A Decision Procedure for Detecting Atomicity Violations for Communicating Processes with Locks

Nicholas Kidd[1], Peter Lammich[2], Tayssir Touili[3], and Thomas Reps[1,4]

[1] University of Wisconsin, {kidd,reps}@cs.wisc.edu
[2] Westfälische Wilhelms-Universität Münster, peter.lammich@uni-muenster.de
[3] LIAFA, CNRS & Université Paris Diderot, touili@liafa.jussieu.fr
[4] GrammaTech, Inc.

Abstract. We present a new decision procedure for detecting property violations in pushdown models for concurrent programs that use lock-based synchronization, where each thread's lock operations are properly nested (à la synchronized methods in Java). The technique detects violations expressed as *indexed phase automata* (PAs)—a class of non-deterministic finite automata in which the only loops are self-loops.

Our interest in PAs stems from their ability to capture atomic-set serializability violations. (Atomic-set serializability is a relaxation of atomicity to only a user-specified set of memory locations.) We implemented the decision procedure and applied it to detecting atomic-set-serializability violations in models of concurrent Java programs. Compared with a prior method based on a semi-decision procedure, not only was the decision procedure 7.5X faster overall, but the semi-decision procedure timed out on about 68% of the queries versus 4% for the decision procedure.

1 Introduction

Pushdown systems (PDSs) are a formalism for modeling the interprocedural control flow of recursive programs. Likewise, *multi*-PDSs have been used to model the set of all interleaved executions of a concurrent program with a finite number of threads [1–7]. This paper presents a decision procedure for multi-PDS model checking with respect to properties expressed as *indexed phase automata* (PAs)—a class of non-deterministic finite automata in which the only loops are self-loops. The decision procedure handles (i) reentrant locks, (ii) an *unbounded* number of context switches, and (iii) an *unbounded* number of lock acquisitions and releases by each PDS. The decision procedure is *compositional*: each PDS is analyzed independently with respect to the PA, and then a single compatibility check is performed that ties together the results obtained from the different PDSs.

Our interest in PAs stems from their ability to capture *atomic-set serializability* (AS-serializability) violations. AS-serializability was proposed by Vaziri et al. [8] as a relaxation of the atomicity property [9] to only a user specified set of fields of an object. (A detailed example is given in §2.) In previous work by some of the authors [10], we developed techniques for verifying AS-serializability for concurrent Java programs. Our tool first abstracts a concurrent Java program into EML, a modeling language based on multi-PDSs and a finite number of *reentrant* locks. The drawback of the approach that we have used to date is that an EML program is compiled into a communicating pushdown system (CPDS) [4, 5], for which the required model-checking problem is undecidable. (A semi-decision procedure is used in [10].)

Kahlon and Gupta [7] explored the boundary between decidability and undecidability for model checking multi-PDSs that synchronize via nested locks. One of their results is an algorithm to decide if a multi-PDS satisfies an (indexed) LTL formula that makes use of only atomic propositions, the “next” operator X, and the “eventually” operator F. In the case of a 2-PDS, the algorithm uses an automaton-pair $M = (A, B)$ to represent a set of configurations of a 2-PDS, where an automaton encodes the configurations of a single PDS in the usual way [11, 12]. For a given logical formula, the Kahlon-Gupta algorithm is defined inductively: from an automaton-pair that satisfies a subformula, they define an algorithm that computes a new automaton-pair for a larger formula that has one additional (outermost) temporal operator.

We observed that PAs can be compiled into an LTL formula that uses only the X and F operators. (App. A presents an algorithm that performs the encoding.) Furthermore, [13] presents a sound and precise technique that uses only *non-reentrant* locks to model EML's reentrant locks. Thus, combining previous work [10, 13] with the Kahlon-Gupta algorithm provides a decision procedure for verifying AS-serializability of concurrent Java programs!

(Briefly, the technique for replacing reentrant locks with non-reentrant locks pushes a special marker onto the stack the *first* time a lock is acquired, and records the acquisition in a PDS's state space. All subsequent lock acquires and their matching releases do not change the state of the lock or the PDS. Only when the special marker is seen again is the lock then released. This technique requires that lock acquisition and releases be properly scoped, which is satisfied by Java's synchronized blocks. Consequently, we consider only non-reentrant locks in the remainder of the paper.)

Unfortunately, [7] erroneously claims that the disjunction operation distributes across two automaton-pairs. That is, for automaton-pairs $M_1 = (A_1, B_1)$ and $M_2 = (A_2, B_2)$, they claim that the following holds: $M_1 \vee M_2 = (A_1 \vee A_2, B_1 \vee B_2)$. This is invalid because *cross-terms* arise when attempting to distribute the disjunct. For example, if $B_1 \cap B_2 = \emptyset$, then there can be configurations of the form $\langle a_1 \in A_1, b_2 \in B_2 \rangle$ that would be accepted by $(A_1 \vee A_2, B_1 \vee B_2)$ but should not be in $M_1 \vee M_2$.

To handle this issue properly, a corrected algorithm must use a *set* of automaton-pairs instead of single automaton-pair to represent a set of configurations of a 2-PDS.[5] Because the size of the set is exponential in the number of locks, in the worst case, their algorithm may perform an exponential number of individual reachability queries to handle one temporal operator. Furthermore,

[5] We confirmed these observations in e-mail with Kahlon and Gupta [14].

once reachability from one automaton-pair has been performed, the resulting automaton pair must be split into a set so that incompatible configurations are eliminated. Thus, it is not immediately clear if the (corrected) Kahlon-Gupta algorithm is amenable to an implementation that would be usable in practice.

This paper presents a new decision procedure for checking properties specified as PAs on multi-PDSs that synchronize via nested locks.[6] Unlike the (corrected) Kahlon-Gupta algorithm, our decision procedure uses only *one* reachability query per PDS. The key is to use *tuples* of lock histories (§5): moving from the lock histories used by Kahlon and Gupta to tuples-of-lock histories introduces a mechanism to maintain the correlations between the intermediate configurations. Hence, our decision procedure is able to make use of only a *single* compatibility check over the tuples-of-lock histories that our analysis obtains for each PDS. The benefit of this approach is shown in the following table, where Procs denotes the number of processes, $\mathcal{L}$ denotes the number of locks, and $|\mathsf{PA}|$ denotes the number of states in property automaton PA:

	PDS State Space	Queries
Kahlon-Gupta [7] (corrected)	$O(2^{\mathcal{L}})$	$O(\lvert\mathsf{PA}\rvert \cdot \mathsf{Procs} \cdot 2^{\mathcal{L}})$
This paper (§6)	$O(\lvert\mathsf{PA}\rvert \cdot 2^{\mathcal{L}})$	Procs

Because our algorithm isolates the exponential cost in the PDS state space, that cost can often be side-stepped using symbolic techniques, such as BDDs, as explained in §7.

This paper makes the following contributions:

- We define a decision procedure for multi-PDS model-checking for PAs. The decision procedure handles (i) reentrant locks, (ii) an *unbounded* number of context switches, (iii) an *unbounded* number of lock acquisitions and releases by each PDS, and (iv) a bounded number of phase transitions.
- The decision procedure is *compositional*: each PDS is analyzed independently with respect to the PA, and then a single compatibility check is performed that ties together the results obtained from the different PDSs.
- We leverage the special form of PAs to give a symbolic implementation that is more space-efficient than standard BDD-based techniques for PDSs [15].
- We used the decision procedure to detect AS-serializability violations in automatically-generated models of four programs from the ConTest benchmark suite [16], and obtained substantially better performance than a prior method based on a semi-decision procedure [10].

The rest of the paper is organized as follows: §2 provides motivation. §3 defines multi-PDSs and PAs. §4 reviews Kahlon and Gupta's decomposition result. §5 presents lock histories. §6 presents the decision procedure. §7 describes a symbolic implementation. §8 presents experimental results. §9 describes related work. Apps. A, B, and C contain additional material; for correctness proofs of Thms. 2 and 3, see [17].

[6] We do not consider multi-PDSs that use wait-notify synchronization because reachability analysis of multi-PDSs with wait-notify is undecidable [7].

```java
class Stack {
  Object[] storage = new Object[10];
  int item = -1;
  public static Stack makeStack(){
    return new Stack();
  }
  public synchronized Object pop(){
    Object res = storage[item];
    storage[item--] = null;
    return res;
  }
  public synchronized void push(Object o){
    storage[++item] = o;
  }
  public synchronized boolean empty(){
    return (item == -1);
  }
}
```

```java
class Client {
  // @atomic
  public synchronized Object get(Stack s){
    if(!s.empty()) { return s.pop(); }
    else return null;
  }
  public static Client makeClient(){
    return new Client();
  }
  public static void main(String[] args){
    Stack stack = Stack.makeStack();
    stack.push(new Integer(1));
    Client client1 = makeClient();
    Client client2 = makeClient();
    new Thread("1") { client1.get(stack); }
    new Thread("2") { client2.get(stack); }
  }
}
```

$$1\colon\ \overbrace{\underline{Abeg_1}\,\overbrace{(_{\mathsf{s}}\underline{R_1(i)})_{\mathsf{s}}}^{\texttt{empty()}}\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ \overbrace{(_{\mathsf{s}}R_1(i)\underline{R_1(s)}R_1(i)W_1(s)W_1(i))_{\mathsf{s}}}^{\texttt{pop()}}\,Aend_1}^{\texttt{get()}}$$

$$2\colon\ \ldots\ldots\ \overbrace{Abeg_2\,\underbrace{(_{\mathsf{s}}R_2(i))_{\mathsf{s}}}_{\texttt{empty()}}\,\underbrace{(_{\mathsf{s}}R_2(i)R_2(s)R_2(i)\underline{W_2(s)W_2(i)})_{\mathsf{s}}}_{\texttt{pop()}}\,Aend_2}^{\texttt{get()}}\ \ldots\ldots\ldots\ldots\ldots\ldots$$

Fig. 1. Example program and problematic interleaving that violates atomic-set serializability. R and W denote a read and write access, respectively. i and s denote fields `item` and `storage`, respectively. *Abeg* and *Aend* denote the beginning and end, respectively, of an atomic code block. The subscripts "1" and "2" are thread ids. "$(_{\mathsf{s}}$" and "$)_{\mathsf{s}}$" denote the acquire and release operations, respectively, of the lock of `Stack stack`.

2 Motivating Example

Fig. 1 shows a simple Java implementation of a stack. Class `Client` is a test harness that performs concurrent accesses on a single stack. `Client.get()` uses the keyword "`synchronized`" to protect against concurrent calls on the same `Client` object. The annotation "`@atomic`" on `Client.get()` specifies that the programmer intends for `Client.get()` to be executed atomically.

The program's synchronization actions do not ensure this, however. The root cause is that the wrong object is used for synchronization: parameter "`Stack s`" of `Client.get()` should have been used, instead of `Client.get()`'s implicit `this` parameter. This mistake permits the interleaved execution shown at the bottom of Fig. 1, which would result in an exception being thrown.

This is an example of an *atomic-set serializability* (AS-serializability)—a relaxation of serializability to only a specified set of shared-memory locations—violation [8] with respect to `s.item` and `s.storage`. AS-serializability violations can be completely characterized by a set of fourteen problematic access patterns [8].[7] Each problematic pattern is a finite sequence of reads and writes by two

[7] This result relies on an assumption that programs do not always satisfy: an atomic code section that writes to one member of a set of correlated locations writes to all locations in that set (e.g., `item` and `storage` of `Stack s`).

threads to one or two shared memory locations. For the program in Fig. 1 and problematic pattern "$Abeg_1; R_1(i); W_2(s); W_2(i); R_1(s)$", the accesses that match the pattern are underlined in the interleaving shown at the bottom of Fig. 1.

The fourteen problematic access patterns can be encoded as an *indexed phase automaton* (PA). The PA that captures the problematic accesses of Fig. 1 is shown in Fig. 2. Its states—which represent the *phases* that the automaton passes through to accept a string—are chained together by *phase transitions*; each state has a self-loop for symbols that cause the automaton to not change state. ("Indexed" refers to the fact that the index of the thread performing an action is included in the label of each transition.)

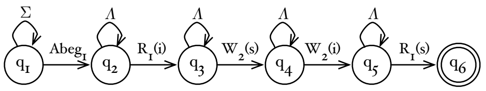

Fig. 2. The PA that accepts the problematic access pattern in the program from Fig. 1. Σ is the set of all actions, and Λ is $\Sigma \setminus \{Aend_1\}$.

The PA in Fig. 2 "guesses" when a violation occurs. That is, when it observes that thread 1 enters an atomic code block, such as `get()`, the atomic-code-block-begin action $Abeg_1$ causes it either to transition to state q_2 (i.e., to start the next phase), or to follow the self-loop and remain in q_1. This process continues until it reaches the accepting state. Note that the only transition that allows thread 1 to exit an atomic code block ($Aend_1$) is the self-loop on the initial state. Thus, incorrect guesses cause the PA in Fig. 2 to become "stuck" in one of the states $q_1 \ldots q_5$ and not reach final state q_6.

3 Program Model and Property Specifications

Definition 1. *A* **(labeled) pushdown system** *(PDS) is a tuple* $\mathcal{P} = (P, Act, \Gamma, \Delta, c_0)$*, where* P *is a finite set of control states,* Act *is a finite set of actions,* Γ *is a finite stack alphabet, and* $\Delta \subseteq (P \times \Gamma) \times Act \times (P \times \Gamma^*)$ *is a finite set of rules. A rule* $r \in \Delta$ *is denoted by* $\langle p, \gamma \rangle \overset{a}{\hookrightarrow} \langle p', u' \rangle$*. A PDS* **configuration** $\langle p, u \rangle$ *is a control state along with a stack, where* $p \in P$ *and* $u \in \Gamma^*$*, and* $c_0 = \langle p_0, \gamma_0 \rangle$ *is the initial configuration.* Δ *defines a transition system over the set of all configurations. From* $c = \langle p, \gamma u \rangle$*,* $\mathcal{P}$ *can make a transition to* $c' = \langle p', u'u \rangle$ *on action* a*, denoted by* $c \xrightarrow{a} c'$*, if there exists a rule* $\langle p, \gamma \rangle \overset{a}{\hookrightarrow} \langle p', u' \rangle \in \Delta$*. For* $w \in Act^*$*,* $c \xrightarrow{w} c'$ *is defined in the usual way. For a rule* $r = \langle p, \gamma \rangle \overset{a}{\hookrightarrow} \langle p', u' \rangle$*,* $\mathsf{act}(r)$ *denotes* r*'s action label* a*.*

A *multi-PDS* consists of a finite number of PDSs $\mathcal{P}_1, \ldots, \mathcal{P}_n$ that synchronize via a finite set of locks $\mathsf{Locks} = \{l_1, \ldots, l_{\mathcal{L}}\}$ (i.e., $\mathcal{L} = |\mathsf{Locks}|$). The actions Act of each PDS consist of lock-acquires ("$(_i$") and releases ("$)_i$") for $1 \leq i \leq \mathcal{L}$, plus symbols from Σ, a finite alphabet of non-parenthesis symbols.

The intention is that each PDS models a thread, and lock-acquire and release actions serve as synchronization primitives that constrain the behavior of the multi-PDS. We assume that locks are acquired and released in a well-nested fashion; i.e., locks are released in the opposite order in which they are acquired.

The choice of what actions appear in Σ depends on the intended application. For verifying AS-serializability (see §2 and §7), Σ consists of actions to read and write a shared-memory location m (denoted by $R(m)$ and $W(m)$, respectively), and to enter and exit an atomic code section ($Abeg$ and $Aend$, respectively).

Formally, a program model is a tuple $\Pi = (\mathcal{P}_1, \ldots, \mathcal{P}_n, \mathsf{Locks}, \Sigma)$. A *global configuration* $g = (c_1, \ldots, c_n, o_1, \ldots, o_{\mathcal{L}})$ is a tuple consisting of a configuration for each PDS and a valuation that indicates the owner of each lock: for each $1 \leq i \leq \mathcal{L}$, $o_i \in \{\bot, 1, \ldots, n\}$ indicates the identity of the PDS that holds lock l_i. The value $\bot$ signifies that a lock is currently not held by any PDS. The initial global configuration is $g_0 = (c_0^1, \ldots, c_0^n, \bot, \ldots, \bot)$. A global configuration $g = (c_1, c_2, \ldots, c_n, o_1, \ldots, o_{\mathcal{L}})$ can make a transition to another global configuration $g' = (c_1', c_2, \ldots, c_n, o_1', \ldots, o_{\mathcal{L}}')$ under the following conditions:

- If $c_1 \xrightarrow{a} c_1'$ and $a \notin \{(_i,)_i\}$, then $g' = (c_1', c_2, \ldots, c_n, o_1, \ldots, o_{\mathcal{L}})$.
- If $c_1 \xrightarrow{(_i} c_1'$ and $g = (c_1, c_2, \ldots, c_n, o_1, \ldots, o_{i-1}, \bot, o_{i+1}, \ldots, o_{\mathcal{L}})$, then $g' = (c_1', c_2, \ldots, c_n, o_1, \ldots, o_{i-1}, 1, o_{i+1}, \ldots, o_{\mathcal{L}})$.
- If $c_1 \xrightarrow{)_i} c_1'$ and $g = (c_1, c_2, \ldots, c_n, o_1, \ldots, o_{i-1}, 1, o_{i+1}, \ldots, o_{\mathcal{L}})$, then $g' = (c_1', c_2, \ldots, c_n, o_1, \ldots, o_{i-1}, \bot, o_{i+1}, \ldots, o_{\mathcal{L}})$.

For $1 < j \leq n$, a global configuration $(c_1, \ldots, c_j, \ldots, c_n, o_1, \ldots, o_{\mathcal{L}})$ can make a transition to a global configuration $(c_1, \ldots, c_j', \ldots, c_n, o_1', \ldots, o_{\mathcal{L}}')$ in a similar fashion.

A program property is specified as an indexed phase automaton, defined as follows:

Definition 2. *An* **indexed phase automaton** *(PA) is a tuple (Q, Id, Σ, δ), where Q is a finite, totally ordered set of states $\{q_1, \ldots, q_{|Q|}\}$, Id is a finite set of thread identifiers, Σ is a finite alphabet, and $\delta \subseteq Q \times Id \times \Sigma \times Q$ is a transition relation. The transition relation δ is restricted to respect the order on states: for each transition $(q_x, i, a, q_y) \in \delta$, either $y = x$ or $y = x + 1$. We call a transition of the form (q_x, i, a, q_{x+1}) a* **phase transition**. *The initial state is q_1, and the final state is $q_{|Q|}$.*

The restriction on δ in Defn. 2 ensures that the only loops in a PA are "self-loops" on states. We assume that there is only one phase transition of the form $(q_x, i, a, q_{x+1}) \in \delta$, $0 < x < |Q|$. (A PA that has multiple such transitions can be factored into a set of PAs, each of which satisfy this property.) Finally, we only consider PAs that recognize a non-empty languauge, which means that a PA must have exactly $(|Q| - 1)$ phase transitions.

For the rest of the paper we consider 2-PDSs, and fix $\Pi = (\mathcal{P}_1, \mathcal{P}_2, \mathsf{Locks}, \Sigma)$ and $\mathcal{A} = (Q, Id, \Sigma, \delta)$; however, the techniques extend easily to any finite number of PDSs, and our implementation handles multiple PDSs (see App. B). Given Π and $\mathcal{A}$, the model-checking problem of interest is to determine if there is an execution that begins at the initial global configuration g_0 that drives $\mathcal{A}$ to its accepting state.

4 Path Incompatibility

The decision procedure analyzes the PDSs of Π independently, and then checks if two runs exist that can be performed in interleaved parallel fashion under the lock-constrained transitions of Π. To do this, it makes use of a decomposition result, due to Kahlon and Gupta [7, Thm. 1], which we now review.

Suppose that PDS $\mathcal{P}_k$, for $k \in \{1, 2\}$, when started in (single-PDS) configuration c_k and executed alone, is able to reach configuration c'_k using the rule sequence ρ_k. Let $\text{LocksHeld}(\mathcal{P}_k, (b_1, b_2, o_1, \ldots, o_{\mathcal{L}}))$ denote $\{l_i \mid o_i = k\}$; i.e., the set of locks held by PDS $\mathcal{P}_k$ at global configuration $(b_1, b_2, o_1, \ldots, o_{\mathcal{L}})$.

Along a rule sequence ρ_k and for an initially-held lock l_i and finally-held lock l_f, we say that the *initial release* of l_i is the first release of l_i, and that the *final acquisition* of l_f is the last acquisition of l_f. Note that for execution to proceed along ρ_k, $\mathcal{P}_k$ must hold an initial set of locks at c_k that is a superset of the set of initial releases along ρ_k; i.e., not all initially-held locks need be released. Similarly, $\mathcal{P}_k$'s final set of locks at c'_k must be a superset of the set of final acquisitions along ρ_k.

Theorem 1. *(**Decomposition Theorem** [7].) Suppose that PDS $\mathcal{P}_k$, when started in configuration c_k and executed alone, is able to reach configuration c'_k using the rule sequence ρ_k. For $\Pi = (\mathcal{P}_1, \mathcal{P}_2, \mathsf{Locks}, \Sigma)$, there does not exist an interleaving of paths ρ_1 and ρ_2 from global configuration $(c_1, c_2, o_1, \ldots, o_{\mathcal{L}})$ to global configuration $(c'_1, c'_2, o'_1, \ldots, o'_{\mathcal{L}})$ iff one or more of the following five conditions hold:*

1. *$\mathit{LocksHeld}(\mathcal{P}_1, (c_1, c_2, o_1, \ldots, o_{\mathcal{L}})) \cap \mathit{LocksHeld}(\mathcal{P}_2, (c_1, c_2, o_1, \ldots, o_{\mathcal{L}})) \neq \emptyset$*
2. *$\mathit{LocksHeld}(\mathcal{P}_1, (c'_1, c'_2, o'_1, \ldots, o'_{\mathcal{L}})) \cap \mathit{LocksHeld}(\mathcal{P}_2, (c'_1, c'_2, o'_1, \ldots, o'_{\mathcal{L}})) \neq \emptyset$*
3. *In ρ_1, $\mathcal{P}_1$ releases lock l_i before it initially releases lock l_j, and in ρ_2, $\mathcal{P}_2$ releases l_j before it initially releases lock l_i.*
4. *In ρ_1, $\mathcal{P}_1$ acquires lock l_i after its final acquisition of lock l_j, and in ρ_2, $\mathcal{P}_2$ acquires lock l_j after its final acquisition of lock l_i,*
5. (a) *In ρ_1, $\mathcal{P}_1$ acquires or uses a lock that is held by $\mathcal{P}_2$ throughout ρ_2, or*
   (b) *in ρ_2, $\mathcal{P}_2$ acquires or uses a lock that is held by $\mathcal{P}_1$ throughout ρ_1.*

Intuitively, items 3 and 4 capture *cycles* in the dependence graph of lock operations. That is, when either item is satisfied, a deadlock must occur for all possible interleavings of ρ_1 and ρ_2, and thus no such interleaving can exist. The remaining items model standard lock semantics: only one thread may hold a lock at a given time.

5 Extracting Information from PDS Rule Sequences

To employ Thm. 1, we now develop methods to extract relevant information from a rule sequence ρ_k for PDS $\mathcal{P}_k$. As in many program-analysis problems that involve matched operations [18]—in our case, lock-acquire and lock-release—it is useful to consider *semi-Dyck languages* [19]: languages of matched parentheses in which each parenthesis symbol is *one-sided*. That is, the symbols "(" and ")" match in the string "()", but do not match in ")(".[8]

Let Σ be a finite alphabet of non-parenthesis symbols; The semi-Dyck language of well-balanced parentheses over $\Sigma \cup \{(_i,)_i \mid 1 \leq i \leq \mathcal{L}\}$ can be defined by the following context-free grammar, where e denotes a member of Σ:

$$\textit{matched} \;\rightarrow\; \epsilon \;\mid\; e \;\textit{matched} \;\mid\; (_i \;\textit{matched}\;)_i \;\textit{matched} \quad \text{[for } 1 \leq i \leq \mathcal{L}]$$

Because we are interested in paths that can begin and end while holding a set of locks, we define the following partially-matched parenthesis languages:

$$\textit{unbalR} \;\rightarrow\; \epsilon \mid \textit{unbalR} \;\; \textit{matched} \;\;)_i \qquad\qquad \textit{unbalL} \;\rightarrow\; \epsilon \mid (_i \;\; \textit{matched} \;\; \textit{unbalL}$$

The language of words that are possibly unbalanced on each end is defined by

$$\textit{suffixPrefix} \;\rightarrow\; \textit{unbalR} \;\; \textit{matched} \;\; \textit{unbalL}$$

Example 1. Consider the following *suffixPrefix* string, in which the positions between symbols are marked A–W. Its *unbalR*, *matched*, and *unbalL* components are the substrings A–N, N–P, and P–W, respectively.

$$\underset{\widehat{\mathrm{A}}}{}\,)_1\,\underset{\widehat{\mathrm{B}}}{}\,(_2\,\underset{\widehat{\mathrm{C}}}{}\,)_2\,\underset{\widehat{\mathrm{D}}}{}\,)_3\,\underset{\widehat{\mathrm{E}}}{}\,(_2\,\underset{\widehat{\mathrm{F}}}{}\,(_4\,\underset{\widehat{\mathrm{G}}}{}\,(_5\,\underset{\widehat{\mathrm{H}}}{}\,)_5\,\underset{\widehat{\mathrm{I}}}{}\,)_4\,\underset{\widehat{\mathrm{J}}}{}\,(_6\,\underset{\widehat{\mathrm{K}}}{}\,)_6\,\underset{\widehat{\mathrm{L}}}{}\,)_2\,\underset{\widehat{\mathrm{M}}}{}\,)_7\,\underset{\widehat{\mathrm{N}}}{}\,(_6\,\underset{\widehat{\mathrm{O}}}{}\,)_6\,\underset{\widehat{\mathrm{P}}}{}\,(_4\,\underset{\widehat{\mathrm{Q}}}{}\,(_2\,\underset{\widehat{\mathrm{R}}}{}\,)_2\,\underset{\widehat{\mathrm{S}}}{}\,(_2\,\underset{\widehat{\mathrm{T}}}{}\,(_7\,\underset{\widehat{\mathrm{U}}}{}\,)_7\,\underset{\widehat{\mathrm{V}}}{}\,(_8\,\underset{\widehat{\mathrm{W}}}{}$$

Let $w_k \in L(\textit{suffixPrefix})$ be the word formed by concatenating the action symbols of the rule sequence ρ_k. One can see that to use Thm. 1, we merely need to extract the relevant information from w_k. That is, items 3 and 4 require extracting (or recording) information from the *unbalR* and *unbalL* portions of w_k, respectively; item 5 requires extracting information from the *matched* portion of w_k; and items 1 and 2 require extracting information from the initial and final parse configurations of w_k.

The information is obtained using acquisition histories (AH) and release histories (RH) for locks, as well as ρ_k's release set (R), use set (U), acquisition set (A), and held-throughout set (HT).

- The *acquisition history* (AH) [7] for a finally-held lock l_i is the union of the set $\{l_i\}$ with the set of locks that are acquired (or acquired and released) after the final acquisition of l_i.[9]
- The *release history* (RH) [7] of an initially-held lock l_i is the union of the set $\{l_i\}$ with the set of locks that are released (or acquired and released) before the initial release of l_i.
- The *release set* (R) is the set of initially-released locks.
- The *use set* (U) is the set of locks that form the *matched* part of w_k.
- The *acquisition set* (A) is the set of finally-acquired locks.
- The *held-throughout set* (HT) is the set of initially-held locks that are not released.

[8] The language of interest is in fact regular because the locks are non-reentrant. However, the semi-Dyck formulation provides insight into how one extracts the relevant information from a rule sequence.

[9] This is a slight variation from [7]; we include l_i in the acquisition history of lock l_i.

A *lock history* is a six-tuple $(\mathsf{R}, \widehat{\mathsf{RH}}, \mathsf{U}, \widehat{\mathsf{AH}}, \mathsf{A}, \mathsf{HT})$, where R, U, A, and HT are the sets defined above, and $\widehat{\mathsf{AH}}$ ($\widehat{\mathsf{RH}}$) is a tuple of $\mathcal{L}$ acquisition (release) histories, one for each lock l_i, $1 \leq i \leq \mathcal{L}$. Let $\rho = [r_1, \ldots, r_n]$ be a rule sequence that drives a PDS from some starting configuration to an ending configuration, and let $\mathcal{I}$ be the set of locks held at the beginning of ρ. We define an abstraction function $\eta(\rho, \mathcal{I})$ from rule sequences and initially-held locks to lock histories; $\eta(\rho, \mathcal{I})$ uses an auxiliary function, post, which tracks R, $\widehat{\mathsf{RH}}$, U, $\widehat{\mathsf{AH}}$, A, and HT for each successively longer prefix.

$$\begin{aligned} \eta([], \mathcal{I}) &= (\emptyset, \emptyset^{\mathcal{L}}, \emptyset, \emptyset^{\mathcal{L}}, \emptyset, \mathcal{I}) \\ \eta([r_1, \ldots, r_n], \mathcal{I}) &= \mathsf{post}(\eta([r_1, \ldots, r_{n-1}], \mathcal{I}), \mathsf{act}(r_n)), \text{ where} \end{aligned}$$

$$\mathsf{post}((\mathsf{R}, \widehat{\mathsf{RH}}, \mathsf{U}, \widehat{\mathsf{AH}}, \mathsf{A}, \mathsf{HT}), a) =$$
$$\begin{cases} (\mathsf{R}, \widehat{\mathsf{RH}}, \mathsf{U}, \widehat{\mathsf{AH}}, \mathsf{A}, \mathsf{HT}) & \text{if } a \notin \{(_i,)_i\} \\ (\mathsf{R}, \widehat{\mathsf{RH}}, \mathsf{U}, \widehat{\mathsf{AH}}', \mathsf{A} \cup \{l_i\}, \mathsf{HT}) & \text{if } a = (_i \\ \quad \text{where } \widehat{\mathsf{AH}}'[j] = \begin{cases} \{l_i\} & \text{if } j = i \\ \emptyset & \text{if } j \neq i \text{ and } l_j \notin \mathsf{A} \\ \widehat{\mathsf{AH}}[j] \cup \{l_i\} & \text{if } j \neq i \text{ and } l_j \in \mathsf{A} \end{cases} & \\ (\mathsf{R}, \widehat{\mathsf{RH}}, \mathsf{U} \cup \{l_i\}, \widehat{\mathsf{AH}}', \mathsf{A} \backslash \{l_i\}, \mathsf{HT} \backslash \{l_i\}) & \text{if } a =)_i \text{ and } l_i \in \mathsf{A} \\ \quad \text{where } \widehat{\mathsf{AH}}'[j] = \begin{cases} \emptyset & \text{if } j = i \\ \widehat{\mathsf{AH}}[j] & \text{otherwise} \end{cases} & \\ (\mathsf{R} \cup \{l_i\}, \widehat{\mathsf{RH}}', \mathsf{U}, \widehat{\mathsf{AH}}, \mathsf{A}, \mathsf{HT} \backslash \{l_i\}) & \text{if } a =)_i \text{ and } l_i \notin \mathsf{A} \\ \quad \text{where } \widehat{\mathsf{RH}}'[j] = \begin{cases} \{l_i\} \cup \mathsf{U} \cup \mathsf{R} & \text{if } j = i \\ \widehat{\mathsf{RH}}[j] & \text{otherwise} \end{cases} & \end{cases}$$

Example 2. Suppose that ρ is a rule sequence whose labels spell out the string from Example 1, and $\mathcal{I} = \{1, 3, 7, 9\}$. Then $\eta(\rho, \mathcal{I})$ returns the following lock history (only lock indices are written):

$$\begin{array}{l} (\{1,3,7\},\ \langle\{1\}, \emptyset, \{1,2,3\}, \emptyset, \emptyset, \emptyset, \{1,2,3,4,5,6,7\}, \emptyset, \emptyset\rangle, \\ \ \{6\},\ \langle\emptyset, \{2,7,8\}, \emptyset, \{2,4,7,8\}, \emptyset, \emptyset, \emptyset, \{8\}, \emptyset\rangle,\ \{2,4,8\},\ \{9\}). \end{array}$$

Note: R and A are included above only for clarity; they can be recovered from $\widehat{\mathsf{RH}}$ and $\widehat{\mathsf{AH}}$, as follows: $\mathsf{R} = \{i \mid \widehat{\mathsf{RH}}[i] \neq \emptyset\}$ and $\mathsf{A} = \{i \mid \widehat{\mathsf{AH}}[i] \neq \emptyset\}$. In addition, from $\mathsf{LH} = (\mathsf{R}, \widehat{\mathsf{RH}}, \mathsf{U}, \widehat{\mathsf{AH}}, \mathsf{A}, \mathsf{HT})$, it is easy to see that the set $\mathcal{I}$ of initially-held locks is equal to $(\mathsf{R} \cup \mathsf{HT})$, and the set of finally-held locks is equal to $(\mathsf{A} \cup \mathsf{HT})$.

Definition 3. *Lock histories* $LH_1 = (R_1, \widehat{RH}_1, U_1, \widehat{AH}_1, A_1, HT_1)$ *and* $LH_2 = (R_2, \widehat{RH}_2, U_2, \widehat{AH}_2, A_2, HT_2)$ *are* **compatible***, denoted by* $Compatible(LH_1, LH_2)$*, iff all of the following five conditions hold:*

$$\begin{array}{ll} 1. (R_1 \cup HT_1) \cap (R_2 \cup HT_2) = \emptyset & 2. (A_1 \cup HT_1) \cap (A_2 \cup HT_2) = \emptyset \\ 3.\ \nexists i, j\ .\ l_j \in \widehat{AH}_1[i] \wedge l_i \in \widehat{AH}_2[j] & 4.\ \nexists i, j\ .\ l_j \in \widehat{RH}_1[i] \wedge l_i \in \widehat{RH}_2[j] \end{array}$$
$$5. (A_1 \cup U_1) \cap HT_2 = \emptyset \wedge (A_2 \cup U_2) \cap HT_1 = \emptyset$$

Each conjunct verifies the absence of the corresponding incompatibility condition from Thm. 1: conditions 1 and 2 verify that the initially-held and finally-held locks of ρ_1 and ρ_2 are disjoint, respectively; conditions 3 and 4 verify the absence of cycles in the acquisition and release histories, respectively; and condition 5 verifies that ρ_1 does not use a lock that is held throughout in ρ_2, and vice versa.

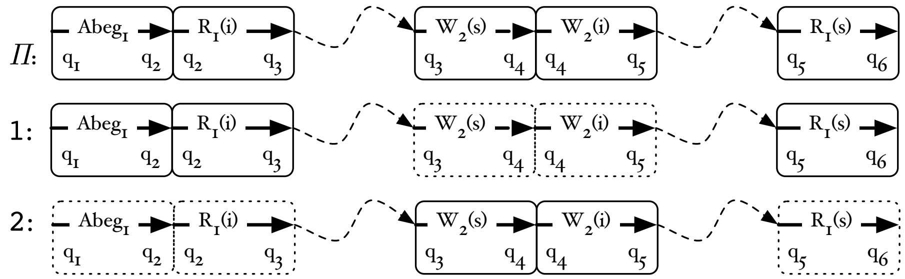

Fig. 3. *Π*: bad interleaving of Fig. 2, showing only the actions that cause a phase transition. 1: the same interleaving from Thread 1's point of view. The dashed boxes show where Thread 1 guesses that Thread 2 causes a phase transition. 2: the same but from Thread 2's point of view and with the appropriate guesses.

6 The Decision Procedure

As noted in §4, the decision procedure analyzes the PDSs independently. This decoupling of the PDSs has two consequences.

First, when $\mathcal{P}_1$ and $\mathcal{A}$ are considered together, independently of $\mathcal{P}_2$, they cannot directly "observe" the actions of $\mathcal{P}_2$ that cause $\mathcal{A}$ to take certain phase transitions. Thus, $\mathcal{P}_1$ must *guess* when $\mathcal{P}_2$ causes a phase transition, and vice versa for $\mathcal{P}_2$. An example of the guessing is shown in Fig. 3. The interleaving labeled "*Π*" is the bad interleaving from Fig. 2, but focuses on only the PDS actions that cause phase transitions. The interleaving labeled "1" shows, via the dashed boxes, where $\mathcal{P}_1$ guesses that $\mathcal{P}_2$ caused a phase transition. Similarly, the interleaving labeled "2" shows the guesses that $\mathcal{P}_2$ must make.

Second, a post-processing step must be performed to ensure that only those behaviors that are consistent with the lock-constrained behaviors of *Π* are considered. For example, if $\mathcal{P}_1$ guesses that $\mathcal{P}_2$ performs the $W_2(s)$ action to make the PA transition from state q_3 to state q_4 (the dashed box for interleaving "1" in Fig. 3) while it is still executing the `empty()` method (see Fig. 2), the behavior is inconsistent with the semantics of *Π*. This is because both threads would hold the lock associated with the shared "`Stack s`" object. The post-processing step ensures that such behaviors are not allowed.

6.1 Combining a PDS with a PA

We now present a construction that combines a PDS $\mathcal{P}$ with a PA $\mathcal{A}$ to form a new PDS $\mathcal{P}^{\mathcal{A}}$. The resulting PDS $\mathcal{P}^{\mathcal{A}}$ is an unlabeled PDS. $\mathcal{LH}$ denotes the set of all lock histories; $\widehat{\mathcal{LH}} = \mathcal{LH}^{|Q|}$ denotes the set of all tuples of lock histories of length $|Q|$. We denote a typical lock history by LH, and a typical tuple of lock histories by $\widehat{\mathsf{LH}}$. $\widehat{\mathsf{LH}}[i]$ denotes the i^{th} component of $\widehat{\mathsf{LH}}$.

The construction uses *tuples* of lock histories because when a phase transition occurs along a path in $\mathcal{P}_i^{\mathcal{A}}$'s transition system, it is necessary to remember the lock history of the path taken during the current phase. A tuple maintains the sequence of lock histories for one or more paths taken through a sequence of phases. These are used later to check whether some scheduling of an execution of Π can actually perform all of the required phase transitions.

Our construction makes use of the phase-transition function on LHs defined as follows: $\mathsf{ptrans}((\mathsf{R}, \widehat{\mathsf{RH}}, \mathsf{U}, \widehat{\mathsf{AH}}, \mathsf{A}, \mathsf{HT})) = (\emptyset, \emptyset^{\mathcal{L}}, \emptyset, \emptyset^{\mathcal{L}}, \emptyset, \mathsf{A} \cup \mathsf{HT})$. This function is used to encode the start of a new phase: the set of initially-held locks is the set of locks held at the end of the previous phase.

Let $\mathcal{P}_i = (P_i, Act_i, \Gamma_i, \Delta_i, \langle p_0, \gamma_0\rangle)$ be a PDS, Locks be a set of locks of size $\mathcal{L}$, $\mathcal{A} = (Q, Id, \Sigma, \delta)$ be a PA, and $\widehat{\mathsf{LH}}$ be a tuple of lock histories of length $|Q|$. We define the PDS $\mathcal{P}_i^{\mathcal{A}} = (P_i^{\mathcal{A}}, \emptyset, \Gamma_i, \Delta_i^{\mathcal{A}}, \langle p_0^{\mathcal{A}}, \gamma_0\rangle)$, where $\mathcal{P}_i^{\mathcal{A}} \subseteq P_i \times Q \times \widehat{\mathcal{LH}}$. The initial control state is $p_0^{\mathcal{A}} = (p_0, q_1, \widehat{\mathsf{LH}}_\emptyset)$, where $\widehat{\mathsf{LH}}_\emptyset$ is the lock-history tuple $(\emptyset, \emptyset^{\mathcal{L}}, \emptyset, \emptyset^{\mathcal{L}}, \emptyset, \emptyset)^{|Q|}$. Each rule $r \in \Delta_i^{\mathcal{A}}$ performs only a *single* update to the tuple $\widehat{\mathsf{LH}}$, at an index x determined by a transition in δ. The update is denoted by $\widehat{\mathsf{LH}}[x \mapsto e]$, where e evaluates to an LH. Two kinds of rules are introduced to account for whether a transition in δ is a phase transition or not:

1. **Non-phase Transitions**: $\widehat{\mathsf{LH}}' = \widehat{\mathsf{LH}}[x \mapsto \mathsf{post}(\widehat{\mathsf{LH}}[x], a)]$.
   (a) For each rule $\langle p, \gamma\rangle \xhookrightarrow{a} \langle p', u\rangle \in \Delta_i$ and transition $(q_x, i, a, q_x) \in \delta$, there is a rule $r = \langle (p, q_x, \widehat{\mathsf{LH}}), \gamma\rangle \xhookrightarrow{} \langle (p', q_x, \widehat{\mathsf{LH}}'), u\rangle \in \Delta_i^{\mathcal{A}}$.
   (b) For each rule $\langle p, \gamma\rangle \xhookrightarrow{a} \langle p', u\rangle \in \Delta_i$, $a \in \{(_k,)_k\}$, and each $q_x \in Q$, there is a rule $r = \langle (p, q_x, \widehat{\mathsf{LH}}), \gamma\rangle \xhookrightarrow{} \langle (p', q_x, \widehat{\mathsf{LH}}'), u\rangle \in \Delta_i^{\mathcal{A}}$.
2. **Phase Transitions**: $\widehat{\mathsf{LH}}' = \widehat{\mathsf{LH}}[(x+1) \mapsto \mathsf{ptrans}(\widehat{\mathsf{LH}}[x])]$.
   (a) For each rule $\langle p, \gamma\rangle \xhookrightarrow{a} \langle p', u\rangle \in \Delta_i$ and transition $(q_x, i, a, q_{x+1}) \in \delta$, there is a rule $r = \langle (p, q_x, \widehat{\mathsf{LH}}), \gamma\rangle \xhookrightarrow{} \langle (p', q_{x+1}, \widehat{\mathsf{LH}}'), u\rangle \in \Delta_i^{\mathcal{A}}$.
   (b) For each transition $(q_x, j, a, q_{x+1}) \in \delta$, $j \neq i$, and for each $p \in P_i$ and $\gamma \in \Gamma_i$, there is a rule $r = \langle (p, q_x, \widehat{\mathsf{LH}}), \gamma\rangle \xhookrightarrow{} \langle (p, q_{x+1}, \widehat{\mathsf{LH}}'), \gamma\rangle \in \Delta^{\mathcal{A}}$.

Rules defined by item 1(a) make sure that $\mathcal{P}_i^{\mathcal{A}}$ is constrained to follow the self-loops on PA state q_x. Rules defined by item 1(b) allow for $\mathcal{P}_i^{\mathcal{A}}$ to perform lock acquires and releases. Recall that the language of a PA is only over the non-parenthesis alphabet Σ, and does not constrain the locking behavior. Consequently, a phase transition cannot occur when $\mathcal{P}_i^{\mathcal{A}}$ is acquiring or releasing a lock. Rules defined by item 2(a) handle phase transitions caused by $\mathcal{P}_i^{\mathcal{A}}$. Finally, rules defined by item 2(b) implement $\mathcal{P}_i^{\mathcal{A}}$'s guessing that another PDS $\mathcal{P}_j^{\mathcal{A}}$, $j \neq i$, causes a phase transition, in which case $\mathcal{P}_i^{\mathcal{A}}$ has to move to the next phase, as well.

6.2 Checking Path Compatibility

For a generated PDS $\mathcal{P}_k^{\mathcal{A}}$, we are interested in the set of paths that begin in the initial configuration $\langle p_0^{\mathcal{A}}, \gamma_0\rangle$ and drive $\mathcal{A}$ to its final state $q_{|Q|}$. Each such

```
   input  : A 2-PDS $\Pi = (\mathcal{P}_1, \mathcal{P}_2, \mathsf{Locks}, \Sigma)$ and a PA $\mathcal{A}$.
   output: $true$ if $\Pi$ can drive $\mathcal{A}$ to its final state.
1  let $\mathcal{A}^1_{post^*} \leftarrow post^*_{\mathcal{P}_1^{\mathcal{A}}}$; let $\mathcal{A}^2_{post^*} \leftarrow post^*_{\mathcal{P}_2^{\mathcal{A}}}$;
2  foreach $p_1 \in P_1,\ \widehat{LH}_1$ s.t. $\exists u_1 \in \Gamma_1^* : \langle (p_1, q_{|Q|}, \widehat{LH}_1), u_1 \rangle \in L(\mathcal{A}^1_{post^*})$ do
3      foreach $p_2 \in P_2,\ \widehat{LH}_2$ s.t. $\exists u_2 \in \Gamma_2^* : \langle (p_2, q_{|Q|}, \widehat{LH}_2), u_2 \rangle \in L(\mathcal{A}^2_{post^*})$ do
4          if $\mathit{Compatible}(\widehat{LH}_1, \widehat{LH}_2)$ then
5              return $true$;
6  return $false$;
```

Algorithm 1: The decision procedure. The two tests of the form "$\exists u_k \in \Gamma_k^* : \langle (p_k, q_{|Q|}, \widehat{\mathsf{LH}}_k), u_k \rangle \in L(\mathcal{A}^k_{post^*})$" can be performed by finding *any* path in $\mathcal{A}^k_{post^*}$ from state $(p_k, q_{|Q|}, \widehat{\mathsf{LH}}_k)$ to the final state.

path ends in some configuration $\langle (p_k, q_{|Q|}, \widehat{\mathsf{LH}}_k), u \rangle$, where $u \in \Gamma^*$. Let ρ_1 and ρ_2 be such paths from $\mathcal{P}_1^{\mathcal{A}}$ and $\mathcal{P}_2^{\mathcal{A}}$, respectively. To determine if there exists a compatible scheduling for ρ_1 and ρ_2, we use Thm. 1 on each component of the lock-history tuples $\widehat{\mathsf{LH}}_1$ and $\widehat{\mathsf{LH}}_2$ from the ending configurations of ρ_1 and ρ_2:

$$\mathsf{Compatible}(\widehat{\mathsf{LH}}_1, \widehat{\mathsf{LH}}_2) \Longleftrightarrow \bigwedge_{i=1}^{|Q|} \mathsf{Compatible}(\widehat{\mathsf{LH}}_1[i], \widehat{\mathsf{LH}}_2[i]).$$

Due to recursion, $\mathcal{P}_1^{\mathcal{A}}$ and $\mathcal{P}_2^{\mathcal{A}}$ could each have an infinite number of such paths. However, each path is abstracted as a tuple of lock histories $\widehat{\mathsf{LH}}$, and there are only a finite number of tuples in $\widehat{\mathcal{LH}}$; thus, we only have to check a finite number of $(\widehat{\mathsf{LH}}_1, \widehat{\mathsf{LH}}_2)$ pairs. For each PDS $\mathcal{P}^{\mathcal{A}} = (P^{\mathcal{A}}, \mathit{Act}, \Gamma, \Delta, c_0^{\mathcal{A}})$, we can identify the set of relevant $\widehat{\mathsf{LH}}$ tuples by computing the set of all configurations that are reachable starting from the initial configuration, $post^*_{\mathcal{P}^{\mathcal{A}}}(c_0^{\mathcal{A}})$, using standard automata-based PDS techniques [11, 12]. (Because the initial configuration is defined by the PDS $\mathcal{P}^{\mathcal{A}}$, henceforth, we merely write $post^*_{\mathcal{P}^{\mathcal{A}}}$.) That is, because the construction of $\mathcal{P}^{\mathcal{A}}$ removed all labels, we can create a $\mathcal{P}$*-(multi)-automaton* [11] $\mathcal{A}_{post^*}$ that accepts exactly the set of configurations $post^*_{\mathcal{P}^{\mathcal{A}}}$.

The algorithm to check whether Π can drive $\mathcal{A}$ to its final state is given in Alg. 1.

Theorem 2. *For multi-PDS $\Pi = (\mathcal{P}_1, \mathcal{P}_2, \mathsf{Locks}, \Sigma)$ and PA $\mathcal{A}$, there exists an execution of Π that drives $\mathcal{A}$ to its final state iff Alg. 1 returns true.*

7 A Symbolic Implementation

Alg. 1 solves the multi-PDS model-checking problem for PAs. However, an implementation based on symbolic techniques is required because it would be infeasible to perform the final explicit enumeration step specified in Alg. 1, lines 2–5. One possibility is to use Schwoon's BDD-based PDS techniques [15]; these represent the transitions of a PDS's control-state from one configuration to another as a relation, using BDDs. This approach would work with relations over $Q \times \mathcal{LH}$, which requires using $|Q|^2|\mathsf{LH}|^2$ BDD variables, where $|\mathsf{LH}| = 2\mathcal{L} + 2\mathcal{L}^2$.

This section describes a more economical encoding that needs only $(|Q|+1)|\mathsf{LH}|$ BDD variables. Our approach leverages the fact that when a property is specified with a phase automaton, once a PDS makes a phase transition from q_x to q_{x+1}, the first x entries in $\widehat{\mathsf{LH}}$ tuples are no longer subject to change. In this situation, Schwoon's encoding contains redundant information; our technique eliminates this redundancy.

We explain the more economical approach by defining a suitable weight domain for use with a *weighted PDS* (WPDS) [4, 20]. A WPDS $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$ is a PDS $\mathcal{P} = (P, Act, \Gamma, \Delta, c_0)$ augmented with a *bounded idempotent semiring* $\mathcal{S} = (D, \otimes, \oplus, \overline{1}, \overline{0})$ (see App. C), and a function $f : \Delta \to D$ that assigns a semiring element $d \in D$ to each rule $r \in \Delta$. When working with WPDSs, the result of a *post** computation is a *weighted automaton*. For the purposes of this paper, we view the weighted automaton $\mathcal{A}_{post^*} = post^*_{\mathcal{W}}$ as a function from a regular set of configurations C to the *sum-over-all-paths* from c_0 to all $c \in C$; i.e., $\mathcal{A}_{post^*}(C) = \bigoplus\{v \mid \exists c \in C : c_0 \xrightarrow{r_1 \ldots r_n} c, v = f(r_1) \otimes \ldots \otimes f(r_n)\}$, where $r_1 \ldots r_n$ is a sequence of rules that transforms c_0 into c. For efficient algorithms for computing both $\mathcal{A}_{post^*}$ and $\mathcal{A}_{post^*}(C)$, see [4, 20].

Definition 4. *Let S be a finite set; let $A \subseteq S^{m+1}$ and $B \subseteq S^{p+1}$ be relations of arity $m+1$ and $p+1$, respectively. The* **generalized relational composition** *of A and B, denoted by "$A\ ;\ B$", is the following subset of S^{m+p}:*

$$A\ ;\ B = \{\langle a_1, \ldots, a_m, b_2, \ldots, b_{p+1}\rangle \mid \langle a_1, \ldots, a_m, x\rangle \in A \wedge \langle x, b_2, \ldots, b_{p+1}\rangle \in B\}.$$

Definition 5. *Let S be a finite set, and θ be the maximum number of phases of interest. The set of all* **θ-term formal power series over z, with relation-valued coefficients of different arities**, *is*

$$\mathcal{RFPS}[\mathcal{S}, \theta] = \{\textstyle\sum_{i=0}^{\theta-1} c_i z^i \mid c_i \subseteq S^{i+2}\}.$$

A **monomial** *is written as $c_i z^i$ (all other coefficients are understood to be $\emptyset$); a monomial $c_0 z^0$ denotes a* **constant**. *The* **multi-arity relational weight domain over S and θ** *is defined by $(\mathcal{RFPS}[\mathcal{S}, \theta], \times, +, Id, \emptyset)$, where $\times$ is polynomial multiplication in which generalized relational composition and $\cup$ are used to multiply and add coefficients, respectively, and terms $c_j z^j$ for $j \geq \theta$ are dropped; $+$ is polynomial addition using $\cup$ to add coefficients; Id is the constant $\{\langle s, s\rangle \mid s \in S\} z^0$; and $\emptyset$ is the constant $\emptyset z^0$.*

We now define the WPDS $\mathcal{W}_i = (\mathcal{P}_i^{\mathcal{W}}, \mathcal{S}, f)$ that results from taking the product of PDS $\mathcal{P}_i = (P_i, Act_i, \Gamma_i, \Delta_i, \langle p_0, \gamma_0\rangle)$ and phase automaton $\mathcal{A} = (Q, Id, \Sigma, \delta)$. The construction is similar to that in §6.1, i.e., a cross product is performed that pairs the control states of $\mathcal{P}_i$ with the state space of $\mathcal{A}$. The difference is that the lock-history tuples are removed from the control state, and instead are modeled by $\mathcal{S}$, the multi-arity relational weight domain over the finite set $\mathcal{LH}$ and $\theta = |Q|$. We define $\mathcal{P}_i^{\mathcal{W}} = (P_i \times Q, \emptyset, \Gamma_i, \Delta_i^{\mathcal{W}}, \langle (p_0, q_1), \gamma_0\rangle)$, where $\Delta_i^{\mathcal{W}}$ and f are defined as follows:

1. **Non-phase Transitions**: $f(r) = \{\langle \mathsf{LH}_1, \mathsf{LH}_2\rangle \mid \mathsf{LH}_2 = \mathsf{post}(\mathsf{LH}_1, a)\} z^0$.

```
input  : A 2-PDS $(\mathcal{P}_1, \mathcal{P}_2, \mathsf{Locks}, \Sigma)$ and a PA $\mathcal{A}$.
output: $\mathit{true}$ if there is an execution that drives $\mathcal{A}$ to the accepting state.
let $\mathcal{A}^1_{post^*} \leftarrow post^*_{\mathcal{W}_1}$; let $\mathcal{A}^2_{post^*} \leftarrow post^*_{\mathcal{W}_2}$;
let $c^1_{|Q|-1} z^{|Q|-1} = \mathcal{A}^1_{post^*}(\{\langle (p_1, q_{|Q|}), u \rangle \mid p_1 \in P_1 \wedge u \in \Gamma_1^*\})$;
let $c^2_{|Q|-1} z^{|Q|-1} = \mathcal{A}^2_{post^*}(\{\langle (p_2, q_{|Q|}), u \rangle \mid p_2 \in P_2 \wedge u \in \Gamma_2^*\})$;
return $\exists \langle \mathit{LH}_0, \widehat{\mathit{LH}}_1 \rangle \in c^1_{|Q|-1}, \langle \mathit{LH}_0, \widehat{\mathit{LH}}_2 \rangle \in c^2_{|Q|-1} : \mathit{Compatible}(\widehat{\mathit{LH}}_1, \widehat{\mathit{LH}}_2)$;
```

Algorithm 2: The symbolic decision procedure.

(a) For each rule $\langle p, \gamma \rangle \xhookrightarrow{a} \langle p', u \rangle \in \Delta_i$, $a \in \{(_k,)_k\}$, and for each $q_x \in Q$, there is a rule $r = \langle (p, q_x), \gamma \rangle \hookrightarrow \langle (p', q_x), u \rangle \in \Delta_i^{\mathcal{W}}$.

(b) For each rule $\langle p, \gamma \rangle \xhookrightarrow{a} \langle p', u \rangle \in \Delta_i$ and transition $(q_x, i, a, q_x) \in \delta$, there is a rule $r = \langle (p, q_x), \gamma \rangle \hookrightarrow \langle (p', q_x), u \rangle \in \Delta_i^{\mathcal{W}}$.

2. **Phase Transitions**: $f(r) = \{\langle \mathsf{LH}, \mathsf{LH}, \mathsf{ptrans}(\mathsf{LH}) \rangle \mid \mathsf{LH} \in \mathcal{LH}\} z^1$.

(a) For each rule $\langle p, \gamma \rangle \xhookrightarrow{a} \langle p', u \rangle \in \Delta_i$ and transition $(q_x, i, a, q_{x+1}) \in \delta$, there is a rule $r = \langle (p, q_x), \gamma \rangle \hookrightarrow \langle (p', q_{x+1}), u \rangle \in \Delta_i^{\mathcal{W}}$.

(b) For each transition $(q_x, j, a, q_{x+1}) \in \delta$, $j \neq i$, and for each $p \in P_i$ and $\gamma \in \Gamma_i$, there is a rule $r = \langle (p, q_x), \gamma \rangle \hookrightarrow \langle (p, q_{x+1}), \gamma \rangle \in \Delta^{\mathcal{W}}$.

A multi-arity relational weight domain is parameterized by the quantity θ—the maximum number of phases of interest—which we have picked to be $|Q|$. We must argue that weight operations performed during model checking do not cause this threshold to be exceeded. For configuration $\langle (p, q_x), u \rangle$ to be reachable from the initial configuration $\langle (p_0, q_1), \gamma_0 \rangle$ of some WPDS $\mathcal{W}_i$, PA $\mathcal{A}$ must make a sequence of transitions from states q_1 to q_x, which means that $\mathcal{A}$ goes through exactly $x-1$ phase transitions. Each phase transition multiplies by a weight of the form $c_1 z^1$; hence, the weight returned by $\mathcal{A}_{post^*}(\{\langle (p, q_x), u \rangle\})$ is a monomial of the form $c_{x-1} z^{x-1}$. The maximum number of phases in a PA is $|Q|$, and thus the highest-power monomial that arises is of the form $c_{|Q|-1} z^{|Q|-1}$. (Moreover, during $post^*_{\mathcal{W}_k}$ as computed by the algorithm from [20], only monomial-valued weights ever arise.)

Alg. 2 states the algorithm for solving the multi-PDS model-checking problem for PAs. Note that the final step of Alg. 2 can be performed with a single BDD operation.

Theorem 3. *For multi-PDS $\Pi = (\mathcal{P}_1, \mathcal{P}_2, \mathsf{Locks}, \Sigma)$ and PA $\mathcal{A}$, there exists an execution of Π that drives $\mathcal{A}$ to the accepting state iff Alg. 2 returns true.*

8 Experiments

Our experiment concerned detecting AS-serializability violations (or proving their absence) in models of concurrent Java programs. The experiment was designed to compare the performance of Alg. 2 against that of the communicating-pushdown system (CPDS) semi-decision procedure from [10]. Alg. 2 was implemented using the WALI WPDS library [21] (the multi-arity relational weight domain is included in the WALI release 3.0). The weight domain uses the BuDDy BDD library [22]. All experiments were run on a dual-core 3 GHz Pentium Xeon processor with 4 GB of memory.

We analyzed four Java programs from the ConTest benchmark suite [16]. Our tool requires a few source annotations to be added to each program. We have annotated eleven on the twenty-seven programs that ConTest documentation identifies as having "non-atomic" bugs. Our front-end currently handles eight of the eleven (the AST rewriting of [10] currently does not support certain Java constructs). Finally, after abstraction, four of the eight EML models did not use locks, so we did not analyze them further. The four that we used in our study are SoftwareVerificationHW, BugTester, BuggyProgram, and shop.

For each program, the front-end of the EMPIRE tool [10] was used to create an EML program. An EML program has a set of shared-memory locations, S_{Mem}, a set of locks, S_{Locks}, and a set of EML processes, S_{Procs}. Five of the fourteen PAs used for detecting AS-serializability violations check behaviors that involve a single shared-memory location; the other nine check behaviors that involve a pair of shared-memory locations. For each of the five PAs that involve a single shared location, we ran one query for each $m \in S_{\mathsf{Mem}}$. For each of the nine PAs that involve a pair of shared locations, we ran one query for each $(m_1, m_2) \in S_{\mathsf{Mem}} \times S_{\mathsf{Mem}}$. In total, each tool ran 2,147 queries. Fig. 4 shows log-log scatter-plots of the execution times, classified into the 43 queries for which Alg. 2 reported an AS-serializability violation (left-hand graph), and the 2,095 queries for which Alg. 2 verified correctness (right-hand graph).

Although the CPDS-based method is a semi-decision procedure, it is capable of both (i) verifying correctness, and (ii) finding AS-serializability violations [10]. (The third possibility is that it times out.) Comparing the total time to run all queries, Alg. 2 ran 7.5X faster (136,235 seconds versus 17,728 seconds). The CPDS-based method ran faster than Alg. 2 on some queries, although never more than about 8X faster; in contrast, Alg. 2 was more than two orders of magnitude faster on some queries. Moreover, the CPDS-based method timed out on about 68% of the queries—both for the ones for which Alg. 2 reported an AS-serializability violation (29 timeouts out of 43 queries), as well as the ones for which Alg. 2 verified correctness (1,425 timeouts out of 2,095 queries). Alg. 2 exceeded the 200-second timeout threshold on nine queries. The CPDS-based method also timed out on those queries. When rerun with no timeout threshold, Alg. 2 solved each of the nine queries in 205–231 seconds.

	Query Category	
Impl.	CPDS succeeded Alg. 2 succeeded (685 of 2,147)	CPDS timed out Alg. 2 succeeded (1,453 of 2,147)
CPDS	6,006	130,229
Alg. 2	2,428	15,310

Fig. 5. Total time (in seconds) for examples classified according to whether CPDS succeeded or timed out.

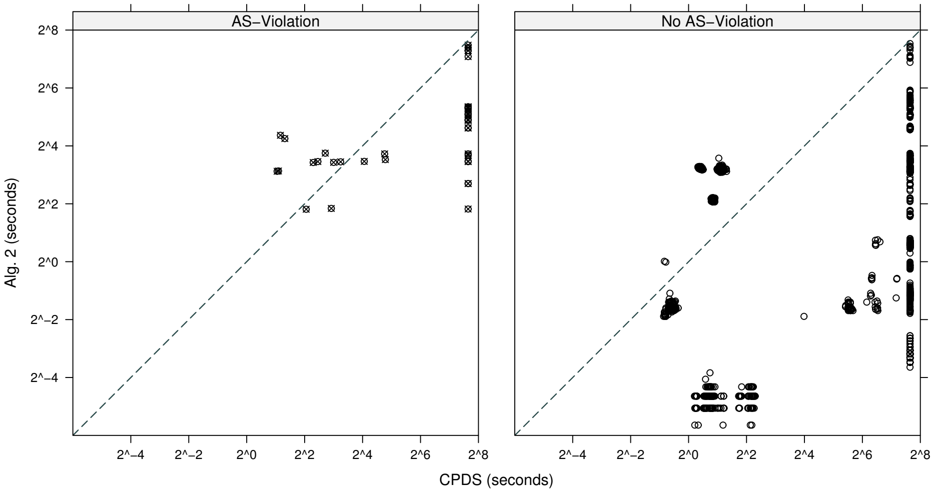

Fig. 4. Log-log scatter-plots of the execution times of Alg. 2 (y-axis) versus the CPDS semi-decision procedure [10] (x-axis). The dashed lines denote equal running times; points below and to the right of the dashed lines are runs for which Alg. 2 was faster. The timeout threshold was 200 seconds; the minimal reported time is .25 seconds. The vertical bands near the right-hand axes represent queries for which the CPDS semi-decision procedure timed out. (The horizontal banding is due to the fact that, for a given program, Alg. 2 often has similar performance for many queries.)

Fig. 5 partitions the examples according to whether CPDS succeeded or timed out. The 1,453 examples on which CPDS timed out (col. 3 of Fig. 5) might be said to represent "harder" examples. Alg. 2 required 15,310 seconds for these, which is about 3X more than the 1,453/685 × 2,428 = 5,150 seconds expected if the queries in the two categories were of equal difficulty for Alg. 2. Roughly speaking, therefore, the data supports the conclusion that what is harder for CPDS is also harder for Alg. 2.

9 Related Work

The present paper introduces a different technique than that used by Kahlon and Gupta [7]. To decide the model-checking problem for PAs (as well as certain generalizations not discussed here), one needs to check pairwise reachability of *multiple* global configurations in succession. Our algorithm uses WPDS weights that are sets of lock-history tuples, whereas Kahlon and Gupta use sets of pairs of configuration automata.

There are similarities between the kind of splitting step needed by Qadeer and Rehof to enumerate states at a context switch [1] in context-bounded model checking and the splitting step on sets of automaton-pairs needed in the algorithm of Kahlon and Gupta [7] to enumerate compatible configuration pairs [14]. Kahlon and Gupta's algorithm performs a succession of pre^* queries; after each one, it splits the resulting set of automaton-pairs to enforce the invariant that succeeding queries are only applied to compatible configuration pairs. In contrast, our algorithm (i) analyzes each PDS independently using *one post** query per PDS, and then (ii) ties together the answers obtained from the different PDSs by performing a single compatibility check on the sets of lock-history tuples that result. Because our algorithm does not need a splitting step on intermediate results, it avoids enumerating compatible configuration pairs, thereby enabling BDD-based symbolic representations to be used throughout.

The Kahlon-Gupta decision procedure has not been implemented [14], so a direct performance comparison was not possible. It is left for future work to determine whether our approach can be applied to the decidable sub-logics of LTL identified in [7].

Our approach of using sets of tuples is similar in spirit to the use of matrix [2] and tuple [3] representations to address context-bounded model checking [1]. In this paper, we bound the number of phases, but permit an unbounded number of context switches and an unbounded number of lock acquisitions and releases by each PDS. The decision procedure is able to explore the entire state space of the model; thus, our algorithm is able to verify properties of multi-PDSs instead of just performing bug detection.

Dynamic pushdown networks (DPNs) [23] extend parallel PDSs with the ability to create threads dynamically. Lammich et al. [24] present a generalization of acquisition histories to DPNs with well-nested locks. Their algorithm uses chained *pre** queries, an explicit encoding of acquisition histories in the state space, and is not implemented.

References

1. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: TACAS. (2005)
2. Lal, A., Touili, T., Kidd, N., Reps, T.: Interprocedural analysis of concurrent programs under a context bound. In: TACAS. (2008)
3. Lal, A., Reps, T.: Reducing concurrent analysis under a context bound to sequential analysis. In: CAV. (2008)
4. Bouajjani, A., Esparza, J., Touili, T.: A generic approach to the static analysis of concurrent programs with procedures. In: POPL. (2003)
5. Chaki, S., Clarke, E., Kidd, N., Reps, T., Touili, T.: Verifying concurrent message-passing C programs with recursive calls. In: TACAS. (2006)
6. Kahlon, V., Ivancic, F., Gupta, A.: Reasoning about threads communicating via locks. In: CAV. (2005)
7. Kahlon, V., Gupta, A.: On the analysis of interacting pushdown systems. In: POPL. (2007)
8. Vaziri, M., Tip, F., Dolby, J.: Associating synchronization constraints with data in an object-oriented language. In: POPL. (2006)
9. Flanagan, C., Qadeer, S.: A type and effect system for atomicity. In: PLDI. (2003)
10. Kidd, N., Reps, T., Dolby, J., Vaziri, M.: Finding concurrency-related bugs using random isolation. In: VMCAI. (2009)
11. Bouajjani, A., Esparza, J., Maler, O.: Reachability analysis of pushdown automata: Application to model checking. In: CONCUR. (1997)

12. Finkel, A., B.Willems, Wolper, P.: A direct symbolic approach to model checking pushdown systems. Elec. Notes in Theor. Comp. Sci. **9** (1997)
13. Kidd, N., Lal, A., Reps, T.: Language strength reduction. In: SAS. (2008)
14. Kahlon, V., Gupta, A.: Personal communication (January 2009)
15. Schwoon, S.: Model-Checking Pushdown Systems. PhD thesis, TUM (2002)
16. Eytani, Y., Havelund, K., Stoller, S.D., Ur, S.: Towards a framework and a benchmark for testing tools for multi-threaded programs. Conc. and Comp.: Prac. and Exp. **19**(3) (2007)
17. Kidd, N., Lammich, P., Reps, T., Touili, T.: A decision procedure for detecting atomicity violations for communicating processes with locks. Technical Report 1649, Univ. of Wisconsin (Jan. 2009) Available at http://www.cs.wisc.edu/wpis/abstracts/tr1649.abs.html.
18. Reps, T.: Program analysis via graph reachability. Inf. and Softw. Tech. **40** (1998)
19. Harrison, M.: Introduction to Formal Language Theory. Addison-Wesley, Reading, MA (1978)
20. Reps, T., Schwoon, S., Jha, S., Melski, D.: Weighted pushdown systems and their application to interprocedural dataflow analysis. SCP **58** (2005)
21. Kidd, N., Lal, A., Reps, T.: WALi: The Weighted Automaton Library (Feb. 2009) http://www.cs.wisc.edu/wpis/wpds/download.php.
22. BuDDy: A BDD package (Jul. 2004) http://buddy.wiki.sourceforge.net/.
23. Bouajjani, A., Müller-Olm, M., Touili, T.: Regular symbolic analysis of dynamic networks of pushdown systems. In: CONCUR. (2005)
24. Lammich, P., Müller-Olm, M., Wenner, A.: Predecessor sets of dynamic pushdown networks with tree-regular constraints. In: CAV. (2009) To appear in.

A An LTL Formula for PA $\mathcal{A}$

Intuitively, for a N-PDS, a PA $\mathcal{A}$ can be expressed as a N-indexed LTL formula $\varphi_{\mathcal{A}}$ using only the "eventually" F and "next" X operators: self-loops are captured with an F and phase-transitions with an X. Let the predicate S_{q_x} denote an atomic-logical formula meaning that the control state all (augmented) PDSs satisfies q_x. That is, the control state of the PDS that results from taking the standard cross product of PDS $\mathcal{P}$ with PA $\mathcal{A}$ is of the form (p, q_x). The following function can be used to translate a PA into an N-indexed LTL formula:

$$\begin{aligned} H(q_{|Q|}) &= S_{q_{|Q|}} \\ H(q_x) &= \mathsf{F}(S_{q_x} \wedge \mathsf{X}_1(\ldots(\mathsf{X}_\mathsf{N}(S_{q_{x+1}} \wedge H(q_{x+1})))\ldots) \end{aligned}$$

Specifically, $\varphi_{\mathcal{A}} = H(q_1)$.

B Generalizing to More Than Two PDSs

Generalizing the decision procedure from a 2-PDS system to an N-PDS system only requires defining how to check compatibility for N lock-history tuples. Because the set of reachable configurations, and hence the set of lock-history tuples, are computed independently for each PDS, the construction from §6.1 that combines a PDS $\mathcal{P}$ with a PA $\mathcal{A}$ to form a new PDS $\mathcal{P}^{\mathcal{A}}$ does not change when generalizing to N PDSs.

Generalizing the compatibility check to N lock-history tuples requires generalizing Thm. 1. The extension of items 1, 2, and 5 to N lock-history tuples is straightforward. Items 3 and 4 define incompatibility to be a cycle of length two in the acquisition and release histories, respectively. Hence, the generalized condition requires checking for a cycle in the acquisition and release histories that has a length anywhere from two to N. We use the notation $\mathsf{Compatible}(\mathsf{LH}_1, \ldots, \mathsf{LH}_N)$ to denote the generalized check. Then Alg. 1 is modified to contain N **foreach** loops, and the compatibility check at line 4 is replaced with $\mathsf{Compatible}(\widehat{\mathsf{LH}}_1, \ldots, \widehat{\mathsf{LH}}_N)$.

Similarly, Alg. 2 is modified to construct N WPDSs, perform N *post** operations (line 1), compute N sum-over-all-paths values $c^1_{|Q|-1}z^{|Q|-1}, \ldots, c^N_{|Q|-1}z^{|Q|-1}$ (lines 2–3), and finally perform the check

$$\exists \langle \mathsf{LH}_0, \widehat{\mathsf{LH}}_1 \rangle \in c^1_{|Q|-1}, \ldots, \langle \mathsf{LH}_0, \widehat{\mathsf{LH}}_N \rangle \in c^N_{|Q|-1} : \mathsf{Compatible}(\widehat{\mathsf{LH}}_1, \ldots, \widehat{\mathsf{LH}}_N).$$

As in Alg. 2, the compatibility check can be performed via a single BDD operation by defining the N-way compatibility relation. This is how it is performed in our implementation.

C Bounded Idempotent Semirings

Definition 6. *A* **bounded idempotent semiring** *is a tuple* $(D, \otimes, \oplus, \overline{1}, \overline{0})$, *where* D *is a set whose elements are called* **weights***;* $\overline{1}, \overline{0} \in D$*; and* $\otimes$ *(the* **extend** *operation) and* $\oplus$ *(the* **combine** *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 (i.e., for all $a \in D$, $a \oplus a = a$).*
2. *$(D, \otimes)$ is a monoid with $\overline{1}$ as its neutral element.*
3. *$\otimes$ distributes over $\oplus$, i.e., for all $a, b, c \in D$ we have*
$$a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c) \text{ and } (a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c) \ .$$
4. *$\overline{0}$ is an annihilator with respect to $\otimes$, i.e., for all $a \in D$, $a \otimes \overline{0} = \overline{0} = \overline{0} \otimes a$*
5. *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.*

A multi-arity relational weight domain over S and θ, as defined in Defn. 5, meets the requirements of Defn. 6 because of (i) the properties of polynomial addition and truncated polynomial multiplication, (ii) the fact that the set of all relations of finite arity ≥ 2 and the operation of generalized relational composition defined in Defn. 4 (";") is a monoid, and (iii) ";" is both left- and right-distributive over union of arity-k relations.