_{\mathbb{B}}$	$\overline{0}_{\mathbb{B}}$	$\overline{1}_{\mathbb{B}}$

$\oplus_{\mathcal{S}}$	$\overline{0}_{\mathcal{S}}$	$\neg\overline{0}_{\mathcal{S}}$
$\overline{0}_{\mathcal{S}}$	$\overline{0}_{\mathcal{S}}$	$\neg\overline{0}_{\mathcal{S}}$
$\neg\overline{0}_{\mathcal{S}}$	$\neg\overline{0}_{\mathcal{S}}$	$\neg\overline{0}_{\mathcal{S}}$

$\oplus_{\mathbb{B}}$	$\overline{0}_{\mathbb{B}}$	$\overline{1}_{\mathbb{B}}$
$\overline{0}_{\mathbb{B}}$	$\overline{0}_{\mathbb{B}}$	$\overline{1}_{\mathbb{B}}$
$\overline{1}_{\mathbb{B}}$	$\overline{1}_{\mathbb{B}}$	$\overline{1}_{\mathbb{B}}$

This is because for any two weights z_1 and z_2 in $\mathcal{S}$ such that $z_1 \neq \overline{0}_{\mathcal{S}}$ and $z_2 \neq \overline{0}_{\mathcal{S}}$, the following holds: $\alpha_{\mathbb{B}}(z_1) \otimes_{\mathbb{B}} \alpha_{\mathbb{B}}(z_2) = \overline{1}_{\mathbb{B}}$; however, $z_1 \otimes z_2 = \overline{0}_{\mathcal{S}}$ is possible (e.g., suppose that z_1 and z_2 are non-empty relations and their relational composition results in the empty relation).

The operation *reduce* that produces $\mathcal{P}$ from EWPDS $\mathcal{E} = (\mathcal{P}, \mathcal{S}, f, g)$ defines a new EWPDS $\mathcal{E}_{\mathbb{B}} = (\mathcal{P}, \mathcal{S}_{\mathbb{B}}, f_{\mathbb{B}}, g_{\mathbb{B}})$, where $f_{\mathbb{B}}(r) = \alpha_{\mathbb{B}}(f(r))$, and $g_{\mathbb{B}}(r) = \lambda w_1.\lambda w_2.w_1 \otimes_{\mathbb{B}} w_2$. Interestingly, $\mathcal{P} = \mathcal{E}_{\mathbb{B}}$. This is easy to see as all runs of $\mathcal{E}_{\mathbb{B}}$ have the weight $\overline{1}_{\mathbb{B}}$, which simply indicates reachability. This is precisely what a run of $\mathcal{P}$ represents. Notice that $L(\mathcal{P}) \supseteq L(\mathcal{E})$ because of the argument made above. Finally, we apply *Construction 2* to produce the EWPDS $\mathcal{E}_R$.

The operation *reduce* that produces $\mathcal{E}_R$ from EWPDS $\mathcal{E}_A = (\mathcal{P}, \mathcal{S}_A, f_A, g_A)$ also makes use of the $\alpha_{\mathbb{B}}$ function. That is, $\mathcal{E}_R = (\mathcal{P}, \mathcal{S}_R, f_R, g_R)$, where $\mathcal{S}_R$ is the relational weight domain defined by *Construction 2*, $f_R(r) = \{q \xrightarrow{\alpha_{\mathbb{B}}(y)} q' \mid q \xrightarrow{y} q' \in f_A(r)\}$, and $g_R(r)$ is the merging function defined by Eqn. (2). Notice that $f_R(r)$ is exactly the weight that is defined in *Construction 2*; i.e., for a step rule $\langle p, n_1\rangle \hookrightarrow \langle p', n_2\rangle$, $f_R(r) = \delta_{i|n_1}$. This is the result of the usage of $\alpha_{\mathbb{B}}$ to "mask off" the weights from the weighted relations. Notice that the merging function defined by Eqn. (2) is an abstraction of the merging function defined by Eqn. (6). This follows from the argument used above for $\alpha_{\mathbb{B}}$. Finally, reducing weighted relations to be relations has the effect of reducing the φ-function of *Construction 3* to the φ-function of *Construction 2*. Thus, the two paths that join at (i) in Fig. 2 produce the same EWPDS $\mathcal{E}_R$.

To complete the proof, we need to show that $L^{\varphi}(\mathcal{E}_A) \subseteq L^{\varphi}(\mathcal{E}_R)$. This follows from Lem. 12. That is, Lem. 12 tells us that for a run of an EWPDS where the weight domain is a weighted relation, one can view the weighted valuation of the run as having two parts, the relation and the weight. For EWPDSs $\mathcal{E}_R$ and $\mathcal{E}_A$, the relational part is the same and the weighted part of a weighted valuation of a run of $\mathcal{E}_R$ overapproximates the weighted part of the weighted valuation of the same run of $\mathcal{E}_A$, and hence $L^{\varphi}(\mathcal{E}_A) \subseteq L^{\varphi}(\mathcal{E}_R)$.

From Lem. 10, we know that $L^{\varphi}(\mathcal{E}_R) \subseteq L(A, Q)$, and hence $L^{\varphi}(\mathcal{E}_A) \subseteq L(A, Q)$.

□

Lemma 15. $L^{\varphi}(\mathcal{E}_A) \supseteq L(A, Q) \cap L(\mathcal{E})$

Proof. Let $nw = (w, v)$ be a nested word in $L(A, Q) \cap L(\mathcal{E})$ such that $|w| = j$. From the definition of $L(A, Q)$, there must exist a run $[t_1, \ldots, t_j]$ that A can use to read nw. For the PDS component $\mathcal{P}$ of $\mathcal{E}$ and $\mathcal{E}_A$, $\Gamma^\beta = \emptyset$. From the definition of $L(\mathcal{E})$ and by Lem. 2, there must exist a run $[r_1, \ldots, r_j]$ of $\mathcal{E}$ such that $NWofRun([r_1, \ldots, r_j]) = nw$. The proof is a simulation proof and is performed by induction on the length j. That is, we show that there exists a run of $\mathcal{E}_A$ such that the run simultaneously models the runs $[t_1, \ldots, t_j]$ of A and $[r_1, \ldots, r_j]$ of $\mathcal{E}$.

Base case: If $j = 0$, then the runs of A and $\mathcal{E}$ are empty (i.e., $[]$), and thus $nw = (\epsilon, \emptyset)$. By definition, $NWofRun([]) = (\epsilon, \emptyset)$, and the empty run of $\mathcal{E}_A$ generates nw. Consequently, $nw \in L(\mathcal{E}_A)$.

Inductive step: We assume that $\mathcal{E}_A$ has simulated the initial $j - 1$ steps of the runs of both $\mathcal{E}$ and A. Specifically, let

- $nw' = NWofRun([r_1, \ldots, r_{j-1}]) \in L(\mathcal{E}) \cap L(A, Q)$.
- $y' = v_{\mathcal{E}}([r_1, \ldots, r_{j-1}]) \neq \overline{0}$.
- $y = v_{\mathcal{E}}([r_1, \ldots, r_j]) \neq \overline{0}$.
- $z' = v_{\mathcal{E}_A}([r_1, \ldots, r_{j-1}]) \neq \overline{0}_A$.
- $z = v_{\mathcal{E}_A}([r_1, \ldots, r_j]) \neq \overline{0}_A$.
- q' be the state of A after making $j - 1$ transitions.
- q be the state of A after making j transitions.

Because $\mathcal{E}_A$ and $\mathcal{E}$ have the same underlying PDS, we know that any run of $\mathcal{E}$ is a run of $\mathcal{E}_A$. We now need to show that for each $q_0 \xrightarrow{y'} q' \in z'$ such that $y' \neq \overline{0}$, we have $q_0 \xrightarrow{y} q \in z$. The fact that the weight on the weighted relation is y follows from Lem. 12. The existence of $q_0 \xrightarrow{y} q \in z$ follows from the fact that weighted relations are generalizations of relations. Thus, the proof for Lem. 11 applies.
□

Safety Query To verify that $\mathcal{E}$ adheres to the property specified by A, we slightly modify Eqn. (5):

$$L^{\varphi_{err}}(\mathcal{E}_A) = \emptyset, \text{where } \varphi_{err}(z) = q_0 \xrightarrow{y} q_{err} \in z \wedge y \neq \overline{0} \tag{11}$$

Similar to Eqn. (5), we are only interested in nested words that drive A to the error state q_{err}. However, Eqn. (11) modifies the function φ_{err} from Eqn. (5) by also taking into account the weighted valuation of a run. Again, we can solve Eqn. (11) by computing a reachability query on $\mathcal{E}_A$ as follows: $\text{MOVP}(\{\langle p_0, n_{\text{main}}\rangle\}, \mathcal{U})(q_0, q_{err}) = \overline{0}$, where $\mathcal{U} = \{\langle p, S\rangle \mid p \in P, S \in \Gamma^*\}$.

Notice that in solving the safety query, we have computed more information than simple reachability—we have computed the set of abstract states that the program can be in when the propery specification is violated. That is, $y = \text{MOVP}(\{\langle p_0, n_{\text{main}}\rangle\}, \mathcal{U})(q_0, q_{err})$ is a weight from the original EWPDS abstraction. By returning z to the program analyst (or client analyzer), we are able to provide more information than had we only computed reachability.

8.3 Observations

Observe that when combining $\mathcal{E}$ and A, one might be inclined to use a "paired" weight domain for $\mathcal{E}_A$. A paired weight domain would have weights of the form (d, w), where d models A and w is a rule's weight from $\mathcal{E}$. The problem with this weight domain is that it loses the correlation between the runs. Consider two runs ρ_1 and ρ_2 such that

- $c \Rightarrow^{\rho_1} c'$ and $c \Rightarrow^{\rho_2} c'$
- $nw_1 = NWofRun(\rho_1) \in L(A, q_{err})$ and $y_1 = v_{\mathcal{E}}(\rho_1) = \overline{0}$
- $nw_2 = NWofRun(\rho_2) \notin L(A)$, and $y_2 = v_{\mathcal{E}}(\rho_2) \neq \overline{0}$

Notice that neither nw_1 nor nw_2 are in $L(A) \cap L(\mathcal{E})$. However, the paired weight domain would cause an EWPDS reachability query to overapproximate the result of the intersection. This is because an EWPDS reachability query coalesces runs that reach the same configuration. The weight computed by $\mathcal{E}_A$ at configuration c' for runs ρ_1 and ρ_2 would be: $(q_0 \rightarrow q_{err}, \overline{0}) \oplus (\emptyset, y_2) = (q_0 \rightarrow q_{err}, y_2)$. Thus, $\mathcal{E}_A$ would incorrectly (though soundly) state that there exists a run of $\mathcal{E}$ that drives A to the error state.

In contrast, with weighted relations, the valuation of ρ_1 and ρ_2 by $\mathcal{E}_A$ maintains the correlation. Specifically, $q_0 \xrightarrow{\overline{0}} q_{err} \in v_{\mathcal{E}_A}(\rho_1)$ and $q_0 \xrightarrow{\overline{0}} q_{err} \in v_{\mathcal{E}_A}(\rho_2)$. Thus, using weighted relations allows $\mathcal{E}_A$ to simultaneously model both $\mathcal{E}$ and A.

9 Related Work

We presented three constructions for property checking: *Explicit NWA plus PDS*, *Symbolic NWA plus PDS*, and *Symbolic NWA plus EWPDS*, where each construction is an extension of the former. Compared to standard approaches to property checking, *Symbolic NWA plus EWPDS* allows one to simultaneously check properties

1. stated in a more expressive specification language
2. on program models that support more powerful abstractions
3. while furnishing a broader range of diagnostic information when property violations are detected.

BLAST [18] and SLAM [19] proved the feasibility of performing property checking on software. They model a program as a Boolean program, and the property is specified by a finite automaton. Thus, they do not support items 1 and 2.

Alur and Madhusudan [5] propose that both the program and the property should be expressed as NWAs. While this provides for item 1, NWAs do not support item 2. Furthermore, it is not clear how to accomplish item 3 without first transforming an NWA into an EWPDS.

Visibly pushdown automata [20] are another formalism that has been proposed for property checking of programs. According to Alur and Madhusudan, they exhibit the same properties as NWAs [5], and thus do not support items 2 and 3.

References

1. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: CAV, London, UK, Springer-Verlag (1997) 72–83
2. Müller-Olm, M., Seidl, H.: Precise interprocedural analysis through linear algebra. In: POPL. (2004)
3. Reps, T., Schwoon, S., Jha, S., Melski, D.: Weighted pushdown systems and their application to interprocedural dataflow analysis. SCP (2005)
4. Bouajjani, A., Esparza, J., Touili, T.: A generic approach to the static analysis of concurrent programs with procedures. In: POPL. (2003)
5. Alur, R., Madhusudan, P.: Adding nesting structure to words. In: DLT. (2006)
6. Chaudhuri, S., Alur, R.: Instrumenting C programs with nested word monitors. In: SPIN. (2007)
7. Lal, A., Kidd, N., Reps, T., Touili, T.: Abstract error projection. In: SAS. (2007)
8. Lal, A., Reps, T., Balakrishnan, G.: Extended weighted pushdown systems. In: CAV. (2005)
9. Lal, A., Touili, T., Kidd, N., Reps, T.: Interprocedural analysis of concurrent programs under a context bound. Technical Report TR-1598, Univ. of Wisconsin (July 2007)
10. Schwoon, S.: Model-Checking Pushdown Systems. PhD thesis, TUM (2002)
11. Bouajjani, A., Esparza, J., Maler, O.: Reachability analysis of pushdown automata: Application to model-checking. In: Proc. CONCUR. (1997)
12. Büchi, J.: Finite Automata, their Algebras and Grammars. Springer-Verlag (1988) D. Siefkes (ed.).
13. Esparza, J., Hansel, D., Rossmanith, P., Schwoon, S.: Efficient algorithms for model checking pushdown systems. In: CAV. (2000)
14. Finkel, A., B.Willems, Wolper, P.: A direct symbolic approach to model checking pushdown systems. Elec. Notes in Theor. Comp. Sci. (1997)
15. Knoop, J., Steffen, B.: The interprocedural coincidence theorem. In: CC. (1992)
16. Bryant, R.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. on Comp. (1986)
17. Bahar, R., Frohm, E., Gaona, C., Hachtel, G., Macii, E., Pardo, A., Somenzi, F.: Algebraic decision diagrams and their applications. In: CAD. (1993)
18. Henzinger, T., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL. (2002)
19. Ball, T., Rajamani, S.: Automatically validating temporal safety properties of interfaces. In: SPIN. (2001)
20. Alur, R., Madhusudan, P.: Visibly pushdown languages. In: STOC. (2004)