Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

A Decision Procedure for Detecting Atomicity Violations for Communicating Processes with Locks[⋆]

Nicholas Kidd[1⋆⋆], Peter Lammich[2], Tayssir Touili[3], Thomas Reps[4,5]

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

Received: date / Revised version: date

Abstract. The problem of interest is to verify data consistency of a concurrent Java program. In particular, we present a new decision procedure for verifying that a class of data races caused by inconsistent accesses on *multiple fields* of an object cannot occur (so-called *atomic-set serializability*). Atomic-set serializability generalizes the ordinary notion of a data race (i.e., inconsistent coordination of accesses on a *single* memory location) to a broader class of races that involve accesses on *multiple* memory locations.

Previous work by some of the authors presented a technique to abstract a concurrent Java program into an EML program, a modeling language based on *pushdown systems* and a finite set of *reentrant* locks. Our previous work used only a semi-decision procedure, and hence provides a definite answer only some of the time. In this paper, we rectify this shortcoming by developing a decision procedure for verifying data consistency, i.e., atomic-set serializability, of an EML program. When coupled with the previous work, it provides a decision procedure for verifying data consistency of a concurrent Java program.

We implemented the decision procedure, and applied it to detect both single-location and multi-location data races in models of concurrent Java programs. Compared with the prior method based on a semi-decision procedure, not only was the decision procedure 34 times faster overall, but the semi-decision procedure timed out on about 50% of the queries, whereas the decision procedure timed out on none of the queries.

1 Introduction

To leverage the increased processing power of modern multicore processors, programmers are left with the burden of writing concurrent programs. A popular language for writing concurrent programs is Java. In Java, a concurrent program consists of multiple threads that execute in parallel and communicate through shared memory. Because the number of threads is typically greater than the number of available processors (or cores), a thread scheduler determines which threads will execute and for how long. From a programmer's perspective, the thread scheduler is non-deterministic.

Writing correct concurrent programs is a notoriously difficult task because the programmer must account not only for the sequential behavior of an individual thread, but also for non-deterministic interference from other (external) threads. Non-deterministic interference can result in *data-consistency errors*, a class of programming errors that sequential programs are not prone to. Loosely speaking, a data-consistency error occurs when a thread of execution exposes intermediate computational results to external threads, or when it observes external computational results when executing a sequence of operations that define one (larger) logically-atomic operation.

To guard against data-consistency errors, shared-memory concurrent Java programs use locks to protect accesses to shared-memory locations. Locks enforce mutual exclusion: only one thread may hold the lock at a given time. (In Java, each object has a lock associated with it, and the lock is acquired (released) when a `synchronized` method is entered (exited) for an object. §2 presents a detailed example of a concurrent Java program and the use of `synchronized` methods to implement synchronization between concurrently executing threads.) Because of mutual exclusion, and assuming that accesses to shared objects are consistently protected by locks in each thread, the thread that currently holds a lock l that protects a shared object o can access and update o's fields without external interference (e.g., an update by another thread).

[⋆] Supported by NSF under grants CCF-0540955, CCF-0524051, and CCF-0810053, by AFRL under contract FA8750-06-C-0249, and by ONR under grant N00014-09-1-0510.

[⋆⋆] Work performed while the author was at the University of Wisconsin.

If threads do not consistently acquire the lock l when attempting to access a field f of the shared object o, and at least one of the accesses is a write access, a data-consistency violation known as a *data race* can occur. The name reflects the fact that two threads are "racing" to access the field f.

Another example of a data-consistency error concerns a commercial networking tool (Visser, 2009). The scenario is as follows. A server maintains a mapping $\mathcal{M}$ from users to workers (threads that are responsible for servicing user requests). When a request req is received by the server for a user u, $\mathcal{M}$ is queried to find the worker w that u is mapped to. After the lookup, but before w is notified of the request, the worker w terminates, and the mapping for u is updated to another worker w'. Hence, the server's attempt to dispatch u's request to w fails. This data-consistency error is an example of an *atomicity* violation. That is, the lookup and dispatch were supposed to appear to execute atomically. In this example, all methods were properly synchronized, i.e., no data races were involved in this data-consistency error. Moreover, this particular data-consistency violation involved *multiple* shared-memory locations.

Multi-location data-consistency errors are a third category of errors due to concurrency, with the other two being data races and atomicity violations. Multi-location data-consistency errors arise because programs often have (usually unstated) consistency relationships between multiple shared-memory locations (e.g., a mapping from user objects to worker objects). A recent survey by Lu et al. (2008) of two popular open-source software applications, Apache and Firefox, showed that multi-location data-consistency errors accounted for *one third* of non-deadlock consistency errors. Thus, it is crucial that tools and techniques be able to verify the absence of multi-location data-consistency errors.

Vaziri et al. (2006) propose *atomic-set serializability* (AS-serializability) as a data-centric correctness criterion for concurrent Java programs. AS-serializability is a property of a program execution, and is a relaxation of *serializability*. An execution is serializable if its outcome is equivalent to an execution where all transactions are executed serially. AS-serializability relaxes serializability to be only with respect to specific memory locations specified as a subset of the fields of a Java class and referred to as an *atomic set*. An important result of Vaziri et al. (2006) is that AS-serializability violations can be completely characterized by a set of fourteen problematic-access patterns, each of which can be specified by a finite-state machine. (§2 presents a detailed example of atomic sets, AS-serializability, and problematic-access patterns.) AS-serializability encompasses data races (single-field atomic sets), multi-location data-consistency errors (multi-field atomic sets), and atomicity (all of memory forms one atomic set).

In previous work by some of the authors (Kidd et al., 2009b), we developed techniques for verifying AS-serializability for concurrent Java programs.[1] Our tool, EMPIRE, first abstracts a concurrent Java program into EML, a modeling language based on *multi-pushdown systems* (multi-PDSs) (defined in §3), where each PDS models a thread of the original Java program, and a finite number of *reentrant* locks.[2] Given a generated EML program, the problem of interest is then to verify that no interleaved execution of the EML program contains one of the fourteen problematic-access patterns. The drawback of the approach that we have used to date is that a generated EML program, along with a specification of a problematic-access pattern, are compiled into a communicating pushdown system (CPDS) (Bouajjani et al., 2003; Chaki et al., 2006), for which the required model-checking problem is undecidable. (A semi-decision procedure is used in (Kidd et al., 2009b).)

In the present paper, we address the limitation of using CPDS-based model checking (i.e., undecidability), by developing a new decision procedure for detecting AS-serializability violations in EML programs. (Actually, we show decidability for a model-checking formalism that can be used to encode AS-serializability violation detection; however, for the purposes of this introduction, we will provide intuition in terms of AS-serializability.) Because the set of behaviors of an EML program is an over-approximation of the set of behaviors of a concurrent Java program, showing that an AS-serializability violation cannot occur verifies data consistency of the original concurrent Java program.

Three steps led to us reaching a decision procedure:

1. For each of the fourteen problematic-access patterns, the non-deterministic finite automaton (NFA) that recognizes interleaved executions of an EML program containing that pattern always has a special form: the only loops are self-loops on states. We call such an automaton an *indexed-phase automaton* (IPA). (IPAs are formally defined in §3.)
2. Like Java locks, EML locks are reentrant and are acquired and released by entering an exiting an EML function. Kidd et al. (2008) present the language-strength-reduction transformation, which allows a reentrant lock to be replaced by a *non-reentrant* lock without sacrificing soundness or precision when the acquisitions and releases of the lock are synchronized with a thread's calls and returns. Thus, we are able to focus on only non-reentrant locks while remaining faithful to Java locks.
3. The scheduling constraints by an individual PDS's use of the non-reentrant locks—that result from applying the language-strength-reduction transformation—can be summarized by a finite PDS path abstraction known as *lock histories*, first developed by Kahlon and Gupta (2007).

The special form of IPAs, coupled with the ability to finitely summarize the locking constraints of a PDS, directly led to the decision procedure that is the focus of this paper.

Related Work. The results described in this paper are related to results obtained by Kahlon and Gupta (2007). Whereas Kahlon and Gupta were interested in verifying that a property

[1] We say that a concurrent Java program Prog is AS-serializable if every possible execution of Prog is AS-serializable.

[2] The tool does not currently support dynamic thread creation.

	PDS control locations	Queries	Cost
Chaining	$O(2^{\mathcal{L}})$	$O(2^{\mathcal{L}\cdot\lvert\mathcal{A}\rvert} \cdot \lvert S_{\mathsf{Procs}}\rvert)$	$O(2^{\mathcal{L}} \cdot 2^{\mathcal{L}\cdot\lvert\mathcal{A}\rvert} \cdot \lvert S_{\mathsf{Procs}}\rvert)$
Tupling	$O(2^{\mathcal{L}\cdot\lvert\mathcal{A}\rvert})$	$\lvert S_{\mathsf{Procs}}\rvert$	$O(2^{\mathcal{L}\cdot\lvert\mathcal{A}\rvert}) \cdot \lvert S_{\mathsf{Procs}}\rvert$

Table 1. Comparison between the (corrected) chaining approach of Kahlon and Gupta (2007) and our tupling approach. $\mathcal{L}$ denotes the number of locks, $|\mathcal{A}|$ denotes the number of states of an IPA $\mathcal{A}$, and $|S_{\mathsf{Procs}}|$ denotes the number of EML processes (PDSs).

expressed in (a fragment of) indexed-LTL holds for multi-PDSs, we are interested in verifying that no execution of a multi-PDS is accepted by an IPA that recognizes execution traces containing AS-serializability violations. However, we later determined that an IPA can be encoded in one of the fragments of LTL that Kahlon and Gupta showed to be decidable. A detailed comparison between the two approaches requires us to first present the program model, lock histories, and our new decision procedure, and is thus reserved until §11. However, we present a comparison of the relative complexities of the two approaches in Tab. 1, where $\mathcal{L}$ is the number of EML locks, $\mathcal{A}$ is the set of states of the IPA of interest, and S_{Procs} is the set of EML processes (PDSs). We highlight two points:

1. Our approach ("Tupling") avoids an exponential in the number of locks $\mathcal{L}$ when compared to the approach of Kahlon and Gupta ("Chaining"). (See the rightmost column in Tab. 1.)
2. Our approach ("Tupling") isolates the exponential cost in the PDS state space, which is preferred because that cost can often be side-stepped using symbolic techniques, such as BDDs, as explained in §8.

Contributions. This paper makes the following contributions:

- We define a decision procedure for verifying AS-serializability of an EML program. The decision procedure handles (i) *reentrant* locks (via the language-strength-reduction transformation of Kidd et al. (2008)), (ii) an *unbounded* number of context switches, (iii) an *unbounded* number of lock acquisitions and releases by each PDS, and (iv) determines whether the sequence of interleaved memory accesses of a problematic access pattern is present (or not). Because the set of behaviors of a generated EML program is an over-approximation of the set of behaviors of the concurrent Java program from which it was generated, verifying AS-serializability of an EML program also verifies AS-serializability of the concurrent Java program.
- The decision procedure is *modular*: each PDS is analyzed independently with respect to IPA, and then a single compatibility check is performed that ties together the results obtained from the analysis of the different PDSs.
- We leverage the special form of IPAs to give a symbolic implementation that is more space-efficient than standard BDD-based techniques for PDSs (Schwoon, 2002).
- We used the decision procedure to detect AS-serializability violations in automatically-generated models of four concurrent Java programs from the ConTest benchmark suite (Eytani et al., 2007a). On the corresponding set of queries, the decision procedure was *34* times faster overall (i.e., the total running time for all queries) when compared to the semi-decision procedure. Moreover, with a timeout threshold of 300 seconds, for each query where the semi-decision procedure timed out (roughly 50% of the queries), the decision procedure succeeded within the allotted time, and actually performed more work because the decision procedure explored the *entire* state space.

Organization. The remainder of the paper is organized as follows: §2 motivates our work by presenting a concurrent Java program that contains an AS-serializability violation, and discusses the difficulties in AS-serializability violation detection. §3 defines multi-PDSs and IPAs. §4 presents the problem statement along with a more detailed overview of the steps that were required to obtain the result that AS-serializability-violation detection is decidable. §5 reviews a decomposition result due to Kahlon and Gupta. §6 presents lock histories. §7 presents the decision procedure for a 2-PDS. §8 presents a symbolic implementation of our 2-PDS decision procedure. §9 generalizes the 2-PDS decision procedure to handle general multi-PDSs. §10 presents experimental results. §11 gives a detailed comparison with a certain decision procedure of Kahlon and Gupta. §12 describes related work. §13 presents some conclusions.

2 Atomic-Set Serializability Violation Detection

Vaziri et al. (2006) introduce *atomic sets*, *unit-of-work* methods, and *atomic-set serializability* (AS-serializability), where

- an atomic set is a set of fields of a Java class for which there exists an unspecified data-structure invariant;
- a unit-of-work method is a method that when executed serially, is guaranteed to reestablish the invariant of an atomic set upon completion;
- and AS-serializability is a correctness criterion of (interleaved) executions where an execution is said to be atomic-set serializable if its outcome is equivalent to an execution in which all unit-of-work methods are executed serially. In other words, an execution e is atomic-set serializable if for each atomic set S, the projection of e with respect to S is serializable.

AS-serializability can be viewed as a relaxation of Flanagan and Qadeer's atomicity property (Flanagan and Qadeer, 2003) to a user-specified set of memory locations. That is, using the terminology of Vaziri et al. (2006), atomicity is equivalent to atomic-set serializability where all of memory forms one atomic set. We next illustrate both atomic sets and atomic-set serializability via a concrete example.

The Java program shown in Listing 1 defines two classes, `Stack` and `SafeWrap`. Class `Stack` is a minimal implementation of a stack that contains two fields:

`data` is an array that stores the objects that have been pushed onto the stack,

`count` is a zero-based counter that denotes the number of items on the stack (zero-based so that the current value of `count` is the index into the array `data` that is the top of the stack).

The fields are annotated with `@`**`atomic`**`(S)` to specify that they belong to the atomic set S. As mentioned above, the invariant is that `count` is a zero-based counter of the number of objects stored in `data`. All non-static methods of `Stack` are **`synchronized`** to implement mutual exclusion of concurrent accesses by multiple threads.[3] The annotation `@`**`atomic`** specifies that the non-static methods are also unit-of-work methods—they are intended to execute atomically with respect to the fields of atomic set S.

Class `SafeWrap` does not define an atomic set. Instead, it is a "wrapper" class that (attempts to) "harden" the implementation of `Stack`. That is, method `Stack.pop` does not perform bounds checking before indexing into the array `data` (line 11). Thus, invoking `Stack.pop` on an empty stack—one in which `count` has the value `-1`—would result in an `ArrayIndexOutOfBoundsException` being thrown. The method `SafeWrap.popwrap` addresses this liability by first checking that the input parameter `Stack s` has a size greater than 0 (line 38) before invoking `Stack.pop`. The `@`**`atomic`** annotation on method `SafeWrap.popwrap` specifies that it is a unit-of-work method and thus should execute atomically. Because class `SafeWrap` does not define an atomic set, atomic execution is just with respect to the parameter `Stack s`.

The programmer attempted to ensure that `SafeWrap.popwrap` always executes atomically by declaring the method to be **`synchronized`**. Unfortunately, in this case the object that *should* have been synchronized on is the parameter `Stack s`. By performing synchronization on the wrong object (namely, the implicit **`this`** parameter of `SafeWrap.popwrap`), the interleaved execution shown in Fig. 1, which results in an `ArrayIndexOutOfBoundsException` being raised, is an allowable behavior of the system. The execution proceeds as follows: Initially, the stack contains one item. Thread T_1 begins execution and checks that the stack is non-empty by invoking `Stack.size`. The check succeeds, and so T_1's next action is to invoke `Stack.pop`. Before doing so, thread T_2 successfully executes `SafeWrap.popwrap`, which removes the item from the stack, leaving it empty. When T_1 resumes execution, it invokes `Stack.pop` on an empty

Listing 1. Stack Java program that contains an AS-serializability violation.

```
 1  class Stack {
 2     // @atomic(S)
 3     Object[] data = new Object[10];
 4
 5     // @atomic(S)
 6     int count = -1;
 7
 8     // @atomic
 9     synchronized Object pop(){
10        Object res = data[count];
11        data[count--] = null;
12        return res;
13     }
14
15     // @atomic
16     synchronized void push(Object o) {
17        data[++count] = o;
18     }
19
20     // @atomic
21     synchronized int size() {
22        return count+1;
23     }
24
25     // @atomic
26     synchronized replaceTop(Object o) {
27        pop(); push(o);
28     }
29
30     static Stack makeStack() {
31        return new Stack();
32     }
33  }
34
35  class SafeWrap {
36     // @atomic
37     synchronized Object popwrap(Stack s) {
38        if (s.size() > 0) return s.pop();
39        else return null;
40     }
41
42     static SafeWrap makeSafeWrap() {
43        return new SafeWrap();
44     }
45
46     static void main(String[] args){
47        Stack stack = Stack.makeStack();
48        stack.push(new Integer(1));
49        new Thread("1") {
50           makeSafeWrap().popwrap(stack);
51        }
52        new Thread("2") {
53           makeSafeWrap().popwrap(stack);
54        }
55     }
56  }
```

[3] In Java, every object has a lock associated with it. A **`synchronized`** method is one in which the lock of the receiving object is acquired upon entering the method, and released before upon exiting.

$$T_1:\ \overbrace{\underline{[_1}\ \overbrace{[_1(_s\underline{R_1(c)})_s]_1}^{\texttt{size}} \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \overbrace{[_1(_sR_1(c)\underline{R_1(d)}\maltese}^{\texttt{pop}}}^{\texttt{popwrap}}$$

$$T_2:\ \cdots\cdots\cdots \overbrace{[_2\underbrace{\underline{[_2(_sR_2(c))_s]_2}}_{\texttt{size}}\ \underbrace{[_2(_sR_2(c)R_2(d)R_2(c)\underline{W_2(d)W_2(c)})_s]_2}_{\texttt{pop}}]_2}^{\texttt{popwrap}} \cdots\cdots\cdots$$

Fig. 1. An interleaved execution of thread T_1 and T_2 that contains an AS-serializability violation. R and W denote a read and write access, respectively. c and d denote fields `count` and `data`, respectively. "[" and "]" denote the beginning and end, respectively, of a unit-of-work method. The subscripts "1" and "2" are thread ids. "$(_\mathbf{s}$" and "$)_\mathbf{s}$" denote the acquire and release operations, respectively, of the lock of `Stack s` that is the input parameter to `SafeWrap.popwrap`.

stack, which raises an exception. The point at which the exception is raised is indicated by the ✠ symbol at the end of thread T_1's execution sequence. Note that the execution in Fig. 1 does not contain a data race (in fact the program in Listing 1 is data-race free).

An important result of Vaziri et al. (2006) is that AS-serializability violations can be completely characterized by a set of fourteen problematic access patterns (see (Vaziri et al., 2006) for a complete listing).[4] Each problematic access pattern is a finite sequence of reads and writes by two threads to one or two shared memory locations. For the program in Listing 1 and problematic access pattern 12 instantiated as follows:

$$[_1; R_1(i); W_2(s); W_2(i); R_1(s),$$

the accesses that match the pattern are underlined in the interleaving shown in Fig. 1.

Fig. 2 presents the non-deterministic finite automaton (NFA) $\mathcal{A}_2$ that recognizes execution traces that contain the problematic-access pattern described above. Notice that $\mathcal{A}_2$ has a special form—the only loops are self-loops on states. We call such an automaton an *indexed phase automaton* (IPA). "Indexed" denotes that the index of a PDS is included on the edge label of a transition. "Phased" denotes that a word accepted by an IPA can be divided into phases, where a phase constitutes all symbols of the word that cause an IPA to follow a self loop, i.e., remain in the same state. A transition between states is called a *phase transition*. Notice that an IPA can only perform a *bounded* number of phase transitions. Bounding the number of phase transitions—global synchronizations—is a key step towards a decision procedure.

The focus of this paper is a technique to statically verify that no execution trace of a multi-PDS is recognized by an IPA (e.g., the IPA in Fig. 2). Because each of the fourteen problematic-access patterns can be encoded as an IPA, and because AS-serializability violations can be completely characterized by the fourteen patterns, our decision procedure can be used to verify AS-serializability of concurrent Java programs.

2.1 What are the Difficulties?

In previous work (Kidd et al., 2009b), we addressed AS-serializability violation detection by encoding the problem as a CPDS model-checking problem. The results of this paper show that CPDS model-checking was, in some sense, a too-powerful hammer: the CPDS model-checking problem is, in general, undecidable, and thus we were not able to obtain answers for roughly 50% of the CPDS queries posed in the experimental evaluation of (Kidd et al., 2009b).

We discovered that the problem was decidable only after working on the problem for several years (and developing the methods discussed in Kidd et al. (2008, 2009b)). There are several reasons why it is not immediately apparent that the problem is decidable:

1. EML locks, like Java locks, are reentrant. The straightforward approach to encoding a reentrant lock requires a counter to track the depth of nested calls to **`synchronized`** methods, and a counter requires an infinite state space.
2. Like EML locks, units of work are also reentrant, and the straightforward approach again requires an infinite-state counter to track the depth of nested calls to unit-of-work methods.
3. For an AS-serializability violation to occur, an interleaved execution must be found such that the read and write accesses to memory locations occur in a specified order (e.g., the order shown in Fig. 2). Moreover, the interleaved execution must respect the semantics of locks, which constitute global state of an EML program. The need to reason precisely about the owners of locks, and the validity of individual transitions of the PDSs that model EML processes with respect to locking semantics, induces an apparent tight coupling between the PDSs.

2.2 Overcoming the Difficulties

At a high level, we overcome the apparent difficulties discussed above by transforming the problem in several ways—in each step without losing precision—so that the threads can be decoupled.

1. The language-strength-reduction transformation of Kidd et al. (2008) provides a mechanism to replace reentrant

[4] 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`).

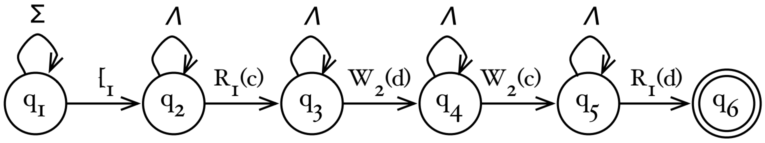

Fig. 2. The IPA $\mathcal{A}_2$ that recognizes execution traces of the program from Listing 1 that contain the AS-serializability violation specified by the problematic-access pattern "$[_1; R_1(i); W_2(s); W_2(i); R_1(s)$". Σ denotes the input alphabet of $\mathcal{A}_2$, and Λ is defined as $\Sigma \setminus \{]\}$. Once $\mathcal{A}_2$ guesses that a violation will occur by making a transition to state q_3, it must observe a violation before the unit-of-work end symbol "]" appears in a trace. Otherwise, it will become stuck in a state q_{3-6}.

locks with *non-reentrant* locks while still being able to explore the entire state space. [5] That is, precision is not lost because the transformed model contains the same set of behaviors as the original model.

2. In much the same way, the language-strength-reduction transformation also provides a mechanism to eliminate the need to use a counter to count the depth of nested calls to unit-of-work methods. In fact, this is a key transformation in showing that AS-serializability violation detection of EML programs is decidable because otherwise, the fourteen problematic-access patterns could not be encoded as IPAs (discussed in §4).
3. The techniques presented in this paper provide a mechanism to analyze the PDSs that model EML processes independently of each other. At a high level, a summary of the locking behavior of each individual PDS—or rather a summary of the constraints that the PDS's use of locks places on the set of possible interleaved executions—is computed by independent analysis runs (one run per PDS). A post-processing step then determines whether there exists an interleaved execution that satisfies the constraints computed for each PDS.

3 Program Model and Property Specifications

This section formally defines the *multi-pushdown system* (multi-PDS) programming model, and *indexed-phase automata* (IPAs), which are used to specify the property of interest.

Definition 1. A ***(labeled) pushdown system*** (PDS) is a tuple $\mathcal{P} = (P, \mathsf{Lab}, \Gamma, \Delta, c_0)$, where P is a finite set of control states, Lab is a finite set of action labels (actions), Γ is a finite stack alphabet, and $\Delta \subseteq (P \times \Gamma) \times \mathsf{Lab} \times (P \times \Gamma^*)$ is a finite set of rules. A rule $r \in \Delta$ is denoted by $\langle p, \gamma \rangle \xhookrightarrow{a} \langle p', u' \rangle$. 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 $\langle p, \gamma \rangle \xhookrightarrow{a} \langle p', u' \rangle \in \Delta$. For $w \in \mathsf{Lab}^*$, $c \xrightarrow{w} c'$ is defined in the usual way. For a rule $r = \langle p, \gamma \rangle \xhookrightarrow{a} \langle p', u' \rangle$, $\mathsf{act}(r)$ denotes r's action label a.

Without loss of generality, a pushdown rule is restricted to have at most two stack symbols appear on the right-hand side, i.e., for $\langle p, \gamma \rangle \xhookrightarrow{a} \langle p', u' \rangle \in \Delta$, $|u'| \leq 2$ (Schwoon, 2002). Rules with zero, one, and two right-hand-side stack symbols are called *pop*, *step*, and *push* rules, respectively. We use Δ_0, Δ_1, and Δ_2 to denote the set of pop, step, and push rules in Δ, respectively.

For a PDS $\mathcal{P} = (P, \mathsf{Lab}, \Gamma, \Delta, c_0)$, we sometimes need to reason about the set of PDS paths that cause $\mathcal{P}$ to make a sequence of transitions that take it from configuration c to configuration c', which will be denoted by *paths*(c, c'). A PDS path ρ is a sequence of PDS rules $[r_1, \ldots, r_n]$, and the intention is that the rules are applied in order, left to right. Note that, due to recursion and looping, the size of the set *paths*(c, c') can be infinite.

Given a set of configurations C, we define the set of *forwards* reachable configurations from C as:

$$\mathit{post}^*(C) =_{\mathrm{df}} \{c' \mid \exists c \in C \ : \ c \Rightarrow^* c'\},$$

and the set of *backwards* reachable configurations from C as:

$$\mathit{pre}^*(C) =_{\mathrm{df}} \{c' \mid \exists c \in C \ : \ c' \Rightarrow^* c\}.$$

3.1 Multi-PDS Programming Model

Informally, a *multi-PDS* consists of a finite set of PDSs and a finite set of locks. The intention is that each PDS models a thread, and that the PDSs acquire and release locks to perform global synchronization. We assume that locks are acquired and released in a well-nested fashion—locks are released in the opposite order in which they are acquired—and in synchrony with a PDS's push and pop rules (Δ_2 and Δ_0, respectively). In fact, the latter assumption is all that is necessary as it implies the former.

We present two execution semantics of multi-PDSs: (i) the *reentrant semantics* equips a multi-PDS with reentrant locks; and (ii) the *non-reentrant semantics* equips a multi-PDS with only non-reentrant locks. As discussed in §2, one can use the language-strength-reduction transformation of Kidd et al. (2008) to *automatically* reduce a multi-PDS with reentrant semantics to a multi-PDS with non-reentrant semantics when the locking operations of each PDS meet our assumptions.

[5] Briefly, the technique for replacing reentrant locks with non-reentrant locks involves transforming a PDS so that it pushes a special marker onto the stack the *first* time a lock is acquired, and also records the fact that the PDS holds the lock in the PDS's state space. All subsequent lock acquires and their matching releases do not change the state of the lock or the state of the PDS. Only when the special marker is seen again is the lock then released. The technique requires that lock acquisitions and releases are synchronized with procedure calls and returns (á la Java's synchronized methods), and/or properly scoped (á la Java's synchronized blocks).

Definition 2. A ***multi-PDS*** Π is a tuple $(\mathcal{P}_1, \ldots, \mathcal{P}_n, S_{\mathsf{Locks}}, \Sigma)$, where each PDS $\mathcal{P}_j = (P_j, \mathsf{Lab}_j, \Gamma_j, \Delta_j, c_0^j)$, $S_{\mathsf{Locks}} = \{l_1, \ldots, l_{\mathcal{L}}\}$ is a finite set of $\mathcal{L}$ locks (i.e., $\mathcal{L} = |S_{\mathsf{Locks}}|$), and Σ is a finite alphabet of non-locking symbols. The action labels Lab of each PDS consist of lock-acquires ("$(_i$") and lock-releases ("$)_i$") for $1 \leq i \leq \mathcal{L}$, plus symbols from Σ. A ***global configuration*** $(c_1, \ldots, c_n, \bar{o})$ is a tuple consisting of:

- a local configuration c_j for each PDS $\mathcal{P}_i$, $1 \leq j \leq n$; and
- an ***ownership array*** $\bar{o}$ of length $\mathcal{L}$, in which each entry indicates the owner of a given lock: for each $1 \leq i \leq \mathcal{L}$, $\bar{o}[i] \in (\{\bot, 1, \ldots, n\} \times \mathcal{N})$ is a pair where the first component indicates the identity j of the PDS $\mathcal{P}_j$ that holds lock l_i ($\bot$ signifies that l_i is currently not held by any PDS), and the second component is a non-negative number that indicates the number of times that a PDS has (re)acquired a lock.

The ***initial global configuration*** $g_0 = (c_0^1, \ldots, c_0^n, \bar{o}_0)$, where c_0^i is the initial configuration of PDS $\mathcal{P}_i$, $1 \leq i \leq n$, and $\bar{o}_0$ is the initial ownership array that maps each entry $\bar{o}[i], 1 \leq i \leq \mathcal{L}$, to the ownership pair $(\bot, 0)$. For an ownership array $\bar{o}$, an update at position i for lock l_i to a new ownership pair p is denoted by $\bar{o}[i \mapsto p]$.

Remark 1. The choice of what symbols (actions) appear in Σ depends on the intended application. For the target application of verifying AS-serializability, Σ consists of symbols to read and write a shared-memory location m (denoted by $R(m)$ and $W(m)$, respectively), and to begin and end a unit of work ([and], respectively).

3.1.1 Reentrant Semantics

For multi-PDS $\Pi = (\mathcal{P}_1, \ldots, \mathcal{P}_n, S_{\mathsf{Locks}}, \Sigma)$, the *Reentrant Semantics* allows for a lock $l \in S_{\mathsf{Locks}}$ to be reacquired by the PDS that owns the lock. In particular, two global configurations g and g' are in the relation $\rightsquigarrow$, denoted by $g \rightsquigarrow g'$, iff $g = (c_1, \ldots, c_j, \ldots, c_n, \bar{o})$ and following condition holds:

1. $c_j \xrightarrow{a} c'_j$, $a \in \Sigma$, and $g' = (c_1 \ldots, c'_j, \ldots, c_n, \bar{o})$.
2. $c_j \xrightarrow{(_i} c'_j$, $\bar{o}[i] = (\bot, 0)$,
   and $g' = (c_1, \ldots, c'_j, \ldots, c_n, \bar{o}[i \mapsto (j, 1)])$.
3. $c_j \xrightarrow{(_i} c'_j$, $\bar{o}[i] = (j, z)$,
   and $g' = (c_1, \ldots, c'_j, \ldots, c_n, \bar{o}[i \mapsto (j, z+1)])$.
4. $c_j \xrightarrow{)_i} c'_j$, $\bar{o}[i] = (j, z)$, $z > 1$,
   and $g' = (c_1, \ldots, c'_j, \ldots, c_n, \bar{o}[i \mapsto (j, z-1)])$.
5. $c_j \xrightarrow{)_i} c'_j$, $\bar{o}[i] = (j, 1)$,
   and $g' = (c_1, \ldots, c'_j, \ldots, c_n, \bar{o}[i \mapsto (\bot, 0)])$.

The reflexive transitive closure of $\rightsquigarrow$ is denoted by $g \rightsquigarrow^* g'$.

An *execution trace* is a sequence of global configurations $g_0 \rightsquigarrow g_1 \rightsquigarrow \ldots \rightsquigarrow g_k$. Let $\Sigma^{\mathcal{L}}$ be the set $\Sigma \cup \{(_i,)_i \mid 1 \leq i \leq \mathcal{L}\}$. A *trace word* w is a sequence of PDS-identifier/action pairs that records the identifier j for PDS $\mathcal{P}_j$ and the action label $a \in \Sigma^{\mathcal{L}}$ of $\mathcal{P}_j$ that caused global configuration g_i to make a transition to global configuration g_{i+1}, i.e., because PDS $\mathcal{P}_j$ made the transition $c_j \xrightarrow{a} c'_j$, $a \in \Sigma^{\mathcal{L}}$. For a trace word w, the *observable trace word* w_Σ of w is the projection of w to only inlude PDS-identifier/action pairs of the form (j, a), where $a \in \Sigma$. In other words, an observable trace word does not include any PDS-identifier/action pairs that have a locking action, either a lock-acquire or lock-release.

3.1.2 Non-Reentrant Semantics

When lock acquisitions and releases are synchronized with a PDS's stack, one can use the language-strength-reduction transformation to replace reentrant locks with non-reentrant locks. In essence, for a PDS $\mathcal{P} = (P, \mathsf{Lab}, \Gamma, \Delta, c_0)$, the transformation defines a new PDS $\mathcal{P}' = (P', \mathsf{Lab}, \Gamma', \Delta', c'_0)$, where

- $P' = P \times 2^{S_{\mathsf{Locks}}}$, where a member (p, s) of P' includes the original control state p of P and a set s that records the set of locks that $\mathcal{P}'$ currently holds.
- $\Gamma' = \Gamma \cup (\Gamma \times S_{\mathsf{Locks}})$ consists of the original stack alphabet Γ, and, in addition, a new set of symbols that enables $\mathcal{P}'$ to record on the stack the first time that a lock $l \in S_{\mathsf{Locks}}$ has been acquired.
- Δ' contains, for each rule $r \in \Delta$, a set of rules that maintain and update the set of held locks s for a control state (p, s), and record on the stack via a symbol (γ, l) that a lock has been acquired for the first time. The exact definition of Δ' is beyond the scope of this paper, and we refer the reader to Kidd et al. (2008) for further details.
- $c'_0 = \langle (p_0, \emptyset), \gamma_0 \rangle$ is the initial configuration paired with $\emptyset$ to indicate that $\mathcal{P}'$ does not hold any locks.

After transforming $\mathcal{P}$ to be $\mathcal{P}'$, all nested lock acquisitions and releases have been removed. Hence, we can refine the Reentrant Semantics to disallow configurations where an ownership array $\bar{o}$ contains at position i for lock l_i an ownership pair (j, z) such that $z > 1$. To distinguish between the Non-Reentrant and Reentrant Semantics, we will use j and $\bot$ to denote the ownership pair $(j, 1)$ and $(\bot, 0)$, respectively. The shorthand is sound because there can be no ambiguity, i.e., (j, z) where $z > 1$ is not allowed. Moreover, to distinguish the Non-Reentrant Semantics, we will use $\rightarrow$ instead of $\rightsquigarrow$ to denote the transition relation between global configurations. In particular, two global configurations g and g' are in the relation $\rightarrow$, denoted by $g \rightarrow g'$, iff $g = (c_1, \ldots, c_j, \ldots, c_n, \bar{o})$ and one of the following conditions holds:

1. $c_j \xrightarrow{a} c'_j$, $a \notin \{\, (_i, \ )_i \,\}$,
   and $g' = (c_1 \ldots, c'_j, \ldots, c_n, \bar{o})$.
2. $c_j \xrightarrow{(_i} c'_j$, $\bar{o}[i] = \bot$,
   and $g' = (c_1, \ldots, c'_j, \ldots, c_n, \bar{o}[i \mapsto j])$.
3. $c_j \xrightarrow{)_i} c'_j$, $\bar{o}[i] = j$,
   and $g' = (c_1, \ldots, c'_j, \ldots, c_n, \bar{o}[i \mapsto \bot])$.

Note that items 1,2, and 3 correspond to items 1,2, and 5 of the Reentrant Semantics. The reflexive transitive closure of $\rightarrow$ is denoted by $g \rightarrow^* g'$. Finally, an execution trace

$g_0 \rightarrow g_1 \rightarrow \ldots \rightarrow g_k$, trace word w, and observable trace word w_Σ are defined as for the Reentrant Semantics.

Because a multi-PDS Π with Reentrant Semantics can be automatically transformed into another multi-PDS Π' with Non-Reentrant Semantics, we consider only multi-PDSs with Non-Reentrant Semantics in the remainder of the paper.

3.2 Property Specification

A program property is specified as an indexed phase automaton.

Definition 3. An ***indexed phase automaton*** (IPA) is a tuple $(Q, \mathit{Id}, \Sigma, \delta)$, 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 $\delta \subseteq Q \times \mathit{Id} \times \Sigma \times Q$ 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. 3 ensures that the only loops in an IPA are self-loops on states. We assume that for every x, $1 \leq x < |Q|$, there is only one phase transition of the form $(q_x, i, a, q_{x+1}) \in \delta$. (An IPA that has multiple such transitions can be factored into a set of IPAs, each of which satisfy this property.) Finally, we only consider IPAs that recognize a non-empty language, which means that an IPA must have exactly $(|Q| - 1)$ phase transitions. IPAs enjoy the *bounded-phase-transition property*.

Property 1 (Bounded Phase Transition). For an IPA $\mathcal{A} = (Q, \mathit{Id}, \Sigma, \delta)$, any run of $\mathcal{A}$ that accepts a word w will make only a *bounded* number of phase transitions. That is, an accepting run of $\mathcal{A}$ on word w will make exactly $(|Q| - 1)$ phase transitions.

For expository purposes, through §9 we only consider 2-PDSs, and fix $\Pi = (\mathcal{P}_1, \mathcal{P}_2, S_{\mathsf{Locks}}, \Sigma)$ and $\mathcal{A} = (Q, \mathit{Id}, \Sigma, \delta)$. §9 shows how to generalize the techniques to multi-PDSs. The implementation, discussed in §8, is for the general case.

4 Problem Statement

Problem 1. Given Π and IPA $\mathcal{A}$, the model-checking problem of interest is to determine if there is an observable trace word w_Σ for an execution trace $g_0 \rightarrow g_1 \rightarrow \ldots \rightarrow g_k$ of Π such that w_Σ is recognized by $\mathcal{A}$.

Before defining the decision procedure to solve Problem 1, we discuss some of the difficulties that need to be overcome.

4.1 Bounded Global Synchronizations

For formalisms that use multiple PDSs to model program threads, reachability analyses that require an a priori unbounded number of global synchronizations are in general undecidable (Ramalingam, 2000). For the CPDSs used by Kidd et al. (2009b), a global synchronization is a communicating action. For the concurrent-PDSs of Qadeer and Rehof (2005) used for context-bounded analysis, a global synchronization is a context switch. To guarantee that a reachability query terminates, both formalisms artificially bound the number of global synchronizations.

Ignoring locks, for Problem 1 a global synchronization occurs when Π makes a transition from a global configuration g to some global configuration g' that causes the IPA $\mathcal{A}$ to make a phase transition. Because of the Bounded-Phase-Transition Property, the number of global synchronizations is bounded, which is a key property that renders the problem decidable.

Remark 2. Returning to AS-serializability violation detection, the language-strength-reduction transformation (Kidd et al., 2008) that eliminates the need to count the depth of nested calls to unit-of-work methods was paramount for bounding the number of global synchronizations because otherwise, the number of "phase transitions" would be a priori unbounded. Informally, with reentrant unit-of-work methods, the language of execution traces that contain an AS-serializability violation is a context-free language that essentially "counts" the depth of nested calls by encoding calls and returns via matched-parentheses. The property specification for detecting AS-serializability violations (i.e., the language of execution traces that contain a problematic access pattern), could not have been encoded as an IPA, which means that our decision procedure could not have been used for AS-serializability violation detection.

Still ignoring locks, the essence of the decision procedure is to identify a *sequence* of global configurations $g_1, \ldots, g_{|Q|}$. The sequence consists of a global configuration for each phase of the IPA $\mathcal{A}$. Because any execution of Π begins from g_0, the first global configuration in the sequence must be g_0. The rest of the global configurations in the sequence are the points at which a phase transition occurs in the IPA $\mathcal{A}$.

Example 1. The IPA $\mathcal{A}_2$ shown in Fig. 2 recognizes execution traces of the 2-PDS generated from the Java program shown in Listing 1 which contains an AS-serializability violation specified by problematic access pattern 12. For the 2-PDS to drive $\mathcal{A}_2$ to its accepting state, it must pass through a sequence of global configurations, beginning from the initial global configuration g_0, such that the transition taken from global configuration g_i to global configuration g_{i+1} in the sequence causes $\mathcal{A}_2$ to make a phase transition. For $\mathcal{A}_2$, there are six phases, and thus the sequence of global configurations will be of length six: one for the initial global configuration plus one for each of the five phase transitions.

Because an execution trace that is recognized by an IPA must cause $|Q| - 1$ phase transitions, the sequence of global configurations must be of length $|Q|$. If such a sequence of configurations exists, then it is possible to drive the IPA $\mathcal{A}$ to its accepting state.

4.2 Accounting for Locks

Kahlon et al. (2005) presents a decision procedure for checking reachability of a set of global configurations of a multi-PDS. That is, for a set of global configurations G, they are interested in answering the query:

> Does there exist a global configuration g in G such that $g_0 \rightarrow^* g$?

To do so, their decision procedure computes lock-usage summaries, known as *acquisition histories*, for the PDS paths leading to the target set of single-PDS configurations for each PDS. A post-processing step then compares the summaries to determine if the target set of global configurations is reachable. For the following discussion, we note that acquisition histories are a finite abstraction—albeit of size $O(2^{\mathcal{L}})$—and can thus be embedded in the control locations of a single PDS. Embedding enables a standard, single-PDS reachability query to be used to compute the lock-usage summaries.

Remark 3. The language-strength-reduction transformation provides a mechanism to replace reentrant locks with non-reentrant locks. Thus, we could use the techniques of Kahlon et al. (2005) to check for reachability of a single set of global configurations. However, as discussed in the previous subsection, we require the ability to check for reachability of a bounded sequence of global configurations.

To check for reachability for a bounded sequence of global configurations, there are two known techniques, both based on *lock histories*, an extension of acquisition histories and formally defined in §6. The first technique, which is used in our decision procedure, is to use *tuples* of lock histories. Tupling enables a mechanism to "remember" or "record" the (set of) lock histories that arise at each global configuration in the desired sequence, and, in addition, maintains the correlations between the lock histories that arise in the sequence. That is, it is not sufficient to merely remember a sequence of sets of lock histories, one set per global configuration in the target sequence, because the correlations that hold between the lock histories that arise along an individual sequence are lost (and, as will be discussed shortly, could be incompatible). Tupling ensures that these correlations are maintained. For AS-serializability-violation detection, the tuple will consist of a lock history for each state of the IPA that accepts traces that contain an AS-serializability violation. For $\mathcal{A}_2$, the tuple would have six lock histories, one for each state q_i, $1 \leq i \leq 6$.

The second technique, due to Kahlon and Gupta (2007), uses a sequence of chained reachability queries to compute a sequence of global configurations, one for each phase transition in the IPA. For $\mathcal{A}_2$, the chain of reachability queries would begin by computing a *set of sets* of global configurations that are compatible (via lock histories) such that $\mathcal{A}_2$ performs a transition from state q_1 to state q_2.[6] Why a set of sets of global configurations? Chaining, unlike tupling, does not provide a mechanism to maintain correlations between the lock histories of intermediate global configurations. Thus, the algorithm of Kahlon and Gupta (2007) must perform chained reachability queries on sets of sets of global configurations.

§11 presents a detailed comparison of the two approaches. For now, we remind the reader that Tab. 1 presents a comparison of the relative complexities of using the two approaches, i.e., tupling versus chaining, for solving Problem 1, and that items 1 and 2 of §1 highlight the benefit of our tupling approach.

5 Path Incompatibility

The decision procedure analyzes the PDSs of Π independently, and then checks if there exists a run from each PDS 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 (2007, Thm. 1), which we now review.

Suppose that Π is in global configuration $g = (c_1, c_2, \bar{o})$. Let $\text{LocksHeld}(\mathcal{P}_k, g)$, $k \in \{1, 2\}$, denote $\{l_i \mid \bar{o}[i] = k\}$; i.e., the set of locks held by PDS $\mathcal{P}_k$ at global configuration g. Furthermore, suppose that PDS $\mathcal{P}_k$, when started in (single-PDS) configuration c_k and executed alone, is able to reach configuration c'_k using the rule sequence $\rho_k = [r_1, \ldots]$.

Before the execution of ρ_k, PDS $\mathcal{P}_k$ has a (possibly empty) set of initially-held locks, i.e., the set $\text{LocksHeld}(\mathcal{P}_k, g)$. After the execution of ρ_k, PDS $\mathcal{P}_k$ will have a (possibly empty) set of finally-held locks. Along 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.

Consider the case that $\Pi = (\mathcal{P}_1, \mathcal{P}_2, \mathsf{Locks}, \Sigma)$ is in global configuration $g = (c_1, c_2, \bar{o})$, and that $\mathcal{P}_k$, while executed alone, can make a transition from configuration c_k to configuration c'_k using rule sequence ρ_k. Kahlon and Gupta's decomposition result characterizes the conditions under which it is not possible for Π to make a transition from global configuration g to $g' = (c'_1, c'_2, \bar{o}')$. Informally, by the semantics of locks, it must be the case that the set of locks held by $\mathcal{P}_1$ and $\mathcal{P}_2$ at global configurations g and g' are disjoint—two PDSs cannot hold the same lock at the same time. Similarly, if $\mathcal{P}_1$ holds a lock l_i throughout ρ_1, then $\mathcal{P}_2$ cannot acquire l_i, and likewise for $\mathcal{P}_2$.

[6] The algorithm presented in Kahlon and Gupta (2007) is stated incorrectly and uses only a set of global configurations. We will discuss this error in more detail in §11 where we present a detailed comparison between our decision procedure and the one of Kahlon and Gupta (2007).

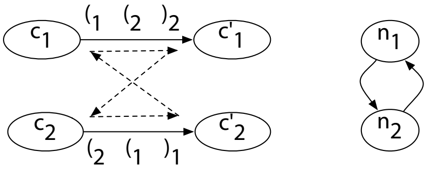

Fig. 3. Individual executions of $\mathcal{P}_1$ and $\mathcal{P}_2$ from configurations c_1 and c_2 to configurations c'_1 and c'_2, respectively. The symbols $(_i$ and $)_i$ denote acquiring and releasing lock l_i, respectively. The dashed arrows denote the sequential and inter-PDS locking dependences due to locking operations. The nodes n_1 and n_2 on the right are the nodes in the graph G of locking dependences. The cycle in the right-hand graph is a proof that a scheduling of ρ_1 and ρ_2 does not exist; it implies the cycle in the scheduling of locking operations indicated by the dashed cycle in the left-hand graph.

Intuition into Kahlon and Gupta's decomposition result can be obtained by considering a dependence graph that represents the locking operations of the PDSs. Let us focus on finally-held locks. (The dual of the following discussion applies to initially-held locks.) Consider a graph $G = (N, E)$, where N is a set of nodes, and E is a set of edges. There is a node n_i in N for each lock l_i. There is an edge (n_i, n_j) if for some PDS $\mathcal{P}_k$, l_i is a finally-held lock, and l_j is acquired (and possibly released) after the final acquisition of l_i in ρ_k. Edges capture (i) the sequential dependences of lock operations by PDS $\mathcal{P}_k$ when executing rule sequence ρ_k, and (ii) the inter-PDS dependences of lock operations. With respect to point (ii), observe that if l_i is a finally-held lock by $\mathcal{P}_1$, then if $\mathcal{P}_2$ acquires l_i in ρ_2, it must also release l_i in ρ_2 before $\mathcal{P}_1$ acquires l_i for the final time. Similar reasoning applies to a finally-held lock l_j of $\mathcal{P}_2$ and lock operations on l_j by $\mathcal{P}_1$.

If there is a cycle in graph G, then a scheduling of ρ_1 and ρ_2 does not exist. Consider the case where G has a cycle consisting of the edges (n_1, n_2) and (n_2, n_1). Because there are outgoing edges from n_1 and n_2, the locks l_1 and l_2 must be finally-held locks. Moreover, they must be finally-held locks of two different PDSs: as we assume non-reentrant locks, $\mathcal{P}_k$ cannot make a final acquisition of l_1, then make a final acquisition of l_2, and then reacquire l_1. Assume that l_1 is finally-held by $\mathcal{P}_1$ and l_2 is finally-held by $\mathcal{P}_2$. The edge (n_1, n_2) means that l_2 is acquired after the final acquisition of l_1 by $\mathcal{P}_1$; the edge (n_2, n_1) means that l_1 is acquired after the final acquisition of l_2 by $\mathcal{P}_2$. Because l_1 is finally-held by $\mathcal{P}_1$, and because the set of finally-held locks of $\mathcal{P}_1$ and $\mathcal{P}_2$ are disjoint, there must be a corresponding release of l_1 by $\mathcal{P}_2$. This scenario is shown in Fig. 3.

For a scheduling to exist, the sequential and inter-PDS dependences due to locking operations must be satisfied. The cycle in the graph G on the right of Fig. 3 shows that no such scheduling exists. That is, the cycle in G captures the cycle that is formed by the dashed edges on the left in Fig. 3, which denote sequential and inter-PDS dependences due to locking operations. The dashed edges are explained as follows:

1. The dashed edge from $(_1$ to $)_2$ denotes that the final acquisition of l_1 by $\mathcal{P}_1$ must come before the release of l_2 by $\mathcal{P}_1$.
2. The dashed edge from $)_2$ to $(_2$ denotes that the release of l_2 by $\mathcal{P}_1$ must come before the final acquisition of l_2 by $\mathcal{P}_2$: $\mathcal{P}_1$ must complete its locking operations on l_2 before $\mathcal{P}_2$ finally acquires l_2.
3. The dashed edge from $(_2$ to $)_1$ denotes that the final acquisition of l_2 by $\mathcal{P}_2$ must come before the release of l_1 by $\mathcal{P}_2$.
4. The dashed edge from $)_1$ to $(_1$ denotes that the release of l_1 by $\mathcal{P}_2$ must come before the final acquisition of l_1 by $\mathcal{P}_1$: $\mathcal{P}_2$ must complete its locking operations on l_1 before $\mathcal{P}_1$ can finally acquire l_1.

Here we have a contradiction: the release of l_1 by $\mathcal{P}_2$ must come *before* the final acquisition of l_1 by $\mathcal{P}_1$ because of item 4; however, it must also come *after* the final acquisition of l_1 by $\mathcal{P}_1$ because of items 1–3 (i.e., the chain of dependences that follows from items 1–3). This argument shows that a cycle in G is a proof that a scheduling of ρ_1 and ρ_2 does not exist.

Similar reasoning to that just given can be used to (i) define a graph H where edges in H denote dependences due to initially-released locks, and (ii) show that a cycle in H is a proof that a scheduling of ρ_1 and ρ_2 does not exist. We now formally state Kahlon and Gupta's decomposition result.

Theorem 1. ***(Decomposition Theorem** (Kahlon and Gupta, 2007).) 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 rule sequences ρ_1 and ρ_2 from global configuration $g = (c_1, c_2, \bar{o})$ to global configuration $g' = (c'_1, c'_2, \bar{o}')$ iff one or more of the following five conditions hold:*

1. *LocksHeld$(\mathcal{P}_1, g) \cap$ LocksHeld$(\mathcal{P}_2, g) \neq \emptyset$: $\mathcal{P}_1$ and $\mathcal{P}_2$ both hold the same lock initially.*
2. *LocksHeld$(\mathcal{P}_1, g') \cap$ LocksHeld$(\mathcal{P}_2, g') \neq \emptyset$: $\mathcal{P}_1$ and $\mathcal{P}_2$ both hold the same lock finally.*
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.

With respect to the discussion above about graphs G and H of locking dependences, Items 3 and 4 correspond to cycles in graphs G and H, respectively. The remaining items model standard lock semantics: only one thread may hold a lock at a given time.

6 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 (Reps, 1998)—in our case, lock-acquire and lock-release—it is useful to consider *semi-Dyck languages* (Harrison, 1978): languages of matched parentheses (Dyck languages) in which each parenthesis symbol is *one-sided* (semi-Dyck languages). That is, the symbols "(" and ")" match in the string "()", but do not match in ")(".[7]

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 σ denotes a member of Σ and $1 \leq i \leq \mathcal{L}$:

$$\begin{array}{rl} \textit{matched} \rightarrow & \epsilon \\ \mid & \sigma \ \textit{matched} \\ \mid & (_i \ \textit{matched} \ )_i \ \textit{matched} \end{array}$$

Because we are interested in paths (rule sequences) that can begin and end while holding a set of locks, we also need to consider prefixes and suffixes of $\mathsf{Lang}(\textit{matched})$, which are languages of partially-matched parentheses. In particular,

- The words in the language of suffixes of $\mathsf{Lang}(\textit{matched})$ may have extra right-parenthesis symbols: every left parenthesis "$(_i$" is balanced by a succeeding right parenthesis "$)_i$", but the converse need not hold. This is called an *unbalanced-right* language.
- The words in the language of prefixes of $\mathsf{Lang}(\textit{matched})$ may have extra left-parenthesis symbols: every right parenthesis "$)_i$" is balanced by a preceding left parenthesis "$(_i$", but the converse need not hold. This is called an *unbalanced-left* language.

It is useful to define the following context-free grammars:

$$\begin{array}{rcl} \textit{unbalR} & \rightarrow & \epsilon \mid \textit{unbalR matched} \ )_i \\ \textit{unbalL} & \rightarrow & \epsilon \mid (_i \ \textit{matched unbalL} \end{array}$$

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

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

Example 2. 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.

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

Let $w_k \in \mathsf{Lang}(\textit{suffixPrefix})$ 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.

For a single PDS's execution ρ_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) (Kahlon and Gupta, 2007) 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.[8]
- The *release history* (RH) (Kahlon and Gupta, 2007) of an initially-held lock l_i, where l_i is not held throughout, 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 are first acquired and then released, i.e., 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.

A lock history is a six-tuple $(\mathsf{R}, \widehat{\mathsf{RH}}, \mathsf{U}, \widehat{\mathsf{AH}}, \mathsf{A}, \mathsf{HT})$:

- R, U, A, and HT are the release, use, acquisition, and held-throughout sets, respectively.
- $\widehat{\mathsf{RH}}$ is a tuple of $\mathcal{L}$ release histories, one for each lock l_i, $1 \leq i \leq \mathcal{L}$.
- $\widehat{\mathsf{AH}}$ is a tuple of $\mathcal{L}$ acquisition 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. In Fig. 4, 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_η, which tracks R, $\widehat{\mathsf{RH}}$, U, $\widehat{\mathsf{AH}}$, A, and HT for each successively longer prefix.

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

$$\begin{array}{ll} \mathsf{R} : \{1, 3, 7\} & \widehat{\mathsf{RH}} : \langle \{1\}, \emptyset, \{1, 2, 3\}, \emptyset, \emptyset, \emptyset, \{1, 2, 3, 4, 5, 6, 7\}, \emptyset, \emptyset \rangle \\ \mathsf{U} : \{6\} & \widehat{\mathsf{AH}} : \langle \emptyset, \{2, 7, 8\}, \emptyset, \{2, 4, 7, 8\}, \emptyset, \emptyset, \emptyset, \{8\}, \emptyset \rangle \\ \mathsf{A} : \{2, 4, 8\} & \mathsf{HT} : \{9\} \end{array}$$

Remark 4. R and A are included above only for clarity; they can be recovered from $\widehat{\mathsf{RH}}$ and $\widehat{\mathsf{AH}}$, as follows: $\mathsf{R} = \{i \mid$

[7] 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.

[8] This is a slight variation from Kahlon and Gupta (2007); we include l_i in the acquisition history of lock l_i.

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

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

Fig. 4. The function eta, which produces a lock histroy LH from a PDS rule sequence $[r_1,\ldots,r_n]$ and a set of initially-held locks $\mathcal{I}$, and the helper function post_η, which produces an updated lock history from a lock history and a PDS action a.

$\widehat{\mathsf{RH}}[i] \neq \emptyset\}$ 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 4. Lock histories $\mathsf{LH}_1 = (\mathsf{R}_1, \widehat{\mathsf{RH}}_1, \mathsf{U}_1, \widehat{\mathsf{AH}}_1, \mathsf{A}_1, \mathsf{HT}_1)$ and $\mathsf{LH}_2 = (\mathsf{R}_2, \widehat{\mathsf{RH}}_2, \mathsf{U}_2, \widehat{\mathsf{AH}}_2, \mathsf{A}_2, \mathsf{HT}_2)$ are ***compatible***, denoted by $\mathsf{Compat}(\mathsf{LH}_1, \mathsf{LH}_2)$, iff all of the following five conditions hold:

1. $(\mathsf{R}_1 \cup \mathsf{HT}_1) \cap (\mathsf{R}_2 \cup \mathsf{HT}_2) = \emptyset$
2. $(\mathsf{A}_1 \cup \mathsf{HT}_1) \cap (\mathsf{A}_2 \cup \mathsf{HT}_2) = \emptyset$
3. $\nexists i, j \,.\, l_j \in \widehat{\mathsf{AH}}_1[i] \wedge l_i \in \widehat{\mathsf{AH}}_2[j]$
4. $\nexists i, j \,.\, l_j \in \widehat{\mathsf{RH}}_1[i] \wedge l_i \in \widehat{\mathsf{RH}}_2[j]$
5. $(\mathsf{A}_1 \cup \mathsf{U}_1) \cap \mathsf{HT}_2 = \emptyset \wedge (\mathsf{A}_2 \cup \mathsf{U}_2) \cap \mathsf{HT}_1 = \emptyset$

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.

7 The Decision Procedure

As noted in §5, 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. 5. The interleaving labeled "Π" is an example interleaved execution that is accepted by the IPA $\mathcal{A}_2$ from §2, namely, the execution shown in Fig. 1. In Fig. 5, only the PDS actions that cause phase transitions are shown. 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, for $\mathcal{P}_2$ to perform the $W_2(d)$ actions, $\mathcal{P}_2$ must hold the lock associated with allocated `Stack` object (alternatively, the underlined $W_2(d)$ action in the interleaved execution shown in Fig. 1 is between the actions $(_s$ and $)_s$, which denote acquiring and releasing the lock associated with the parameter `Stack s` of `SafeWrap.popwrap`). If $\mathcal{P}_1$ guesses that $\mathcal{P}_2$ performs the $W_2(d)$ action at a point when it currently holds the lock associated with the parameter `Stack s`, then the behavior is inconsistent with the semantics of Π because both threads would hold the lock associated with the randomly-isolated object. The post-processing step ensures that such behaviors are not allowed.

7.1 *Combining a* PDS *with an* IPA

To define a modular algorithm, we must be able to analyze $\mathcal{P}_1$ and $\mathcal{A}$ independently of $\mathcal{P}_2$, and likewise for $\mathcal{P}_2$ and $\mathcal{A}$ relative to $\mathcal{P}_1$. Our approach is to combine $\mathcal{A}$ and $\mathcal{P}_i$, $1 \leq i \leq 2$, to define a new PDS $\mathcal{P}_i^{\mathcal{A}}$ using a cross-product-like construction. The main difference is that lock histories and lock-history updates are incorporated in the construction.

Recall that the goal is to determine if there exists an execution of Π that drives $\mathcal{A}$ to its final state. Because of the bounded-phase-transition property, we know that any such execution must make $|Q| - 1$ phase transitions. Hence, a valid interleaved execution must be able to reach $|Q|$ global configurations, one for each of the $|Q|$ phases.

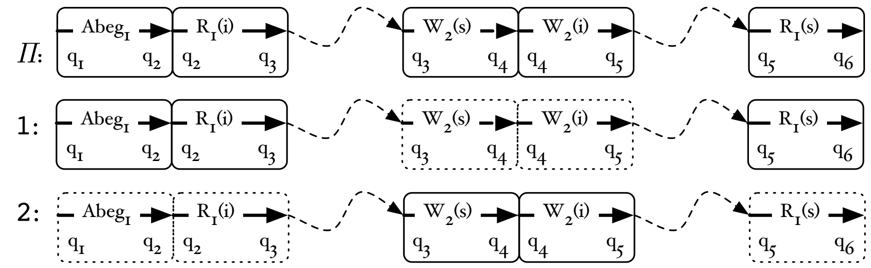

Fig. 5. Π: a bad interleaving that is recognized by $\mathcal{A}_2$ (see page 6), showing only the actions that cause a phase transition. 1: the same interleaving from Thread 1's point of view. The dashed boxes show where Thread 1 guesses that Thread 2 causes a phase transition. 2: the same but from Thread 2's point of view where dashed boxes where where Thread 2 guesses that Thread 1 causes a phase transition.

Lock histories encode the constraints that a PDS path places on the set of possible interleaved executions of Π. A desired path of an individual PDS must also make $|Q| - 1$ phase transitions, and hence our algorithm keeps track of $|Q|$ lock histories, one for each phase. This is accomplished by encoding into the state space of $\mathcal{P}_i^{\mathcal{A}}$ a *tuple* of $|Q|$ lock histories. A tuple maintains the sequence of lock histories for one or more paths taken through a sequence of phases. In addition, a tuple maintains the *correlation* between the lock histories of each phase, which is necessary to ensure that only valid executions are considered. The rules of $\mathcal{P}_i^{\mathcal{A}}$ are then defined to update the lock-history tuple accordingly. The lock-history tuples are used later to check whether some scheduling of an execution of Π can actually perform all of the required phase transitions.

Let $\mathcal{LH}$ denote the set of all lock histories, and let $\widehat{\mathcal{LH}} = \mathcal{LH}^{|Q|}$ denote 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 $\widehat{\mathsf{LH}}$. $\widehat{\mathsf{LH}}[k]$ denotes the k^{th} component of $\widehat{\mathsf{LH}}$.

Our construction makes use of the phase-transition function on LHs defined as follows:

$$\mathsf{ptrans}((\mathsf{R}, \widehat{\mathsf{RH}}, \mathsf{U}, \widehat{\mathsf{AH}}, \mathsf{A}, \mathsf{HT})) = (\emptyset, \emptyset^{\mathcal{L}}, \emptyset, \emptyset^{\mathcal{L}}, \emptyset, \mathsf{A} \cup \mathsf{HT}).$$

The ptrans 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 $\mathcal{P}_i = (P_i, \mathsf{Lab}_i, \Gamma_i, \Delta_i, \langle p_0, \gamma_0 \rangle)$ be a PDS, S_{Locks} be a set of locks of size $\mathcal{L}$, $\mathcal{A} = (Q, Id, \Sigma, \delta)$ be an IPA, 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 $P_i^{\mathcal{A}} \subseteq P_i \times Q \times \widehat{\mathcal{LH}}$. The initial control state is $p_0^{\mathcal{A}} = (p_0, q_1, \widehat{\mathsf{LH}}_\emptyset)$, where $\widehat{\mathsf{LH}}_\emptyset$ is the empty 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 is an expression that evaluates to an LH. Two kinds of rules are introduced to account for whether a transition in δ is a phase transition or not. (The update to $\widehat{\mathsf{LH}}$ is listed after each rule kind.)

1. **Non-phase Transitions**: $\widehat{\mathsf{LH}}' = \widehat{\mathsf{LH}}[x \mapsto \mathsf{post}_\eta(\widehat{\mathsf{LH}}[x], a)]$.

   (a) For each rule $\langle p, \gamma \rangle \overset{a}{\hookrightarrow} \langle p', u \rangle \in \Delta_i$ and transition $(q_x, i, a, q_x) \in \delta$, there is a rule of the form: $\langle (p, q_x, \widehat{\mathsf{LH}}), \gamma \rangle \hookrightarrow \langle (p', q_x, \widehat{\mathsf{LH}}'), u \rangle \in \Delta_i^{\mathcal{A}}$. Rules of this form ensure that $\mathcal{P}_i^{\mathcal{A}}$ is constrained to follow the self-loops on IPA state q_x.

   (b) For each rule $\langle p, \gamma \rangle \overset{a}{\hookrightarrow} \langle p', u \rangle \in \Delta_i$, $a \in \{(_k,)_k\}$, and each $q_x \in Q$, there is a rule of the form: $\langle (p, q_x, \widehat{\mathsf{LH}}), \gamma \rangle \hookrightarrow \langle (p', q_x, \widehat{\mathsf{LH}}'), u \rangle \in \Delta_i^{\mathcal{A}}$. Rules of this form record the acquisition or release of the lock l_k in the lock-history tuple $\widehat{\mathsf{LH}}$ at index x. (Recall that the language of an IPA 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.)

2. **Phase Transitions**: $\widehat{\mathsf{LH}}' = \widehat{\mathsf{LH}}[(x{+}1) \mapsto \mathsf{ptrans}(\widehat{\mathsf{LH}}[x])]$.

   (a) For each rule $\langle p, \gamma \rangle \overset{a}{\hookrightarrow} \langle p', u \rangle \in \Delta_i$ and transition $(q_x, i, a, q_{x+1}) \in \delta$, there is a rule of the form: $\langle (p, q_x, \widehat{\mathsf{LH}}), \gamma \rangle \hookrightarrow \langle (p', q_{x+1}, \widehat{\mathsf{LH}}'), u \rangle \in \Delta_i^{\mathcal{A}}$. Rules of this form perform a phase transition on the lock-history tuple $\widehat{\mathsf{LH}}$ for PDS $\mathcal{P}_i$.

   (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 of the form: $\langle (p, q_x, \widehat{\mathsf{LH}}), \gamma \rangle \hookrightarrow \langle (p, q_{x+1}, \widehat{\mathsf{LH}}'), \gamma \rangle \in \Delta^{\mathcal{A}}$. Rules of this form 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.

Given $\mathcal{P}^{\mathcal{A}}$, one can compute the set of all reachable configurations via the query $\mathcal{A}_{post^*} = post^*_{\mathcal{P}^{\mathcal{A}}}(\langle p_0^{\mathcal{A}}, \gamma_0 \rangle)$ using standard automata-based PDS techniques (Bouajjani et al., 1997; Finkel et al., 1997). (Because the initial configuration is defined by the PDS $\mathcal{P}^{\mathcal{A}}$, henceforth, we merely write $post^*_{\mathcal{P}^{\mathcal{A}}}$.) A configuration $c \in \mathcal{A}_{post^*}$ will be of the form $\langle (p, q, \widehat{\mathsf{LH}}), u \rangle$, where p is a state of the original PDS $\mathcal{P}$, q is a state of the IPA $\mathcal{A}$, $\widehat{\mathsf{LH}}$ is a lock-history tuple, and $u \in \Gamma^*$ is a reachable stack. The lock-history tuple $\widehat{\mathsf{LH}}$ encodes the locking constraints of all paths from $\langle p_0^{\mathcal{A}}, \gamma_0 \rangle$ to the configuration c.

Algorithm 1: The decision procedure.

```
  input  : A 2-PDS $\Pi = (\mathcal{P}_1, \mathcal{P}_2, \mathsf{Locks}, \Sigma)$ and a IPA $\mathcal{A}$.
  output: true if $\Pi$ can drive $\mathcal{A}$ to its accepting state.
1 let $\mathcal{A}^1_{post^*} \leftarrow post^*_{\mathcal{P}^{\mathcal{A}}_1}$; let $\mathcal{A}^2_{post^*} \leftarrow post^*_{\mathcal{P}^{\mathcal{A}}_2}$;
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 Compat($\widehat{LH}_1, \widehat{LH}_2$) then
5             return true;
6 return false;
```

7.2 Checking Path Compatibility

For a generated PDS $\mathcal{P}^{\mathcal{A}}_k$, we are interested in the set of paths that begin in the initial configuration $\langle p^{\mathcal{A}}_0, \gamma_0 \rangle$ and drive $\mathcal{A}$ to its accepting state $q_{|Q|}$. Each such 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 $\mathcal{P}^{\mathcal{A}}_1$ and $\mathcal{P}^{\mathcal{A}}_2$, 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:

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

Due to recursion, $\mathcal{P}^{\mathcal{A}}_1$ and $\mathcal{P}^{\mathcal{A}}_2$ 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 $\mathcal{P}^{\mathcal{A}} = (P^{\mathcal{A}}, \mathsf{Lab}, \Gamma, \Delta, c^{\mathcal{A}}_0)$, the set of relevant $\widehat{\mathsf{LH}}$ tuples are found in the $\mathcal{P}^{\mathcal{A}}_k$-automaton $\mathcal{A}^k_{post^*}$ that results from the *post** operation. That is, one merely enumerates the state space of $\mathcal{A}^k_{post^*}$, extracting the $\widehat{\mathsf{LH}}$ tuples that are members of a state $(p, q_{|Q|}, \widehat{\mathsf{LH}})$. By only considering states that have an IPA state-component of $q_{|Q|}$, we ensure that the PDS $\mathcal{P}_k$ performed the required $(|Q| - 1)$ phase transitions.

Alg. 1 gives the algorithm to check whether Π can drive $\mathcal{A}$ to its accepting state. The two tests on lines 2 and 3 of the form "$\exists u_k \in \Gamma_k^* : \langle (p_k, q_{|Q|}, \widehat{\mathsf{LH}}_k), u_k \rangle \in L(\mathcal{A}^k_{post^*})$", where $L(\mathcal{A}^k_{post^*})$ is the language of the $\mathcal{P}_k$-automaton, can be performed by finding *any* path in $\mathcal{A}^k_{post^*}$ from state $(p_k, q_{|Q|}, \widehat{\mathsf{LH}}_k)$ to the accepting state.

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

Proof. The proof builds on Thm. 1 by showing that for runs ρ_1 and ρ_2 of PDSs $\mathcal{P}_1$ and $\mathcal{P}_2$, respectively, that reach the target set of configurations of their respective PDSs, there exists a scheduling of ρ_1 and ρ_2 for each phase by the proof of Thm. 1. Because each phase can be scheduled, there exists a scheduling for runs ρ_1 and ρ_2.

In particular, from Thm. 1, we know that rule sequences ρ_1 and ρ_2 from PDSs $\mathcal{P}_1$ and $\mathcal{P}_2$, respectively, where ρ_1 and ρ_2 begin execution from configurations with a disjoint set of initially held locks $\mathcal{I}_1$ and $\mathcal{I}_2$, there exists a compatible scheduling of ρ_1 and ρ_2 iff $\mathsf{Compat}(\eta(\rho_1, \mathcal{I}_1), \eta(\rho_2, \mathcal{I}_2))$.

If Alg. 1 returns true, then there exists two tuples of lock histories, $\widehat{\mathsf{LH}}_1$ and $\widehat{\mathsf{LH}}_2$, where $\widehat{\mathsf{LH}}_1$ ($\widehat{\mathsf{LH}}_2$) is an abstraction of a rule sequence ρ_1 (ρ_2) from the initial configuration of PDS $\mathcal{P}_1$ ($\mathcal{P}_2$) that drives the IPA $\mathcal{A}$ to the accepting state, such that $\mathsf{Compat}(\widehat{\mathsf{LH}}_1, \widehat{\mathsf{LH}}_2)$. Because of the Decomposition Theorem and the definition of $\mathsf{Compat}(\widehat{\mathsf{LH}}_1, \widehat{\mathsf{LH}}_2)$, there must exist a scheduling of ρ_1 and ρ_2 that adheres to the interleaved semantics of of Π. That is, there must exist an interleaved scheduling of ρ_1 and ρ_2 that causes Π, starting from the initial global configuration g_0, to pass through a *sequence* of configurations such that a phase transition occurs at each intermediate configuration, and finally to reach a configuration such that the IPA $\mathcal{A}$ is in its accepting state.

If Alg. 1 returns false, then there does not exist two such tuples of locks histories. This can occur if one (or both) of the PDSs does not have a path that can drive the IPA $\mathcal{A}$ to the accepting state, and thus it is not possible for an interleaved execution of Π to drive IPA $\mathcal{A}$ to the accepting state. Alternatively, there are a pair of tuples for rule sequences that drive the respective PDSs to their final states, but the tuples are not in the Compat relation. From the definition of Compat, there must be some phase such that the lock histories are incompatible, and thus no interleaved execution exists. □

8 A Symbolic Implementation

Alg. 1 solves the multi-PDS model-checking problem for IPAs. 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 (Schwoon, 2002); 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 $Q \times \widehat{\mathcal{LH}}$, 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 an IPA, 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.

Our symbolic implementation encodes the lock history tuples as a *semiring* or *weight domain* for use with *weighted pushdown systems* (WPDSs) (Reps et al., 2005), where a

WPDS equips a PDS with a semiring, and annotates the PDS rules with elements from the semiring's domain. Reps et al. (2005) showed the strong connection between interprocedural dataflow analysis and reachability queries on WPDSs. We now formally define these concepts.

Definition 5. A ***bounded idempotent semiring*** is a tuple $\mathcal{S} = (D, \oplus, \otimes, \overline{0}, \overline{1})$, where D is a finite set of elements called ***weights***, $\overline{0}, \overline{1} \in D$, and $\oplus$ (the ***combine*** operation) and $\otimes$ (the ***extend*** operation) are binary operations on D such that

1. $(D, \oplus)$ is an commutative monoid with neutral element $\overline{0}$, where $\oplus$ is idempotent: $\forall x \in D, x \oplus x = x$.
2. $(D, \otimes)$ is a monoid with neutral element $\overline{1}$.
3. $\otimes$ distributes over $\oplus$: $\forall x, y, z \in D$,
$$x \otimes (y \oplus z) = (x \otimes y) \oplus (x \otimes z)$$
$$\text{and } (x \oplus y) \otimes z = (x \otimes z) \oplus (y \otimes z).$$
4. $\overline{0}$ is an annihilator with respect to $\otimes$: $\forall x \in D, x \otimes \overline{0} = \overline{0} = \overline{0} \otimes x$.
5. In the partial order $\sqsubseteq$ defined by $\forall x, y \in D,\ x \sqsubseteq y$ iff $x \oplus y = x$, there are no infinite descending chains.

Definition 6. A ***weighted PDS*** (WPDS) is a tuple $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$, where $\mathcal{P} = (P, \Gamma, \mathsf{Lab}, \Delta, c_0)$ 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 from PDS rules to weights. We abuse notation by defining $f : \Delta^* \to D$ as f overloaded to operate on a rule sequence $\sigma = [r_1, \ldots, r_n]$ as follows: $f(\sigma) = f(r_1) \otimes \ldots \otimes f(r_n)$.

For a WPDS $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$ and configurations c and c', the set of PDS paths *paths*(c, c') is defined on the underlying PDS $\mathcal{P}$. For a set of configurations C, reachability queries for PDSs—*post** and *pre**—are generalized for WPDSs as follows:

$$\mathit{post}^*(C) =_{\mathrm{df}}$$
$$\{\, (c', w) \ \mid\ \exists c \in C : c \Rightarrow^* c' \wedge w = \bigoplus_{\rho \in \mathit{paths}(c,c')} f(\rho)\}$$

$$\mathit{pre}^*(C) =_{\mathrm{df}}$$
$$\{\, (c', w) \ \mid\ \exists c \in C : c' \Rightarrow^* c \wedge w = \bigoplus_{\rho \in \mathit{paths}(c,c')} f(\rho)\}$$

We now present the particular semiring that we use in our analysis algorithm, the *multi-arity relational weight domain*, whose elements are generalized relations, which we call *θ-term formal power series*.

Definition 7. 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 8. Let S be a finite set, and $\theta > 0$ be a bound. The set of all ***θ-term formal power series over z, with relation-valued coefficients of different arities***, is

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

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 $(\mathcal{RFPS}[S, \theta], \times, +, \mathit{Id}, \emptyset)$, 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 $\emptyset z^0$.

Remark 5. A multi-arity relational weight domain over S and θ, as defined in Defn. 8, meets the requirements of a bounded idempotent semiring (Defn. 5) 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. 7 (";") is a monoid, and (iii) ";" is both left- and right-distributive over union of arity-k relations.

We now define the WPDS $\mathcal{W}_i = (\mathcal{P}_i^{\mathcal{W}}, \mathcal{S}, f)$ that results from taking the product of PDS $\mathcal{P}_i = (P_i, \mathsf{Lab}_i, \Gamma_i, \Delta_i, \langle p_0, \gamma_0\rangle)$ and phase automaton $\mathcal{A} = (Q, \mathit{Id}, \Sigma, \delta)$. The construction is similar to that in §7.1, 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**: $f(r) = \{\langle \mathsf{LH}, \mathsf{post}_\eta(\mathsf{LH}, a)\rangle \mid \mathsf{LH} \in \mathcal{LH}\} z^0$.
   (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 $r = \langle (p, q_x), \gamma\rangle \hookrightarrow \langle (p', q_x), u\rangle \in \Delta_i^{\mathcal{W}}$.
   (b) For each rule $\langle p, \gamma\rangle \xhookrightarrow{a} \langle p', u\rangle \in \Delta_i$, $a \in \{(_k,)_k\}$, and for each $q_x \in Q$, there is a rule $r = \langle (p, q_x), \gamma\rangle \hookrightarrow \langle (p', q_x), u\rangle \in \Delta_i^{\mathcal{W}}$.
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 $r = \langle (p, q_x), \gamma\rangle \hookrightarrow \langle (p, q_{x+1}), \gamma\rangle \in \Delta^{\mathcal{W}}$.

Algorithm 2: The symbolic decision procedure.

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

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 $\mathcal{W}_i$, IPA $\mathcal{A}$ 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 $c_{x-1} z^{x-1}$, i.e., c_{x-1} is a relation of arity $x + 1$ (a subset of $\mathcal{LH}^{x+1}$). The maximum number of phases in a IPA 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 Reps et al. (2005), only monomial-valued weights ever arise.)

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

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

Proof (Sketch). The proof proceeds by showing that the multi-arity relations that annotate the rules of $\mathcal{W}$ simulate the change in control state of the rules of $\mathcal{P}^{\mathcal{A}}$, and vice versa. This, combined with the proofs of correctness of algorithms for solving reachability problems in PDSs (Bouajjani et al., 1997; Finkel et al., 1997) and WPDSs (Bouajjani et al., 2003; Reps et al., 2005), proves that Alg. 2 computes the same result as Alg. 1. The proof then reduces to the proof of correctness for Alg. 1, which is given in §7. The full proof of simulation is given in App. A.

9 Generalizing to More Than Two PDSs

Because the set of reachable configurations, and hence the set of lock-history tuples, are computed independently for each PDS, the construction from §7.1 that combines a PDS $\mathcal{P}$ with a IPA $\mathcal{A}$ to form a new PDS $\mathcal{P}^{\mathcal{A}}$ does not change when generalizing to N PDSs. Hence, the only modification required to define a decision procedure for an N-PDS is to generalize the compatibility check for N lock-history tuples.

Generalizing the compatibility check to N lock-history tuples requires a generalization of the *Decomposition Theorem* (Thm. 1, page 10). The extension of items 1, 2, and 5 to N lock-history tuples is straightforward.

1. $\mathrm{LocksHeld}(\mathcal{P}_1, g) \cap \ldots \cap \mathrm{LocksHeld}(\mathcal{P}_N, g) \neq \emptyset$.
2. $\mathrm{LocksHeld}(\mathcal{P}_1, g') \cap \ldots \cap \mathrm{LocksHeld}(\mathcal{P}_N, g') \neq \emptyset$.
5. (a) In ρ_i, $\mathcal{P}_i$ acquires or uses a lock that is held by $\mathcal{P}_j$, $j \neq i$ throughout ρ_j, or
   (b) in ρ_i, $\mathcal{P}_i$ acquires or uses a lock that is held by $\mathcal{P}_j$, $j \neq i$ throughout ρ_j.

Items 1 and 2 ensure that the set of locks held at the initial and final global configurations, respectively, are disjoint. Item 5 ensures that a PDS $\mathcal{P}$ does not acquire or use a lock that is held throughout by another PDS $\mathcal{P}'$.

In Thm. 1, Items 3 and 4 define incompatibility to be a cycle of length two in the acquisition and release histories, respectively. The generalization for an N-PDS is to check for a cycle of length anywhere from 2 to N. For example, consider a 3-PDS with three (or more) locks. The absence of a cycle of length three in a tuple of acquisition histories would then be defined as:

$$\nexists i, j, k : l_i \in \widehat{\mathsf{AH}}_1[j] \wedge l_j \in \widehat{\mathsf{AH}}_2[k] \wedge l_k \in \widehat{\mathsf{AH}}_3[i].$$

The absence of a cycle of length three is defined similarly for release histories. We use the notation $\mathsf{Compat}(\mathsf{LH}_1, \ldots, \mathsf{LH}_N)$ 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 $\mathsf{Compat}(\widehat{\mathsf{LH}}_1, \ldots, \widehat{\mathsf{LH}}_N)$.

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

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

As in Alg. 2, the compatibility check can be performed via a single BDD operation by defining the N-way compatibility relation.

10 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 IPAMC, which implements Alg. 2, against CPDSMC, which implements the communicating-pushdown system (CPDS) semi-decision procedure from (Kidd et al., 2009b). IPAMC is implemented using the WALI WPDS library (Kidd et al., 2009a), and the multi-arity relational weight domain uses the BuDDy BDD library (BuDDy, 2004). (The

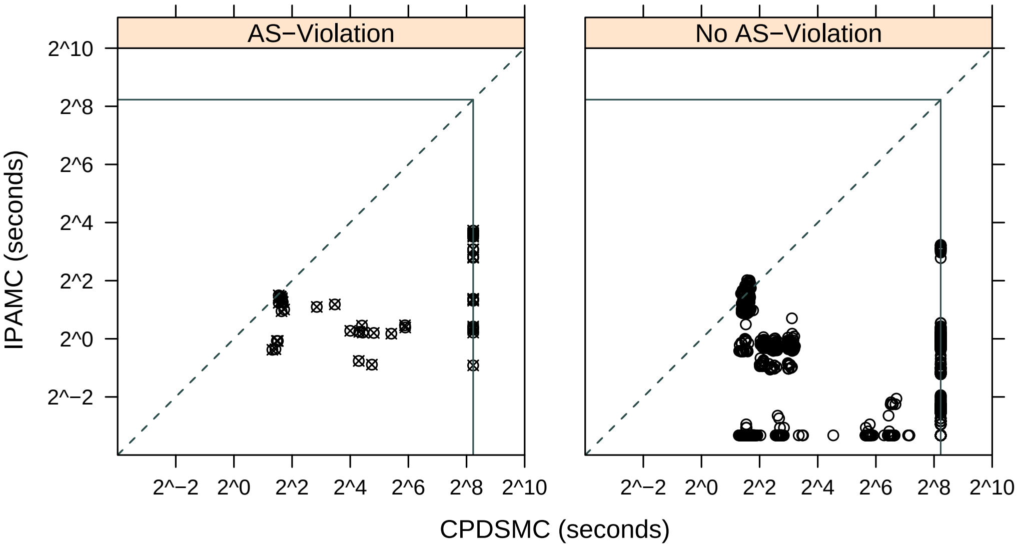

Fig. 6. Log-log scatter-plots of the execution times of IPAMC (y-axis) versus CPDSMC (x-axis). The left-hand graph shows the 49 queries for which IPAMC reported an AS-serializability violation; the right-hand graph shows the 2,096 queries for which IPAMC verified correctness. The dashed lines denote equal running times; points below and to the right of the dashed lines are runs for which IPAMC was faster. The timeout threshold was 300 seconds, and is marked by the solid vertical and horizontal lines that form an inner box. The minimum reported time is 0.1 second.

multi-arity relational weight domain is included in WALI starting with release 3.0.) 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 (Eytani et al., 2007b). Our tool, EMPIRE, requires that the allocation site of interest be annotated in the source program. We annotated eleven of 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 (Kidd et al., 2009b) 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 (Kidd et al., 2009b) 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 IPAs 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 IPAs that involve a single shared location, we ran one query for each $m \in S_{\mathsf{Mem}}$. For each of the nine IPAs 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 2,145 queries.

The purpose of the experiment was to determine the answer to the queries performed by Kidd et al. (2009b) on which CPDSMC exhausted resources (roughly 50%), and to compare the performance of the symbolic-decision procedure (Alg. 2) implemented in IPAMC to the semi-decision procedure that is implemented in CPDSMC. The analysis summaries are shown in Tab. 2. As expected, the decision procedure returned a definitive answer for all queries.

	Query Category		
	CPDSMC succeeded (1,074)	CPDSMC timed out (1,071)	Total (2,145)
CPDSMC	10,000	82,400	92,400
IPAMC	1,400	1,300	2,700
Speedup	7X	62X	34X

Table 3. Total time (in seconds) for examples classified according to whether CPDSMC succeeded or timed out.

Although the CPDS-based method is a semi-decision procedure, it is capable of both (i) verifying correctness, and (ii) finding AS-serializability violations (see Chaki et al. (2006); Kidd et al. (2009b)). (The third possibility is that it fails to provide a definite answer because it times out or runs out of memory.) Fig. 6 presents log-log scatter plots of the execution times of IPAMC (y-axis) versus CPDSMC (x-axis). The "inner box" in each scatter plot marks the timeout threshold of 300 seconds.[9] The vertical bands on the "inner box" in each scatter plot are queries where CPDSMC timed out. The dashed-diagonal line denotes equal running times. Points below and to the right of the dashed line are queries where IPAMC was faster than CPDSMC. Because almost every point is below and to the right of the dashed-diagonal line in Fig. 6, we can see that IPAMC performed better than CPDSMC on nearly every query.

Tab. 3 presents a comparison of the total time to execute all queries. The total times are partitioned according to

[9] A 300-second timeout is used because that is the point where CPDSMC is able to answer roughly half of the queries.

Benchmark	# Queries	Viol	OK	OOM	OOT
SoftwareVerificationHW	15	6 (4)	9 (5)	0	0 (6)
BugTester	615	0	615 (460)	0 (155)	0
BuggyProgram	615	16 (0)	599	0	0 (16)
shop	900	27 (6)	873 (0)	0 (839)	0 (55)
Totals	2145	49 (10)	2096 (1064)	0 (994)	0 (77)

Table 2. For each benchmark, column "# Queries" gives the number of queries that were generated (i.e., the number of IPAs generated for a benchmark's model Π); columns "Viol" and "OK" give the breakdown of query satisfaction (i.e., an execution of Π is and is not, respectively, recognized by a generated IPA); columns "OOM" and "OOT" specify the number of queries that exhausted memory and time resources, respectively. The counts in parentheses show where CPDSMC did not perform as well as IPAMC.

Program	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SoftwareVerificationHW	X	X	X		X									
BugTester														
BuggyProgram		✓		✓							✓	✓	✓	
shop	✓	✓		X		X	X	X	X	X	✓	X	X	X

Table 4. Marked entries denote violations reported by EMPIRE. An entry marked with "✓" was found using both IPAMC and CPDSMC. An entry marked with "X" was found only using IPAMC.

whether CPDSMC succeeded or timed out. Comparing the total time to run all queries, IPAMC ran 34X faster (92,400 seconds versus 2,700 seconds). For queries on which both IPAMC and CPDSMC returned definitive answers, IPAMC ran 7X faster (10,000 seconds versus 1,400 seconds). Moreover, CPDSMC timed out, or ran out of memory, on about 50% of the queries—both for the ones for which IPAMC reported an AS-serializability violation (39 timeouts out of 49 queries), as well as the ones for which IPAMC verified correctness (1,064 timeouts out of 2,096 queries).

Tab. 4 breaks down the AS-serializability violations found according to the problematic access pattern that occurred. Entries marked with an "X" are AS-serializability violations that EMPIRE found using IPAMC but *did not* find using CPDSMC because CPDSMC exhausted the available resources.[10] The number of additional AS-serializability violations detected clearly shows the benefit of using a decision procedure over a semi-decision procedure.

11 Comparison with the Kahlon-Gupta Decision Procedure

We now present a more detailed comparison of the decision procedure from §7 with the (corrected) decision procedure of Kahlon and Gupta (2007).

The Kahlon-Gupta decision procedure takes as input a multi-PDS and an LTL formula φ. For our comparison, we will only consider a formula φ that consists of the temporal operators eventually F and next X, and a 2-PDS $\Pi = (\mathcal{P}_1, \mathcal{P}_2, S_{\mathsf{Locks}})$.

The Kahlon-Gupta decision procedure also uses lock histories; however, unlike our decision procedure (§7), they do not use lock-history tuples but merely a single lock history. Thus, each PDS $\mathcal{P}_i$ is augmented so that its set of control locations P_i includes a lock history. To model a global configuration of Π, they use a configuration pair (c_1, c_2), where c_1 and c_2 are configurations of $\mathcal{P}_1$ and $\mathcal{P}_2$, respectively. A set of global configurations G is represented as a pair of sets of configurations (C_1, C_2). That is, $G = (C_1, C_2)$ represents the set of global configurations $\{\ (c_1, c_2) \mid c_1 \in C_1, c_2 \in C_2\ \}$. Finally, for a set of global configurations $G = (C_1, C_2)$, their algorithm must maintain the invariant that for each pair of configurations $(c_1, c_2) \in G$, lock histories LH_1 and LH_2 that annotate the control locations of c_1 and c_2, respectively, are in the Compat relation—i.e., that $\mathsf{Compat}(\mathsf{LH}_1, \mathsf{LH}_2)$ holds.

The Kahlon-Gupta decision procedure is defined inductively. For a given logical formula φ, and 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. For example, let $G = (C_1, C_2)$ be the automaton-pair that satisfies a subformula. If the next-outermost temporal operator is F, then they would define $G' = (C_1', C_2')$, where C_i' is obtained by performing a *pre** query on PDS $\mathcal{P}_i$ beginning from C_i. The automaton-pair G' is thus the pairing of the automata that result from the two *pre** queries.

We observed that the decision procedure as presented in Kahlon and Gupta (2007) contains an error, which Kahlon and Gupta confirmed in email correspondence (Kahlon and Gupta, 2009). For two automata-pairs $G = (C_1, C_2)$ and $G' = (C_1', C_2')$, they claimed that disjunction distributes

[10] Due to a limitation of our implementation in not producing witness traces for failed queries (sometimes called "counterexamples"), we are not able to check whether the additional AS-serializability violations found using IPAMC are actual bugs or false positives. The lack of witness traces is not a fundamental limitation of the approach used in IPAMC, just of our current implementation; in principle, it is possible to extend IPAMC to produce witness traces.

across automata-pairs—i.e., that $G \vee G' = (C_1 \vee C_1', C_2 \vee C_2')$. Disjunction does not distribute because it loses correlations that need to be maintained. To illustrate this point, consider the following two sets of global configurations: $G = (\{c_1\}, \{c_2\})$ and $G' = (\{c_1'\}, \{c_2'\})$. If one takes the disjunction $G \vee G'$ as defined by Kahlon and Gupta (2007), the result would be $G \vee G' = (\{c_1, c_1'\}, \{c_2, c_2'\})$, which allows for the global configuration (c_1, c_2') to be in the disjunction when it is not in G or G'. Moreover, because correlations related to the Compat relation can be lost, the necessary invariant discussed above can be violated.

We can now explain why *sets* of automaton pairs are required to correct their algorithm. The Kahlon-Gupta algorithm must maintain the invariant that for an automaton pair $G = (C_1, C_2)$, the lock-history component of all configuration pairs $(c_1 \in C_1, c_2 \in C_2)$ must be in the compatible relation (i.e., $\mathsf{Compat}(\mathsf{LH}_1, \mathsf{LH}_2)$, where LH_i, $1 \leq i \leq 2$, is the lock history component of the control location of configuration c_i). To maintain the invariant, after computing the individual reachability query on each automaton C_i (e.g., *pre*$^*(C_i)$), the resulting automata cannot be simply paired back together because disjunction does not distribute. Instead, sets of automaton-pairs must be defined so that (i) the invariant continues to hold, and (ii) the compatibility invariant is maintained.

To translate a 2-PDS $\mathit{\Pi}$ and an IPA $\mathcal{A}$ into the input format of Kahlon and Gupta (2007), one would have to perform two steps.

1. The input formula φ is specified over the control locations of the individual PDSs. Thus, the control locations of the PDSs of $\mathit{\Pi}$ must be expanded to include the states of $\mathcal{A}$. To do so, one would need to perform the cross product of $\mathcal{P}_i$, $1 \leq i \leq 2$, and $\mathcal{A}$.
2. The IPA $\mathcal{A}$ must be compiled into an LTL formula. Intuitively, for a 2-PDS, an IPA $\mathcal{A}$ can be expressed as a 2-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 proposition meaning that the control state of each (augmented) PDS satisfies q_x. That is, the control state of the PDS that results from the cross product of PDS $\mathcal{P}$ with IPA $\mathcal{A}$ is of the form (p, q_x). The following function can be used to translate an IPA $\mathcal{A}$ into a 2-indexed LTL formula:
$$\begin{aligned} H(q_{|Q|}) &= S_{q_{|Q|}} \\ H(q_x) &= \mathsf{F}(S_{q_x} \wedge \mathsf{X}(\mathsf{X}(S_{q_{x+1}} \wedge H(q_{x+1})))) \end{aligned}$$
In particular, $\varphi_{\mathcal{A}}$ is $H(q_1)$.

The Kahlon-Gupta decision procedure would proceed by augmenting the input PDSs with lock histories (not lock-history tuples). For all compatible lock histories LH_1 and LH_2 (i.e., $\forall \mathsf{LH}_1, \mathsf{LH}_2 \in \mathcal{LH} : \mathsf{Compat}(\mathsf{LH}_1, \mathsf{LH}_2)$), the query of interest is then whether any of the following configuration pairs are reachable from the initial configuration:

$$\{(\langle (p_1, q_{|Q|}, \mathsf{LH}_1), u_1 \rangle, \langle (p_2, q_{|Q|}, \mathsf{LH}_2), u_2 \rangle) \mid \\ p_1 \in P_1, u_1 \in \Gamma_1^*, p_2 \in P_2, u_2 \in \Gamma_2^*\}.$$

For each state q_x of $\mathcal{A}$, the function $H(q_x)$ introduces three temporal operators. Thus, the (corrected) Kahlon-Gupta decision procedure would require $(3 * |Q|)$ inductive “steps” to be performed on each PDS, where a step for the temporal operators X and F requires a single-step post query and a reachability *post** query, respectively. Each step operates on a set of automaton-pairs. In the worst case, the size of the set of automaton-pairs is of size exponential in the number of locks. Thus, in the worst case, their algorithm must perform $(3 * |Q|) * 2^{|S_{\mathsf{Locks}}|}$ queries for each PDS.

To implement the (corrected) Kahlon-Gupta algorithm, there are two problems that appear to be difficult to overcome. First, the number of queries is exponential in the number of locks, which is not desirable because the cost of each query is also exponential in the number of locks—the cost of a PDS *pre** query has a linear factor in the size of the control locations, which is exponential in the number of locks because of the use of lock histories. Second, the straightforward approach for reestablishing the invariant after performing the individual *pre** queries on the PDSs $\mathcal{P}_1$ and $\mathcal{P}_2$—i.e., defining the set of automaton-pairs—is to enumerate the states of the automata C_1 and C_2 that result from the *pre** queries of $\mathcal{P}_1$ and $\mathcal{P}_2$, respectively. Enumeration is not desirable because it requires enumerating over two sets that are each of size exponential in the number of locks (i.e., the lock histories).

This paper introduces a different technique than that used by Kahlon and Gupta. Our algorithm uses WPDS weights that are sets of lock-history tuples, whereas Kahlon and Gupta use sets of pairs of configuration automata. Our algorithm also has a much different structure than theirs. The (corrected) Kahlon and Gupta 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.

By moving from lock histories to *tuples* of lock histories, the decision procedure presented in §7 does not require multiple reachability queries. Consequently, it does not need to perform the disjunction of automata that result from intermediate reachability queries as is required by Kahlon and Gupta (2007). The use of lock-history tuples has the following benefits:

1. We avoid the need to perform an exponential number of queries on each PDS because sets of automaton-pairs are not required.
2. Because tupling maintains correlations between the intermediate configurations of an individual PDS $\mathcal{P}_i$, we do not need to (re)establish the invariant that Kahlon and Gupta (2007) did for performing chained reachability queries. Besides avoiding the need to operate on automaton-pairs

as discussed above, not being forced to (re)establish the invariant avoids the enumeration of the control locations of automata C_1 and C_2 that result from the intermediate *pre** queries of Kahlon and Gupta (2007).

We note that tupling is not free: the size of the set of control locations of each PDS has an extra exponential factor, namely, the size of the set of states Q of $\mathcal{A}$. However, isolating exponential factors in the PDS control locations is favored because symbolic techniques such as BDDs can often represent exponentially large state spaces in an efficient manner. Finally, Tab. 5 repeats the comparison table from the beginning of the paper to emphasize that worst-case running time of our algorithm has one less exponential factor when compared to the (corrected) Kahlon-Gupta algorithm. In particular, see the rightmost column.

12 Related Work

§11 has already presented a detailed comparison with the work of Kahlon and Gupta (2007). In short, Kahlon and Gupta provide an algorithm that can be used to solve the problem addressed in this paper; however, as discussed in detail in §11, their paper contains an error, and a corrected algorithm is much more costly than the asymptotic costs claimed in their paper. Overall, the asymptotic cost of our algorithm is lower than the asymptotic cost of their algorithm (see Tab. 5).

Another approach to model checking concurrent software is *context-bounded analysis*, first defined by Qadeer and Rehof (2005) and later improved by Lal et al. (2008). Context-bounded analysis bounds the number of context switches that are explored (while letting processes perform an arbitrary number of computation steps in between context switches). In contrast, our approach bounds the number of phases, but permits an unbounded number of context switches and an unbounded number of lock acquisitions and releases by each PDS. Moreover, the decision procedures from §7 and §8 are able to explore the entire state space of the model; thus, our algorithms are able to verify properties of multi-PDSs instead of just performing bug detection.

Dynamic pushdown networks (DPNs) (Bouajjani et al., 2005) extend parallel PDSs with the ability to create threads dynamically. Lammich et al. (2009) present a generalization of acquisition histories to DPNs with properly-nested locks. Their algorithm uses chained *pre** queries, an explicit encoding of acquisition histories in the state space, and is not implemented.

13 Conclusion

To sum up, we show that the following problem is decidable:

> Given a program consisting of a fixed, finite number of threads that can use (i) reentrant locks, (ii) an unbounded number of context switches, and (iii) an unbounded number of lock acquisitions and releases, determine whether the lock-constrained executions of the program contain any sequence of interleaved memory accesses that match a given problematic access pattern.

	LTL/Atomicity	CBA
Explicit (splitting)	Kahlon and Gupta (2007)	Qadeer and Rehof (2005)
Symbolic (tupling)	Kidd et al. (2009c) (atomicity)	Lal et al. (2008)

Table 6. Related work on LTL/atomicity checking and context-bounded analysis (CBA). Each row specifies whether the approach uses an explicit modeling of the reachable configurations, which requires *splitting*, or a symbolic modeling via the use of *tupling*

In addition to the algorithms presented in this paper, the result also relies on the language-strength-reduction transformation Kidd et al. (2008), which is used to (i) replace reentrant locks with non-reentrant locks while retaining the same set of behaviors, and (ii) eliminating the counters that count the depth of nested calls to unit-of-work methods. In the intended application, the threads are Java threads, and the problematic access patterns are those for AS-serializability violations. However, the decision procedure is applicable for any problem in which the program can be modeled as a multi-PDS and the property can be specified as an IPA.

The symbolic decision procedure (Alg. 2) has been implemented and used to detect AS-serializability violations in automatically generated models of four concurrent Java programs from the CONTEST benchmark suite (Eytani et al., 2007b). When compared to a previous approach that uses CPDS model checking, which implements only a semi-decision procedure, the implementation of Alg. 2 was *34* times faster overall. Moreover, with a timeout threshold of 300 seconds, for each query where the semi-decision procedure timed out or ran out of memory (roughly 50% of the queries), the decision procedure succeeded within the allotted time, and actually performed more work because the decision procedure explored the *entire* state space.

Another contribution of the paper is that it sheds light on what appears to be a general principle for reducing the cost of model checking concurrent software, namely, the advantages of *tupling* over *splitting* for maintaining required correlations. The issue arises both in our setting—-verifying data-consistency properties—as well as in context-bounded analysis (Qadeer and Rehof, 2005; Lal et al., 2008); see Tab. 6. In both settings, the model checker needs to establish facts about reachability for a sequence of intermediate global configurations of a multi-PDS, while maintaining information about certain kinds of correlations between intermediate configurations.

– The decision procedure presented in this paper uses tupling to maintain the correlations between intermediate configurations, whereas the (corrected) Kahlon and Gupta (2007) algorithm uses splitting. As shown in Tab. 5, the

	PDS control locations	Queries	Cost
Chaining	$O(2^{\mathcal{L}})$	$O(2^{\mathcal{L}\cdot\lvert\mathcal{A}\rvert}\cdot\lvert S_{\mathsf{Procs}}\rvert)$	$O(2^{\mathcal{L}}\cdot 2^{\mathcal{L}\cdot\lvert\mathcal{A}\rvert}\cdot\lvert S_{\mathsf{Procs}}\rvert)$
Tupling	$O(2^{\mathcal{L}\cdot\lvert\mathcal{A}\rvert})$	$\lvert S_{\mathsf{Procs}}\rvert$	$O(2^{\mathcal{L}\cdot\lvert\mathcal{A}\rvert}\cdot\lvert S_{\mathsf{Procs}}\rvert)$

Table 5. Comparison between the (corrected) chaining approach of Kahlon and Gupta (2007) and our tupling approach. $\mathcal{L}$ denotes the number of locks, $|\mathcal{A}|$ denotes the number of states of an IPA, and $|S_{\mathsf{Procs}}|$ denotes the number of EML processes (PDSs).

cost of the tupling-based method is asymptotically lower than that of the splitting-based method: tupling avoids a factor that is exponential in the number of locks. Tupling also isolates an exponential cost in the PDS control locations, which is beneficial because that cost can often be side-stepped using symbolic techniques, such as BDDs.

- In context-bounded analysis (Qadeer and Rehof, 2005; Lal et al., 2008), the algorithm of Qadeer and Rehof (2005) uses splitting to maintain correlations. In that setting, they enumerate the global state space at a context switch so that they can maintain correlations between the global data state and the (unbounded-size) program stack for a given thread. In contrast, Lal et al. (2008) use a form of tupling to maintain correlations, which also enables them to isolate an exponential cost in the PDS control locations.[11]

In general, the use of tupling is preferred because

1. Tupling may permit an exponential-cost enumeration step to be avoided (and thus lower the worst-case asymptotic cost).
2. Tupling enables symbolic techniques to be employed, which often allows the remaining exponential factors to be overcome in practice.

References

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

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

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

BuDDy (2004). A BDD package. `http://buddy.wiki.sourceforge.net/`.

Chaki, S., Clarke, E., Kidd, N., Reps, T., and Touili, T. (2006). Verifying concurrent message-passing C programs with recursive calls. In *TACAS*.

Eytani, Y., Havelund, K., Stoller, S. D., and Ur, S. (2007a). Towards a framework and a benchmark for testing tools for multi-threaded programs. *Conc. and Comp.: Prac. and Exp.*, 19(3).

Eytani, Y., Havelund, K., Stoller, S. D., and Ur, S. (2007b). Towards a framework and a benchmark for testing tools for multi-threaded programs. *Conc. and Comp.: Prac. and Exp.*, 19(3).

Finkel, A., B.Willems, and Wolper, P. (1997). A direct symbolic approach to model checking pushdown systems. *Elec. Notes in Theor. Comp. Sci.*, 9.

Flanagan, C. and Qadeer, S. (2003). A type and effect system for atomicity. In *PLDI*.

Harrison, M. (1978). *Introduction to Formal Language Theory*. Addison-Wesley, Reading, MA.

Kahlon, V. and Gupta, A. (2007). On the analysis of interacting pushdown systems. In *POPL*.

Kahlon, V. and Gupta, A. (2009). Personal communication.

Kahlon, V., Ivancic, F., and Gupta, A. (2005). Reasoning about threads communicating via locks. In *CAV*.

Kidd, N., Lal, A., and Reps, T. (2008). Language strength reduction. In *SAS*.

Kidd, N., Lal, A., and Reps, T. (2009a). WALi: The Weighted Automaton Library. `http://www.cs.wisc.edu/wpis/wpds/download.php`.

Kidd, N., Reps, T., Dolby, J., and Vaziri, M. (2009b). Finding concurrency-related bugs using random isolation. In *VMCAI*.

Kidd, N. A., Lammich, P., Touili, T., and Reps, T. (2009c). A decision procedure for detecting atomicity violations for communicating processes with locks. In *SPIN*.

Lal, A., Touili, T., Kidd, N., and Reps, T. (2008). Interprocedural analysis of concurrent programs under a context bound. In *TACAS*.

Lammich, P., Müller-Olm, M., and Wenner, A. (2009). Predecessor sets of dynamic pushdown networks with tree-regular constraints. In *CAV*.

Lu, S., Park, S., Seo, E., and Zhou, Y. (2008). Learning from mistakes—a comprehensive study on real world concurrency bug characteristics. In *ASPLOS*.

Qadeer, S. and Rehof, J. (2005). Context-bounded model checking of concurrent software. In *TACAS*.

Ramalingam, G. (2000). Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Transactions on Programming Languages and Systems*, 22.

Reps, T. (1998). Program analysis via graph reachability. *Inf. and Softw. Tech.*, 40.

Reps, T., Schwoon, S., Jha, S., and Melski, D. (2005). Weighted pushdown systems and their application to in-

[11] The work of Lal et al. (2008) equips WPDS semirings with a *tensor* operation that can be viewed as means for defining tuples for weight domains of infinite size. For finite weight domains, such as lock histories, explicit tuples suffice.

terprocedural dataflow analysis. *SCP*, 58.

Schwoon, S. (2002). *Model-Checking Pushdown Systems*. PhD thesis, TUM.

Vaziri, M., Tip, F., and Dolby, J. (2006). Associating synchronization constraints with data in an object-oriented language. In *POPL*.

Visser, W. (2009). Personal communication.

A Proof of Thm. 3

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

Proof. The proof proceeds as follows: (i) show by induction that Alg. 1 and Alg. 2 compute the same lock-history tuples for related PDS paths; and (ii) combine the previous step with the proof of correctness for WPDSs. We use the following definitions.

1. $\mathcal{P} = (P, \mathsf{Lab}, \Gamma, \Delta, c_0)$ is a PDS
2. $\mathcal{A} = (Q, \mathit{Id}, \Sigma, \delta)$ is a IPA
3. $\mathcal{P}^{\mathcal{A}} = \big(P^{\mathcal{A}}, \emptyset, \Gamma, \Delta^{\mathcal{A}}, \langle (p_0, q_1, \widehat{\mathsf{LH}}_0), \gamma_0 \rangle\big)$ is the (unlabeled) PDS that results from combining $\mathcal{P}$ with $\mathcal{A}$ as defined in §7.1
4. $\mathcal{W} = \big((P \times Q, \emptyset, \Gamma, \Delta^{\mathcal{W}}, \langle (p_0, q_1), \gamma_0 \rangle), \mathcal{S}, f\big)$ is the WPDS that results from combining $\mathcal{P}$ with $\mathcal{A}$ as defined in §8
5. $\rho^{\mathcal{P}} = [r_1^{\mathcal{P}}, \ldots, r_n^{\mathcal{P}}]$ is a rule sequence from $\mathcal{P}$
6. $\rho^{\mathcal{A}} = [r_1^{\mathcal{A}}, \ldots, r_n^{\mathcal{A}}]$ is a rule sequence from $\mathcal{P}^{\mathcal{A}}$
7. $\rho^{\mathcal{W}} = [r_1^{\mathcal{W}}, \ldots, r_n^{\mathcal{W}}]$ is a rule sequence from $\mathcal{W}$
8. $\mathit{val}(\rho^{\mathcal{W}}) = f(r_1^{\mathcal{W}}) \otimes \ldots \otimes f(r_n^{\mathcal{W}})$ is the weighted valuation of $\rho^{\mathcal{W}}$
9. $\mathsf{inflate}(c_{x-1} z^{x-1}, x) = c_{x-1} z^{x-1}; \big\{ \langle \mathsf{LH}, \mathsf{LH}, \mathsf{LH}_0^{|Q|-x} \rangle \mid \mathsf{LH} \in \mathcal{LH} \big\} z^{|Q|-x}$
10. $\mathsf{deflate}(c_{|Q|-1} z^{|Q|-1}, x) = \big\{ \langle \mathsf{LH}_1, \ldots, \mathsf{LH}_x, \mathsf{LH}_{x+1} \rangle \mid \langle \mathsf{LH}_1, \ldots, \mathsf{LH}_x, \mathsf{LH}_{x+1}, \ldots, \mathsf{LH}_{|Q|-1} \rangle \in c_{|Q|-1} \big\} z^{x-1}$

Item 9 defines the inflate function that takes a monomial of arity m and transforms it into a monomial of arity $|Q| - 1$. This is necessary for comparing the result of executing a rule sequence $\rho^{\mathcal{A}}$ of $\mathcal{P}^{\mathcal{A}}$ with executing a rule sequence $\rho^{\mathcal{W}}$ of $\mathcal{W}$ because $\rho^{\mathcal{W}}$ might not have performed $|Q| - 1$ phase transitions. The function inflate "appends" the empty lock history LH_0 to the end of the monomial $c_{x-1} z^{x-1}$. This coincides with the fact that a path from the initial configuration of $\mathcal{P}^{\mathcal{A}}$ only modifies the lock-history tuple entries for the phases that it has been in or is currently executing in. The function deflate simply undoes the result of inflate, i.e., $c_{x-1} z^{x-1} = \mathsf{deflate}(\mathsf{inflate}(c_{x-1} z^{x-1}, x), x)$.

Let $c_0^{\mathcal{A}} \Rightarrow^{\rho^{\mathcal{A}}} c^{\mathcal{A}}$ denote that $\mathcal{P}^{\mathcal{A}}$ makes a transition to a configuration $c^{\mathcal{A}}$ from configuration $c_0^{\mathcal{A}}$ when executing rule sequence $\rho^{\mathcal{A}}$. Similarly, let $c_0^{\mathcal{W}} \Rightarrow^{\rho^{\mathcal{W}}} c^{\mathcal{W}}$ denote that $\mathcal{W}$ makes a transition to a configuration $c^{\mathcal{W}}$ from configuration $c_0^{\mathcal{W}}$ when executing rule sequence $\rho^{\mathcal{W}}$. We show the following:

$$\begin{aligned} & c_0^{\mathcal{A}} \Rightarrow^{\rho^{\mathcal{A}}} \langle (p, q_x, \widehat{\mathsf{LH}}), u \rangle \\ & \quad \Longleftrightarrow \\ & c_0^{\mathcal{W}} \Rightarrow^{\rho^{\mathcal{W}}} \langle (p, q_x), u \rangle \wedge \langle \mathsf{LH}_0, \widehat{\mathsf{LH}} \rangle \in \mathsf{inflate}(\mathit{val}(\rho^{\mathcal{W}}), x). \end{aligned}$$

The proofs in both directions are by induction on the length of a rule sequence.

Show ⇒.
For rule sequence $\rho^{\mathcal{A}} = [r_1^{\mathcal{A}}, \ldots, r_n^{\mathcal{A}}]$, assume that $c_0^{\mathcal{A}} \Rightarrow^{\rho^{\mathcal{A}}} \langle (p, q_x, \widehat{\mathsf{LH}}), u \rangle$. We show how to construct a rule sequence $\rho^{\mathcal{W}} = [r_1^{\mathcal{W}}, \ldots, r_n^{\mathcal{W}}]$ such that (i) $c_0^{\mathcal{W}} \Rightarrow^{\rho^{\mathcal{W}}} \langle (p, q_x), u \rangle$ and

(ii) $\langle \mathsf{LH}_0, \widehat{\mathsf{LH}}\rangle \in \mathsf{inflate}(\mathit{val}(\rho^{\mathcal{W}}), x)$. For each case, we rely on the fact that the generalized relational product always composes on the rightmost tuple-component in the left-hand-side operand. This allows us to show that the "effect" of extending weights when firing a rule sequence of $\mathcal{W}$ mimics the explicit change in the control state of $\mathcal{P}^{\mathcal{A}}$ that occurs when firing a rule sequence of $\mathcal{P}^{\mathcal{A}}$.

- **Base case:** $n = 1$.

  For the base case, there is only one rule: $r_1^{\mathcal{A}} = \langle (p_0, q_1, \widehat{\mathsf{LH}}_0), \gamma_0\rangle \hookrightarrow \langle (p, q_x, \widehat{\mathsf{LH}}), u\rangle$. From the definition of $\mathcal{P}^{\mathcal{A}}$, there must be the rule $r_1^{\mathcal{P}} = \langle p_0, \gamma_0\rangle \xhookrightarrow{a} \langle p, u\rangle$ in the original PDS $\mathcal{P}$, and a transition $(q_1, i, a, q_x) \in \delta$. Thus, by the definition of $\mathcal{W}$, there must be the rule $r_1^{\mathcal{W}} = \langle (p_0, q_1), \gamma_0\rangle \hookrightarrow \langle (p, q_x), u\rangle$. We perform a case analysis on $r_1^{\mathcal{A}}$ to show that $\langle \mathsf{LH}_0, \widehat{\mathsf{LH}}\rangle \in \mathsf{inflate}(\mathit{val}(r_1^{\mathcal{W}}), x)$.

  1. If $x = 1$, then
     (a) $\widehat{\mathsf{LH}} = \widehat{\mathsf{LH}}_0[1 \mapsto \mathsf{post}_\eta(\widehat{\mathsf{LH}}_0[1], a)] =$
     $\langle \mathsf{post}_\eta(\mathsf{LH}_0, a), \mathsf{LH}_0^{|Q|-1}\rangle$
     (b) $f(r_1^{\mathcal{W}}) = \{\langle \mathsf{LH}, \mathsf{post}_\eta(\mathsf{LH}, a)\rangle \mid \mathsf{LH} \in \mathcal{LH}\}z^0$
     (c) $\mathsf{inflate}(\mathit{val}([r_1^{\mathcal{W}}]), 1) =$
     $\{\langle \mathsf{LH}, \mathsf{post}_\eta(\mathsf{LH}, a), \mathsf{LH}_0^{|Q|-1}\rangle \mid \mathsf{LH} \in \mathcal{LH}\}z^{|Q|-1}$
     (d) $\langle \mathsf{LH}_0, \widehat{\mathsf{LH}}\rangle \in \mathsf{inflate}(\mathit{val}([r_1^{\mathcal{W}}]), 1)$
  2. Otherwise $x = 2$, then
     (a) $\widehat{\mathsf{LH}} = \widehat{\mathsf{LH}}_0[2 \mapsto \mathsf{ptrans}(\widehat{\mathsf{LH}}_0[1])] =$
     $\langle \mathsf{LH}_0, \mathsf{ptrans}(\mathsf{LH}_0), \mathsf{LH}_0^{|Q|-2}\rangle$
     (b) $f(r_1^{\mathcal{W}}) = \{\langle \mathsf{LH}, \mathsf{LH}, \mathsf{ptrans}(\mathsf{LH})\rangle \mid \mathsf{LH} \in \mathcal{LH}\}z^1$
     (c) $\mathsf{inflate}(\mathit{val}([r_1^{\mathcal{W}}]), 2) =$
     $\{\langle \mathsf{LH}, \mathsf{LH}, \mathsf{ptrans}(\mathsf{LH}), \mathsf{LH}_0^{|Q|-2}\rangle$
     $\mid \mathsf{LH} \in \mathcal{LH}\}z^{|Q|-1}$
     (d) $\langle \mathsf{LH}_0, \widehat{\mathsf{LH}}\rangle \in \mathsf{inflate}(\mathit{val}([r_1^{\mathcal{W}}]), 2)$

- **Inductive step.**

  Now consider the rule sequence $\rho_n^{\mathcal{A}} = [r_1^{\mathcal{A}}, \ldots, r_{n-1}^{\mathcal{A}}, r_n^{\mathcal{A}}]$, and assume that for the first $n-1$ rules of the sequence, $c_0^{\mathcal{A}} \Rightarrow^{\rho_{n-1}^{\mathcal{A}}} \langle (p, q_x, \widehat{\mathsf{LH}}), \gamma u\rangle$. Furthermore, let us use the notation $\widehat{\mathsf{LH}} = \langle \mathsf{LH}^1, \ldots, \mathsf{LH}^x, \mathsf{LH}_0^{x+1}, \ldots, \mathsf{LH}_0^{|Q|}\rangle$ so that we can deconstruct the $\widehat{\mathsf{LH}}$ tuple. (Note that it must be the case that at all tuple indices greater than x the lock history is LH_0 by construction.) By the induction hypothesis we have the following: there exists a rule sequence $\rho_n^{\mathcal{W}} = [r_1^{\mathcal{W}}, \ldots, r_{n-1}^{\mathcal{W}}]$ such that $c_0^{\mathcal{W}} \Rightarrow^{\rho_{n-1}^{\mathcal{W}}} \langle (p, q_x), \gamma u\rangle$ and $\langle \mathsf{LH}_0, \widehat{\mathsf{LH}}\rangle \in \mathsf{inflate}(\mathit{val}(\rho_{n-1}^{\mathcal{W}}), x)$. In addition, the following holds: $\langle \mathsf{LH}_0, \mathsf{LH}^1, \ldots, \mathsf{LH}^x\rangle \in \mathsf{deflate}(\mathsf{inflate}(\mathit{val}(\rho_{n-1}^{\mathcal{W}}), x), x)$
  Let $r_n^{\mathcal{A}} = \langle (p, q_x, \widehat{\mathsf{LH}}), \gamma\rangle \hookrightarrow \langle (p', q_y, \widehat{\mathsf{LH}}'), u'\rangle$, then $c_0^{\mathcal{A}} \Rightarrow^{\rho_n^{\mathcal{A}}} \langle (p', q_y, \widehat{\mathsf{LH}}'), u'u\rangle$. From the definition of $\mathcal{P}^{\mathcal{A}}$, there must exist a rule $\langle p, \gamma\rangle \xhookrightarrow{a} \langle p', u'\rangle \in \Delta$ and transition $(q_x, i, a, q_y) \in \delta$. Thus, from the definition of $\mathcal{W}$, there exists a rule $r_n^{\mathcal{W}} = \langle (p, q_x), \gamma\rangle \hookrightarrow \langle (p', q_y), u'\rangle \in \Delta^{\mathcal{W}}$, and $c_0^{\mathcal{W}} \Rightarrow^{\rho_n^{\mathcal{W}}} \langle (p', q_y), u'u\rangle$, which satisfies condition (i) above. To show that condition (ii) above is satisfied, i.e., that $\langle \mathsf{LH}_0, \widehat{\mathsf{LH}}'\rangle \in \mathsf{inflate}(\mathit{val}(\rho_n^{\mathcal{W}}), y)$, we perform a case analysis on the rule $r_n^{\mathcal{A}} = \langle (p, q_x, \widehat{\mathsf{LH}}), \gamma\rangle \hookrightarrow \langle (p', q_y, \widehat{\mathsf{LH}}'), u'\rangle$.

  1. If $x = y$, then
     (a) $\widehat{\mathsf{LH}}' = \widehat{\mathsf{LH}}[x \mapsto \mathsf{post}_\eta(\widehat{\mathsf{LH}}[x], a)] =$
     $\langle \mathsf{LH}^1, \ldots, \mathsf{post}_\eta(\mathsf{LH}^x, a), \mathsf{LH}_0^{x+1}, \ldots, \mathsf{LH}_0^{|Q|}\rangle$
     (b) $f(r_n^{\mathcal{W}}) = \{\langle \mathsf{LH}, \mathsf{post}_\eta(\mathsf{LH}, a)\rangle \mid \mathsf{LH} \in \mathcal{LH}\}z^0$
     (c) $\langle \mathsf{LH}_0, \mathsf{LH}^1, \ldots, \mathsf{LH}^x\rangle \in$
     $\mathsf{deflate}(\mathsf{inflate}(\mathit{val}(\rho_{n-1}^{\mathcal{W}}), x), x)$, by the induction hypothesis
     (d) $\langle \mathsf{LH}_0, \mathsf{LH}^1, \ldots, \mathsf{post}_\eta(\mathsf{LH}^x, a)\rangle \in$
     $\mathsf{deflate}(\mathsf{inflate}(\mathit{val}(\rho_{n-1}^{\mathcal{W}}), x), x) \otimes f(r_n^{\mathcal{W}})$
     (e) $\mathsf{inflate}(\mathit{val}(\rho_n^{\mathcal{W}}), x) = \mathsf{inflate}(\mathit{val}(\rho_{n-1}^{\mathcal{W}}) \otimes f(r_1^{\mathcal{W}}), x)$
     (f) $\langle \mathsf{LH}_0, \widehat{\mathsf{LH}}'\rangle \in \mathsf{inflate}(\mathit{val}(\rho_n^{\mathcal{W}}), x)$
  2. Otherwise $y = x + 1$, and
     (a) $\widehat{\mathsf{LH}}' = \widehat{\mathsf{LH}}[y \mapsto \mathsf{ptrans}(\widehat{\mathsf{LH}}[x])] =$
     $\langle \mathsf{LH}^1, \ldots, \mathsf{LH}^x, \mathsf{ptrans}(\mathsf{LH}^x), \ldots, \mathsf{LH}_0^{|Q|}\rangle$
     (b) $f(r_n^{\mathcal{W}}) = \{\langle \mathsf{LH}, \mathsf{LH}, \mathsf{ptrans}(\mathsf{LH})\rangle \mid \mathsf{LH} \in \mathcal{LH}\}z^1$.
     (c) $\langle \mathsf{LH}_0, \mathsf{LH}^1, \ldots, \mathsf{LH}^x\rangle \in$
     $\mathsf{deflate}(\mathsf{inflate}(\mathit{val}(\rho_{n-1}^{\mathcal{W}}), x), x)$, by the induction hypothesis
     (d) $\langle \mathsf{LH}_0, \mathsf{LH}^1, \ldots, \mathsf{LH}^x, \mathsf{ptrans}(\mathsf{LH}^x)\rangle \in$
     $\mathsf{deflate}(\mathsf{inflate}(\mathit{val}(\rho_{n-1}^{\mathcal{W}}), x), x) \otimes f(r_n^{\mathcal{W}})$
     (e) $\mathsf{inflate}(\mathit{val}(\rho_n^{\mathcal{W}}), y) = \mathsf{inflate}(\mathit{val}(\rho_{n-1}^{\mathcal{W}}) \otimes f(r_1^{\mathcal{W}}), y)$
     (f) $\langle \mathsf{LH}_0, \widehat{\mathsf{LH}}'\rangle \in \mathsf{inflate}(\mathit{val}(\rho_n^{\mathcal{W}}), y)$

Show $\Leftarrow$.
For a rule sequence $\rho_n^{\mathcal{W}} = [r_1^{\mathcal{W}}, \ldots, r_n^{\mathcal{W}}]$, assume that $c_0^{\mathcal{W}} \Rightarrow^{\rho^{\mathcal{W}}} \langle (p, q_x), u\rangle$ and that $\langle \mathsf{LH}_0, \widehat{\mathsf{LH}}\rangle \in \mathsf{inflate}(\mathit{val}(\rho_n^{\mathcal{W}}), x)$. We show how to construct a rule sequence $\rho^{\mathcal{A}} = [r_1^{\mathcal{A}}, \ldots, r_n^{\mathcal{A}}]$ such that $c_0^{\mathcal{A}} \Rightarrow^{\rho^{\mathcal{A}}} \langle (p, q_x, \widehat{\mathsf{LH}}), u\rangle$. The proof is by induction on the length n of the rule sequence.

- **Base case:** $n = 1$.

  For the base case, there is only one rule: $r_1^{\mathcal{W}} = \langle (p_0, q_1), \gamma_0\rangle \hookrightarrow \langle (p, q_x), u\rangle$. From the definition of $\mathcal{W}$, there must exist the rule $r_1^{\mathcal{P}} = \langle p_0, \gamma_0\rangle \xhookrightarrow{a} \langle p, u\rangle \in \Delta$, and a transition $(q_1, i, a, q_x) \in \delta$. Thus, by the definition of $\mathcal{P}^{\mathcal{A}}$, there must be a rule $r_1^{\mathcal{A}} = \langle (p_0, q_1, \widehat{\mathsf{LH}}_0), \gamma_0\rangle \hookrightarrow \langle (p, q_x, \widehat{\mathsf{LH}}'), u\rangle$. We perform a case analysis on $r_1^{\mathcal{W}}$ to show that $\langle \mathsf{LH}_0, \widehat{\mathsf{LH}}\rangle \in \mathsf{inflate}(\mathit{val}([r_1^{\mathcal{W}}]), x) \Rightarrow \widehat{\mathsf{LH}}' = \widehat{\mathsf{LH}}$.

  1. If $x = 1$, then
     (a) $f(r_1^{\mathcal{W}}) = \{\langle \mathsf{LH}, \mathsf{post}_\eta(\mathsf{LH}, a)\rangle \mid \mathsf{LH} \in \mathcal{LH}\}z^0$
     (b) $\langle \mathsf{LH}_0, \mathsf{post}_\eta(\mathsf{LH}_0, a), \mathsf{LH}_0^{Q-1}\rangle \in \mathsf{inflate}(\mathit{val}([r_1^{\mathcal{W}}]), x)$
     (c) $\widehat{\mathsf{LH}}' = \widehat{\mathsf{LH}}_0[1 \mapsto \mathsf{post}_\eta(\widehat{\mathsf{LH}}_0[1], a)] =$
     $\langle \mathsf{post}_\eta(\mathsf{LH}_0, a), \mathsf{LH}_0^{|Q|-1}\rangle$.
     (d) $\widehat{\mathsf{LH}}' = \widehat{\mathsf{LH}}$
  2. Otherwise $x = 2$, then

  (a) $f(r_1^{\mathcal{W}}) = c_1 z^1 = \{\langle \mathsf{LH}, \mathsf{LH}, \mathsf{ptrans}(\mathsf{LH})\rangle \mid \mathsf{LH} \in \mathcal{LH}\} z^1$.
  (b) $\langle \mathsf{LH}_0, \mathsf{LH}_0, \mathsf{ptrans}(\mathsf{LH}_0), \mathsf{LH}_0^{|Q-2|}\rangle \in \mathsf{inflate}(\mathit{val}([r_1^{\mathcal{W}}]), 2)$.
  (c) $\widehat{\mathsf{LH}}' = \widehat{\mathsf{LH}}_0[2 \mapsto \mathsf{ptrans}(\widehat{\mathsf{LH}}_0[1])] = \langle \mathsf{LH}_0, \mathsf{ptrans}(\mathsf{LH}_0), \mathsf{LH}_0^{|Q|-2}\rangle$.
  (d) $\widehat{\mathsf{LH}}' = \widehat{\mathsf{LH}}$

- **Inductive step.**

  Now consider the rule sequence $\rho_n^{\mathcal{W}} = [r_1^{\mathcal{W}}, \ldots, r_{n-1}^{\mathcal{W}}, r_n^{\mathcal{W}}]$, and assume that for the first $n-1$ rules of the sequence, $c_0^{\mathcal{W}} \Rightarrow^{\rho_{n-1}^{\mathcal{W}}} \langle (p, q_x), \gamma u\rangle$. Let $r_n^{\mathcal{W}} = \langle (p, q_x), \gamma\rangle \hookrightarrow \langle (p', q_y), u'\rangle$, then $c_0^{\mathcal{W}} \Rightarrow^{\rho_n^{\mathcal{W}}} \langle (p', q_y), u'u\rangle$. By the induction hypothesis we have the following: for $\langle \mathsf{LH}_0, \widehat{\mathsf{LH}}\rangle \in \mathsf{inflate}(\mathit{val}(\rho_{n-1}^{\mathcal{W}}), x)$, there exists a rule sequence $\rho_{n-1}^{\mathcal{A}} = [r_1^{\mathcal{A}}, \ldots, r_{n-1}^{\mathcal{A}}]$ such that $c_0^{\mathcal{A}} \Rightarrow^{\rho_{n-1}^{\mathcal{A}}} \langle (p, q_x, \widehat{\mathsf{LH}}), \gamma u\rangle$. Furthermore, let $\widehat{\mathsf{LH}} = \langle \mathsf{LH}^1, \ldots, \mathsf{LH}^x, \mathsf{LH}_0^{x+1}, \ldots, \mathsf{LH}_0^{|Q|}\rangle$; then $\langle \mathsf{LH}_0, \mathsf{LH}^1, \ldots, \mathsf{LH}^x\rangle \in \mathit{val}(\rho_{n-1}^{\mathcal{W}})$.
  From the definition of $\mathcal{W}$, there must exist a rule $\langle p, \gamma\rangle \xhookrightarrow{a} \langle p', u'\rangle \in \Delta$ and transition $(q_x, i, a, q_y) \in \delta$. From the definition of $\mathcal{P}^{\mathcal{A}}$, there must exist a rule $r_n^{\mathcal{A}} = \langle (p, q_x, \widehat{\mathsf{LH}}), \gamma\rangle \hookrightarrow \langle (p', q_y, \widehat{\mathsf{LH}}'), u'\rangle \in \Delta^{\mathcal{A}}$, and $c_0^{\mathcal{A}} \Rightarrow^{\rho_n^{\mathcal{W}}} \langle (p', q_y, \widehat{\mathsf{LH}}'), u'u\rangle$. We perform a case analysis on $r_n^{\mathcal{W}}$ to show that $\langle \mathsf{LH}_0, \widehat{\mathsf{LH}}\rangle \in \mathsf{inflate}(\mathit{val}(\rho_n^{\mathcal{W}}), x) \Rightarrow \widehat{\mathsf{LH}}' = \widehat{\mathsf{LH}}$.

  1. If $x = y$, then
     (a) $f(r_n^{\mathcal{W}}) = \{\langle \mathsf{LH}, \mathsf{post}_\eta(\mathsf{LH}, a)\rangle \mid \mathsf{LH} \in \mathcal{LH}\} z^0$
     (b) $\langle \mathsf{LH}_0, \mathsf{LH}^1, \ldots, \mathsf{LH}^x\rangle \in \mathit{val}(\rho_{n-1}^{\mathcal{W}})$, by the induction hypothesis
     (c) $\langle \mathsf{LH}_0, \mathsf{LH}^1, \ldots, \mathsf{post}_\eta(\mathsf{LH}^x, a)\rangle \in \mathit{val}(\rho_{n-1}^{\mathcal{W}}) \otimes f(r_n^{\mathcal{W}})$
     (d) $\langle \mathsf{LH}_0, \mathsf{LH}^1, \ldots, \mathsf{post}_\eta(\mathsf{LH}^x, a)\rangle \in \mathit{val}(\rho_n^{\mathcal{W}})$
     (e) $\langle \mathsf{LH}_0, \mathsf{LH}^1, \ldots, \mathsf{post}_\eta(\mathsf{LH}^x, a), \mathsf{LH}_0^{x+1}, \ldots, \mathsf{LH}_0^{|Q|}\rangle \in \mathsf{inflate}(\mathit{val}(\rho_n^{\mathcal{W}}), x)$
     (f) $\widehat{\mathsf{LH}}' = \widehat{\mathsf{LH}}[x \mapsto \mathsf{post}_\eta(\widehat{\mathsf{LH}}[x], a)] = \langle \mathsf{LH}^1, \ldots, \mathsf{post}_\eta(\mathsf{LH}^x, a), \mathsf{LH}_0^{x+1}, \ldots, \mathsf{LH}_0^{|Q|}\rangle$
     (g) $\widehat{\mathsf{LH}}' = \widehat{\mathsf{LH}}$
  2. Otherwise $y = x + 1$, and
     (a) $f(r_n^{\mathcal{W}}) = c_1 z^1 = \{\langle \mathsf{LH}, \mathsf{LH}, \mathsf{ptrans}(\mathsf{LH})\rangle \mid \mathsf{LH} \in \mathcal{LH}\} z^1$
     (b) $\langle \mathsf{LH}_0, \mathsf{LH}^1, \ldots, \mathsf{LH}^x\rangle \in \mathit{val}(\rho_{n-1}^{\mathcal{W}})$, by the induction hypothesis
     (c) $\langle \mathsf{LH}_0, \mathsf{LH}^1, \ldots, \mathsf{LH}^x, \mathsf{ptrans}(\mathsf{LH}^x)\rangle \in \mathit{val}(\rho_{n-1}^{\mathcal{W}}) \otimes f(r_n^{\mathcal{W}})$
     (d) $\langle \mathsf{LH}_0, \mathsf{LH}^1, \ldots, \mathsf{LH}^x, \mathsf{ptrans}(\mathsf{LH}^x)\rangle \in \mathit{val}(\rho_n^{\mathcal{W}})$
     (e) $\langle \mathsf{LH}_0, \mathsf{LH}^1, \ldots, \mathsf{LH}^x, \mathsf{ptrans}(\mathsf{LH}^x), \ldots, \mathsf{LH}_0^{|Q|}\rangle \in \mathsf{inflate}(\mathit{val}(\rho_n^{\mathcal{W}}), x)$
     (f) $\widehat{\mathsf{LH}}' = \widehat{\mathsf{LH}}[y \mapsto \mathsf{ptrans}(\widehat{\mathsf{LH}}[x])] = \langle \mathsf{LH}^1, \ldots, \mathsf{LH}^x, \mathsf{ptrans}(\mathsf{LH}^x), \ldots, \mathsf{LH}_0^{|Q|}\rangle$
     (g) $\widehat{\mathsf{LH}}' = \widehat{\mathsf{LH}}$

We have proved that the multi-arity relations that annotate the rules of $\mathcal{W}$ simulate the change in control state of the rules of $\mathcal{P}^{\mathcal{A}}$, and vice versa. This, combined with the proofs of correctness of algorithms for solving reachability problems in PDSs (Bouajjani et al., 1997; Finkel et al., 1997) and WPDSs (Bouajjani et al., 2003; Reps et al., 2005), proves that Alg. 2 computes the same result as Alg. 1, and thus completes the proof of correctness. □