STATIC VERIFICATION OF DATA-CONSISTENCY PROPERTIES

by

Nicholas A. Kidd

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2009

For JAK.

ACKNOWLEDGMENTS

First and foremost, I thank my wife, Katie, for her continued support and patience. When we moved to Paris for a year to accompany my advisor Professor Thomas Reps during his sabbatical, Katie did not hesitate to put her career on hold so that I could focus on my dissertation research. Katie, you made the completion of the dissertation possible, and your confidence and love were the keys to success. The dissertation is as much yours as it is mine. We did it!

I thank my parents, Baron and Kathy, for raising me in a household that stresses education and a strong work ethic. I am still amazed and thankful that they made the trip to Vienna in 2006 where I gave my first conference presentation. Whatever nerves I had before the presentation, it was reassuring to see their smiling faces in the audience. Mom and dad, you have always supported my decisions, and your challenges and guidance have made me who I am today.

I thank my office mate Akash Lal. Our many chess games are some of my fondest memories of graduate school. Akash, you always took the time to listen to my ideas, to help me flesh them out, and to give me a new perspective on the problems that I was trying to solve. I am proud to have been your coauthor, and I sincerely hope that our collaboration does not end at UW.

I thank Matthew Allen and William Benton, the 12-to-1 lunch gang. There is no better mental break than recapping the weekend's football games, and discussing fantasy-football strategies. Your taste in ice cream and pie is exquisite, and I will try to bring it back to Indiana.

I thank Junghee Lim, who with Akash, is my friend *de Paris*. I still miss the delicious meals, sightseeing trips, and photography lessons. Two Paris alumni done, you are the last. Finish strong, I know you will.

I thank Peter Lammich for introducing me to acquisition histories, and for his collaboration on the decision procedure. Peter, working together has been a pleasure, and I look forward to continuing to do so. I intend to take you up on the offer to go biking in the Black Forest.

I thank Tayssir Touili for introducing me to communicating pushdown systems, Mandana Vaziri for introducing me to atomic sets, and Tayssir and Ahmed Bouajjani for hosting me in Paris.

I thank my fellow PL students (past and present) at UW: Gogul Balakrishnan, Amanda Burton, Evan Driscoll, Matt Elder, Denis Gopan, Bill Harris, Steve Jackson, Alexey Loginov, Mulhern, Tristan Ravitch, and Aditya Thakur. Thank you for the insightful discussions, and I wish you all the best.

I thank my thesis committee—Professor Ben Liblit, Professor Somesh Jha, Professor Susan Horwitz, and Dr. Shaz Qadeer—whose feedback greatly improved the dissertation.

Finally, I thank my advisor Professor Thomas Reps. It is probably not customary to thank one's advisor at the end, but Tom is no ordinary advisor! Tom is a mentor: he taught me how to do research, never let me be bothered when a research idea did not pan out, and was always there at two in the morning before a paper deadline. Tom is also a great friend. Katie, Junghee, Akash, and I were fortunate to spend a year in Paris with Tom and Susan Horwitz during their sabbaticals. It was my favorite and most productive year of graduate school. Besides research, we all had the opportunity to really get to know Tom and Susan outside the confines of 1210 West Dayton Street, and those are the times I will remember most. Tom, thank you for everything. I hope that you enjoyed these past seven years as much as I did.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Writing *correct* shared-memory concurrent programs is hard. Not only must a programmer reason about the correctness of the sequential execution of code, but also about the possible side effects caused by interleaved execution of concurrently executing threads. Incorrect use of synchronization primitives can lead to data-consistency errors, which can have drastic consequences (cf. the Northeast Blackout of 2003).

This dissertation presents techniques to verify statically that the programmer's use of synchronization primitives preserves data consistency. The notion of data consistency used throughout the dissertation is *atomic-set serializability* (AS-serializability), which was first proposed by Vaziri et al. (2006). 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.

The approach taken is to use software model checking to verify that every possible execution of a concurrent program is AS-serializable. First, an abstract program is generated from a concrete program. The abstract program is defined such that it over-approximates the set of behaviors of the concrete program. Second, a software model checker explores all possible executions of the abstract program.

The challenge is to define abstractions and techniques that account for the multitude of sources of unboundedness in a concrete program. Concrete programs have dynamic memory allocation, recursion, dynamic thread creation, and reentrant locks, to name a few.

The contributions of the dissertation are generic techniques to permit model checking to be performed in the presence of several sources of unboundedness. My research addressed the problem of determining whether all possible executions of a certain class of models of concurrent Java programs are AS-serializable.

Somewhat surprisingly, given the many sources of unboundedness allowed in the models considered, I was able to show that the problem is decidable, and gave a practical algorithm for the problem. The technique has been implemented in a tool called Empire, which has been used to find known bugs in concurrent Java programs.

1 INTRODUCTION

1.1 The Challenge of Concurrency

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.[1]

To guard against data-consistency errors, shared-memory concurrent 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. When a thread needs to access a shared-memory location $\mathtt{m}$, it is responsible for acquiring the lock $\mathtt{l}$ that protects $\mathtt{m}$. Because of mutual exclusion, and assuming that accesses to $\mathtt{m}$ are consistently protected by $\mathtt{l}$ in each thread, the thread that currently holds $\mathtt{l}$ can access and update $\mathtt{m}$ without external interference (e.g., an update by another thread). Furthermore, proper synchronization requires

[1] Ch. 2 gives the formal definition of *atomic-set serializability*, the notion of data-consistency correctness that is used throughout the dissertation.

that the thread holds $\mathtt{l}$ for the *entire* sequence of operations that constitutes one larger logically-atomic operation. Failure to protect the entire sequence of operations can result in a data-consistency error.

If threads do not consistently acquire the lock $\mathtt{l}$ when attempting to access $\mathtt{m}$, 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 a shared-memory location $\mathtt{m}$. For some programming languages, such as C and C++, the program behavior is *undefined* in the presence of data races.

Data races, however, do not characterize the full set of *single-location* data-consistency errors that are possible. For example, if one augments a program that is prone to a data race with a new lock $\mathtt{l}_{\text{new}}$, and encloses every program statement between new program statements that acquire and release $\mathtt{l}_{\text{new}}$, the data race would be removed (using the technical definition); however, the same executions or behaviors are still present.[2]

Another example concerns a data-consistency error in 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 $\mathtt{u}$, $\mathcal{M}$ is queried to find the worker w that $\mathtt{u}$ is mapped to. After the lookup, but before w is notified of the request, the worker w terminates, and the mapping for $\mathtt{u}$ is updated to another worker w'. Hence, the server's attempt to dispatch $\mathtt{u}$'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. Note that 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. Multi-location data-consistency errors arise because programs

[2] The same behaviors are present assuming a strong-memory model.

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.

1.2 My Approach

This dissertation focuses on techniques to assist the programmer in reasoning about the data consistency of a concurrent Java program. The framework is that the programmer supplies (a small number of) data-consistency specifications, and the techniques presented in later chapters check that the program satisfies the desired data-consistency properties.

The approach taken is to use *software-model checking*. Model checking was pioneered by Clarke and Emerson (1982) and Queille and Sifakis (1982) as a technique for verifying properties of systems. Model checking follows a modular approach to system verification: instead of verifying a large complex property of a system, model checking aims to verify that the system satisfies multiple (simpler) properties. Oftentimes the property of interest is specified as a *temporal* property. A temporal property is a correctness specification for each possible behavior of a system. For software model checking, the system is a program and the set of all behaviors is the set of all possible program executions. For example, if the program of interest performs file I/O, a temporal property for correct usage of a file would be that a file must be opened before it can be read from or written to.

Data-consistency errors can be specified as a temporal property of an *interleaved* execution of a concurrent program. An interleaved execution consists of the steps of all program threads from beginning to end in a serial order. For example, consider an interleaved execution (trace) that has a data race. The trace must contain a sequence of steps by two threads where they access a shared-memory location without proper synchronization. The focus of the dissertation is static techniques to verify that no interleaved execution of a concurrent program contains data-consistency errors. The techniques developed in the dissertation apply to data races and to data-consistency errors involving multiple shared-memory locations.

In general, it is not possible to explore all possible (concrete) executions of a program Prog—either sequential or concurrent—because Prog can have too many sources of *unboundedness*. For example, Prog can have an unbounded number of inputs and outputs, and a precise modeling of such artifacts is not possible. Dynamic memory allocation gives rise to an a priori unbounded amount of memory consumption. Recursion gives rise to an a priori unbounded stack size. The values that a program variable can hold, such as an integer, are for practical purposes unbounded. Concurrency further complicates matters by introducing an unbounded number of threads of execution, and also an unbounded number of interleaved executions.

Because it is infeasible to reason about all possible (concrete) executions of Prog, *abstraction* is used to generate an abstract program $\mathsf{Prog}^\sharp$ of Prog that uses an abstract or approximate semantics (Cousot and Cousot, 1977). That is, abstraction approximates sources of unboundedness. To give one example of abstraction, consider replacing *integer-valued* variables with *sign-valued* variables, where a sign value is even or odd. Sign-valued variables, and their correspondingly defined abstract operators, such as $+^\sharp$ for $+$, over-approximate the unbounded integer-value space with the finite (i.e., bounded) sign-valued space.

For property verification, the set of behaviors (traces) of $\mathsf{Prog}^\sharp$ must be a superset or over-approximation of the actual behaviors (traces) of the concrete program Prog. Thus, if one can verify that a property holds for $\mathsf{Prog}^\sharp$, one can conclude that the property holds for Prog.[3] Software model checking focuses on

[3] The converse is not always true: a property violation in $\mathsf{Prog}^\sharp$ does not always signify a

verifying that a (temporal) property holds for program models, i.e., for $\mathsf{Prog}^\sharp$.

Addressing Unboundedness

Addressing unboundedness is a central theme of the dissertation. The choice of what to abstract away is a consequence of the stated goal of data-consistency verification. The first step is to define for an input program Prog an abstract program $\mathsf{Prog}^\sharp$ whose set of behaviors over-approximate the behaviors (concrete executions) of Prog.

Data-consistency violations are specified as a sequence of reads and writes to shared-memory locations. Thus, we abstract away the *values* of data and define $\mathsf{Prog}^\sharp$ to operate only on *abstract locations*. We bound the number of threads, which removes the a priori unbounded number of threads due to Java's dynamic thread allocation. We use a demand-driven approach to memory allocation by focusing on objects that can be allocated at a specified allocation site. Focusing on a specific allocation site still allows for an unbounded number of object allocations; however, the *random isolation abstraction* that we describe in Ch. 5 allows us to reason soundly about the set of all objects that can be allocated at that site.

After abstraction, $\mathsf{Prog}^\sharp$ contains (i) a finite number of threads that (ii) access a finite number of abstract locations using (iii) a finite number of locks for synchronization. However, there are still four sources of unboundedness.

1. Each thread is (possibly) recursive, which allows for an unbounded stack size.

2. Because threads have an a priori unbounded stack size, the number of interleaved executions is also unbounded.

3. Modeled locks are reentrant (like Java locks). A reentrant lock is a lock that may be (re)acquired multiple times by the thread that holds the lock, and

property violation in Prog. This is a consequence of over-approximation.

must be released the same number of times before it can be acquired by another thread. Because of recursion, the number of times that a thread may (re)acquire a lock is unbounded.

4. The last source of unboundedness comes from the data-consistency specification. In our setting, the user specifies certain program methods as being *units of work*; these are methods that must (appear to) execute atomically.[4] Because recursive methods may be specified as units of work, the number of times that a thread "begins" a unit of work, i.e., invokes a specified method, is also a priori unbounded.

These four sources of unboundedness lead us at first to believe that data-consistency verification is an undecidable problem. After all, one generally needs only two sources of unboundedness to simulate a Turing machine. Thus, we turned to *communicating pushdown system model checking* as the software-model-checking technique for attempting to explore all behaviors of $\mathsf{Prog}^\sharp$.

CPDS **Model Checking**

Communicating pushdown system (CPDS) model checking is a software-model-checking technique that attempts to solve undecidable problems. A CPDS is a formalism for modeling a concurrent message-passing program that uses global rendezvous-style synchronization. Each thread is modeled as a *pushdown system* (PDS), which is a formalism for precisely modeling recursive behaviors of a thread. PDSs and CPDSs are formally defined in Chs. 3 and 4, respectively, but for this discussion, we can view a PDS as defining a *context-free language* (CFL) and a CPDS as defining a set of CFLs. For example, a PDS that models a program thread defines a CFL whose alphabet symbols consist of reads and writes to shared-memory locations, symbols for acquiring and releasing a lock, and symbols denoting the begin and end of a unit-of-work method. We devel-

[4] Ch. 2 gives the formal definition of "execute atomically" for the atomic-set-serializability correctness specification.

oped methods to encode locks and data-consistency specifications as PDSs. Because of reentrancy, we need a CFL rather than a regular language, i.e., we need matched-parenthesis languages in which a parenthesis symbol denotes the acquisition/release of a lock or entering/exiting an atomic block.

CPDS model checking can then be viewed as attempting to determine the emptiness of the intersection of the set of CFLs, which is a known undecidable problem. Because the problem is undecidable, the CPDS model checker CPDSMC implements only a *semi-decision procedure*, where a semi-decision procedure is an algorithm that attempts to answer a "yes/no" query, but because of undecidability, may also return with a result of "maybe". Even though it implements only a semi-decision procedure, in Ch. 5 we show that the use of CPDSMC is effective in practice.

Language Strength Reduction

To improve the efficiency of model analysis, I developed a technique, known as *language-strength reduction*, to eliminate reentrant uses of locks and reentrant uses of unit-of-work methods. Removing reentrancy allows for the languages that model locks and data-consistency specifications to be formulated as *regular* languages instead of CFLs. Because the respective languages are now regular, two of the four sources of unboundedness have been removed (i.e., an unbounded stack is no longer required for components that model these aspects of program behavior).

However, there still exist two sources of unboundedness, namely, recursive procedure calls and interleavings. Because of these two sources of unboundedness, we continue with the use of CPDS model checking for analyzing generated models. The benefit of removing two of the four sources of unboundedness was a total speedup of 1.8 when CPDSMC was used to analyze models generated from the CONTEST benchmark suite (Eytani et al., 2007a). Moreover, two models that previously exhausted all resources were now able to be analyzed.

A Decision Procedure

The final breakthrough is a technique to deal with the unbounded number of interleavings, which leaves only one source of unboundedness, namely recursion, which PDS technology can easily manage. The end result is that we show that data-consistency verification is in fact a *decidable* problem (see Ch. 7).

The technique "decouples" the PDSs that model the bounded number of Java threads. That is, instead of directly exploring all possible interleaved executions, which induces a tight coupling between the PDSs, a finite summary of the behaviors of each PDS is computed individually. After all summaries have been computed, they are analyzed by a post-processing step to check that the data-consistency specification holds for $\mathsf{Prog}^\sharp$.

Besides the importance of showing that the problem is decidable, the model checker that implements the decision procedure of Ch. 7 is 8.5 times faster overall than the version of CPDSMC from Ch. 6 when analyzing abstract programs. Moreover, CPDSMC timed out—returned "maybe"— for 68% of the queries that were asked. The decision-procedure implementation provided the answers to these queries.

The techniques described in the dissertation have been implemented in a tool called EMPIRE, which works as follows. The input is (i) a concurrent Java program, (ii) a data-consistency specification for a Java class T of interest, and (iii) an allocation site χ that allocates an object of class T (i.e., a "`new T(...)`" statement). EMPIRE then generates a model of the program that is specific to T and χ, and checks that the data-consistency specification is a property of the model. Because the model over-approximates the set of behaviors of the input program, if EMPIRE is able to verify data-consistency of the model, then it also establishes the data-consistency of all objects allocated at χ in the program.

1.3 Dissertation Overview

Ch. 2 discusses data consistency and provides examples that illustrate data-consistency errors. It also defines atomic-set serializability, the formal definition of data consistency that is used throughout the dissertation. We also present a comparison of atomic-set serializability to other notions of data-consistency.

Ch. 3 provides background definitions for pushdown systems (PDSs) and weighted pushdown systems (WPDSs). Both PDSs and WPDSs are used to model a sequential recursive program. Accompanying both definitions are examples that illustrate how the formalism is used to perform program analysis.
Contribution: WALi: the **W**eighted **A**utomaton **Li**brary (Kidd et al., 2009a) that implements the discussed PDS and WPDS algorithms.

Ch. 4 defines communicating pushdown systems (CPDSs): the formalism for modeling *concurrent* recursive message-passing programs used for most of the dissertation. Due to its expressive power, reachability analysis of a CPDS is in general undecidable. However, the given semi-decision procedure and several (unpublished) variations have been shown to be useful in practice.
Contribution: The CPDS model-checking algorithm (Chaki et al., 2006) and several variations on the basic algorithm.

Ch. 5 presents *random isolation*, a program abstraction that is used to reason about the locking behavior of programs that use dynamic memory allocation, which is crucial for the analysis of concurrent programs that use lock-based synchronization. Random isolation is also a generic technique for proving a property about identical elements in a set of unbounded size (e.g., all objects that can be allocated at an allocation point in a program).

Contribution: The random-isolation abstraction and a source-to-source transformation for implementing the abstraction (Kidd et al., 2009b).

Ch. 6 presents *language strength reduction*, a technique that allows a program model to use non-reentrant locks in place of reentrant locks without a loss in precision or soundness (i.e., the exact set of interleaved executions are still considered in the transformed model). The name comes from the fact that the language that describes a reentrant lock is context-free—matched-parenthesis words model lock acquisitions and releases—whereas the language that describes a non-reentrant lock is regular—it consists of *non-nested* matched-parenthesis words. Hence, the transformation performs a strength-reduction operation on the language that describes a lock.
Contribution: The language-strength-reduction transformation (Kidd et al., 2008, 2007).

Ch. 7 presents a decision procedure for verifying atomic-set serializability. After language strength reduction, the program abstraction has a *finite-data* model, which enables the definition of a decision procedure. To define the decision procedure, CPDSs are replaced by another *multi*-PDS model that is less powerful, yet still able to verify atomic-set serializability of the program abstraction.
Contribution: A decision procedure for verifying atomic-set serializability of generated models (Kidd et al., 2009c).

Ch. 8 concludes and discusses opportunities for future work.

2 DATA CONSISTENCY

Concurrency is a mechanism used to increase the performance of a program. The use-case for concurrency can differ depending on the application.

- Server applications use concurrency to service multiple requests at the same time, which increases throughput in much the same way that large retail stores use multiple checkout lanes. For example, a web server uses multiple threads of execution to satisfy simultaneously multiple requests for web pages.

- GUI applications use concurrency so that an application is responsive to user requests while other operations are performed in the background. For example, a text editor could use one thread to wait for user input while another separate thread executes a spell checker.

- Scientific applications use concurrency to apply the same task on distinct elements in a set. For example, matrix multiplication simultaneously computes multiple entries in the resultant matrix from the same two input matrices.

- Other uses of concurrency arise depending on the needs of the program, the expertise of the programmer, and the perceived benefit of using concurrency.

For many concurrent applications, shared state is used to synchronize the concurrent processes. Server applications must synchronize accesses to the data that is being served, and GUI applications must synchronize accesses to the data that is being visualized and modifications to that data (e.g., the user may edit a word in a text editor while the spell checker highlights the same word). Scientific applications typically do not perform synchronization because the concurrent threads operate on disjoint data sets.

T_1				T_2			
1a	t1	=	x	2a	t2	=	y
1b	x	=	y	2b	y	=	x
1c	y	=	t1	2c	x	=	t2

Figure 2.1: Two threads attempt to swap the values x and y: T_1 performs "swap(x,y)" while T_2 performs "swap(y,x)".

Data-consistency errors can arise because multiple threads of execution attempt to read and update shared state, but the inter-thread order in which the accesses occur is non-deterministic (i.e., the application does not control the thread scheduler). If accesses to shared state are not properly protected, many problems can occur, such as lost updates, reads of stale data, and, depending on the programming language, the behavior may even be undefined.

Let us illustrate some of these problems with a concrete example. Consider the code fragment shown in Fig. 2.1. There are two shared-memory locations, x and y, and two threads of execution, T_1 and T_2. Together, the three statements that thread T_1 executes perform a "swap(x,y)". Similarly, the net effect of T_2's execution is to perform a "swap(y,x)". To perform the "swap" operations, T_1 (T_2) uses the temporary storage variable t1 (t2) to hold the initial value of x (y). For this example, assume that the "swap" operation constitutes one logically-atomic operation for each thread. Furthermore, assume that before the execution of either thread the value of x is not equal to the value of y (i.e., "$x \neq y$").

Using x_0 and y_0 to denote the values of x and y before the execution of the threads, the intended semantics of the example code is that at the end of the execution of both threads, $x = x_0$ and $y = y_0$. This is easy to see because for two memory locations, a double swap is the same as the identity transformation.

If each thread's code is run serially (i.e., the interleaved execution is "1a-c ; 2a-c" or "2a-c ; 1a-c"), then it is easy to verify that once threads T_1 and T_2 have finished execution, it will be the case that "$x = x_0 \land y = y_0$". Unfortunately, due to the absence of synchronization operations, not all executions have the intended outcome. Consider the interleaved execution "1a;2a;2b;2c;1b;1c" shown in Tab. 2.1. In Tab. 2.1, the left column shows the statement that is executed, and the remaining columns denote the values of the variables after the statement is executed. For each row, only an updated value is written in a column. That is, if a value is not given for a variable, it remains the same as the next-most-recent value in that column. For example, in row 2b, the value of variable x can be found in the x-column entry of row 1a. The final row shows the values of each variable after all statements have been executed. One can see that the outcome is such that the values of *both* x and y are equal to the original value of x, i.e., "$x = x_o \wedge y = x_o$". This is clearly not a desired behavior!

	x	y	t1	t2
1a	x_o	y_o	x_o	0
2a				y_o
2b		x_o		
2c	y_o			
1b	x_o			
1c		x_o		
	x_o	x_o	x_o	y_o

Table 2.1: An interleaved execution of the program in Fig. 2.1 that violates the intended semantics. Each column denotes the value of a specific variable after each program statement is executed. Only state changes are listed. The final row presents the value of each variable after all program statements have been executed.

The fact that the desired property does not hold is a consequence of a data-consistency error. The programming error for the code in Fig. 2.1 is the absence of operations to synchronize accesses to shared variables. In this case, the error can be easily diagnosed by examining the code; however, data-consistency errors are in general extremely difficult to diagnose and debug because their very nature is dependent on non-deterministic events, which includes the decisions made by the thread scheduler and inputs from external sources, such as the user of an application. One can imagine a case where the code in Fig. 2.1 always executes each thread serially. This could occur when the test machine is a uniprocessor and performing three memory operations can be easily completed within one execution context. In this scenario, program testing would not detect an error because only non-buggy interleavings are considered. However, when upgrading or migrating to a multi-processor (multi-core) machine, the code of each thread could then be run in parallel and the buggy interleavings could be exercised.

Data-consistency errors are often non-local properties. That is, diagnosing the root cause of a data-consistency error requires reasoning about more than just a piece of sequential code. For example, if there is a synchronization error in the spell-checker code or the user-input code of a text-editing program, diagnosing and creating a solution requires reasoning about the potential interactions between two separate pieces of code. The fact that programs often deliberately violate data-consistency conditions, the fact that errors often arise only for subtle reasons involving non-determinism, the need to perform non-local reasoning to diagnosis data-consistency errors, all make it difficult for programmers to be sure that their code maintains (or reestablishes) desired data-consistency conditions. Thus, it is crucial that tools and techniques be developed to assist the programmer in reasoning about the data consistency of a program. Before presenting the approach taken in this dissertation to verifying data consistency, we need to formally define what it means for a program to be data consistent.

The rest of this chapter discusses (i) *data-race freedom*, (ii) *serializability* and *atomicity*, and (iii) *atomic-set serializability* and *atomic-set atomicity*. Tab. 2.2 presents the relationship between these data-consistency correctness criteria. Within a column, a lower entry subsumes all higher entries. That is, establishing that a program satisfies a lower entry implies that it satisfies a higher entry. Along a row, we indicate the terminology used depending on whether the data-consistency property is specified on a single trace or the set of all traces of a

§ where discussed	*Trace Property*	*Set-of-Traces Property*
§2.1	data-race free	data-race free
§2.2	serializability	atomicity
§2.3	atomic-set serializability	atomic-set atomicity

Table 2.2: Notions of data consistency that have been proposed for traces and sets of traces.

program. Finally, the leftmost column specifies the section that discusses the data-consistency correctness criteria of an individual row.

2.1 Data-Race Freedom

A *data race* is said to occur when two threads attempt to access the same memory location without synchronization, and at least one of them is a write. There are many data races in the code shown in Fig. 2.1 because the example does not contain any synchronization operations. For example, after each thread executes its first statement, T_1 can write to x while T_2 can read the value of x, which is a data race on the shared-memory location x. In fact, in this same scenario there is also a data race on shared-memory location y because T_1 can attempt to read the value of y while T_2 is attempting to write a new value to y.

Data races occur in practice and can have devastating results. In 2003, the Northeast Blackout left most of the Northeastern part of the United States and Canada without power (Wikipedia, 2009). The Northeast Blackout was more extensive than it otherwise might have been because a data race caused the backup generator to fail to respond, which caused a cascade of blackouts across the North American continent.

Data-race detection has a long history in the research community. The seminal work on the dynamic detection of data races is the Eraser tool, which introduced the *lockset* algorithm, and was developed by Savage et al. (1997). It its most basic form, the lockset algorithm checks whether a program enforces the discipline that a memory location m is consistently protected by a lock l. During program execution, a set of locks L_m is associated with a memory location m. Initially, the set of locks L_m is the set of all locks in the program. When a thread T accesses m, the set of locks L_m is updated to be the intersection of L_m and set of locks L_T that T currently holds. If, after the intersection, L_m is equal to the empty set, then there is the potential for a data race.

Others have developed static techniques to check the same locking discipline that Eraser checked dynamically. For example, the LOCKSMITH tool by Pratikakis et al. (2006) uses an existential type system to check whether a memory location is consistently protected by the same lock, and Naik and Aiken (2007) presented a static analysis to detect data races based on conditional must not aliasing. The basic premise of that work is to demonstrate the absence of data races by showing that, if whenever two locks are different, then their guarded locations must be different.

Proving the absence of data races, i.e., data-race freedom, goes a long way towards establishing data consistency. However, by definition, data-race detection is unable to capture higher levels of data consistency (in the sense of Tab. 2.2). That is, the definition of a data race only considers two instructions executed by two threads, which is a very limited scope. For example, the code shown in Fig. 2.2 is the same as the code shown in Fig. 2.1 except that a global lock l has been added, and each statement acquires the lock l before it is executed and releases the lock l when execution completes. The code shown in Fig. 2.2 *does not* contain a data race because every memory access is protected with synchronization on the lock l. However, the bad interleavings discussed at the beginning of this chapter are all still possible. Although the use of synchronization by the code in Fig. 2.2 is clearly broken, it illustrates the point that enforcing a locking discipline—in particular, the discipline checked by ERASER and LOCKSMITH—does not account for the fact that the programmer intends for *regions* of code to be protected and to be executed atomically.

T_1				T_2			
1.a	lock(l){t1	=	x }	2.a	lock(l){t2	=	y }
1.b	lock(l){x	=	y }	2.b	lock(l){y	=	x }
1.c	lock(l){y	=	t1}	2.c	lock(l){x	=	t2}

Figure 2.2: Two threads attempt to swap the values x and y: T_1 performs "swap(x,y)" while T_2 performs "swap(y,x)". Each statement is guarded by the lock l, which guarantees that the program is data-race free.

2.2 Serializability and Atomicity

Serializability is a stronger notion of data consistency than data-race freedom. The term serializability comes from the database community (Papadimitriou, 1986; Bernstein et al., 1987). For databases, the goal is to execute multiple SQL queries (transactions) concurrently yet maintain the illusion that the transactions are executed serially. An execution schedule is serializable if it is equivalent to a schedule in which the transactions are executed in some serial order.

Various notions of equivalence relations between execution schedules have been proposed, including *conflict equivalence* and *view equivalence* (Bernstein et al., 1987). We briefly discuss conflict equivalence and note that conflict equivalence is a stronger relation than view equivalence (conflict equivalence implies view equivalence). Conflict equivalence uses the notion of *conflicting actions* as an equivalence criterion. A pair of actions is said to conflict if they occur in two separate transactions, access the same data, and at least one is a write. Two schedules S_1 and S_2 are *conflict equivalent* if they execute the same transactions and the order of conflicting actions is the same. A schedule is *conflict serializable* if it is conflict equivalent to a serial schedule.

There have been many proposals for notions of serializability relevant for program execution as well as analyses to detect serializability violations (Artho et al., 2003; Lu et al., 2006; Xu et al., 2005; Wang and Stoller, 2006; Flanagan and Freund, 2004, 2008). In general, a code region that should appear to execute atomically, i.e., denotes a logically-atomic operation, forms a transaction. An interleaved program execution is then serializable if it is equivalent to an execution in which the transactions are executed in some serial order.

For the code shown in Fig. 2.1, the executions (1) "1a-c ; 2a-c" and (2) "2a-c ; 1a-c" are the only serializable executions (i.e., no interleaving is possible). Briefly, an execution that is equivalent to the serial execution (1) must read the value of y that is written by statement 1c to ensure that the order of the conflicting actions is the same. Since statement 2a reads y, it must come after statement 1c and thus no interleaved execution is equivalent to serial execution (1). A similar argument shows that there does not exist an interleaved execution that is equivalent to serial execution (2): statements 1a and 2c are conflicting actions.

Flanagan and Qadeer (2003) extend serializability from a property of a single execution to a property of a set of executions. That is, they are concerned with defining a correctness criterion for *all* possible program executions. A program is said to have the *atomicity* property if all possible executions are serializable.

Flanagan and Qadeer (2003) developed a type system for atomicity. The type system is such that if a program passes the type checker, then it has the atomicity property. With respect to serializability and atomicity, the transactions are the methods of a concurrent Java program. Their type system errs on the side of caution, i.e., a program may fail to type check but in fact have the atomicity property. The conservative approximation used in the type system is to check that each method (transaction) of a concurrent Java program is *atomic*.

> *If a method is atomic, then any interaction between that method and steps of other threads is guaranteed to be benign, in the sense that these interactions do not change the program's overall behavior*
>
> — Flanagan and Qadeer (2003)

If all methods of a program are atomic, then all executions must be serializable. Hence, a program that type checks has the atomicity property. As mentioned above, the converse it not necessarily true.

As a data-consistency correctness criterion, serializability and atomicity are superior to data-race freedom. In each access to shared data takes place in an atomic method, then the program must be data-race free. Furthermore, a higher-level of data consistency can be captured, such as the fact that correct execution of the code in Fig. 2.1 must execute the "swap" operations serially. Unfortunately, atomicity can be overly restrictive because it is a *control-centric* property—a property of a region of code that is a control structure of a programming language.

To show how serializability and atomicity can be too restrictive, consider the Java program in Listing 2.1, which defines two classes. Class `Counter` implements a performance counter to track the number of times non-static methods are invoked. Class `Stack` implements a stack and has two fields:

- `Object data[]` is an array that implements the backing store for a `Stack`.
- `int count` is a counter that keeps track of the number of items currently on a stack.

Let us assume that the `@atomic` annotations on the methods of class `Stack` specify that the annotated methods should appear to execute atomically. (For now, ignore the annotation `@atomic(S)` on the fields of class `Stack`.)

Consider the following interleaved execution of threads T_1 and T_2 from Listing 2.1:

$$\begin{array}{ll} T_1: & \overbrace{\texttt{inc}\ldots\ldots\ldots\overbrace{\texttt{inc}}^{\texttt{pop}}\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\overbrace{\texttt{inc}}^{\texttt{push}}}^{\texttt{replaceTop}} \\ T_2: & \ldots\ldots\texttt{get}\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \end{array}$$

The interleaved execution lists the method calls that are performed by threads T_1 and T_2, respectively. The calls to `Counter.inc`, `Counter.get`, `Stack.pop`, `Stack.push`, and `Stack.replaceTop` have been abbreviated as `inc`, `get`, `pop`, `push`, and `replaceTop`, respectively. A horizontal sequence of dots denotes

Listing 2.1: Stack program.

```
class Counter {
    private static int counter = 0;
    public static synchronized void inc() { counter++; }
    public static int get() { return counter; }
}
class Stack {
    public static final int MAX=10;
    @atomic(S) Object[] data = new Object[MAX];
    @atomic(S) int count = -1;
    @atomic public synchronized Object pop(){
        Counter.inc();
        Object res = data[count];
        data[count--] = null;
        return res;
    }
    @atomic public synchronized void push(Object o) {
        Counter.inc(); data[++count] = o;
    }
    @atomic public synchronized int size() {
        Counter.inc(); return count+1;
    }
    @atomic public synchronized replaceTop(Object o) {
        Counter.inc(); pop(); push(o);
    }
    public static Stack makeStack() { return new Stack(); }
}
class Test() {
    public static void main(String[] args) {
        Stack stack = Stack.makeStack();
        stack.push(new Integer(1));
        new Thread("1") { stack.replaceTop(new Integer(2)); }
        new Thread("2") { Counter.get(); }
    }
}
```

that a thread is not currently executing (e.g., it has been swapped out by the thread scheduler). For the method calls of T_1, an overbrace is used to denote the duration of execution of a parent method, and the methods that it invokes are contained in the overbrace.

The interleaved execution contains a serializibility violation. The problem is the multiple calls to `Counter.inc`. Each call to `Counter.inc`, or rather the update performed by `Counter.inc`, by T_1 is a conflicting action with the call to `Counter.get` by T_2. Because `Stack.replaceTop` is supposed to execute atomically—denoted by the `@atomic` annotation—a serializable execution must schedule the call to `Counter.get` by T_2 completely before or after the call to `Stack.replaceTop` by T_1. The interleaved execution shown above violates this requirement, and hence contains a serializability violation.

Because of possible interleavings like the one shown above, the method `Stack.replaceTop` fails the atomicity property.[1] For the example program in Listing 2.1, the interleaving shown above should be allowed. That is, the serializability violation is benign. To allow such interleavings, data-consistency correctness must move from the control-centric nature of atomicity to a data-centric correctness criterion that is able to reason about relationships between memory locations (i.e., fields of a class), or for the above example the lack of relationships.

2.3 Atomic-Set Serializability

Vaziri et al. (2006) propose *atomic-set serializability* (AS-serializability) as a correctness criterion for data consistency. AS-serializability addresses the fact that serializability can be an overly conservative correctness criterion because it ignores relationships that may exist between shared memory locations, such as invariants and consistency properties. By ignoring relationships, serializability

[1] Technically, Flanagan and Qadeer (2003) uses the theory of left and right movers, due to Lipton (1975), to determine atomicity of a method. For this discussion, we omit these details.

may not accurately reflect the intentions of the programmer for correct behavior, resulting in false positives. For the program shown in Listing 2.1, interleaved updates to the performance counter implemented by class `Counter` do not cause a data-consistency error. That is, the `Counter.inc` method is properly synchronized, which eliminates data races, and its value has no effect on the outcome of the program.

AS-serializability is a data-centric correctness criterion that is able to consider relationships that exist between memory locations. It is based on the notion of *atomic sets*. An atomic set is a set of memory locations for which an invariant holds and thus must be updated atomically. Note that an atomic set merely specifies that there exists an invariant, but does not define the invariant itself.

In Listing 2.1, the annotations `@atomic(S)` on fields `Stack.data` and `Stack.count` denote that those fields are members of the atomic set "`S`". The (unspecified) relationship between `data` and `count` is that the current value of `count` is the index of the position in `data` where the top-of-stack item is stored. The existence of this relationship is reflected by both fields being members of the atomic set "`S`".

Associated with atomic sets are *units of work*, regions of code (methods) that guarantee to maintain the invariant of an atomic set when executed serially. We call a unit of work that writes to all members of an atomic set a *write-complete* unit of work. In Listing 2.1, the units of work are methods that have the `@atomic` annotation (e.g., the method `Stack.pop`). The write-complete units of work are the methods `Stack.pop`, `Stack.push`, and `Stack.replaceTop`.

Finally, atomic sets can be dynamically extended to include the atomic sets of method parameters, which are referred to as *unitfor* parameters. This is specified by an `@atomic` annotation on method parameters.[2] Dynamic extension of an atomic set captures the fact that if a unit of work is dependent on a method parameter, execution of that unit of work must (appear to) be atomic for the

[2] (Vaziri et al., 2006) use the annotation "unitfor" to mark method parameters that should be dynamically absorbed into an atomic set.

atomic sets of that parameter as well.

An execution is *atomic-set serializable* (AS-serializable) if its projection on each atomic set is serializable (Vaziri et al., 2006). Recalling the earlier discussion that the method `Stack.replaceTop` fails the atomicity property, we can now show that AS-serializability holds for `Stack.replaceTop`. Specifically, the method `Stack.replaceTop` and the methods `Stack.pop` and `Stack.push` that it invokes are all `synchronized` methods. For these methods, the atomic set with respect to which they must execute atomically is the atomic set "S". Because the methods are synchronized, no other thread can access the fields that are members of "S" until the method `Stack.replaceTop` completes execution. Because the field `Counter.counter` is not a member of "S", the write accesses that cause serializability violations for some interleavings are of no consequence. Thus, all executions of `Stack.replaceTop` must be AS-serializable. Because all executions of `Stack.replaceTop` are AS-serializable, we say that this method has the *AS-atomicity* property. In fact, the relationship between serializability (atomicity) and AS-serializability (AS-atomicity) is now clear: serializability is the degenerate case of AS-serializability when all of memory is defined to be in one atomic set.

AS-serializability Violations

An execution that is not AS-serializable is said to have an AS-serializability violation. A nice result from Vaziri et al. (2006) is that if all units of work are write-complete, then AS-serializability violations can be completely characterized by a set of fourteen problematic access patterns. Tab. 2.3 presents the fourteen problematic access patterns.[3] Each pattern consists of a sequence of reads (R) and writes (W) to an atomic-set member l, or to two atomic-set members l_1 and l_2. Problematic access patterns 1 through 5 capture single-location data-consistency errors, while problematic access patterns 6 through 14 capture multi-location data-consistency errors. Each memory access occurs

[3] The patterns in Tab. 2.3 appear in Vaziri et al. (2006) and Hammer et al. (2008).

Id	*Problematic Access Pattern*	*Description*
1.	$R_u(l)\ W_{u'}(l)\ W_u(l)$	Value read is stale by the time an update is made in u.
2.	$R_u(l)\ W_{u'}(l)\ R_u(l)$	Two reads of the same location can yield different values in u.
3.	$W_u(l)\ R_{u'}(l)\ W_u(l)$	An intermediate state is observed by u'.
4.	$W_u(l)\ W_{u'}(l)\ R_u(l)$	Value read may not be the same as the one written last in u.
5.	$W_u(l)\ W_{u'}(l)\ W_u(l)$	Value written by u' can be lost.
6.	$W_u(l_1)\ W_{u'}(l_1)\ W_{u'}(l_2)\ W_u(l_2)$	Memory can be left in an inconsistent state.
7.	$W_u(l_1)\ W_{u'}(l_2)\ W_{u'}(l_1)\ W_u(l_2)$	same as above
8.	$W_u(l_1)\ W_{u'}(l_2)\ W_u(l_2)\ W_{u'}(l_1)$	same as above.
9.	$W_u(l_1)\ R_{u'}(l_1)\ R_{u'}(l_2)\ W_u(l_2)$	State observed can be inconsistent.
10.	$W_u(l_1)\ R_{u'}(l_2)\ R_{u'}(l_1)\ W_u(l_2)$	same as above
11.	$R_u(l_1)\ W_{u'}(l_1)\ W_{u'}(l_2)\ R_u(l_2)$	same as above.
12.	$R_u(l_1)\ W_{u'}(l_2)\ W_{u'}(l_1)\ R_u(l_2)$	same as above.
13.	$R_u(l_1)\ W_{u'}(l_2)\ R_u(l_2)\ W_{u'}(l_1)$	same as above.
14.	$W_u(l_1)\ R_{u'}(l_2)\ W_u(l_2)\ R_{u'}(l_1)$	same as above.

Table 2.3: The fourteen problematic access patterns. Patterns 1–5 involve a single memory location; patterns 6–14 involve a pair of memory locations.

during a unit of work u; the subscript on the memory access denotes which unit of work it belongs to. For example, problematic access pattern 1 is defined as "$R_u(l)\ W_{u'}(l)\ W_u(l)$", which describes an AS-serializability violation that involves an atomic-set member l. The first read and last write belong to unit of work u. The intervening write belongs to a unit of work u' that is executed by another thread.

Informally, each pattern contains a pair of conflicting actions that cannot be conflict equivalent to a serial execution because the conflicting actions form a cyclic dependency and thus no serial order can be found. For the first problematic access pattern, the first accesses, $R_u(l)$ and $W_{u'}(l)$, are conflicting actions and the last two accesses, $W_{u'}(l)$ and $W_u(l)$, are conflicting actions.

If we draw an edge between units of work to represent the order of conflicting actions, then the first two statements induce the edge $u \to u'$ and the last two induce the edge $u' \to u$. These two edges form a cycle. From serializability theory, it is known that only executions that induce acyclic conflict graphs are serializable (Bernstein et al., 1987). Vaziri et al. (2006) proved that the absence of all fourteen patterns is a sufficient condition to show that the execution is AS-serializable (i.e., the conflict graphs are acyclic with respect to the atomic sets and their units of work).

To give a concrete illustration of an AS-serializability violation, let us return to the program in Listing 2.1. Class `Stack` assumes that the programmer correctly uses the stack, and omits safety checks for the method `Stack.pop`. That is, `Stack.pop` does not verify that the stack is non-empty before attempting to remove the top-of-stack item. If `Stack.pop` is invoked on an empty stack, then the field `Stack.count` will be `-1`. The array access to the field `Stack.data` will result in an `ArrayIndexOutOfBoundsException` being raised because `Stack.count` has the value `-1`.

The class `SafeWrap` defined below addresses this oversight by defining the method `SafeWrap.popwrap`, which first checks that the parameter "`Stack s`" is not empty before invoking the method `Stack.pop`. For the class `SafeWrap`, the method `SafeWrap.popwrap` is a unit of work, denoted by the `@atomic` annotation, and the parameter "`Stack s`" is a unitfor parameter, also denoted by the `@atomic` annotation. (Note that the `@atomic` annotation is a specification, and not an implementation.) Recall that the atomic sets of a unitfor parameter are dynamically incorporated into the atomic sets of the invoking object, i.e., the `this` parameter, for the duration of a unit of work. Because class `SafeWrap` does not define any atomic sets, the atomic set with respect to which `SafeWrap.popwrap` must execute atomically is the atomic set "`S`" defined by class `Stack`.

```
class SafeWrap {
  @atomic public synchronized Object popwrap(@atomic Stack s) {
    return (s.size() >= 0) ? s.pop() : null;
  }
  public static SafeWrap makeSafeWrap() { return new SafeWrap(); }

  public static void main(String[] args){
    Stack stack = Stack.makeStack();
    stack.push(new Integer(1));
    new Thread("1") { makeSafeWrap().popwrap(stack); }
    new Thread("2") { makeSafeWrap().popwrap(stack); }
  }
}
```

Unfortunately, the attempt to "harden" the code by adding error checking has introduced an AS-serializability violation. The problem is that the program synchronizes on the wrong object. In this case, the method `SafeWrap.popwrap` has the `synchronized` annotation, whereas the method should have been written to synchronize on the parameter "`Stack s`". This allows the interleaved execution shown in Fig. 2.3.

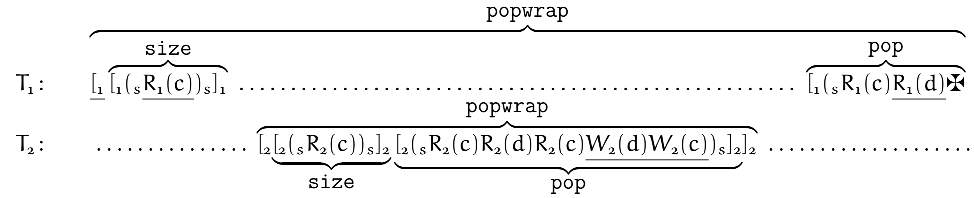
popwrap
size
pop
T₁ : [₁ [₁(ₛR₁(c))ₛ]₁ .... [₁(ₛR₁(c)R₁(d)✠
popwrap
T₂ : .... [₂[₂(ₛR₂(c))ₛ]₂ [₂(ₛR₂(c)R₂(d)R₂(c)W₂(d)W₂(c))ₛ]₂]₂ ....
size
pop

Figure 2.3: 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. The subscripts "1" and "2" are thread ids. "$(_s$" and "$)_s$" denote the acquire and release operations, respectively, of the lock of `Stack` s that is the input parameter to `SafeWrap.popwrap`.

The interleaved execution shown in Fig. 2.3 contains problematic access pattern 12. The data accesses that are involved in the pattern are underlined in Fig. 2.3. Because of the AS-serializability violation, the method

SafeWrap.popwrap can raise an ArrayIndexOutOfBoundsException even when it is invoked on a non-empty stack. This is the case for the interleaved execution shown in Fig. 2.3. 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 stack, which raises an exception. The point at which the exception is raised is denoted by the ✠ symbol at the end of thread T_1's execution sequence.

The focus of the dissertation is on static techniques to verify that interleaved executions like the one just discussed cannot happen—in other words, to verify that a program has the AS-atomicity property.

2.4 Summary

This chapter presented three notions of data-consistency: data-race freedom, serializability and atomicity, and AS-serializability and AS-atomicity. The latter two correctness criteria provide a means of reasoning about data consistency at larger granularity than data-race freedom. In our comparison of serializability (atomicity) to AS-serializability (AS-atomicity), we have shown that AS-serializability (AS-atomicity) is a relaxation of serializability (atomicity). The relaxation provides a finer notion of data consistency.

This dissertation presents techniques to verify AS-atomicity of a concurrent Java program. To the best of my knowledge, there are only two pieces of work related to verifying AS-atomicity. The first is the seminal work on atomic sets by Vaziri et al. (2006), where atomic sets and AS-serializability were first defined. In their setting, they extended the Java language with support for atomic sets, units of work, and unitfor parameters. Given a program in their atomic-set-extended Java, all program executions of the program are guaranteed to have the AS-atomicity property. The distinction between this dissertation and their work is that we focus on analyzing plain old Java programs. That is, the techniques to be presented are capable of analyzing existing Java programs with a limited amount of programmer-supplied annotations. In fact, as will be discussed in Ch. 5, the main annotation is simply the program allocation site that allocates objects with respect to which the programmer desires to verify AS-atomicity. Moreover, the atomic-set-extended Java language is not yet available, and thus their system does not currently assist a programmer with verifying data-consistency, i.e., AS-atomicity, of a program.

Hammer et al. (2008) presents a runtime technique to detect AS-serializability violations in Java programs. Because their technique performs dynamic detection of AS-serializability violations, it cannot guarantee the absence of AS-serializability violations for all executions. That is, the technique of Hammer et al. (2008) cannot verify AS-atomicity of a Java program, but merely detects bugs in specific concurrent executions.

3 DEFINITIONS

The focus of the dissertation is on techniques to verify AS-atomicity of a concurrent Java program. Ch. 1 presented a broad overview of the approach. From an input Java program Prog, an abstract program $\mathsf{Prog}^\sharp$ is first defined (Ch. 5). Then, $\mathsf{Prog}^\sharp$ is translated into the input format of a software-model checker. The dissertation uses two software-model checkers, namely, Cpdsmc (Chs. 5 and 6) and Ipamc (Ch. 7). Each model checker uses a *pushdown system* (PDS) as its underlying thread abstraction.

This chapter formally defines pushdown systems and *weighted* pushdown systems, a generalization of pushdown systems that we use to implement symbolic model checkers, among other things. The definitions are taken from Reps et al. (2003, 2005, 2007). Readers familiar with pushdown systems and weighted pushdown systems may skip to Ch. 4.

3.1 Pushdown Systems

Pushdown systems are used to describe *pure sequential programs* (Bouajjani et al., 1997; Finkel et al., 1997). They have the property that for recursive programs, infinite sets of program configurations can be represented symbolically using regular languages.

Definition 3.1. A *pushdown system* (PDS) is a tuple $\mathcal{P} = (\mathsf{P}, \Gamma, \mathsf{Lab}, \Delta, c_0)$, where P is a finite set of control locations; Γ is a finite set of stack symbols; Lab is a finite set of labels (or actions); $\Delta \subseteq (\mathsf{P} \times \Gamma) \times \mathsf{Lab} \times (\mathsf{P} \times \Gamma^*)$ is a finite set of labeled-transition rules, where a rule is denoted by $r = \langle p, \gamma \rangle \overset{a}{\hookrightarrow} \langle p', u' \rangle$; and $c_0 = \langle p_0, \gamma_0 \rangle$ is the initial configuration of $\mathcal{P}$. A *configuration* c of $\mathcal{P}$ is a pair $\langle p \in \mathsf{P}, u \in \Gamma^* \rangle$. For a rule $r = \langle p, \gamma \rangle \overset{a}{\hookrightarrow} \langle p', u' \rangle$, we use $\mathsf{lab}(r)$ to denote r's label a.

Without loss of generality, a pushdown rule is restricted to have at most two

```
class Prog {
  int _a,_b,_c,_d;
  int a() {
    return _a;
  }
  int b() {
    return _b;
  }
  int ab() {
    int t1 = a();
    int t2 = b();
    return t1+t2;
  }
  int cd() {
    return _c+_d;
  }
  public static void int main(String[] a) {
    int t1 = ab();
    int t2 = cd();
  }
}
```

Figure 3.1: A Java program.

Rule	Control flow modeled
$\langle p, n_1\rangle \xhookrightarrow{s_1} \langle p, n_2\rangle$	Intraprocedural edge $n_1 \rightarrow n_2$
$\langle p, n_c\rangle \xhookrightarrow{s_c} \langle p, e_f\ r_c\rangle$	Call to f, with entry e_f, from n_c that returns to r_c
$\langle p, x_f\rangle \xhookrightarrow{s_f} \langle p, \epsilon\rangle$	Return from f at exit x_f

Table 3.1: The encoding of an ICFG's edges as PDS rules.

stack symbols appear on the right-hand side, i.e., for $\langle p, \gamma\rangle \xhookrightarrow{a} \langle p', u'\rangle \in \Delta$, $|u'| \leqslant 2$ (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.

A PDS $\mathcal{P} = (\mathsf{P}, \Gamma, \mathsf{Lab}, \Delta, c_0)$ provides a way to encode a program's interprocedural control flow: $\mathsf{P} = \{p\}$ is a set containing a single control location p, the stack alphabet Γ is defined to be the set of program nodes—Γ models the program counter; the set of labels Lab is defined to be the set of program statements, and Δ is defined according to Tab. 3.1. The rule templates in Tab. 3.1 are interpreted as follows.

pop The rule $\langle p, x_f \rangle \xhookrightarrow{s_x} \langle p, \epsilon \rangle$ executes the statement s_f, which is a return from procedure f. The program counter is updated to be the return point that was pushed onto the stack by the caller.

step The rule $\langle p, n_1 \rangle \xhookrightarrow{s} \langle p, n_2 \rangle$ executes the statement s associated with node (program counter) n_1, and updates the program counter to node n_2.

push The rule $\langle p, n_c \rangle \xhookrightarrow{s_c} \langle p, e_f\ r_c \rangle$ executes the statement s_c, which is a procedure call. The program counter is updated to be the entry point e_f of the called procedure f. The return point r_c is pushed onto the stack to ensure that when f returns, the program counter will be set to r_c.

Example 3.2. Consider the Java program Prog given in Fig. 3.1. Let Stmts be the set of program statements (e.g., "`return _a`"). The PDS $\mathcal{P}_{3.2} = (\mathsf{P}_{3.2}, \Gamma_{3.2}, \mathsf{Lab}_{3.2}, \Delta_{3.2}, \langle p, n_{17} \rangle)$, where $\mathsf{P}_{3.2} = \{p\}$, $\mathsf{Lab}_{3.2}$ is Stmts, $\Gamma_{3.2} = \{n_i \mid 1 \leqslant i \leqslant 20\}$, and $\Delta_{3.2}$ is the set of PDS rules presented in Fig. 3.2, which are generated using the rule templates given in Tab. 3.1. The PDS $\mathcal{P}_{3.2}$ encodes the interprocedural control flow of Prog.

One is often interested in modeling more than just simple control flow. For example, given the Java program Prog from Fig. 3.1, the *field-accesses-per-method* problem is to determine, for each method m, what is the set of field names that are accessed by m? Such queries can be answered by encoding the appropriate information in the control locations of a PDS. For this particular problem, the appropriate information is a set of (class-name, field-name) pairs. That is, let K be the set of all class names defined by a program, and F be the set of all fields.

a()

$$\begin{array}{lcl}
\langle p, n_3 \rangle & \xhookrightarrow{e_a} & \langle p, n_4 \rangle \\
\langle p, n_4 \rangle & \xhookrightarrow{\texttt{return \_a}} & \langle p, n_5 \rangle \\
\langle p, n_5 \rangle & \xhookrightarrow{x_a} & \langle p, \epsilon \rangle
\end{array}$$

b()

$$\begin{array}{lcl}
\langle p, n_6 \rangle & \xhookrightarrow{e_b} & \langle p, n_7 \rangle \\
\langle p, n_7 \rangle & \xhookrightarrow{\texttt{return \_b}} & \langle p, n_8 \rangle \\
\langle p, n_8 \rangle & \xhookrightarrow{x_b} & \langle p, \epsilon \rangle
\end{array}$$

cd()

$$\begin{array}{lcl}
\langle p, n_{14} \rangle & \xhookrightarrow{e_{cd}} & \langle p, n_{15} \rangle \\
\langle p, n_{15} \rangle & \xhookrightarrow{\texttt{\_c+\_d}} & \langle p, n_{16} \rangle \\
\langle p, n_{16} \rangle & \xhookrightarrow{x_{cd}} & \langle p, \epsilon \rangle
\end{array}$$

ab()

$$\begin{array}{lcl}
\langle p, n_9 \rangle & \xhookrightarrow{e_{ab}} & \langle p, n_{10} \rangle \\
\langle p, n_{10} \rangle & \xhookrightarrow{\texttt{t1 = a()}} & \langle p, n_3\ n_{11} \rangle \\
\langle p, n_{11} \rangle & \xhookrightarrow{\texttt{t2 = b()}} & \langle p, n_6\ n_{12} \rangle \\
\langle p, n_{12} \rangle & \xhookrightarrow{\texttt{return t1+t2}} & \langle p, n_{13} \rangle \\
\langle p, n_{13} \rangle & \xhookrightarrow{x_{ab}} & \langle p, \epsilon \rangle
\end{array}$$

main()

$$\begin{array}{lcl}
\langle p, n_{17} \rangle & \xhookrightarrow{e_{main}} & \langle p, n_{18} \rangle \\
\langle p, n_{18} \rangle & \xhookrightarrow{\texttt{t1 = ab()}} & \langle p, n_9\ n_{19} \rangle \\
\langle p, n_{19} \rangle & \xhookrightarrow{\texttt{t2 = cd()}} & \langle p, n_{14}\ n_{20} \rangle \\
\langle p, n_{20} \rangle & \xhookrightarrow{x_{main}} & \langle p, \epsilon \rangle
\end{array}$$

Figure 3.2: PDS rule set that encodes the interprocedural control flow of the Java program in Fig. 3.1.

For answering the field-accesses-per-method problem, each control location p will be a subset of $K \times F$.

Example 3.3. Consider the Java program Prog from Fig. 3.1. Let $K = \{\mathsf{Prog}\}$ be the set of class names, and $F = \{\_a, \_b, \_c, \_d\}$ be the set of field names. Starting with the single-state PDS $\mathcal{P}_{3.2}$ from Ex. 3.2 that models the control flow of Prog, define the PDS $\mathcal{P}_{3.3} = (2^{K \times F}, \Gamma_{3.2}, \mathsf{Lab}_{3.2}, \Delta_{3.3}, \langle \emptyset, n_{17} \rangle)$, where $\Delta_{3.3}$ is $\Delta_{3.2}$ augmented to update the control locations upon a field access. For a program statement $\mathsf{stmt} \in \mathsf{Stmts}$, let $\mathsf{accesses}(\mathsf{stmt})$ be the set of fields accessed by stmt. For example, if stmt is "return _a" from line 4 in Fig. 3.1, then $\mathsf{accesses}(\mathsf{stmt}) =$

$\{(\mathsf{Prog}, \_\mathsf{a})\}$.[1] Define $\Delta_{3.3}$ to be the the following set of rules:

$$\left\{ \begin{array}{lll} \langle s, n_i \rangle \xhookrightarrow{\mathsf{stmt}} \langle (s \cup \mathsf{accesses}(\mathsf{stmt})), u \rangle & | & \langle p, n_i \rangle \xhookrightarrow{\mathsf{stmt}} \langle p, u \rangle \in \Delta_{3.2} \\ & & \wedge\, s \in 2^{K \times F} \end{array} \right\}.$$

PDS Reachability

Once a property of interest has been encoded in the control locations of a PDS, e.g., the "field-accesses-per-method" property of Ex. 3.3, the query of interest is to determine the set of reachable PDS configurations.

Given a PDS $\mathcal{P} = (P, \Gamma, \mathsf{Lab}, \Delta, c_0)$, we define for each label a in Lab the transition relation $\Rightarrow^a$ between configurations of $\mathcal{P}$ as follows: if $\langle p, \gamma \rangle \xhookrightarrow{a} \langle p', u' \rangle \in \Delta$, then $\langle p, \gamma\ u \rangle \Rightarrow^a \langle p', u'\ u \rangle$ for every $u \in \Gamma^*$. We denote by $\Rightarrow$ the union of the individual transition relations $\Rightarrow^a$ for each $a \in \mathsf{Lab}$, i.e., $\Rightarrow =_{\mathrm{df}} \bigcup_{a \in \mathsf{Lab}} \Rightarrow^a$. Finally, the reflexive transitive closure of $\Rightarrow$ is denoted by $\Rightarrow^*$.

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

$$post^*(C) =_{\mathrm{df}} \{c' \mid \exists c \in C : c \Rightarrow^* c'\}, \tag{3.1}$$

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

$$pre^*(C) =_{\mathrm{df}} \{c' \mid \exists c \in C : c' \Rightarrow^* c\}. \tag{3.2}$$

Reachability queries often result in, and sometimes begin from, an infinite set of configurations. Thus, we require a way to represent symbolically an infinite set of configurations using only a finite amount of storage, which is accomplished by using a non-deterministic finite automaton.

[1] We assume that field names are not redefined by subclasses.

Definition 3.4. For a PDS $\mathcal{P} = (\mathsf{P}, \Gamma, \mathsf{Lab}, c_0, \Delta)$, a $\mathcal{P}$-*automaton* is a tuple $\mathcal{A} = (Q, \Gamma, \rightarrow, P, F)$, where $Q \supseteq P$ is a finite set of states, $\rightarrow \subseteq Q \times \Gamma \times Q$ is a transition relation, P is the set of initial states, and F is the set of final states. We say that a configuration $\langle p, u \rangle$ is recognized by $\mathcal{A}$ if u is accepted by $\mathcal{A}$ when starting from state p. Without loss of generality, we assume that $\mathcal{A}$ has no transitions leading to an initial state, i.e., for all $q \in Q, \gamma \in \Gamma,$ and $p \in P, (q, \gamma, p) \notin \rightarrow$. A set of configurations is *regular* if it is accepted by a $\mathcal{P}$-automaton.

For a PDS $\mathcal{P}$ and a regular set of configurations C, Bouajjani et al. (1997) and Finkel et al. (1997) showed that the sets of configurations defined by *post*$^*(C)$ and *pre*$^*(C)$ are regular sets of configurations, respectively. Indeed, if C is represented as a $\mathcal{P}$-automaton $\mathcal{A}$, then the algorithms for computing *post*$^*(C)$ and *pre*$^*(C)$ produce $\mathcal{P}$-automata $\mathcal{A}_{post^*}$ and $\mathcal{A}_{pre^*}$, respectively, such that the set of configurations recognized by $\mathcal{A}_{post^*}$ and $\mathcal{A}_{pre^*}$ are exactly the sets *post*$^*(C)$ and *pre*$^*(C)$, respectively.

Example 3.5. Consider the PDS $\mathcal{P}_{3.3} = (2^{K \times F}, \Gamma_{3.2}, \mathsf{Lab}_{3.2}, \Delta_{3.3}, \langle \emptyset, n_{17} \rangle)$ defined in Ex. 3.3 for the Java program Prog in Fig. 3.1. Let $M = \{\mathsf{a}, \mathsf{b}, \mathsf{ab}, \mathsf{cd}, \mathsf{main}\}$ be the names of the methods defined by Prog; let $C = \{\langle \emptyset, n_m\ \text{access}_m \rangle \mid m \in M\}$ be a regular set of configurations, where n_m is the program counter for the entry point of method m (e.g., n_6 for method b), and access_m is a uniquely generated stack symbol for method m; and let $\mathcal{A}_{post^*}$ be the $\mathcal{P}_{3.3}$-automaton that recognizes *post*$^*(C)$. For a method m, the solution to the *field-accesses-per-method* problem can be obtained from $\mathcal{A}_{post^*}$ by computing

$$\bigcup \{ s \in 2^{K \times F} \mid \langle s, \text{access}_m \rangle \in \mathcal{A}_{post^*} \}.$$

Remark 3.6. A single reachability query of the PDS $\mathcal{P}_{3.3}$ is sufficient to solve the *field-accesses-per-method* problem. That is, $\mathcal{A}_{post^*}$ summarizes the reachability query, and one then asks multiple acceptance queries of $\mathcal{A}_{post^*}$ to determine the solution for each method m. This form of *staged analysis* generally provides substantial savings when multiple queries will be issued.

PDS **Paths**

For a PDS $\mathcal{P} = (\mathsf{P}, \Gamma, \mathsf{Lab}, \Delta, c_0)$, we sometimes need to reason about the set of PDS paths that cause $\mathcal{P}$ to make a transition from configuration c to configuration c', where a PDS path ρ is a sequence of PDS rules $[r_1, \ldots, r_n]$. We use *paths*(c, c') to denote this set of PDS paths. Note that, due to recursion and looping, the size of the set *paths*(c, c') can be infinite.

For program analysis we are generally interested in PDS paths that begin from the initial PDS configuration c_0. Hence, we distinguish such a PDS path by calling it a *run*. For a set C of PDS configurations, we define $\mathsf{Runs}(\mathcal{P}, \mathsf{C})$ to be the following set of PDS paths: $\{\rho \in \textit{paths}(c_0, c) \mid c \in \mathsf{C}\}$.

The Language of a PDS

The last concept we define for PDSs is that of the *language* of a PDS. The language of a PDS comes into play when dealing with both *communicating* pushdown systems (Ch. 4) and *multi*-pushdown systems (Ch. 7). The basic idea is to treat a PDS as a language generator.

Specifically, for a PDS $\mathcal{P} = (\mathsf{P}, \Gamma, \mathsf{Lab}, \Delta, c_0)$, we generalize the labeling function $\mathsf{lab} : \Delta \to \mathsf{Lab}$ from rules to labels to be a function from PDS paths to label words, $\mathsf{lab} : \Delta^* \to \mathsf{Lab}^*$, as follows:

$$\begin{array}{rcl} \mathsf{lab}([]) & = & \epsilon \\ \mathsf{lab}([\langle p, \gamma \rangle \xhookrightarrow{a} \langle p', u' \rangle]) & = & a \\ \mathsf{lab}([r_1, r_2, \ldots, r_n]) & = & \mathsf{lab}([r_1]) \cdot \mathsf{lab}([r_2, \ldots, r_n]) \end{array}$$

where $\cdot$ denotes the concatenation of labels to form label words.

Definition 3.7. For a PDS $\mathcal{P} = (\mathsf{P}, \Gamma, \mathsf{Lab}, \Delta, c_0)$ and a regular set of configurations C, the language of $\mathcal{P}$ with respect to C is defined as:

$$\mathsf{Lang}(\mathcal{P}, \mathsf{C}) \;\;=_{\mathrm{df}}\;\; \{\, \mathsf{lab}(\rho) \mid \rho \in \mathsf{Runs}(\mathcal{P}, \mathsf{C}) \,\}. \tag{3.3}$$

It is well-known that $\mathsf{Lang}(\mathcal{P}, C)$ is a context-free language.

Example 3.8. Let $\mathcal{P}_{3.8} = (\{p_a, p_b\}, \{\gamma, \bot\}, \{a, b\}, \Delta_{3.8}, \langle p_a, \bot\rangle)$ be a PDS where $\Delta_{3.8}$ is defined as follows:

$$\left\{\langle p_a, \bot\rangle \xhookrightarrow{a} \langle p_a, \gamma\bot\rangle, \langle p_a, \gamma\rangle \xhookrightarrow{a} \langle p_a, \gamma\gamma\rangle, \langle p_a, \gamma\rangle \xhookrightarrow{b} \langle p_b, \epsilon\rangle, \langle p_b, \gamma\rangle \xhookrightarrow{b} \langle p_b, \epsilon\rangle\right\}.$$

The language $\mathsf{Lang}(\mathcal{P}_{3.8}, \{\langle p_b, \bot\rangle\})$ is the canonical context-free language $\{a^n b^n \mid n \geqslant 1\}$.

3.2 Weighted Pushdown Systems

Weighted pushdown systems (WPDSs) generalize PDSs by attaching to a PDS $\mathcal{P}$ a semiring and a function that maps PDS rules to elements in the semiring's domain (Bouajjani et al., 2003; Reps et al., 2003, 2005). We often refer to a semiring as a *weight domain*, and elements of a semiring's domain as *weights*. Reps et al. (2003, 2005) have shown the strong connection between interprocedural dataflow analysis and computing reachability queries on a WPDS.

Definition 3.9. A ***bounded idempotent semiring*** is a tuple $\mathcal{S} = (D, \oplus, \otimes, \bar{0}, \bar{1})$, where D is a finite set of elements called ***weights***, $\bar{0}, \bar{1} \in D$, 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 $\bar{0}$, where $\oplus$ is idempotent: $\forall x \in D, x \oplus x = x$.

2. $(D, \otimes)$ is a monoid with neutral element $\bar{1}$.

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. $\bar{0}$ is an annihilator with respect to $\otimes$: $\forall x \in D, x \otimes \bar{0} = \bar{0} = \bar{0} \otimes x$.

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.

As an example, a common semiring used in program analysis is the semiring whose elements are binary relations over a finite set V.

Definition 3.10. For a finite set V, the *relational semiring* $\mathcal{S}_V = (2^{V \times V}, \cup, ;, \emptyset, \mathit{Id})$ is the semiring whose elements are binary relations over V; the combine operation is union; the extend operation is relational composition (i.e., $\forall R_1, R_2 \in 2^{V \times V} : R_1; R_2 = \{(v_1, v_3) \mid \exists v_2 : (v_1, v_2) \in R_1 \wedge (v_2, v_3) \in R_2\}$); the $\overline{0}$ element is the empty relation; and the $\overline{1}$ element is the identity relation (i.e., $\mathit{Id} = \{(v, v) \mid v \in V\}$).

Definition 3.11. A *weighted PDS* (WPDS) is a tuple $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$, where $\mathcal{P} = (\mathsf{P}, \Gamma, \mathsf{Lab}, \Delta, c_0)$ 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 $\mathit{paths}(c, c')$ is defined on the underlying PDS $\mathcal{P}$. Similarly, for a set of configurations C, the set of PDS runs $\mathsf{Runs}(\mathcal{W}, C)$ is equal to $\mathsf{Runs}(\mathcal{P}, C)$ for the underlying PDS $\mathcal{P}$.

For a set of configurations C, reachability queries for PDSs—*post** and *pre**—are generalized for WPDSs as follows:

$$\mathit{post}^*(C) =_{\mathrm{df}} \{\, (c', w) \mid \exists c \in C : c \Rightarrow^* c' \wedge w = \bigoplus_{\rho \in \mathit{paths}(c, c')} f(\rho)\} \tag{3.4}$$

$$\mathit{pre}^*(C) =_{\mathrm{df}} \{\, (c', w) \mid \exists c \in C : c' \Rightarrow^* c \wedge w = \bigoplus_{\rho \in \mathit{paths}(c, c')} f(\rho)\} \tag{3.5}$$

Given a WPDS, Reps et al. (2003, 2005) present efficient algorithms for computing weighted *post** and *pre** queries. This is accomplished by staging the query. First, similar to PDSs, given a regular set of configurations C, a reachability query computes a weighted automaton that recognizes the set of reachable configurations. Second, the automaton can then be queried to determine the weight that is associated with a (forwards or backwards) reachable configuration c'. However, the automaton is not a $\mathcal{P}$-automaton but a $\mathcal{W}$-automaton, where the difference is that each transition is labeled with a weight.

Definition 3.12. Let $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$ be a WPDS, where $\mathcal{P} = (P, \Gamma, \mathsf{Lab}, \Delta, c_0)$ is a PDS, and $\mathcal{S} = (D, \oplus, \otimes, \bar{0}, \bar{1})$ is a semiring. A $\mathcal{W}$-*automaton* is a tuple $\mathcal{A}_{\mathcal{W}} = (\mathcal{A}, \mathcal{S}, l)$, where $\mathcal{A} = (Q, \Gamma, \rightarrow, P, F)$ is a $\mathcal{P}$-automaton, and l is a function that maps each transition t in $\rightarrow$ to a weight w in D. A configuration $c = \langle p, u \rangle$, is *recognized* by $\mathcal{A}_{\mathcal{W}}$ with weight w if u is accepted by $\mathcal{A}$ when starting from state p, and $w \neq \bar{0}$ is the combine of all accepting paths for u in $\mathcal{A}$ using the function l. For an accepting path $t_1, \ldots, t_n$ of transitions for u starting from p, the weight of $t_1, \ldots, t_n$ is $l(t_1) \otimes \ldots \otimes l(t_n)$ if $\mathcal{A}_{\mathcal{W}}$ is the result of a *pre** query, and is $l(t_n) \otimes \ldots \otimes l(t_1)$ if $\mathcal{A}_{\mathcal{W}}$ is the result of a *post** query. We use $\mathcal{A}_{\mathcal{W}}(c)$ to denote the weight w that $\mathcal{A}_{\mathcal{W}}$ recognizes c with. For a regular set of configurations C, we define $\mathcal{A}_{\mathcal{W}}(C) = \bigoplus \{ \mathcal{A}_{\mathcal{W}}(c) \mid c \in C \}$.

Finally, we are sometimes interested in computing the *combine-over-all-valid-paths* value between two regular sets of configurations C and C'.

Definition 3.13. Let $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$ be a WPDS, where $\mathcal{P} = (P, \Gamma, \mathsf{Lab}, \Delta, c_0)$. Let $C, C' \subseteq P \times \Gamma^*$ be two regular sets of configurations. The *combine-over-all-valid-paths* value $\mathsf{COVP}(C, C')$ is defined as

$$\bigoplus_{\sigma \in \textit{paths}(c, c'),\ c \in C,\ c' \in C'} f(\sigma). \tag{3.6}$$

Reps et al. (2003, 2005) also present an efficient algorithm for computing a COVP query. Essentially, for a WPDS $\mathcal{W}$, a $\mathcal{W}$-automaton $\mathcal{A}_{\mathcal{W}}$ is computed by solving a *post*$^*(C)$ query. The desired weight is then $\mathcal{A}_{\mathcal{W}}(C')$.

As a concrete example, we show how to use a WPDS COVP query to compute a solution to the *field-accesses-per-method* problem for the Java program Prog in Fig. 3.1.

Example 3.14. Consider the program Prog from Fig. 3.1, which has the set of class names K and field names F (as defined in Ex. 3.3). Let $V = 2^{K \times F}$ be the finite set consisting of all subsets of $K \times F$: V is equal to the set of control locations $P_{3.3}$ of the PDS $\mathcal{P}_{3.3}$ from Ex. 3.3. Let $\mathcal{W}_{\mathsf{Prog}} = (\mathcal{P}_{3.2}, \mathcal{S}_V, f)$ be a WPDS where $\mathcal{P}_{3.2} = (P_{3.2}, \Gamma_{3.2}, \mathsf{Lab}_{3.2}, \Delta_{3.2}, \langle p, n_{17} \rangle)$ is the PDS from Ex. 3.2; the semiring domain is binary relations over the subsets of $K \times F$; and f is defined as follows: for PDS rule $r \in \Delta$, if $\mathsf{lab}(r) \in \{$"l = o.f", "o.f = l"$\}$ then $f(r) = \{(s, s \cup (K_o, f)) \mid s \subseteq K \times F\}$, where K_o is the class name of the object-reference o. For a method m and its uniquely generated stack symbol access_m, the query "$\mathsf{COVP}(\{\langle p, e_m\ \mathrm{access}_m \rangle\}, \{\langle p, \mathrm{access}_m \rangle\})$" computes a relation R that can be used to determine the fields that are accessed by m. In particular, the projection of R on the empty set, denoted by $R[\emptyset]$, gives the fields accessed by m.

Remark 3.15. Ex. 3.14 replaces PDS $\mathcal{P}_{3.3}$ from Ex. 3.3, which has multiple control locations, by a WPDS whose PDS component has a single control location, and whose semiring domain is binary relations on the control locations of the original PDS. This technique can be used for an arbitrary PDS (Schwoon, 2002).

An implementation of the WPDS algorithms (Reps et al., 2005) and some not discussed here (Lal et al., 2005; Lal and Reps, 2006) can be found in the freely downloadable library "WALi: **W**eighted **A**utomaton **Li**brary" (Kidd et al., 2009a).

4 COMMUNICATING PUSHDOWN SYSTEMS

The approach taken in the dissertation to verifying AS-atomicity of a concurrent Java program is to use PDS-based software model checking. For the first part of the dissertation, specifically Chs. 5 and 6, we use the PDS-based formalism known as *communicating pushdown systems* (CPDSs).

CPDSs were first proposed by Bouajjani et al. (2003) as a generic model for programs that perform global rendezvous-style synchronization. Conceptually, a CPDS is a set of PDSs. Because each PDS defines a context-free language (CFL), a CPDS can also be viewed as a set of CFLs. From the language point of view, a CPDS-reachability query is tantamount to determining whether the intersection of the set of CFLs is empty, which is a known undecidable problem. Thus, CPDS model checking uses only a *semi-decision procedure* (SDP), i.e., an algorithm that (attempts to) answer an undecidable "yes/no" (reachability) query, but can also return the inconclusive result "maybe".

Bouajjani et al. (2003) proposed four language abstractions for the CFLs of the PDSs of a CPDS. The abstractions are *regular* over-approximations, and thus determining the emptiness of intersection of the (regular) abstract languages is decidable. Because over-approximation is used, if the intersection of the abstract languages of the PDSs is empty, then so too is the intersection of the concrete CFLs of the PDSs. Unfortunately, for the case when the intersection of the abstract languages is non-empty, with the abstractions proposed by Bouajjani et al. (2003) it is not possible to determine if a word in the abstract intersection is in the concrete intersection. Moreover, they do not discuss how to refine the abstractions.

Together with several collaborators, I developed refinable abstractions that address both of these issues (Chaki et al., 2006). In addition, this chapter presents (1) a previously unpublished abstraction that is more precise than the one presented by Chaki et al. (2006), and (2) three heuristics for performing abstraction refinement. We conclude with a case study that uses CPDS model checking for analyzing a model of a Windows Bluetooth driver.

The rest of this chapter is organized as follows. §4.1 presents an overview of the use of refinable abstractions. §4.2 presents the formal definition of CPDSs. §4.3 presents α-SDP, the SDP from Chaki et al. (2006) for performing CPDS reachability analysis. §4.4 presents β-SDP, an improved SDP for performing CPDS reachability analysis. §4.5 presents two heuristics, and their combination, for performing CPDS reachability analysis. §4.6 presents a case study that compares each of the defined SDPs for analyzing a model of a Windows Bluetooth driver. §4.7 presents a summary.

4.1 Overview

Given a set of CFLs $L_1, \ldots, L_n$ for PDSs $\mathcal{P}_1, \ldots, \mathcal{P}_n$, the SDP that is implemented by the CPDS model checker CPDSMC uses abstraction to define a regular over-approximation R_i for each CFL L_i, $1 \leqslant i \leqslant n$, and approximates the intersection result $L = \bigcap_{i=1}^{n} L_i$ by $R = \bigcap_{i=1}^{n} R_i$. The abstractions employed use a form of bounded precision. Namely, for a bound k, the language L_i is divided into two sets: (1) words of length less than k; and (2) words of length greater than or equal to k. The first set is referred to as the *concrete set* because it can be modeled precisely, i.e., no abstraction is required. Likewise, the second set is referred to as the *abstract set* because abstraction of this set is required to ensure decidability of the emptiness of intersection.

Because each R_i, $1 \leqslant i \leqslant n$, is an over-approximation of the corresponding CFL L_i, R is an over-approximation of L; consequently, if $R = \emptyset$ then $L = \emptyset$. The key to the SDP is that if $R \neq \emptyset$, one can determine if R contains a concrete word w from L (i.e., the length of w is less than k). If no such w exists, then the abstractions must be refined, which amounts to increasing the precision bound k. Thus, the SDP uses the succession of approximations (indicated by $k, k', k'', \ldots$) $\bigcap_{i=1}^{n} R_i^k, \bigcap_{i=1}^{n} R_i^{k'}, \bigcap_{i=1}^{n} R_i^{k''}$, and so on, to determine if the actual intersection $L = \bigcap_{i=1}^{n} L_i$ is empty. This process continues until either (i) a concrete word has been found, (ii) the intersection has been shown to be empty, or (iii) CPDSMC has exhausted the available resources.

To define a regular over-approximation of the abstract set, various abstractions have been considered: The *prefix abstraction* (Chaki et al., 2006) precisely models the prefix of length k of each abstract word, but loses precision by allowing any sequence of symbols to follow the prefix. The *suffix abstraction* (Chaki et al., 2006) precisely models the suffix of length k of each abstract word, but loses precision by allowing any sequence of symbols to precede the suffix. The *bifix* abstraction combines the prefix and suffix abstractions so that abstract words are constrained on each end, but loses precision by allowing any sequence of symbols to come in-between. Finally, more precise abstractions for the parts of words that lie beyond the k-bounded threshold can be used (cf. Nederhof's survey (Nederhof, 1999)). For the rest of the chapter, we will only consider the prefix abstraction; however, the following discussion and the presented SDPs apply to each of the possible abstractions described above.

4.2 Definition

A CPDS consists of a set of n PDSs that perform global rendezvous-style synchronization on a set of communicating actions. Informally, the labels that annotate the rules of the PDSs are interpreted as communication actions. A global configuration g of a CPDS is a tuple $(c_1, \ldots, c_n)$ of configurations of the individual PDSs.[1] To model global rendezvous-style synchronization, a CPDS can make a transition from global configuration g to global configuration g' iff each individual PDS can make a (local) transition on the same communicating action a, i.e, each PDS must use its locally defined transition relation $\Rightarrow^a$. This is an important point because it induces a tight coupling between the PDSs of a CPDS, which leads to undecidability for CPDS model checking. The focus of

[1] We differentiate between PDS configurations and CPDS configurations by referring to the latter as global configurations.

Ch. 7 is a technique that can decouple the PDSs for certain problems, which results in a decision procedure for instances of those problems.

Definition 4.1. A *communicating pushdown system* (CPDS) is a tuple $\Pi = (\mathcal{P}_1, \ldots, \mathcal{P}_n, \mathsf{Act})$ of pushdown systems, where $\mathsf{Act} = \bigcup_{i=1}^{n} \mathsf{Lab}_i$ is the union of the individual action sets of the PDSs (Lab_i is the set of actions of $\mathcal{P}_i$). There is a special action τ that belongs to all the sets Lab_i, $1 \leqslant i \leqslant n$, such that for all $a \in \mathsf{Lab} : \tau \cdot a = a = a \cdot \tau$. The action τ is neutral with respect to concatenation, and represents an internal action of a process modeled by a PDS. The non-τ actions correspond to synchronization actions.

For a CPDS Π, a *global configuration* is a tuple $g = (c_1, \ldots, c_n)$ of configurations of $\mathcal{P}_1, \ldots, \mathcal{P}_n$. The initial global configuration $g_0 = (c_0^1, \ldots, c_0^n)$ consists of the initial configurations of the individual PDSs. For each action $a \in \mathsf{Act}$, we define the relation $\xrightarrow{a}$ between global configurations as follows:

- $(c_1, \ldots, c_n) \xrightarrow{\tau} (c_1', \ldots, c_n')$ if there is an index $1 \leqslant i \leqslant n$ such that $c_i \Rightarrow^{\tau} c_i'$ and $c_j' = c_j$ for every $j \neq i$;
- $(c_1, \ldots, c_n) \xrightarrow{a} (c_1', \ldots, c_n')$ if for every i, $1 \leqslant i \leqslant n$, $c_i \Rightarrow^{a} c_i'$: all processes synchronize on a and move simultaneously.

A set G of global configurations is *regular* if it can be represented as a tuple $(C_1, \ldots, C_n)$ of regular sets of configurations of the individual PDSs, i.e., $G = \{(c_1, \ldots, c_n) \mid c_1 \in C_1, \ldots, c_n \in C_n\}$.

From now on, we fix a CPDS $\Pi = (\mathcal{P}_1, \ldots, \mathcal{P}_n, \mathsf{Act})$ and a regular set of global configurations $G = (C_1, \ldots, C_n)$, where for every i, $1 \leqslant i \leqslant n$, $\mathcal{P}_i = (P_i, \Gamma_i, \mathsf{Lab}_i, \Delta_i, c_0^i)$,

Definition 4.2. The language of Π with respect to G, denoted by $\mathsf{Lang}(\Pi, G)$, consists of words formed by concatenating sequences of communicating actions that allow Π to make a transition from the initial global configuration g_0 to some global configuration g in G.

$$\{\, w \in \mathsf{Act}^* \mid \exists g_1, \ldots, g_n \in G \,:\, g_0 \xrightarrow{a_1} \ldots \xrightarrow{a_n} g_n \land w = a_1 \cdot \ldots \cdot a_n \,\}. \quad (4.1)$$

Remark 4.3. Because the special action τ is neutral with respect to concatenation, i.e., $\tau \cdot a = a = a \cdot \tau$ for all a in Act, the length of a word w, denoted by $|w|$, in $\mathsf{Lang}(\Pi, G)$ might not necessarily reflect the number of transitions that Π makes to generate w. That is, if $w = a_1 \cdot \ldots \cdot a_n$, then $|w| \leqslant n$ because some of the action symbols could be τ.

A CPDS is a natural model of concurrent programs where processes synchronize via rendezvous. The special transition relation $\xrightarrow{\tau}$ models a particular process of a concurrent program performing a local transition. A transition relation $\xrightarrow{a}$ models all processes synchronizing on a. Because a CPDS uses global rendezvous-style synchronization—i.e., all PDSs synchronize on an action a and make a transition simultaneously—and because the special action τ is neutral with respect to concatenation, we can see that the following holds:

$$\mathsf{Lang}(\Pi, G) \quad = \quad \bigcap_{i=1}^{n} \mathsf{Lang}(\mathcal{P}_i, C_i). \qquad (4.2)$$

Modeling k-wise synchronization

When using CPDSs to model real systems, it is often the case that not all processes synchronize on every action. For example, if the program model uses pairwise synchronization, each action a would be a member of exactly two action sets Lab_i and Lab_j, $i \neq j$. However, for Eqn. (4.2) to hold, all PDSs must have the same set of actions. Thus, we need to insert everywhere in the paths of $\mathcal{P}_i$ labels that correspond to the synchronization actions that are not in Lab_i, but that the other PDSs can perform. Bouajjani et al. (2003) formalized this encoding via the *shuffle* operation. For $a, b \in \mathsf{Act}$ and $u, v \in \mathsf{Act}^*$, the shuffle operation $⧢$ on words is defined as follows:

$$\begin{array}{rcccl} u ⧢ \epsilon & = & \{u\} & = & \epsilon ⧢ u \\ au ⧢ bv & = & \big(\{a\} \cdot (u ⧢ bv)\big) \cup \big(\{b\} \cdot (au ⧢ v)\big), & & \end{array}$$

and for two languages L and L', $L ⧢ L' = \{u ⧢ v \mid u \in L, v \in L'\}$. Using the shuffle operation, we define the language L_i as follows:

$$L_i =_{\text{df}} \mathsf{Lang}(\mathcal{P}_i, C_i) ⧢ (\mathsf{Lab} \setminus \mathsf{Lab}_i)^*. \tag{4.3}$$

Bouajjani et al. (2003) show that Eqn. (4.2) is then extended as follows:

$$\mathsf{Lang}(\Pi, G) = \bigcap_{i=1}^{n} L_i. \tag{4.4}$$

They also showed that relaxing the synchronization model has no effect on the expressive power of a CPDS—one can simply add "self-loops" to the rule set of each PDS to account for unused actions. That is, define a CPDS $\Pi' = (\mathcal{P}'_1, \ldots, \mathcal{P}'_n)$, where for $1 \leqslant i \leqslant n$, $\mathcal{P}'_i = (P_i, \Gamma_i, \mathsf{Lab}, \Delta'_i, c^i_0)$ is $\mathcal{P}_i$ with the rule set Δ_i augmented as follows:

$$\Delta'_i = \Delta_i \cup \{\langle p, \gamma\rangle \xhookrightarrow{a} \langle p, \gamma\rangle \mid p \in P_i, \gamma \in \Gamma_i, a \in (\mathsf{Act} \setminus \mathsf{Lab}_i)\}.$$

Example 4.4. Ex. 3.8 in Ch. 3 defined the PDS $\mathcal{P}_{3.8} = (\{p_a, p_b\}, \{\gamma, \bot\}, \{a, b\}, \Delta_{3.8}, \langle p_a, \bot\rangle)$ such that $\mathsf{Lang}(\mathcal{P}_{3.8}, \{\langle p_b, \bot\rangle\})$ is the canonical context-free language $\{a^n b^n \mid n \geqslant 1\}$. Let $\mathcal{P}_{4.4}$ be $\mathcal{P}_{3.8}$ with an unused action c and PDS rules added that explicity account for it: $\mathcal{P}_{4.4} = \big(\{p_a, p_b\}, \{\gamma, \bot\}, \{a, b, c\}, \Delta_{4.4}, \langle p_a, \bot\rangle\big)$, where $\Delta_{4.4} = \Delta_{3.8} \cup \big\{\langle p, x\rangle \xhookrightarrow{c} \langle p, x\rangle \mid p \in \{p_a, p_b\}, x \in \{\gamma, \bot\}\big\}$. The language $\mathsf{Lang}\big(\mathcal{P}_{4.4}, \{\langle p_b, \bot\rangle\}\big)$ is $\big\{(c^* a)^n (c^* b)^n c^* \mid n \geqslant 1\big\}$.

The language $\mathsf{Lang}(\Pi, G)$ defined by Eqn. (4.4) is equivalent to the language $\mathsf{Lang}(\Pi', G)$ defined by Eqn. (4.2)—i.e., the following holds:

$$\mathsf{Lang}(\Pi, G) \;=\; \bigcap_{i=1}^{n} L_i \;=\; \bigcap_{i=1}^{n} \mathsf{Lang}(\mathcal{P}'_i, C_i) \;=\; \mathsf{Lang}(\Pi', G).$$

The α-SDP that is presented next only considers a CPDS where the action set of each PDS is equal to Act, i.e., $\mathsf{Lab}_i = \mathsf{Act}$ for each PDS $\mathcal{P}_i$, $1 \leqslant i \leqslant n$. §4.4 presents the β-SDP, which, unlike α-SDP, takes advantage of the case when the action sets are not all equal to Act.

4.3 Reachability Analysis of CPDSs

The goal of CPDS model checking is to determine if a given set G of global configurations is reachable in Π. As discussed in §4.1, CPDS model checking is in general undecidable. Thus, we develop a *semi-decision procedure* that attempts to answer a reachability query. The semi-decision procedure (SDP) returns "Yes" if G is reachable, "No" if G is not reachable, and "Maybe" if available resources are exhausted.[2]

Each SDP presented in the chapter proceeds in two phases. First, abstraction is used to compute a regular over-approximation of $\mathsf{Lang}(\Pi, G)$. Second, the regular over-approximation is checked to determine whether the reachability query can be answered definitively. If so, the user is given the answer. Otherwise, the regular over-approximation is refined and the process continues.

The Prefix Abstraction

The *prefix* abstraction is a *bounded* abstraction that is precise up to a bound k, and after that bound is exhausted, precision is lost.

[2] Available resources generally means the memory available to the analyzer, as well as a time limit.

Definition 4.5. Given a bound k and a language $L \subseteq \mathsf{Act}^*$, the *prefix abstraction* $\alpha_k(L)$ is the language:

$$\left\{w \mid w \in L \wedge |w| < k\right\} \cup \left\{\lfloor w \rfloor^k w' \mid w \in L \wedge |w| \geqslant k \wedge w' \in \mathsf{Act}^*\right\}, \quad (4.5)$$

where for a word w and a bound k, $\lfloor w \rfloor^k$ denotes the prefix of w of length k. A word $w \in \alpha_k(L)$ is called a *concrete* word if $|w| < k$ and an *abstract* word otherwise.

Example 4.6. Given $\mathcal{P}_{4.4}$ from Ex. 4.4 and bound $k = 3$, $\alpha_3\big(\mathsf{Lang}(\mathcal{P}_{4.4}, \{\langle p_b, \bot\rangle\})\big)$ is the language $\left\{ab\right\} \cup \left\{aaaw,\ aabw,\ aacw,\ abcw,\ acaw,\ acbw,\ accw,\ caaw,\ cabw,\ cacw,\ ccaw,\ cccw \mid w \in \mathsf{Lab}^*\right\}$.

For a language L and bound k, the prefix abstraction $\alpha_k(L)$ can be represented by two sets:

1. The *concrete set* consists of all concrete words and is represented exactly.
2. The *abstract set* consists of the prefixes of length k, where each prefix summarizes an infinite set of words.

The representation is finite because there are only a finite number of words w such that $|w| \leqslant k$.

Example 4.7. The language $\alpha_3\big(\mathsf{Lang}(\mathcal{P}_{4.4}, \{\langle p_b, \bot\rangle\})\big)$ from Ex. 4.6 can be represented by the finite sets $\big(\{ab\}, \{aaa,\ aab,\ aac,\ abc,\ aca,\ acb,\ acc,\ caa,\ cab,\ cac,\ cca,\ ccc\}\big)$.

Implementing the Prefix Abstraction

Given a PDS $\mathcal{P}$ and a regular set of configurations C, the prefix abstraction defines a regular approximation to $\mathsf{Lang}(\mathcal{P}, C)$. To compute a prefix abstraction, we use a WPDS COVP query, where the weight domain is the prefix semiring.

Definition 4.8. Given action set Act and a bound k, let D_{α_k} be the powerset of $\bigcup_{0 \leqslant i \leqslant k} \mathsf{Act}^i$ (i.e., D_{α_k} is the set of all subsets of words over Act of length less than or equal to k). For two words $w_1 = a_1 \dots a_i$ and $w_2 = b_1 \dots b_j$, let $w_1 \bowtie_k w_2$ be the word $\lfloor a_1 \dots a_i b_1 \dots b_j \rfloor^k$, and extend $\bowtie_k$ to operate on sets of words in the natural way.[3] The *prefix semiring* $\mathcal{S}_{\alpha_k}$ is defined as $(D_{\alpha_k}, \cup, \bowtie_k, \emptyset, \{\epsilon\})$.

For PDS $\mathcal{P} = (P, \Gamma, \mathsf{Lab}, \Delta, c_0)$ and bound k, define $\mathcal{W}_{\alpha_k} = (\mathcal{P}, \mathcal{S}_{\alpha_k}, f)$ to be a WPDS, where $\mathcal{S}_{\alpha_k}$ is the prefix semiring over Lab and k, and for each rule $r = \langle p, \gamma \rangle \xhookrightarrow{a} \langle p', u' \rangle$ in Δ, $f(r)$ is the weight $\{a\}$. The solution to the query $\mathsf{COVP}(\{c_0\}, C)$ on $\mathcal{W}_{\alpha_k}$ computes $\alpha_k\big(\mathsf{Lang}(\mathcal{P}, C)\big)$.

In practice, one must be careful with the representation of a weight. For a bound k, the size of a weight, i.e., the number of words in the set, is exponential in k. More precisely, it is bound by $O(\mathsf{Lab}^k)$. Thus, an explicit representation using sets of strings will not scale. CPDSMC encodes sets of strings using binary decision diagrams (BDDs) (Bryant, 1986). BDDs are also of exponential size in the worst case; however, their implementation saves space by representing duplicate elements (e.g., common prefixes of words) with shared structures.

The α-SDP

Alg. 4.1 presents α-SDP (originally defined by Chaki et al. (2006)). α-SDP takes as input a CPDS Π and a regular set of configurations G, and attempts to determine if G is reachable in Π. To do so, it makes use of the prefix abstraction. Initially, the prefix bound k is set to 1 (line 1). For a given value of k, the prefix abstraction of each language L_i for PDS $\mathcal{P}_i$ (cf. Eqn. (4.3)) computes a regular over-approximation of L_i (line 3). The intersection of the regular over-approximations is then queried to determine if G is reachable. If the intersection is empty, then G must not be reachable (lines 4–5). This is sound because each $\alpha_k(L_i)$ over-approximates L_i. That is, $\mathcal{A}_\cap = \bigcap_{i=1}^n \alpha_k(L_i) \supseteq \bigcap_{i=1}^n L_i$, and if $\mathcal{A}_\cap = \emptyset$ then $\bigcap_{i=1}^n L_i = \emptyset$. Otherwise, a minimal-length word $w_{\min}$ is extracted from $\mathcal{A}_\cap$, and

[3] Recall that for a word w, the word $\lfloor w \rfloor^k$ is equal to w if $|w| < k$, and the prefix of w of length k otherwise.

Algorithm 4.1: The α-SDP. ($w_{\min}$ is a minimal-length word in $\mathcal{A}_\cap$.)

```
   Input 1: $\Pi = (\mathcal{P}_1, \ldots, \mathcal{P}_n, \mathsf{Lab})$, a CPDS
   Input 2: $G = (C_1, \ldots, C_n)$, a regular set of configurations
   output : Whether $G$ is reachable in $\Pi$.
1  k=1
2  while true do                               /* Resources are available */
3      $\mathcal{A}_\cap = \bigcap_{i=1}^{n} \alpha_k(L_i)$
4      if $\mathcal{A}_\cap = \emptyset$ then                          /* G is not reachable */
5          return no
6      else if $|w_{min}| < k$ then                        /* G is reachable */
7          return yes
8      else
9          k=k+1
10 return maybe                                 /* Exhausted resources */
```

α-SDP checks whether $w_{\min}$ is concrete, i.e., $|w_{\min}| < k$ (line 6). Recall that the prefix abstraction is precise for words whose length is strictly less than k. Thus, if such a word exists then that word must also be in the concrete intersection and hence a member of $\mathsf{Lang}(\Pi, G)$ (line 7). If no such word exists, i.e., all words in $\mathcal{A}_\cap$ have length greater than or equal to k, then k is incremented to refine the abstraction (line 9). This process terminates when either a value of k has been reached that is precise enough to answer the query, or CPDSMC has exhausted all of its available resources.

4.4 Improved Reachability Analysis

The CPDS reachability analysis described in §4.3 is inefficient because of the need to account for the right operand of the shuffle term in Eqn. (4.3). That is, explicitly adding rules to PDS $\mathcal{P}_i$ to account for the unused actions (i.e., $\mathsf{Act} \setminus \mathsf{Lab}_i$) causes (i) the abstraction to lose more precision than necessary, and (ii) answer sets to be possibly of exponentially greater size than with the technique described in this section. In fact, when only using $\alpha_k(L_i)$ to approximate the language of $\mathcal{P}_i$, CPDSMC exhausted all resources on even the simplest of queries. Using the improved $\beta_k(L_i)$ abstraction defined below, CPDSMC took only 2.76 seconds to determine reachability for a Bluetooth model (see §4.6), whereas using the abstraction $\alpha_k(L_i)$, it ran out of memory on a dual-core Xeon processor with 4 GB of memory.

We motivate the β_k abstraction by reviewing CFLs and their prefix abstractions from earlier examples. Ex. 3.8 in Ch. 3 showed that the language $\mathsf{Lang}(\mathcal{P}_{3.8}, \langle p_b, \epsilon\rangle)$ is $\{a^n b^n \mid n \geqslant 1\}$. For $k = 3$, the finite sets that represent $\alpha_3\big(\mathsf{Lang}(\mathcal{P}_{3.8}, \{\langle p_b, \epsilon\rangle\}\big)$ is $\big\{ab, aaa, aab\big\}$. When an unused communicating action c is introduced, Ex. 4.4 showed that the language $\mathsf{Lang}\big(\mathcal{P}_{4.4}, \{\langle p_b, \bot\rangle\}\big)$ is $\big\{(c^*a)^n(c^*b)^n c^* \mid n \geqslant 1\big\}$. Finally, Ex. 4.7 showed that the pair of finite sets that represent $\alpha_3\big(\ \mathsf{Lang}(\mathcal{P}_{4.4}\ , \ \{\langle p_b, \bot\rangle\}\ )\big)$ is $\big(\{ab\}, \{aaa, aab, aac, abc, aca, acb, acc, caa, cab, cac, cca, ccc\}\big)$.

Observe that for the prefix abstraction $R_{\alpha_3} = \alpha_3\big(\mathsf{Lang}(\mathcal{P}_{4.4}, \{\langle p_b, \epsilon\rangle\}\big)$, all of the regularity of the unused action c in $\mathsf{Lang}(\mathcal{P}_{4.4}, \{\langle p_b, \epsilon\rangle\})$, i.e., the c^* in $\big\{(c^*a)^n(c^*b)^n c^* \mid n \geqslant 1\big\}$, has been lost. Moreover, the prefix abstraction R_{α_3} actually recognizes words where a "b" action comes before an "a" action. This is because the abstract word ccc in the pair of finite sets that represents R_{α_3} is interpreted as the set of words $\{cccw \mid w \in \mathsf{Act}^*\}$. Finally, explicitly modifying $\mathcal{P}_i$ to account for unused actions, as in Ex. 4.4, causes the set that represents $\alpha_k(L_i)$ to be exponentially larger in the size of $\mathsf{Act} \setminus \mathsf{Lab}_i$ (e.g., aaa blows up to $\big\{aaa, aac, aca, acc, caa, cac, cca, ccc\big\}$). We avoid this inefficiency *and* define a more precise abstraction, β_k, by leveraging the fact that we can use the shuffle operation to account for unused actions.

Recall that for a PDS $\mathcal{P}_i$, the *concrete* language of interest is L_i defined by Eqn. (4.3). Instead of directly applying the prefix abstraction to L_i (i.e., $\alpha_k(L_i)$), we gain precision by pulling the shuffle operation *outside of* the approximation:

$$\beta_k(L_i) \quad =_{\text{df}} \quad \alpha_k\big(\mathsf{Lang}(\mathcal{P}_i, C_i)\big) \sqcup\!\sqcup (\mathsf{Act} \setminus \mathsf{Lab}_i)^*. \tag{4.6}$$

It is easy to see that $\beta_k(L_i)$ is an over-approximation of L_i, and that incrementing

$\beta_k(L_i)$		$\alpha_k(L_i)$
$\alpha_k\big(\mathsf{Lang}(\mathcal{P}_i, C)\big) \sqcup\!\sqcup (\mathsf{Act} \setminus \mathsf{Lab}_i)^*$	$\subseteq$	$\alpha_k\big(\mathsf{Lang}(\mathcal{P}_i, C) \sqcup\!\sqcup (\mathsf{Act} \setminus \mathsf{Lab}_i)^*\big)$

Figure 4.1: Precision comparison between $\beta_k(L_i)$ and $\alpha_k(L_i)$.

k obtains a more precise over-approximation, i.e., for every i, $L_i \subseteq \beta_{k+1}(L_i) \subseteq \beta_k(L_i)$.

The β_k abstraction avoids the exponential increase that occurs when additional PDS rules are introduced to account for unused actions. Also, β_k provides a *more precise* over-approximation of L_i because β_k performs an *exact* shuffle. The relationship between $\alpha_K(L_i)$ and $\beta_k(L_i)$ is shown in Fig. 4.1.

Moreover, performing the shuffle *after* computing the prefix abstraction is trivial. Let $\mathcal{A}_i = (Q_i, \Sigma_i, \delta_i, q_0, F)$ be the automaton (defined in the usual way) that accepts the language $\alpha_k\big(\mathsf{Lang}(\mathcal{P}_i, C_i)\big)$. The shuffle operation $\mathsf{Lang}(\mathcal{A}_i) \sqcup\!\sqcup (\mathsf{Act} \setminus \mathsf{Lab}_i)^*$ is performed directly on $\mathcal{A}_i$ by augmenting the transition relation δ_i with the following set of additional transitions: $\{(q, a, q) \mid q \in Q_i \wedge a \in (\mathsf{Act} \setminus \mathsf{Lab}_i)\}$.

Example 4.9. Let $\mathsf{Act} = \{a, b, c\}$. The language $\beta_3\big(\mathsf{Lang}(\mathcal{P}_{4.4}, \{\langle p_b, \bot\rangle\})\big)$ is $\{c^*ac^*bc^*\} \cup \{c^*ac^*aw \mid w \in \mathsf{Act}^*\}$. This is a more precise approximation than the language $\alpha_3\big(\mathsf{Lang}(\mathcal{P}_{4.4}, \{\langle p_b, \bot\rangle\})\big) = \big\{ab\big\} \cup \big\{aaaw, aabw, aacw, abcw, acaw, acbw, accw, caaw, cabw, cacw, ccaw, cccw \mid w \in \mathsf{Lab}^*\big\}$ from Ex. 4.6 because for abstract words, the former enforces that two "a" actions occur before a "b" action.

Alg. 4.2 presents the β-SDP, a more precise and efficient SDP than α-SDP. Compared to Alg. 4.1, there are two differences.

1. The β_k abstraction replaces the α_k abstraction on line 3.

2. To determine the concreteness of $w_{\min}$ with respect to a PDS $\mathcal{P}_i$, the unused actions of $\mathcal{P}_i$ (i.e., $\mathsf{Act} \setminus \mathsf{Lab}_i$), must be projected out of the word $w_{\min}$ (line 6). (Projection is explained next.)

Algorithm 4.2: The β-SDP. (w_{min} is a minimal-length word in $\mathcal{A}_\cap$.)

```
   Input 1: $\Pi = (\mathcal{P}_1, \ldots, \mathcal{P}_n, \mathsf{Lab})$, a CPDS
   Input 2: $\mathsf{G} = (C_1, \ldots, C_n)$, a regular set of configurations
   output : Whether G is reachable in $\Pi$.
 1 k=1
 2 while true do
 3     $\underline{\mathcal{A}_\cap = \bigcap_{i=1}^{n} \beta_k(L_i)}$
 4     if $\mathcal{A}_\cap = \emptyset$ then
 5         return no
 6     else if $\underline{max(|\pi_1(w_{min})|, \ldots, |\pi_n(w_{min})|) < k}$ then
 7         return yes
 8     else
 9         k=k+1
10 return maybe
```

Because the β_k abstraction performs the shuffle *after* approximating L_i, the length of a word $w \in \mathcal{A}_\cap$ is no longer sufficient to determine if w is concrete. That is, for a word $w \in \mathcal{A}_\cap$ and PDS $\mathcal{P}_i$, the actions that are not executed by $\mathcal{P}_i$ and that were introduced in $\beta_k(L_i)$ because of the shuffle operation must be projected out (line 6). Let w be the sequence of actions $a_1 \cdots a_{|w|}$; then for each i, $1 \leqslant i \leqslant n$, $\pi_i(w) = \pi_i(a_1) \cdots \pi_i(a_{|w|})$, where the mapping π_i is defined as follows:

$$\pi_i(a) = \begin{cases} a & \text{if } a \in \mathsf{Lab}_i \\ \tau & \text{if } a \in \mathsf{Lab} \setminus \mathsf{Lab}_i. \end{cases}$$

If the length of the projection is less than k (i.e., $|\pi_i(w)| < k$), then this corresponds to a concrete execution for process $\mathcal{P}_i$.

Example 4.10. Let $\Pi_3 = (\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3, \mathsf{Lab})$, where $\mathsf{Lab} = \{\tau, a, b, c, d\}$, $\mathsf{Lab}_1 = \{\tau, b, d\}$, $\mathsf{Lab}_2 = \{\tau, a, c\}$, and $\mathsf{Lab}_3 = \{\tau, a, b, c, d\}$. Assume that at $k = 3$, a minimal-length word $w = abcdb \in \mathcal{A}_\cap$ has been found. Then $\pi_1(w) = bdb$, $\pi_2(w) = ac$, and $\pi_3(w) = abcdb$. Because w is abstract for $k = 3$ ($\pi_1(w) \geqslant 3$ and $\pi_3(w) \geqslant 3$), the β-SDP refines the value of k to be 4. Note that for $\mathcal{P}_2$, the word $\pi_2(w) = ac$ represents a concrete execution because its length is less than 3.

Comparing α_k and β_k: The following theorem compares α-SDP and β-SDP.

Theorem 4.1. *If the α-SDP decides reachability of G in Π at an abstraction k, then the β-SDP will decide reachability of G in Π at an abstraction $k' \leqslant k$.*

Proof. If G is reachable, then the α-SDP finds a concrete word $w \in \mathsf{Lang}(\Pi, G)$ at abstraction k, and $|w| = k-1$. (The length of w must be $k-1$ because if it were less than $k-1$, then α-SDP would have found w an a bound $k' < k$.) Because of over-approximation, if $w \in \mathsf{Lang}(\Pi, G)$, then w will be in $\bigcap_{i=1}^{n} \beta_k(L_i)$, and thus the β-SDP will find w at abstraction k (or another concrete word $w' \in \mathsf{Lang}(\Pi, G)$ such that $|w'| = k-1$). However, because of the increased precision, i.e., $\beta_k(L_i) \subseteq \alpha_k(L_i)$, the β-SDP can find a concrete word w' at an abstraction $k' < k$.

Otherwise, assume the α-SDP proves that $\mathsf{Lang}(\Pi, G) = \emptyset$ at abstraction k. For $1 \leqslant i \leqslant n$, $\beta_k(L_i) \subseteq \alpha_k(L_i)$. Hence, if $\bigcap_{i=1}^{n} \alpha_k(L_i) = \emptyset$, then $\bigcap_{i=1}^{n} \beta_k(L_i) = \emptyset$. Because $\beta_k(L_i) \subseteq \alpha_k(L_i)$, it is possible for the β-SDP to prove emptiness at an abstraction $k' < k$.

□

4.5 Abstraction-Refinement-Policy Extensions

This section presents two abstraction-refinement heuristics, and concludes by showing how they are easily combined to form a third. The first heuristic attempts to be more intelligent in choosing *how much* to refine the abstraction. The second heuristic attempts to be more intelligent in determining *which* abstractions should be refined.

Algorithm 4.3: The *Multi-step* SDP. ($w_{\min}$ is a minimal-length word in $\mathcal{A}_\cap$.)

```
   Input 1: $\Pi = (\mathcal{P}_1, \ldots, \mathcal{P}_n, \mathsf{Lab})$, a CPDS
   Input 2: $G = (C_1, \ldots, C_n)$, a regular set of configurations
   output : Whether $G$ is reachable in $\Pi$.
1  $k = 1$
2  while $\mathit{true}$ do
3      $\mathcal{A}_\cap = \bigcap_{i=1}^{n} \beta_k(L_i)$
4      if $\mathcal{A}_\cap = \emptyset$ then
5          return no
6      else if $\mathit{max}(|\pi_1(w_{\mathit{min}})|, \ldots, |\pi_n(w_{\mathit{min}})|) < k$ then
7          return yes
8      else
9          $\underline{k = \pi_{\mathsf{max}} + 1}$
10 return maybe
```

Multi-Step Abstraction Refinement

The β-SDP, and also the α-SDP, uses a simple abstraction-refinement policy: increase the search depth by one (Algs. 4.1 and 4.2, line 9). The heuristic employed by *Multi-step* SDP attempts to improve on this policy by leveraging information present in the intersection result $\mathcal{A}_\cap$. Specifically, at line 9 in Alg. 4.2, the β-SDP has found that $w_{\min}$ is a minimal-length word, and that $\mathsf{max}(|\pi_1(w_{\min})|, \ldots, |\pi_n(w_{\min})|) \geqslant k$. (We will use π_{max} to denote the result of the max computation.) To determine whether or not $w_{\min}$ is concrete, a value for k equal to $\pi_{\mathsf{max}} + 1$ must be used. The *Multi-step* SDP, presented in Alg. 4.3, defines the next value of k to be $\pi_{\mathsf{max}} + 1$. The difference between β-SDP and *Multi-step* SDP is highlighted, using underlining, on line 9 of Alg. 4.3.

The benefit of the *Multi-step* SDP is that CPDSMC is able to converge more quickly on a value for k that results in finding a counterexample (i.e., a concrete word), and avoids useless work in doing so. This is illustrated in Ex. 4.11.

Example 4.11. Revisiting Ex. 4.10 where $w = abcdb$, the *Multi-step* SDP determines that $\pi_{\mathsf{max}} = 5$ because $\pi_3(w) = abcdb$. Thus, the next abstraction uses a value of 6 for k instead of 4, and the model checker avoids analysis for $k = 4$ and $k = 5$.

Individual Abstraction Refinement

The *Multi-step* SDP addresses the fact that the β-SDP is naïve in determining what the next value of k should be. We now define the *Individual* SDP, which addresses *when* a new abstraction should be computed.

Let w be a word in the intersection $\mathcal{A}_\cap$. Then for PDSs $\mathcal{P}_i$ and $\mathcal{P}_j$, $i \neq j$, it need not be the case that $|\pi_i(w)| = |\pi_j(w)|$ because Lab_i and Lab_j can be different. Due to this difference, if the shortest word w is an abstract word then it is not necessarily an abstract word for *both* $\mathcal{P}_i$ and $\mathcal{P}_j$. That is, if w has the global property of being abstract, it may in fact be (locally) concrete for some subset of the PDSs. If w is concrete for $\mathcal{P}_i$ (i.e., $|\pi_i(w)| < k$), then the approximation $\mathcal{A}_i$ for the language $\mathsf{Lang}(\mathcal{P}_i, C_i)$ need not be refined. The *Individual* SDP, presented in Alg. 4.4, leverages the possibility of local concreteness by applying an *individual*-refinement heuristic.

The three differences between β-SDP and *Individual* SDP are highlighted, using underlining, in Alg. 4.4. First, *Individual* SDP uses n local abstraction levels $k_1, \ldots, k_n$ (line 1). Second, for the word $w_{\min}$ to be concrete, the length of the projected word $\pi_i(w_{\min})$ must be less than k_i, for $1 \leqslant i \leqslant n$ (line 6). Third, if $w_{\min}$ is an abstract word, then each k_i is incremented only as needed (lines 9–11). These extensions allow the *Individual* SDP to focus its refinement efforts on only those PDSs that require it.

Example 4.12. Given Π_3 from Ex. 4.10, assume that $k_1 = k_2 = k_3 = 3$ and that $w = abcdb$ is the same minimal-length word found. Because $\pi_2(w) = ac$ and $|ac| < k_2$, k_2 is not refined for the next round of approximation by *Individual* SDP. Similarly, because $|\pi_1(w)| \geqslant k_1$ and $|\pi_3(w)| \geqslant k_3$, the values for k_1 and k_3 are incremented for the next round. Thus, the next abstraction uses the k_i values $(4, 3, 4)$.

Algorithm 4.4: The *Individual* SDP. ($w_{\min}$ is a minimal-length word in $\mathcal{A}_\cap$.)

```
   Input 1: $\Pi = (\mathcal{P}_1, \ldots, \mathcal{P}_n, \mathsf{Lab})$, a CPDS
   Input 2: $G = (C_1, \ldots, C_n)$, a regular set of configurations
   output : Whether $G$ is reachable in $\Pi$.
1  $\underline{k_1, \ldots, k_n = 1}$
2  while true do
3      $\mathcal{A}_\cap = \bigcap_{i=1}^{n} \beta_{k_i}(L_i)$
4      if $\mathcal{A}_\cap = \emptyset$ then
5          return no
6      else if $\underline{\bigwedge_{i=1}^{n} |\pi_i(w_{min})| < k_i}$ then
7          return yes
8      else
9          for $i = 1 \ldots n$ do
10             if $\pi_i(w) \geqslant k_i$ then
11                 $\underline{k_i = k_i + 1}$
12
13 return maybe
```

There are two benefits to performing abstraction refinement at the level of an individual PDS. First, it is possible that a counterexample can be found using a very coarse approximation for some of the PDSs in Π. This permits the model checker to avoid unnecessarily computing more precise over-approximations (i.e., the β_ks) for such PDSs. Second, before performing the shuffle operation defined by β_{k_i}, the language $\alpha_{k_i}\big(\mathsf{Lang}(\mathcal{P}_i, C_i)\big)$ is represented by a finite set S. Because the size of S can be exponential in the abstraction level k_i, when the model checker is able to use a coarser abstraction for $\mathcal{P}_i$, it does so using smaller sets, and thus using less memory.

Individual Multi-Step Abstraction Refinement

The logical next step takes advantage of these improvements at the same time. To combine the two new abstraction-refinement heuristics, only line 11 in Alg. 4.4 needs to be modified. The change is to incorporate the multi-step heuristic by

Algorithm 4.5: The *Individual Multi-step* SDP. ($w_{\min}$ is a minimal-length word in $\mathcal{A}_\cap$.)

```
Input 1: $\Pi = (\mathcal{P}_1, \dots, \mathcal{P}_n, \mathsf{Lab})$, a CPDS
Input 2: $G = (C_1, \dots, C_n)$, a regular set of configurations
output : Whether $G$ is reachable in $\Pi$.
$k_1, \dots, k_n = 1$
while true do
    $\mathcal{A}_\cap = \bigcap_{i=1}^{n} \beta_{k_i}(L_i)$
    if $\mathcal{A}_\cap = \emptyset$ then
        return no
    else if $\bigwedge_{i=1}^{n} |\pi_i(w_{min})| < k_i$ then
        return yes
    else
        for $i = 1 \dots n$ do
            if $\pi_i(w) \geqslant k_i$ then
                $k_i = \pi_i(w) + 1$

return maybe;
```

defining the next value of k_i to be $|\pi_i(w)| + 1$ instead of $k_i + 1$. By doing so, the *Individual Multi-step* SDP, presented in Alg. 4.5, is able to take advantage of the time and space savings provided by the new abstraction-refinement policies. Ex. 4.13 gives a comparison of the four SDPs.

Example 4.13. Revisiting Ex. 4.12 with $k_1 = k_2 = k_3 = 3$ and $w = \mathtt{abcdb}$, the *Individual Multi-step* SDP would refine k_1 and k_3; however, it is able to make a better choice for the next value of k_3 because $|\pi_3(w)| = 5$. The *Individual Multi-step* SDP would choose the next abstraction to have values $(4, 3, 6)$.

To summarize, the refinement decisions of the four SDPs are as follows:

β-SDP	*Multi-step* SDP	*Individual* SDP	*Individual Multi-step* SDP
$k = 4$	$k = 6$	$(k_1, k_2, k_3) = (4, 3, 4)$	$(k_1, k_2, k_3) = (4, 3, 6)$

4.6 Case Study: A Bluetooth Driver

This section presents a case study of CPDS model checking. The purpose of the case study is to compare CPDS model checking using the SDPs that were presented in this chapter. In addition, we illustrate how a concurrent shared-memory program is modeled as a CPDS.

Background

The programs that are discussed are several versions of a buggy Windows Bluetooth driver. The original bug was found by Qadeer and Wu (2004). The program shown in Listing 4.1 is the model of an actual Windows Bluetooth driver written in a Java-like modeling language called Zing (Qadeer et al., 2004).[4]

For the code shown in Listing 4.1 (and Listing 4.2), starting a process is modeled by an asynchronous method call, and thus the program contains two processes $\mathsf{T}_{\mathsf{Add}}$ and $\mathsf{T}_{\mathsf{Stop}}$ that execute the methods `Add` and `Stop`, respectively. A valid instantiation of either program consists of a finite number of Add processes and one Stop process (Listing 4.1 and Listing 4.2 show instantiations with one Add and one Stop process, respectively.) Processes communicate via the shared variables shown at the top. The first three variables are Boolean flags that have the following meanings:

- The flag `stopped` indicates whether the driver has stopped. It is initialized to `false` in the `main` method, and set to `true` when all of the processes have completed execution.
- The flag `driverStoppingFlag` indicates whether the driver has received a request to stop execution. It is initialized to `false`, and set to `true` when a stop request has been issued (modeled by the `Stop` method).

[4] The programs shown in Listing 4.1 and Listing 4.2 have been simplified to fit on one page. The actual Zing code can be found on the Bluetooth driver web page (Kidd, 2009).

Listing 4.1: Orig. Bluetooth model

```
bool stopped = false;
bool driverStoppingFlag = false;
bool stoppingEvent = false;
int  pendingIo = 1;


// Models the single "stop" request
void Stop() {
    driverStoppingFlag = true;
    IoDecrement();
    WaitForStoppingEvent();
    stopped = true;
}

// Models a single "user" request
void Add() {
    bool status = IoIncrement();
    if (status) {
        // do work here
        assert(!stopped);
    }
    IoDecrement();
}

bool IoIncrement() {
    if (driverStoppingFlag == true) {
        return false;
    }
    else {
        // BUG: Context switch here allows
        //      stopped to be set to true
        InterlockedIncrementPendingIo();
        return true;
    }
}

void IoDecrement() {
    int val = InterlockedDecrementPendingIo();
    if (val == 0) { stoppingEvent = true; }
}

atomic int InterlockedIncrementPendingIo() {
    pendingIo = pendingIo + 1;
    return pendingIo;
}

atomic int InterlockedDecrementPendingIo() {
    pendingIo = pendingIo - 1;
    return pendingIo;
}

void WaitForStoppingEvent() {
    select { wait (stoppingEvent) -> ; }
}

void main() {
    async { Add(); }
    async { Stop(); }
}
```

Listing 4.2: Rev. Bluetooth model

```
bool stopped = false;
bool driverStoppingFlag = false;
bool stoppingEvent = false;
int  pendingIo = 1;


// Models the single "stop" request
void Stop () {
    driverStoppingFlag = true;
    IoDecrement();
    WaitForStoppingEvent();
    stopped = true;
}

// Models a single "user" request
void Add () {
    bool status = IoIncrement();
    if (status) {
        // do work here
        assert(!stopped);
    }
    IoDecrement();
}

bool IoIncrement() {
    int val = InterlockedIncrementPendingIo();
    if(driverStoppingFlag == true) {
        // BUG: Decrement here causes a
        // double decrement by Add
        IoDecrement();
        return false;
    }
    else
        return true;
}

void IoDecrement() {
    int val = InterlockedDecrementPendingIo();
    if(v == 0) { stoppingEvent = true; }
}

atomic int InterlockedIncrementPendingIo() {
    pendingIo = pendingIo + 1;
    return pendingIo;
}

atomic int InterlockedDecrementPendingIo() {
    pendingIo = pendingIo - 1;
    return pendingIo;
}

void WaitForStoppingEvent() {
    select { wait (stoppingEvent) -> ; }
}

void main(){
    async { Add(); }
    async { Stop(); }
}
```

- The flag `stoppingEvent` is set to `true` when all user requests have completed execution. The Stop process discussed below waits for the flag `stoppingEvent` to be set to `true` before stopping the driver, i.e., before releasing the shared resources.

The fourth variable, `pendingIo`, is an integer counter that tracks the number of active processes. It is initialized to 1 to model that the Bluetooth driver has been loaded and has begun executing in the Windows kernel.

The interleaved execution found by Qadeer and Wu (2004) that leads to an assertion failure (line 20 of Listing 4.1) is as follows:

- T_{Add} begins execution and calls `IoIncrement`. The flag `driverStoppingFlag` is false, so it proceeds down the `else` branch. At this point it is interrupted on line 31.

- T_{Stop} begins execution and executes the method `Stop` to completion. First, `driverStoppingFlag` is set to `true`. Second, `IoDecrement` is called. Third, `IoDecrement` calls `InterlockedDecrementPendingIo`, which decrements `pendingIo` and returns `0`. Fourth, because `0` was returned, `stoppingEvent` is set to `true`. Fifth, `WaitForStoppingEvent` is called and it immediately returns because `stoppingEvent` is `true`. Finally, `stopped` is set to true.

- T_{Add} resumes execution on line 31, which increments `pendingIo` by executing `InterlockedIncrementPendingIo`. The method `IoIncrement` returns true, and then T_{Add} proceeds down the `true` branch. At this point, `stopped` is `true` and thus the assertion on line 20 fails.

Briefly, the model-checking approach used by Qadeer and Wu (2004) is as follows: from a program with two threads T_1 and T_2, they define a single-threaded program T that (i) begins executing T_1, (ii) non-deterministically performs a context switch to execute T_2, and (iii) non-deterministically switches back to T_1. By bounding the number of context switches, sometimes referred to as *context-bounded model checking* (Qadeer and Rehof, 2005; Lal et al., 2008; Lal and Reps, 2008), they maintain decidability. In this case, because the bound is 2, the melding of T_1 and T_2 to define T can be obtained by inserting a non-deterministic call to the `main` procedure of T_2 before every statement defined by T_1. Similarly, before every statement defined by T_2, a non-deterministic `return` to T_1 is inserted. Finally, the non-deterministic choice is guarded to ensure that a call and a return can each occur once. Once T has been defined, a standard PDS reachability query on the PDS $\mathcal{P}$ that models T can be used to perform 2-context-bounded model checking.

Every 2-context-bounded model checking problem can be encoded as a 2-PDS CPDS using a prefix bound $k = 3$. The first communicating action passes the global state from PDS $\mathcal{P}_1$, i.e., the control location of $\mathcal{P}_1$, to PDS $\mathcal{P}_2$. The second communicating action returns the updated global state from $\mathcal{P}_2$ to $\mathcal{P}_1$. Hence, the set Act of communicating actions will be exactly the set of control locations of the PDS $\mathcal{P}$ that models the melded thread T. Finally, a prefix bound of 3 ensures precision for two communicating actions.

Context-bounded model checking as defined by Qadeer and Wu (2004) cannot verify a property of a program because it only explores executions with at most 2-context switches, i.e., it uses under-approximation. In a discussion with Qadeer, we pointed out that CPDS model checking can, in some cases, verify a property because CPDS model checking uses (a sequence of) over-approximations. Qadeer set forth the challenge to verify that the bug discovered by Qadeer and Wu (2004) had been corrected in the updated Bluetooth driver program shown in Listing 4.2. In an attempt to do so, we discovered that the program shown in Listing 4.2 actually contained a bug. The bug, and a proposed fix, were reported in Chaki et al. (2006).

The New Bug

The program shown in Listing 4.2 is Qadeer's revised version of the program from Listing 4.1. The only difference is the implementation of the `IoIncrement` method on lines 25–34. (The main differences are underlined in Listing 4.2.) In an attempt to correct the bug discussed above, `IoIncrement` was modified to first increment the counter `pendingIo`, which eliminates the buggy interleaving discussed above.

The program shown in Listing 4.2 has the unspecified invariant that the value of `pendingIo` should be equal to the number of executing processes. The new bug is that certain thread interleavings cause the invariant to be violated. The source of the bug is in the method `IoIncrement`. As the name implies, `IoIncrement` should always increment `pendingIo` by one. However, if `driverStopingFlag` is `true`, then `IoDecrement` is called. The net effect is that the counter is not incremented. When the `Add` method invokes `IoIncrement`, and `driverStoppingFlag` is `true`, it will perform a *double decrement* on `pendingIo`. As just described, in this scenario, the method `IoIncrement` does not actually increment the counter. Thus, the call to `IoDecrement` on line 17 performs a *second* decrement of `pendingIo`.

For an instantiation with only two threads, the bug does not lead to an assertion failure on line 20. For the assertion to fail, there must be at least two Add processes. For an instantiation with two Add processes, $\mathsf{T}^1_{\mathsf{Add}}$ and $\mathsf{T}^2_{\mathsf{Add}}$, and one Stop process $\mathsf{T}_{\mathsf{Stop}}$, the interleaved execution that causes an assertion failure is as follows:

1. $\mathsf{T}^1_{\mathsf{Add}}$ executes lines 12-14, and is then paused before checking the assertion on line 15.

2. $\mathsf{T}_{\mathsf{Stop}}$ executes lines 5–7, which leaves it stuck waiting for the `stoppingEvent` flag to be set to `true`.

3. $\mathsf{T}^2_{\mathsf{Add}}$ executes to completion. In doing so, it performs the double decrement to `pendingIo`, which results in `pendingIo` having the value `0`. The second call to `IoDecrement` sets `stoppingEvent` to `true` because `pendingIo` is `0`.

4. $\mathsf{T}_{\mathsf{Stop}}$ is awakened, and sets `stopped` to `true`.

5. $\mathsf{T}^{1}_{\mathsf{Add}}$ resumes execution. At this point, `stopped` is `true`, which causes the `assert` statement on line 15 to fail.

It is necessary for there to be more than one Add process because the assertion failure requires one Add process to perform the double decrement, which allows the Stop process to set `stopped` to `true`, and a second Add process to then execute (and fail) the assertion check on line 20. We have used CPDSMC to establish that an instantiation with just a single Add process could perform the double decrement, but not cause the assertion to fail.

The CPDS Model

In general, a shared-memory concurrent program modeled as a CPDS consists of two types of PDSs, *state*-PDSs and *process*-PDSs. A state-PDS models a portion of the shared state. For the program in Listing 4.2, there are four shared variables, and hence the CPDS model contains four state-PDSs. A process-PDS models an actual program process. For the program shown in Listing 4.2, process-PDSs model the Add and Stop processes. We will use $\mathsf{S}_{\mathsf{Procs}}$ to denote the finite set of processes, and x to denote the name of a process in $\mathsf{S}_{\mathsf{Procs}}$.

State-PDSs can be further characterized according to the type of shared variable that they model. For a Boolean variable, such as `stopped`, a state-PDS implements a finite-state machine, and is formally defined below.

Definition 4.14. Let b be a Boolean variable. The PDS that models b is defined as $\mathcal{P}_b = (\mathsf{P}_b, \Gamma_b, \mathsf{Lab}_b, \Delta_b, c^b_0)$, where $\mathsf{P}_b = \{p\}$ is a single control location; $\Gamma_b = \{t, f\}$ consists of stack symbols t and f that represent the variable holding `true` and `false`, respectively; $\mathsf{Lab}_b = \{x.\mathsf{set}, x.\mathsf{unset}, x.\mathsf{is\text{-}set}, x.\mathsf{not\text{-}set} \mid x \in \mathsf{S}_{\mathsf{Procs}}\}$ consists of actions to set b, unset b, and query whether b is set or not (is-set not-set, respectively); c^b_0 is $\langle p, t\rangle$ or $\langle p, f\rangle$ depending on whether b is initialized to true or false; and Δ_b is the set consisting of the following rules:

$$\begin{array}{llll}
\langle p, t\rangle \xhookrightarrow{\mathsf{x.unset}} & \langle p, f\rangle, & \langle p, f\rangle \xhookrightarrow{\mathsf{x.unset}} & \langle p, f\rangle, \\
\langle p, f\rangle \xhookrightarrow{\mathsf{x.set}} & \langle p, t\rangle, & \langle p, t\rangle \xhookrightarrow{\mathsf{x.set}} & \langle p, t\rangle, \\
\langle p, t\rangle \xhookrightarrow{\mathsf{x.is\text{-}set}} & \langle p, t\rangle, & \langle p, f\rangle \xhookrightarrow{\mathsf{x.not\text{-}set}} & \langle p, f\rangle,
\end{array}$$

where each rule is instantiated for each process-PDS x. Note that there are no rules of the form $\langle p, t\rangle \xhookrightarrow{\mathsf{x.not\text{-}set}} \langle\langle\rangle\rangle p, \gamma$ and $\langle p, f\rangle \xhookrightarrow{\mathsf{x.is\text{-}set}} \langle\langle p, \gamma\rangle\rangle$.

The set of rules Δ_b defined above models the transitions that can be applied to the state of a Boolean flag. For example, from the configuration $\langle p, t\rangle$, which denotes the Boolean flag b having the value `true`, it can be set to false by a process-PDS x by synchronizing on an $\mathsf{x.unset}$ action. Each rule is templated on a process x to distinguish which process invokes the action. Because access to a shared variable is modeled by pair-wise synchronization between a state-PDS and a process-PDS, the actions used must be specific to the individual process-PDSs. Hence, the state-PDSs template their action sets on the names of the program's processes (e.g., Add for the Bluetooth program).

A state-PDS can model an integer in a very restricted fashion. That is, for an integer i, the only operations that are allowed are (i) increments and decrements to i by 1 (or any fixed constant), and (ii) tests of whether or not i is zero. These are precisely the operations that are needed to model an integer that implements a counter c. The state-PDS that models a counter is defined by Defn. 4.15.

Definition 4.15. Let c be a counter. The PDS that models c is defined as $\mathcal{P}_c = (P_c, \Gamma_c, \mathsf{Lab}_c, \Delta_c, c_0^c)$, where $P_c = \{p\}$ is a single control location; $\Gamma_c = \{1, 0\}$ is a binary alphabet that is used to count in unary; $\mathsf{Lab}_c = \{\mathsf{x.inc}, \mathsf{x.dec}, \mathsf{x.is\text{-}zero}, \mathsf{x.not\text{-}zero} \mid x \in S_{\mathsf{Procs}}\}$ consists of actions to increment (inc) and decrement (dec) the counter, and to test whether or not the counter is zero (is-zero and not-zero, respectively); $c_0^c = \langle p, 1^k\ 0\rangle$ models c being initialized to k by placing k occurrences of the symbol "1" on the stack; and Δ_c is the set consisting of the following rules:

$$\begin{array}{llllll}
\langle p, 0\rangle & \xhookrightarrow{\mathsf{x.inc}} & \langle p, 1\,0\rangle, & \langle p, 1\rangle & \xhookrightarrow{\mathsf{x.inc}} & \langle p, 1\,1\rangle, \\
\langle p, 0\rangle & \xhookrightarrow{\mathsf{x.is\text{-}zero}} & \langle p, 0\rangle, & \langle p, 1\rangle & \xhookrightarrow{\mathsf{x.not\text{-}zero}} & \langle p, 1\rangle, \\
\langle p, 1\rangle & \xhookrightarrow{\mathsf{x.dec}} & \langle p, \epsilon\rangle, & & &
\end{array}$$

where each rule is instantiated for each process-PDS x.

In the set of rules Δ_c defined above, rules that model incrementing the counter always push a 1-symbol onto the stack. A counter that has the value zero is modeled by the top-of-stack symbol being 0. A counter that has a non-zero value is modeled by the top-of-stack symbol being 1. Specifically, a counter that has a value of k is modeled by a stack that has k occurrences of the symbol “1" at the top and a 0-symbol on the bottom. A decrement to the counter is modeled by popping a 1-symbol off the top of the stack. As for the PDS $\mathcal{P}_b$ that models a Boolean variable b, the action set Lab_c for counter c is templated on the names of the program's processes.

For a process x, there is a process-PDS $\mathcal{P}_x = (P_x, \Gamma_x, \mathsf{Lab}_x, \Delta_x, c_0^x)$. The control locations P_x encode a finite amount of local state, which enables modeling Boolean return values from called methods. The stack alphabet Γ_x, rules Δ_x, and initial configuration c_0^x are all defined in the usual way (cf. Tab. 3.1 in Ch. 3). The action set Lab_x encodes both actions that affect the shared state of a program statement, and actions that perform tests on the current valuation of the shared state. For example, the statement “`pendingIo = pendingIo + 1`” on line 35 is modeled by exchanging an $\mathsf{x.inc}$ action with the PDS $\mathcal{P}_{\mathsf{pendingIo}}$ that models the counter `pendingIo`.

Experiments

We repeated the Bluetooth experiments of Chaki et al. (2006) using the SDPs that have been presented in this chapter. The experiments consist of analyzing the Bluetooth models BT_1, BT_2 and BT_3.

	# Add Procs	β-SDP	*Multi-step* SDP	*Individual* SDP	*Individual Multi-step* SDP
BT_1	1	2.8	2.6	1.5	1.4
BT_2	2	31.2	26.1	28.8	24.7
BT_3	2	1.6	1.6	1.6	1.6
BT_3	3	5.4	3.0	5.6	3.2
BT_3	4	OOM	OOM	OOM	OOM

Table 4.1: Time in seconds to analyze the Bluetooth models using the four SDPs listed in the column headings. An "OOM" entry denotes that CPDSMC ran out of memory. For BT_{1-2}, the time reported is for CPDSMC to determine reachability, i.e., find the bug. For BT_3 with 2–3 Add processes, the time reported is for CPDSMC to determine unreachability, i.e., prove that the bug cannot occur for an instantiation with the listed number of Add processes.

BT_1 is the original buggy Bluetooth model that was first reported and presented by Qadeer and Wu (2004), and is shown in Listing 4.1.

BT_2 is the proposed fix for BT_1, and is shown in Listing 4.2. As described above, BT_2 is bug-free for an instantiation with a single Add process, but is buggy for instantiations with two or more Add processes.

BT_3 is our modification to BT_2 that removes the erroneous call to `IoDecrement` on line 23. Removing the call corrects the bug, and has been verified by CPDSMC for instantiations with two and three Add processes.

The time to analyze each model using the four SDPs defined in §4.3 and §4.4 is given in Tab. 4.1. For the reported times, the model BT_1 consisted of a single Add process, while models BT_2 and BT_3 used a two-Add-process instantiation. Note that analysis times are not reported for α-SDP because α-SDP exhausted all resources when analyzing each of the models BT_{1-3}.[5] Overall, each refinement heuristic alone, and their combination, provided a performance boost

[5] This is not a contradiction with the experimental evaluation by Chaki et al. (2006) because CPDSMC was already using β-SDP. Although β-SDP had been unpublished until now, the fact that α-SDP could not analyze the models led to the development of β-SDP.

over β-SDP for analyzing the Bluetooth models. The best performance was obtained by using *Individual Multi-step* SDP (see the right most column in Tab. 4.1). Comparing *Multi-step* SDP to *Individual* SDP, analysis was faster for BT_2 and BT_3 when using *Multi-step* SDP, and for BT_1, *Individual* SDP provided a larger performance gain.

The four-Add-process instantiation of BT_3 exhausted the available memory resources. The issue is that every Bluetooth process can both query and update the state of the counter `pendingIo`, and the PDS $\mathcal{P}_{\text{pendingIo}}$ allows for any combination of queries and updates. For example, the language of $\mathcal{P}_{\text{pendingIo}}$ allows for the Stop process to increment and decrement the counter `pendingIo` any number of times, which is an over-approximation of the Stop process's actual behavior (the Stop process only decrements `pendingIo` once, and never increments `pendingIo`). The net effect is that the language of $\mathcal{P}_{\text{pendingIo}}$ grows exponentially, which causes it to exhaust the available memory. Invalid behaviors, such as the ones present in $\mathsf{Lang}(\mathcal{P}_{\text{pendingIo}})$, could be removed. To do so would require, after each approximation-round i, refining the language $\mathsf{Lang}(\mathcal{P}_{\text{pendingIo}})$ by taking the intersection of $\mathsf{Lang}(\mathcal{P}_{\text{pendingIo}})$ and $\mathsf{Lang}(\mathcal{A}^i_\cap)$, where $\mathcal{A}^i_\cap$ is the intersection of the computed prefix languages for bound i. Intersection would remove words from $\mathsf{Lang}(\mathcal{P}_{\text{pendingIo}})$ that could not occur in an actual interleaved execution (e.g., a word consisting of actions that model increments to `pendingIo` by the Stop process). In essence, for a PDS $\mathcal{P}$, intersecting $\mathsf{Lang}(\mathcal{A}^i_\cap)$ with $\mathsf{Lang}(\mathcal{P})$ before computing the next-more-precise over-approximation $\beta_j(\mathcal{P})$, $j > i$, transfers the "global knowledge" present in $\mathcal{A}^i_\cap$—an over-approximation of the language $\mathsf{Lang}(\Pi, G)$—to $\mathcal{P}$. We have not (yet) explored this possibility.

Considering the similarities of the models for BT_2 and BT_3, it may seem odd that the analysis time for BT_3 is much faster than for BT_2 (for all SDPs listed). This is due to the fact that the analysis of BT_3 requires one less round of approximation to prove that a two-Add-process instantiation is correct, which can be see in Tab. 4.2. Tab. 4.2 shows the sequence of refinements used by the *Individual Multi-step* SDP to analyze each model. An underlined k_i value

BT_1	BT_2	BT_3
(Stop, Add $\mid$ pIo, sF, sE)	(Stop, Add_1, Add_2 $\mid$ pIo,sF,sE)	(Stop, Add_1, Add_2 $\mid$ pIo,sF,sE)
$(2, 2 \mid 2, 2, 2)$	$(2, 2, 2 \mid 2, 2, 2)$	$(2, 2, 2 \mid 2, 2, 2)$
$(\underline{4}, 2 \mid \underline{4}, \underline{4}, \underline{4})$	$(\underline{4}, \underline{4}, \underline{4} \mid \underline{4}, \underline{4}, 2)$	$(\underline{4}, \underline{4}, \underline{4} \mid \underline{4}, \underline{4}, 2)$
$(4, \underline{5} \mid 4, \underline{5}, 4)$	$(\underline{7}, 4, \underline{7} \mid \underline{7}, 4, \underline{7})$	$(\underline{7}, 4, \underline{7} \mid \underline{7}, 4, \underline{7})$
$(4, 5 \mid 4, \underline{6}, 4)$	$(7, 4, 7 \mid \underline{9}, 4, 7)$	

Table 4.2: *Individual Multi-step* SDP's refinement steps for analyzing the Bluetooth models. Bluetooth models BT_2 and BT_3 are instantiated with two Add processes and one Stop process. Each table entry is the k_i-tuple used during an analysis round. The column header gives the component of the Bluetooth model that PDS $\mathcal{P}_i$ models. The vertical bar "|" separates process-PDSs from state-PDSs. Underlined entries mark the k_i values that were updated for the between approximation rounds.

indicates a refinement step for the PDS $\mathcal{P}_i$ of the CPDS. The column headers list the components of the Bluetooth model that are modeled by each PDS $\mathcal{P}_i$, where the abbreviations are: "pIo" for `pendingIo`, "sF" for `StoppingFlag`, and "sE" for `stoppingEvent`.[6] We can see that for BT_3, the *Individual Multi-step* SDP required only three rounds of approximation.

4.7 Summary

In this chapter, we defined *communicating pushdown systems*, and presented multiple semi-decision procedures for attempting to answer a reachability query. We applied CPDS model checking to find a known bug in a Windows Bluetooth driver (BT_1), first reported by Qadeer and Wu (2004). When attempting to verify a proposed fix, analysis of the model BT_2 using a two-Add-process instantiation showed that the proposed fix did not actually correct the bug. We manually corrected the fix to produce the model BT_3, and proved that the assertion failure

[6] Because the flag `stopped` is only modified by the Stop process, we manually combined the PDS $\mathcal{P}_{\text{stopped}}$ with the PDS $\mathcal{P}_{\text{Stop}}$.

cannot occur for two- and three-Add-process instantiations of BT_3.

5 EMPIRE: MODEL EXTRACTION AND ANALYSIS

This chapter presents Empire, a tool to verify that all executions of a concurrent Java program are AS-serializable. Empire takes a demand-driven approach to the verification problem. Instead of verifying AS-serializability of executions for all of a program's objects, the tool focuses on objects that are allocated at a specified allocation site. For Java, an allocation site is a program location associated with a `new` statement. A demand-driven approach reduces the size of generated models, and enables parallelization because many instances of Empire can be used in parallel to verify the program proper.

The input to Empire is a concurrent Java program Prog and an allocation site $\dot{\chi}$ for a class T in Prog.[1] There are two conventions for defining the atomic sets of the class T.

1. Assume all fields defined by class T form one atomic set. This is the approach taken by Hammer et al. (2008) for performing dynamic AS-serializable violation detection.

2. The Empire user specifies the atomic sets of class T.

Similarly, the units of work for class T can be specified by the user, or Empire can use the default assumption that all public methods defined by T are units of work. In addition, a unit-of-work method for a class T' that has a $\mathsf{unitfor}$ parameter of type T is a unit-of-work method for class T. The remainder of this chapter merely assumes that the atomic sets and the unit-of-work methods of T have been specified in some manner.

Empire's front-end uses abstraction to generate a program in Empire's intermediate modeling language EML. An EML program consists of a finite set of processes that read from and write to a finite set of shared-memory locations, and that synchronize on a finite set of locks. From one EML program, Empire

[1] The symbol χ is used to denote an arbitrary allocation site. The dotted version, $\dot{\chi}$, denotes the allocation site specified as input to Empire.

generates multiple CPDSs to pass as input to the CPDS model checker Cpdsmc. The translation from an EML program to a CPDS follows a strategy similar to that used in the Bluetooth case study presented in §4.6 of Ch. 4. Each EML process is modeled as a PDS, and each lock, which constitutes shared state, is also represented by its own PDS. The one complication in compiling an EML program into a CPDS is how to detect an AS-serializability violation. We accomplish this by including a *violation-monitor* PDS that checks one of the fourteen problematic scenarios.[2]

In each of the generated CPDSs, the target set of configurations is reachable if and only if there exists an interleaved execution of the EML program that causes the violation monitor to reach the error state. Thus, a positive reachability result from Cpdsmc for one query means that there is an AS-serializability violation in the EML program. Because of over-approximation, the Java program may or may not have an actual violation. A negative reachability result from Cpdsmc for all queries means that all executions of the EML program have been verified to be AS-serializable, which proves that for all objects allocated at the specified allocation site, all executions of the Java program are AS-serializable.

The choice to model a concurrent Java program as a CPDS has direct implications on program abstraction. A Java program, which has dynamic memory and thread allocation, must be abstracted into an EML program with a finite number of threads and a finite number of objects: each entity—thread or object—is modeled as a PDS, and a CPDS has only a finite number of PDSs. The main technical challenge is to design a finite-entity program abstraction that (i) is a sound overapproximation of the set of program behaviors, and (ii) retains precision about the locking behavior of a program. We address the challenge by defining the *random-isolation abstraction*.

Remark 5.1. A second implication is that translating an EML program into a CPDS tightly couples the EML processes. By tight coupling, we refer to the

[2] To remind the reader, Tab. 2.3 on page 24 lists all of the fourteen problematic access patterns.

fact that each PDS of a CPDS can directly observe every action of the rest of the PDSs. For the translation presented in this chapter, the coupling is in fact necessary. The reason is that the languages of the PDSs for EML locks and the violation-monitor PDS are context-free. As will be fully explained later, the context-freeness comes from the fact that EML locks, like Java locks, are reentrant. Moreover, units of work are reentrant, which causes the language of the violation-monitor PDS to also be context-free. As a teaser, the focus of Ch. 6 is a technique to remove reentrant use of locks and units of work. We then show in Ch. 7 that after reentrancy has been removed, the problem is in fact decidable! But we have a long way to go before we can conclude with the decidability result.

The rest of this chapter is organized as follows: §5.1 reviews AS-serializability violations; §5.2 presents a program abstraction candidate; §5.3 presents the random-isolation abstraction; §5.4 presents the implementation of the random-isolation abstraction; §5.5 presents the EMPIRE Modeling Language EML; §5.6 presents the translation from Java to EML; §5.7 presents the translation from EML to a set of CPDSs; §5.8 presents our experimental evaluation; and §5.9 presents related work.

5.1 Review of AS-serializability Violations

Listing 5.1 repeats the Java program shown in Listing 2.1, which was discussed in §2.2 and §2.3 of Ch. 2. The class `SafeWrap` from §2.3 has been included, and, to simplify the following discussion, the class `Counter`—whose fields are not involved in the AS-serializability violation—has been removed. We now review the AS-serializability violation that was discussed in §2.3 of Ch. 2.

The motivation for introducing class `SafeWrap` was that the method `Stack.pop` does not check that the `Stack` is non-empty before accessing the field `Stack.data` (line 6 of Listing 5.1). The method `SafeWrap.popwrap` is a wrapper for the method `Stack.pop`, and implements the safety check. First, on line 23, `SafeWrap.popwrap` ensures that the size of the parameter "`Stack s`" is greater than zero, which means that it is non-empty. Second, also on line 23, `Stack.pop` is invoked on the parameter "`Stack s`" if it is non-empty, and otherwise the `null` reference is returned. The method `SafeWrap.popwrap` is implemented incorrectly: the `synchronized` keyword that annotates the method performs a synchronization operation on the implicit `this` parameter (line 22), whereas a synchronization operation should have been performed on the parameter "`Stack s`". The bug allows for the interleaved execution that was discussed in §2.3, and is repeated below.

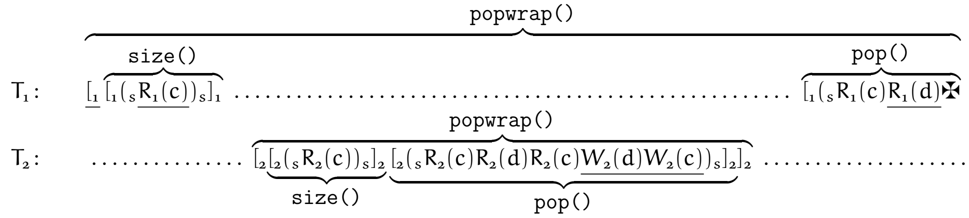

Figure 5.1: An interleaved execution of thread T_1 and T_2 that contains an AS-serializability violation. R and W denote a read and write access, respectively. `c` and `d` denote fields `count` and `data`, respectively. "[" and "]" denote the beginning and end, respectively, of a unit of work. The subscripts "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()`.

For the interleaved execution of threads T_1 and T_2 depicted in Fig. 5.1, thread T_1 begins execution first, invokes `SafeWrap.popwrap`, and then `Stack.size`, which returns 1. Thread T_1 then proceeds down the true branch of the conditional, but is preempted *before* invoking `Stack.pop`. Thread T_2 then begins execution, invoking and completing the methods `SafeWrap.popwrap`, `Stack.size`, and `Stack.pop`. Thread T_2 is able to execute these methods because it is operating on a different `SafeWrap` object than thread T_1, and thread T_1 does not hold the lock associated with the shared `Stack` object. When thread T_1 resumes execution, it invokes `Stack.pop`. The error occurs at this point because the shared object `stack` is now *empty*; however, thread T_1 is about to perform an operation that was guarded by the condition "`s.size() > 0`". Because the field

Listing 5.1: Modified Stack program.

```
class Stack {
   public static final int MAX=10;
   @atomic(S) Object[] data = new Object[MAX];
   @atomic(S) int count = -1;
   public synchronized Object pop(){
      Object res = data[count];
      data[count--] = null;
      return res;
   }
   public synchronized void push(Object o) {
      data[++count] = o;
   }
   public synchronized int size() {
      return count+1;
   }
   public synchronized replaceTop(Object o) {
      pop(); push(o);
   }
   public static Stack makeStack() { return new$_{\dot{\chi}}$ Stack(); }
}
class SafeWrap {
   public synchronized Object popwrap(Stack s) {
      return (s.size() > 0) ? s.pop() : null;
   }
   public static SafeWrap makeSafeWrap() { return new SafeWrap(); }

   public static void main(String[] args){
      Stack stack = Stack.makeStack();
      stack.push(new Integer(1));
      new Thread("1") { makeSafeWrap().popwrap(stack); }
      new Thread("2") { makeSafeWrap().popwrap(stack); }
   }
}
```

Stack.count has the value -1, the array access on line 6 that results from T_1 invoking Stack.pop throws a java.lang.ArrayOutOfBoundsException.

The interleaved execution raises an exception because it contains an AS-serializability violation. In this case, the interleaved execution contains problematic access pattern 12: "$R_1(c); W_2(d); W_2(c); R_1(d)$". The field accesses that are part of the pattern are underlined in Fig. 5.1.

Empire is a tool to verify that errors such as the one just described do not occur in a concurrent Java program Prog. Empire returns answers of the form "the problematic access pattern p is definitely not preset" or "the problematic access pattern p may be present", where p is an integer in the range one through fourteen that specifies the particular problematic access pattern of interest (cf. Tab. 2.3 on page 24 for the complete list). Empire uses abstraction to generate an abstract program $\mathsf{Prog}^\sharp$ such that the set of behaviors of $\mathsf{Prog}^\sharp$ is a *sound over-approximation* of the set of behaviors of Prog. The challenge is to define a finite-data abstraction such that $\mathsf{Prog}^\sharp$ is able to disallow certain thread interleavings by modeling the synchronization of Prog's processes.

5.2 The Allocation-Site Abstraction

A natural choice for a finite-data abstraction is the *allocation-site abstraction* (Jones and Muchnick, 1982). Given an allocation site χ for class T, let $\mathsf{Conc}(\chi)$ denote the set of all concrete objects of class T that can be allocated at χ. The allocation-site abstraction uses a single abstract object $\zeta^\sharp_\chi$ to summarize all of the concrete objects in $\mathsf{Conc}(\chi)$. When the size of $\mathsf{Conc}(\chi)$, denoted by $|\mathsf{Conc}(\chi)|$, is greater than 1, the abstract object $\zeta^\sharp_\chi$ is referred to as a *summary object*. Thus, for each field f defined by T, field $\zeta^\sharp_\chi.f$ is a summary field for the set of fields $\{\zeta.f \mid \zeta \in \mathsf{Conc}(\chi)\}$. Because the program has a finite number of program points, and each class defines a finite number of fields, this results in a finite-data abstraction.

There are five allocation sites in Listing 5.1: line 19 allocates a Stack object;

line 25 allocates a `SafeWrap` object; line 29 allocates an `Integer` object; and lines 30 and 31 allocate `Thread` objects T_1 and T_2, respectively. All allocation sites except the one on line 25 are executed exactly one time for all executions of the program shown in Listing 5.1. The allocation site on line 25 allocates two concrete objects because the method `SafeWrap.makeSafeWrap` is invoked once by thread T_1 and once by thread T_2.

The allocation-site abstraction would define the five abstract objects $\zeta^\sharp_{19}$,$\zeta^\sharp_{25}$, $\zeta^\sharp_{29}$,$\zeta^\sharp_{30}$, and $\zeta^\sharp_{31}$, where abstract object $\zeta^\sharp_i$ represents all concrete objects that could be allocated at the allocation site on line i. For the program in Listing 5.1, the abstract objects $\zeta^\sharp_{19}$,$\zeta^\sharp_{29}$, $\zeta^\sharp_{30}$, and $\zeta^\sharp_{31}$ each represent a singleton set because the program only executes the associated allocation statement once (as discussed above). However, the abstract object $\zeta^\sharp_{25}$ represents two concrete objects because the method `SafeWrap.makeSafeWrap` is invoked twice, once by threads T_1 and T_2, respectively.

For the allocation-site abstraction to be sound, an analysis generally has to perform *weak updates* on each summary object. That is, information for the summary object must be *accumulated* rather than overwritten. A *strong update* of the abstract state generally can only be performed when the analysis can prove that there is exactly one object allocated at χ, i.e., $|\mathsf{Conc}(\chi)| = 1$. For the program in Listing 5.1, strong updates could be performed on all abstract objects except $\zeta^\sharp_{25}$ because abstract object $\zeta^\sharp_{25}$ represents two concrete `SafeWrap` objects. Note that an interprocedural analysis would be required to determine that the abstract object $\zeta^\sharp_{19}$—the abstract object representing objects allocated by the `Stack.makeStack` method—represents a singleton set. The problem is that $\zeta^\sharp_{19}$ is used to allocate *all* `Stack` objects, and thus an interprocedural analysis would be required to prove that the method `Stack.makeStack` is invoked exactly one time for *all* executions of the program. Because of this difficulty, analyses that make use of the allocation-site abstraction typically resort to the assumption that every abstract object is a summary object, and do not bother with proving otherwise.

Empire is concerned with tracking reads and writes to the fields of the T objects allocated at the specified allocation site $\dot{\chi}$. In Listing 5.1, the allocation site of interest is on line 19. This is denoted by the subscripted $\mathtt{new}_{\dot{\chi}}$ statement. The allocation-site abstraction is a sound overapproximation for modeling reads and writes because a read from (write to) the abstract field $\zeta^{\sharp}_{\dot{\chi}}.\mathsf{f}$ corresponds to a possible read from (write to) $\zeta.\mathsf{f}$, for one concrete object ζ in $\mathsf{Conc}(\chi)$. For the progam shown in Listing 5.1, the reads and writes of interest are to the fields `Stack.data` and `Stack.count`. The `@atomic(S)` annotation on the two fields specifies that each field is a member of the atomic set S.

Empire must also model program synchronization. Empire accomplishes this by defining locks in the EML program that correspond to the abstract objects of $\mathsf{Prog}^{\sharp}$. There are two possibilities for defining the semantics of an EML lock. The first possibility is to interpret a lock acquire as a *strong update*, i.e., the program has definitely acquired a particular lock. This would correspond to acquiring the locks of *all possible* instances in $\mathsf{Conc}(\chi)$, which in most circumstances—including the one here—would be unsound. In the example of Listing 5.1, this interpretation of locking combined with the allocation-site abstraction would preclude the interleaved program execution that contains the bug, because the two `SafeWrap` objects would effectively get the same lock, and the two `SafeWrap.popwrap` methods would execute without interleaving.

The second possibility for defining the semantics of EML locks is to interpret lock acquire as a *weak update*, i.e., the program may have acquired a particular lock. A weak update would leave the lock in a "possibly-held" state. When an EML process Proc attempts to acquire a lock that is in the "possibly-held" state, two cases must be considered.

1. The lock is actually held by another EML process Proc' and thus Proc must block until Proc' releases the lock.

2. The lock is not held by another EML process Proc' and thus Proc may acquire the lock.

This semantics is sound because all possible cases are considered. However, the overall effect of this semantics is, in essence, equivalent to an EML program without locks because the program can never reason definitively whether or not a lock is actually held. That is, in the abstract program a thread is always able to acquire a lock, which in essence means that the abstract program operates as if there are no synchronization constraints. In general, this possibility would greatly increase the number of false positives. For instance, in the example of Listing 5.1, if we were to fix the code by adding an additional synchronization block on the parameter `Stack s` inside the body of `SafeWrap.popwrap`, the analysis would still report a bug because locking behavior was modeled imprecisely.

5.3 Random-Isolation Abstraction

Our solution was to develop a new abstraction technique, *random-isolation abstraction*, which is a novel extension of allocation-site abstraction. The random-isolation abstraction is motivated by the following observation:

Observation 5.1. *The concrete objects that can be allocated at a given allocation site* χ*,* $\mathsf{Conc}(\chi)$*, cannot be distinguished by the allocation-site abstraction.*

Obs. 5.1 states that if one chooses to isolate a *random concrete* object $\zeta_{\ddot{\chi}}$ from $\mathsf{Conc}(\dot{\chi})$, the allocation-site abstraction would not be able to distinguish the randomly-chosen concrete object from any of the other concrete objects that are represented by $\zeta^{\sharp}_{\dot{\chi}}$.[3]

The random-isolation abstraction leverages Obs. 5.1 by randomly isolating one of the concrete objects allocated at allocation site $\dot{\chi}$ and tracking it specially in the abstraction. Whereas allocation-site abstraction would use one summary object $\zeta^{\sharp}_{\dot{\chi}}$ to represent all concrete objects $\mathsf{Conc}(\dot{\chi})$ from $\dot{\chi}$, random isolation uses two objects: one summary $\zeta^{\sharp}_{\dot{\chi}}$ and one non-summary $\zeta^{\sharp}_{\ddot{\chi}}$. The object $\zeta^{\sharp}_{\ddot{\chi}}$

[3] We use the double-dotted $\zeta_{\ddot{\chi}}$ to denote a randomly-isolated object.

is a non-summary object because it alone represents the randomly-isolated concrete object $\zeta_{\ddot{\chi}}$. Because $\zeta^{\sharp}_{\ddot{\chi}}$ is a non-summary object, it is safe to perform strong updates to its (abstract) state, which gives us *Random-Isolation Principle 1*.

Random-Isolation Principle 1 (Updates). *Let $\zeta_{\ddot{\chi}} \in \mathit{Conc}(\dot{\chi})$ be a randomly-isolated concrete object. Because $\zeta_{\ddot{\chi}}$ is modeled by a special abstract object $\zeta^{\sharp}_{\ddot{\chi}}$, the random-isolation abstraction enables an analysis to perform strong updates on the state of $\zeta^{\sharp}_{\ddot{\chi}}$.*

Random isolation also provides a powerful methodology for proving properties of a program: a proof that a property ϕ holds for $\zeta^{\sharp}_{\ddot{\chi}}$ proves that ϕ holds for all $\zeta \in \mathsf{Conc}(\dot{\chi})$. Consider a concrete trace of the program in which a concrete object ζ' is allocated at a dynamic instance of $\dot{\chi}$, and ϕ does not hold for ζ'. Because of random isolation, the randomly-isolated object $\zeta_{\ddot{\chi}}$ is just as likely to be ζ' as it is to be any other concrete object. Thus, the prover must consider the case that $\zeta_{\ddot{\chi}}$ is ζ'. Because the property holds for $\zeta^{\sharp}_{\ddot{\chi}}$, and because $\zeta^{\sharp}_{\ddot{\chi}}$ represents ζ' in the trace under consideration, then the property must also hold for ζ', which is a contradiction. This gives us *Random-Isolation Principle 2*:

Random-Isolation Principle 2 (Proofs). *Given a property ϕ and site $\dot{\chi}$, a proof that ϕ holds for the randomly-isolated abstract object $\zeta^{\sharp}_{\ddot{\chi}}$ proves that ϕ holds for every object that is allocated at $\dot{\chi}$. That is, $\phi(\zeta^{\sharp}_{\ddot{\chi}}) \to (\forall_{\zeta \in \mathit{Conc}(\dot{\chi})}.\phi(\zeta))$.*

Before describing the technical details of how random isolation is implemented, we highlight the benefits of random isolation as used in Empire. A generated EML program will have an EML lock for each non-summary object. Because of random isolation, the state of the Java lock that is associated with the randomly-isolated instance $\zeta_{\ddot{\chi}}$ can be modeled precisely by the state of the special abstract object $\zeta^{\sharp}_{\ddot{\chi}}$. That is, the acquiring and releasing of the lock for $\zeta_{\ddot{\chi}}$ by a thread of execution can be modeled by a strong update on the state of $\zeta^{\sharp}_{\ddot{\chi}}$, thus allowing the analyzer to disallow certain thread interleavings when performing state-space exploration on the generated EML program.

In contrast, because sound tracking of the lock state for a summary object generally would result in the "possibly-held" state, EML programs have no locks for summary objects: their modeled behaviors are not restricted by synchronization primitives. This provides a sound, finite model of the locking behavior of $\mathsf{Prog}^{\sharp}$. (It is an over-approximation because the absence of locks on summary objects causes them to gain additional behaviors.)

5.4 Implementing Random Isolation

Random isolation is implemented via a source-to-source transformation, which we explain in the context of the example program shown in Listing 5.1. In Listing 5.1, the allocation site of interest is on line 19, which we repeat below for convenience.

$$\mathsf{public\ static\ Stack\ makeStack()\ \{\ return\ new}_{\dot{\chi}}\ \mathsf{Stack();\ \}} \tag{5.1}$$

Random isolation involves transforming the $\mathsf{new}_{\dot{\chi}}$ statement into

$$\mathsf{(rand()\ \&\&\ test\text{-}and\text{-}set(}\mathcal{F}_{\ddot{\chi}}\mathsf{))\ ?\ new}_{\ddot{\chi}}\ \mathsf{Stack()\ :\ new}_{\dot{\chi}}\ \mathsf{Stack();} \tag{5.2}$$

The site $\dot{\chi}$ from code fragment (5.1) is transformed into a conditional-allocation site, where the conditional "tests-and-sets" a newly introduced global flag $\mathcal{F}_{\ddot{\chi}}$.

Remark 5.2. The "test-and-set" must be an *atomic* operation.[4] Without the use of an atomic "test-and-set", the source-to-source transformation would introduce a race condition that would allow multiple objects to be allocated at the newly introduced allocation site $\ddot{\chi}$.

The global flag $\mathcal{F}_{\ddot{\chi}}$ serves two purposes. First, because it can be set to true only one time, it ensures that only one concrete object $\zeta_{\ddot{\chi}}$ can ever be allocated at the generated site $\ddot{\chi}$ (as for randomly-isolated objects, we use the double-dotted

[4] The phrase "test-and-set" emphasizes that random isolation is not specific to Java. For Java, the method `AtomicBoolean.compareAndSwap` is used.

version $\ddot{\chi}$ to denote the allocation site of the randomly-isolated object). That is, $\mathcal{F}_{\ddot{\chi}}$ guarantees that $|\mathsf{Conc}(\ddot{\chi})| \leqslant 1$ for all possible executions of the program; consequently, the size of the set that is the concretization of $\zeta^{\sharp}_{\ddot{\chi}}$ must also be less than or equal to one.

Second, for performing AS-serializability-violation detection, we only have to be concerned with executions in which $\zeta_{\ddot{\chi}}$ is eventually allocated (i.e., traces in which $|\mathsf{Conc}(\ddot{\chi})| = 1$). This is a consequence of the use of randomness in code fragment 5.2. If there was a trace in which $\zeta_{\ddot{\chi}}$ was not allocated but the trace did in fact contain an AS-serializability violation, then because of the use of randomness, there must exist a similar trace in which the $\zeta_{\ddot{\chi}}$ object was allocated and the violation occurred on that object. That is, the need to only consider traces in which $\zeta_{\ddot{\chi}}$ is (eventually) allocated is a corollary of *Random-Isolation Principle 2*. We can identify all such traces because the global flag $\mathcal{F}_{\ddot{\chi}}$ must eventually be set to true in each trace. Moreover, when translating an EML program into a CPDS, we can directly model the state of $\mathcal{F}_{\ddot{\chi}}$.

While the use of a source-to-source transformation is not strictly necessary, it allows existing object-sensitive analyses to be used with minimal changes. For example, let Pts be the points-to relation computed via a flow-insensitive, object-sensitive points-to analysis in the style of Milanova et al. (2005), and let CG be an object-sensitive call graph.[5] Because these two analysis artifacts are object-sensitive, their respective dataflow facts make a distinction between objects allocated at $\ddot{\chi}$ and objects allocated at $\dot{\chi}$. For example, if T defines a method T.m, then CG will contain at least two nodes for T.m: one for object context $\ddot{\chi}$, and one for object context $\dot{\chi}$. Thus, inside of the control-flow graph for T.m with object context $\ddot{\chi}$, an analysis is able to take advantage of the fact that the special Java `this` variable is referring to the non-summary object $\zeta^{\sharp}_{\ddot{\chi}}$. That is, a unique context of T.m has been created for $\zeta^{\sharp}_{\ddot{\chi}}$ without modifying the

[5] An object-sensitive call graph CG models the interprocedural control flow of a program: there is a node in CG for each method of the program for each context in which it can be invoked (Milanova et al., 2005). An object-sensitive points-to analysis associates points-to facts with the nodes of CG, thus computing different points-to facts for different object contexts of the same method.

analyses!

In some situations, however, a CG node's context is not enough to distinguish between $\zeta^{\sharp}_{\ddot{\chi}}$ and $\zeta^{\sharp}_{\dot{\chi}}$. Consider the code fragment "`synchronized(t) { t.m() }`", where `t` has been defined to be the result of code fragment (5.2), i.e., `t` is either a reference to the randomly-isolated object $\zeta_{\ddot{\chi}}$ or to another object $\zeta_{\dot{\chi}}$ allocated at site $\dot{\chi}$. Because of abstraction, the points-to set for `t` will be defined as follows: $\mathsf{Pts}(\mathtt{t}) = \{\zeta^{\sharp}_{\ddot{\chi}}, \zeta^{\sharp}_{\dot{\chi}}\}$. When entering the `synchronized` block, it is desirable to reason precisely about the state of the lock associated with $\zeta^{\sharp}_{\ddot{\chi}}$; however, because of the imprecision of points-to analysis, it is not directly possible to determine if the lock for $\zeta^{\sharp}_{\ddot{\chi}}$ is acquired or not because the abstraction cannot determine if `t` solely references $\zeta^{\sharp}_{\ddot{\chi}}$. To reason precisely about the state of the lock associated with $\zeta^{\sharp}_{\ddot{\chi}}$, we must be able to distinguish between the case when `t` references $\zeta^{\sharp}_{\ddot{\chi}}$ and when `t` references $\zeta^{\sharp}_{\dot{\chi}}$. That is, the abstract program $\mathsf{Prog}^{\sharp}$ should only acquire $\zeta^{\sharp}_{\ddot{\chi}}$ if "$\mathsf{Pts}(\mathtt{t}) = \{\zeta^{\sharp}_{\ddot{\chi}}\}$".

The case analysis is accomplished via a second source-to-source transformation, defined as follows.

```
if (is_ri(t)) {
  synchronized(t) { t.m(); }
} else {
  synchronized(t) { t.m(); }
}
```

The method "`is_ri`" returns `true` if `t` is a reference to the randomly-isolate object $\zeta_{\ddot{\chi}}$, i.e., the method `is_ri` is defined as "`return t ==` $\zeta_{\ddot{\chi}}$;".

During points-to analysis, the interpretation of the call on `is_ri` performs a case analysis on $\mathsf{Pts}(\mathtt{t})$. Specifically, the abstract interpretation of `is_ri` performs the abstract test "$\mathtt{t} == \zeta^{\sharp}_{\ddot{\chi}}$", which allows the points-to analysis to perform *assume* statements on the outgoing branches (e.g., when following the true branch of the condition, the points-to analysis performs an "`assume` $\mathsf{Pts}(\mathtt{t}) = \{\zeta^{\sharp}_{\ddot{\chi}}\}$"). One can view this as a way to achieve object-sensitivity at the level of a program block instead of just at the method level. Although the second transformation is presented in the context of AS-serializability-violation detection, it is a generic approach that can be applied wherever an analysis needs to distinguish between $\zeta^{\sharp}_{\ddot{\chi}}$ and $\zeta^{\sharp}_{\dot{\chi}}$ to perform a strong update.

Summary

After performing the source-to-source transformation to implement the random-isolation abstraction, the abstract program $\mathsf{Prog}^{\sharp}$ has been defined. $\mathsf{Prog}^{\sharp}$ operates over a set of abstract objects consisting of $\zeta^{\sharp}_{\chi}$ for each allocation site χ in the program (including the special abstract object $\zeta^{\sharp}_{\ddot{\chi}}$ for the generated allocation site $\ddot{\chi}$). Because threads in Java are objects themselves, $\mathsf{Prog}^{\sharp}$ now has a finite number of threads, where each thread is associated with an allocation site. (We can, of course, generate multiple copies of each thread as needed. That is, the number of threads is actually a parameter of $\mathsf{Prog}^{\sharp}$.) The interprocedural control flow of each thread is defined by the set of reachable nodes in the object-sensitive call graph CG from the thread's entry point. Finally, the intraprocedural control flow of each method is defined by the control flow graph (CFG) for the method, where for a method $\mathtt{m}$, $\mathsf{CFG}_{\mathtt{m}}$ consists of a set of Java statements Stmts, a successor relation $\mathsf{Succ} \subseteq \mathsf{Stmts} \times \mathsf{Stmts}$, and has distinct entry and exit statements.

5.5 Empire Modeling Language

From an abstract program $\mathsf{Prog}^{\sharp}$, Empire generates a program in the Empire Modeling Language EML.

An EML program EProg consists of (i) a finite set of shared-memory locations S_{Mem}; (ii) a finite set of reentrant locks S_{Locks}; and (iii) a finite number of concurrently executing processes S_{Procs}.

An EML shared-memory location $\mathtt{m}$ is an abstract memory location: abstract reads and writes can be made on $\mathtt{m}$; however, EML does not have a notion of a value held by $\mathtt{m}$. The lack of values stored at shared-memory locations is an artifact of EMPIRE's goal of verifying AS-serializability, which is a property of the order of interleaved reads and writes of an application, and not the values read and written.

An EML lock is reentrant, meaning that the lock can be reacquired by an EML process that currently owns the lock, and also that the lock must be released the same number of times to become free. EML restricts the acquisition and release of an EML lock to occur within the body of a function, i.e., an EML lock cannot be acquired in a function f and released in another function f'. In addition, the acquisition of multiple EML locks by an EML process must be properly nested: an EML process must release a set of held locks in the order opposite to their acquisition order. The two restrictions are naturally fulfilled when EML is used to model `synchronized` blocks and methods in a Java thread.

An EML process Proc is defined by a set of (possibly) recursive functions, one of which is designated as the `main` function of the process. An EML function f is defined by a *labeled-flow graph* $\mathcal{G}_F$. A labeled-flow graph is a tuple $\mathcal{G}_f = (\mathsf{Nodes}_f, \mathsf{Edges}_f, n_{\mathsf{entry}}, n_{\mathsf{exit}})$, where Nodes_f is a set of nodes, $\mathsf{Edges}_f \subseteq \mathsf{Nodes}_f \times \mathsf{Labels} \times \mathsf{Nodes}_f$ is a set of edges, $n_{\mathsf{entry}} \in \mathsf{Nodes}_f$ is the distinct entry node, and $n_{\mathsf{exit}} \in \mathsf{Nodes}_f$ is the distinct exit node. An edge $e = (n, a, n') \in \mathsf{Edges}_f$ denotes the flow of control from node n to node n'. Non-determinism is introduced by having multiple outgoing edges from the same node. (EML programs have only non-deterministic branches.) The label $a \in \mathsf{Labels}$ represents a semantic action that the EML process performs when transferring control from node n to node n'. EML supports the labels listed in Tab. 5.1.

An edge labeled "start Proc" starts the EML process named Proc. This is used to model the fact that when a Java program begins, only one thread is executing its `main` method, and all other threads cannot begin execution until they have been started by an already executing thread.

Labels	Semantics
call f	invoke function f
read m	read from memory location $m \in S_{\mathsf{Mem}}$
write m	write to memory location $m \in S_{\mathsf{Mem}}$
alloc l	allocate the EML lock $l \in S_{\mathsf{Locks}}$
lock l	acquire the EML lock $l \in S_{\mathsf{Locks}}$
unlock l	release the EML lock $l \in S_{\mathsf{Locks}}$
unitbegin	beginning a unit of work
unitend	ending a unit of work
start Proc	start EML process Proc
skip	a statement whose semantic action is not modeled

Table 5.1: The edge labels Labels of an EML flow graph that represents an EML function and their corresponding semantics.

5.6 EML Generation

Empire defines the EML program EProg as follows. To model the randomly-isolated abstract object $\zeta^{\sharp}_{\ddot{\chi}}$, EProg defines a shared memory location m_f for each field f of the class T, and also an EML lock $l_{\ddot{\chi}}$ to model the lock associated with $\zeta^{\sharp}_{\ddot{\chi}}$. The status of the global flag $\mathcal{F}_{\ddot{\chi}}$ is modeled by the EML lock $l_{\ddot{\chi}}$ being allocated or not. As noted in §5.4, we are only concerned with executions in which $l_{\ddot{\chi}}$ is eventually allocated. Hence we need only be concerned with traces of the EML program in which "alloc $l_{\ddot{\chi}}$" appears somewhere.

Let Threads be the set of all subclasses of `java.lang.Thread`. For each $\theta \in$ Threads, and for each allocation site χ_θ that allocates an instance of θ, EProg defines an EML process $\mathsf{Proc}_{\chi_\theta}$ that models the behavior of one instance of θ that is allocated at χ_θ. Also, EProg defines an EML process $\mathsf{Proc}_{\mathsf{main}}$ that models the Java thread that begins execution of the `main` method.

The functions of an EML process Proc correspond to the Java methods that are reachable in the object-sensitive call graph CG from the entry point (e.g., main) of the Java thread that is being modeled. For a Java method m, there is an EML function f_m. The labeled-flow graph $\mathcal{G}_{f_m}$ is defined from the control-flow

stmt	a_{stmt}	Condition
`o.m()`	call m	
`entry`	unitbegin	`this`$= \zeta^\sharp_{\ddot{\chi}}$, CFG_m is a unit of work
`exit`	unitend	`this`$= \zeta^\sharp_{\ddot{\chi}}$, CFG_m is a unit of work
`x = o.f`	read m_f	$\zeta^\sharp_{\ddot{\chi}} \in \mathsf{Pts}(o)$
`o.f = x`	write m_f	$\zeta^\sharp_{\ddot{\chi}} \in \mathsf{Pts}(o)$
`new`$_{\ddot{\chi}}$ `T`	alloc $l_{\zeta^\sharp_{\ddot{\chi}}}$	
`monitorenter o`	lock $l_{\zeta^\sharp_{\ddot{\chi}}}$	$\mathsf{Pts}(o) = \{\zeta^\sharp_{\ddot{\chi}}\}$
`monitorexit o`	unlock $l_{\zeta^\sharp_{\ddot{\chi}}}$	$\mathsf{Pts}(o) = \{\zeta^\sharp_{\ddot{\chi}}\}$
`o.start()`	start $\mathsf{Proc}_{\chi_\theta}$	$\zeta^\sharp_{\chi_\theta} \in \mathsf{Pts}(o)$.
*	skip	

Table 5.2: Java statement types for CFG_m, their corresponding EML labels, and the condition necessary to generate the EML label. The final row is a catchall for the Java statements that are not modeled in EML.

graph CFG_m that is associated with m in CG. The control-flow graph CFG_m consists of a set of Java statements Stmts, a successor relation $\mathsf{Succ} \subseteq \mathsf{Stmts} \times \mathsf{Stmts}$, and has distinct entry and exit statements. The translation from CFG_m to $\mathcal{G}_{f_m}$ is straightforward. There is a node $n_{\mathsf{stmt}} \in \mathsf{Nodes}_{f_m}$ for each $\mathsf{stmt} \in \mathsf{Stmts}$. There is a labeled edge $(n_{\mathsf{stmt}}, a_{\mathsf{stmt}}, n_{\mathsf{stmt}'})$ for each pair of statements $(\mathsf{stmt}, \mathsf{stmt}') \in \mathsf{Succ}$. The label a_{stmt} models the execution of the Java statement stmt. Tab. 5.2 shows the label a_{stmt} that is generated for a Java statement stmt. Note that (i) `synchronized` blocks have been compiled down to the lower-level Java bytecode statements `monitorenter` and `monitorexit`, and (ii) label generation must know if a method is a unit of work.

5.7 CPDS Generation

An EML program has a set of shared-memory locations, S_{Mem}, a set of EML locks, S_{Locks}, and a set of EML processes, S_{Procs}. Empire generates a number of

CPDSs for a given EML program: a CPDS is generated for each pair $(m, m') \in S_{\mathsf{Mem}} \times S_{\mathsf{Mem}}$ for the fourteen interleaving scenarios. Pairs are used because the interleaving scenarios are defined in terms of at most two locations from an atomic set (cf. Tab. 2.3 in Ch. 2). Moreover, AS-serializability-violation detection is asymmetric in that it is performed with respect to an individual EML process Proc. In total, Empire generates $O(|S_{\mathsf{Procs}}| * 14 * (|S_{\mathsf{Mem}}|^2))$ CPDSs for an EML program. In each generated CPDS Π, there is a PDS for each global component of the EML program.

1. Π contains a PDS for each EML lock to enforce mutual exclusion.

2. Π contains a PDS for each EML process.

3. Π contains a PDS that monitors for a violation.

Modeling an EML Lock

Because an EML lock is reentrant, the language of the PDS that describes such behavior is context-free. The PDS stack is used to count in unary the number of times that a process has acquired the lock.[6] Another process may acquire the lock only when the process that currently holds the lock has released the lock enough times to empty the stack. For an EML lock l, the PDS that models the behaviors of l is defined as $\mathcal{P}_l = (P_l, \mathsf{Lab}_l, \Gamma_l, \Delta_l, \langle \mathsf{unalloc}, \bot \rangle)$, where

- $P_l = \{\mathsf{unalloc}, \mathsf{alloc}\}$: the control locations unalloc and alloc denote whether an EML lock l has been allocated.

- $Lab_l = \{$"x.alloc l", "x.lock l", "x.unlock l" $\mid x \in S_{\mathsf{Procs}}\}$: the actions x.alloc, x.lock, and x.unlock denote that EML process $x \in S_{\mathsf{Procs}}$ allocates, acquires, and releases the lock, respectively. (The property that a lock can be allocated one time by a distinct EML process is enforced by the PDS rules, and is explained below.)

[6] The use of the stack to count in unary was also used to model a counter c for the Bluetooth case study presented in §4.6 of Ch. 4.

- $\Gamma_l = \{\bot\} \cup S_{\mathsf{Procs}}$: the stack symbol $\bot$ is the bottom-of-stack marker. For an EML process $x \in S_{\mathsf{Procs}}$, the stack symbol x denotes that x currently holds the lock. The PDS rules Δ_l maintain the invariant that the PDS stack contains the symbol for only one EML process at a time.

- Δ_l is the union of three rule classes: *allocation*, *acquire*, and *release*.

allocation $\{\langle \mathsf{unalloc}, \bot\rangle \xhookrightarrow{x.\mathsf{alloc}\ l} \langle \mathsf{alloc}, \bot\rangle \mid x \in S_{\mathsf{Procs}}\}$. These rules model the allocation of the EML lock l by an EML process $x \in S_{\mathsf{Procs}}$. Because a lock can never return to the unalloc state, the property that a lock can be allocated only one time is enforced.

acquire $\{\langle \mathsf{alloc}, \gamma\rangle \xhookrightarrow{x.\mathsf{lock}\ l} \langle \mathsf{alloc}, x\,\gamma\rangle \mid x \in S_{\mathsf{Procs}}, \gamma \in \{x, \bot\}\}$. These rules model the acquisition of l. The rules push the name of the EML process that acquires l, effectively incrementing the counter that tracks the reentrant depth of acquisitions of l. Note that an EML process x can only acquire the lock if the lock is either in the unlocked state, modeled by $\gamma = \bot$, or if the top-of-stack symbol is the name of the EML process attempting to acquire the lock, which enforces that only one EML process may hold l at a time.

release $\{\langle \mathsf{alloc}, x\rangle \xhookrightarrow{x.\mathsf{unlock}\ l} \langle \mathsf{alloc}, \epsilon\rangle \mid x \in S_{\mathsf{Procs}}\}$. These rules model the release of l. As for the acquire rules, an EML process may only release the lock if the top-of-stack symbol is the name of the EML process (i.e., if the EML process currently owns the lock). The lock will become free only when the current process has released the lock the same number of times that it acquired the lock, which results in the stack containing only the bottom-of-stack marker $\bot$.

Modeling an EML Process

Generating a PDS $\mathcal{P}$ for an EML process Proc is performed in two stages. First, a single-state PDS $\mathcal{P}_1 = (P_1 = \{p\}, \mathsf{Lab}_1, \Gamma_1, \Delta_1, \langle p, e_{\mathsf{main}}\rangle)$ is generated using the rule templates depicted in Tab. 3.1, with Lab_1 being the set of all distinct EML statements used by Proc prefixed with the EML process's name. For example, if the EML statement is "read $\mathtt{m}$" and the EML process's name is Proc, then Lab_1 will contain "Proc.read $\mathtt{m}$". Including the EML process's name in the PDS action enables the violation monitor and locks to know which EML process performs an action (cf. the PDS rules for an EML lock above). $\mathcal{P}_1$ captures the interprocedural control flow of Proc. There is one exception, the EML statement "start $\mathsf{Proc}_{\chi_\theta}$" does not include Proc as a prefix because a thread can be started by any other thread (but only started one time).

Second, PDS $\mathcal{P}_2$ is PDS $\mathcal{P}_1$ augmented to account for lock allocation. (Recall that a lock must be allocated before it can be used.) The set of control locations P_1 of $\mathcal{P}_1$ is expanded to include a boolean flag for each lock l. If the flag is true then l has been allocated, otherwise an EML process has yet to allocate l.

From Proc's point of view, there are two ways that a lock can be allocated: either Proc allocates a lock or another EML process Proc′ allocates the lock. If Proc allocates the lock l, then there will be a PDS rule of the form $\langle p, \gamma \rangle \xhookrightarrow{\mathsf{Proc}.\mathtt{alloc}\ l} \langle p', u \rangle$. The corresponding rule in PDS $\mathcal{P}_2$'s rule set Δ_2 must ensure that the control location on the left-hand-side has the flag for l set to false, and the control location on the right-hand-side has the flag for l set to true. Otherwise Proc′ allocates l. In this case, Proc has no way of knowing when Proc′ allocates l, and therefore must *guess* when the allocation occurred. Guessing is modeled by non-deterministically invoking the guess method, which simply guesses that another EML process allocates l. Because of non-determinism, the guess method causes Proc to consider all possibilities of lock allocation (i.e., the cross product of S_{Procs} and S_{Locks}).

Formally, PDS $\mathcal{P}_2 = (P_2, \mathsf{Lab}_2, \Gamma_2, \Delta_2, \langle \emptyset, e_{\mathrm{main}} \rangle)$, where

- $P_2 = 2^{S_{\mathsf{Locks}}}$: a control location is a set of flags s denoting which locks have been allocated. The set of control locations P_1 is not used because it is the singleton set $\{p\}$.
- $\mathsf{Lab}_2 = \mathsf{Lab}_1 \cup \{P'.\mathsf{alloc}\ l \mid P' \in (S_{\mathsf{Procs}} \setminus \{\mathsf{Proc}\}), l \in S_{\mathsf{Locks}}\}$: Lab_2 is Lab_1

Action a	Rule $\langle p, \gamma \rangle \xhookrightarrow{a} \langle p, w \rangle \in \Delta_1$	
τ, start $\mathcal{P}'$	$\{ \langle s, \gamma \rangle \xhookrightarrow{a} \langle s, w \rangle$	$\mid s \in 2^{S_{\mathsf{Locks}}} \}$
alloc l	$\{ \langle s, \gamma \rangle \xhookrightarrow{P.a} \langle s \cup \{l\}, w \rangle$	$\mid s \in 2^{S_{\mathsf{Locks}}} \wedge l \notin s\}$
lock/unlock l	$\{ \langle s, \gamma \rangle \xhookrightarrow{P.a} \langle s, w \rangle$	$\mid s \in 2^{S_{\mathsf{Locks}}} \wedge l \in s \}$
read/write m	$\{ \langle s, \gamma \rangle \xhookrightarrow{P.a} \langle s, w \rangle$	$\mid s \in 2^{S_{\mathsf{Locks}}} \wedge l_{\ddot{\chi}} \in s \}$
ubegin/uend	$\{ \langle s, \gamma \rangle \xhookrightarrow{P.a} \langle s, w \rangle$	$\mid s \in 2^{S_{\mathsf{Locks}}} \wedge l_{\ddot{\chi}} \in s \}$
$\star$	$\{ \langle s, \gamma \rangle \xhookrightarrow{\tau} \langle s, \mathsf{guess}\, \gamma \rangle$	$\mid s \in 2^{S_{\mathsf{Locks}}} \}$
$\star$	$\{ \langle s,\ \mathsf{guess} \rangle \xhookrightarrow{P'.\mathsf{alloc}\ l} \langle s \cup \{l\}, \epsilon \rangle$	$\mid s \in 2^{S_{\mathsf{Locks}}} \wedge l \notin s$ $\wedge P' \in (S_{\mathsf{Procs}} \setminus \{P\}) \}$

Table 5.3: Each row defines a set of PDS rules in Δ_2 from a rule in Δ_1. The control location p from a rule in Δ_1 is not repeated because all rules in Δ_1 are single-control-location rules. The condition for generating a rule reflects that certain actions can only occur when a lock has been allocated, e.g., acquiring a lock l can only occur after l has been allocated (see §5.7).

augmented to include actions that allow for $\mathcal{P}_2$ to guess when another EML process Proc′ allocates a lock.

- $\Gamma_2 = \Gamma_1 \cup \{\mathsf{guess}\}$: Γ_2 includes the stack symbol guess that implements the guessing procedure.

- Δ_2 is defined from Δ_1 as shown in Tab. 5.3. Row 2 ensures that no lock is allocated more than once; row 3 ensures that a lock is not used before being allocated; and rows 4 and 5 ensure that the shared-memory locations are not accessed before $\zeta^{\sharp}_{\ddot{\chi}}$ has been allocated. Row 6 defines rules that invoke the "guessing" procedure for each configuration of $\mathcal{P}_2$. Guessing is necessary because an EML process cannot know when another EML process allocates a lock. Row 7 defines rules that implement the guessing procedure: from control location s, $s \subseteq S_{\mathsf{Locks}}$, guess that EML process $P' \in (S_{\mathsf{Procs}} \setminus \{P\})$ allocates a lock $l \in (S_{\mathsf{Locks}} \setminus s)$, and return back to the caller in the control location $\mathsf{s} \cup \{\mathsf{l}\}$. The guessing rule is then labeled with action $\mathsf{P}'.\mathsf{alloc}\ \mathsf{l}$.

Violation Monitor

The *violation monitor* detects when one of the interleaving scenarios occurs during a unit of work for a specific EML process Proc. To do so, it must track (i) the reads and writes to the shared-memory locations $\mathsf{S}_{\mathsf{Mem}}$ by each EML process, and (ii) whether or not the target EML process Proc is executing a unit of work.

Tracking the reads and writes of EML processes requires only a finite amount of state, i.e., state to track which reads and writes of interest have been seen. Recall that units of work are reentrant because unit-of-work methods may be recursive or invoke other unit-of-work methods. Thus, tracking the unit-of-work status of Proc requires an infinite amount of state. Similar to how the PDS for an EML lock uses its stack to count the depth of nested lock acquires, the PDS for the violation monitor uses its stack to count in unary the depth of nested calls to units-of-work methods by Proc.

Following the discussion above, we break down the definition of the PDS $\mathcal{P}_{\mathsf{mon}}$ that implements a violation monitor into two parts. The first part handles the infinite-state portion of $\mathcal{P}_{\mathsf{mon}}$ by defining the PDS $\mathcal{P}_{\mathsf{unit}}$ that tracks the unit-of-work status of the target EML process Proc. The second part handles the finite-state portion of $\mathcal{P}_{\mathsf{mon}}$ by defining the non-deterministic finite automaton (NFA) $\mathcal{A}_{\mathsf{mon}}$ that accepts traces containing the problematic access pattern of interest. Finally, the cross product of $\mathcal{P}_{\mathsf{unit}}$ and $\mathcal{A}_{\mathsf{mon}}$, which implements the intersection of their respective languages, defines $\mathcal{P}_{\mathsf{mon}}$.

$\mathcal{P}_{\mathsf{unit}}$

To model the unit-of-work status of the target EML process Proc, a counter is required because unit-of-work methods can be recursive or invoke other unit-of-work methods. The PDS $\mathcal{P}_{\mathsf{unit}}$ uses its stack to count the depth of nested calls to unit-of-work methods, and is essentially the PDS for a counter defined by Defn. 4.15 in Ch. 4—the difference being the actions that label the rules. Formally, $\mathcal{P}_{\mathsf{unit}} = (P_{\mathsf{unit}}, \Gamma_{\mathsf{unit}}, \mathsf{Lab}_{\mathsf{unit}}, \Delta_{\mathsf{unit}}, \langle p, 0\rangle)$, where

- $P_{\mathsf{unit}} = \{p\}$ consists of a single control location;
- $\Gamma_{\mathsf{unit}} = \{0, 1\}$ is the stack alphabet;
- $\mathsf{Lab}_{\mathsf{unit}} = \{\, [,], a_1, a_0 \,\}$ is the set of actions that count the depth of unit-of-work calls in unary. The actions $[$ and $]$ are shorthands for $\mathsf{Proc.unitbegin}$ and $\mathsf{Proc.unitend}$, respectively, and are used to synchronize with $\mathcal{P}_{\mathsf{Proc}}$ when it enters and exits a unit of work, respectively. The actions a_1 and a_0—denoting that the top-of-stack symbol is 1 and 0, respectively—are used to synchronize with $\mathcal{A}_{\mathsf{mon}}$ so that $\mathcal{A}_{\mathsf{mon}}$ can test the unit-of-work status of Proc.
- Δ_{unit} is defined as follows:[7]

$$\begin{array}{lllllll}
1. & \langle p, 0\rangle & \xhookrightarrow{[} & \langle p, 1\,0\rangle, & \langle p, 1\rangle & \xhookrightarrow{[} & \langle p, 1\,1\rangle, \\
2. & \langle p, 0\rangle & \xhookrightarrow{a_0} & \langle p, 0\rangle, & \langle p, 1\rangle & \xhookrightarrow{a_1} & \langle p, 1\rangle, \\
3. & \langle p, 1\rangle & \xhookrightarrow{]} & \langle p, \epsilon\rangle, & & &
\end{array}$$

When Proc begins a unit of work, the stack symbol 1 is pushed onto the stack (row 1). Likewise, when Proc exits a unit of work, the top-of-stack symbol is popped off (row 3). Note that the top of stack must contain the stack symbol 1 for a pop to occur, and thus can only occur when Proc is executing a unit of work. The rules in row 2 allow for the NFA $\mathcal{A}_{\mathsf{mon}}$ to query the unit-of-work status of Proc. That is, if the stack symbol 1 is on the top of the stack, which signifies that Proc is currently executing a unit of work, then $\mathcal{P}_{\mathsf{unit}}$ can exchange the action a_1 with $\mathcal{A}_{\mathsf{mon}}$. Similarly, if

[7] The rules of Δ_{unit} are the same as the rules to implement a counter from Defn. 4.15. The only difference is that the actions that annotate the rules have been updated to model unit-of-work method calls and returns (versus actions to change the state of a counter).

the stack symbol o is on the top of the stack, then $\mathcal{P}_{\mathsf{unit}}$ can exchange the action a_o with $\mathcal{A}_{\mathsf{mon}}$.

$\mathcal{A}_{\mathsf{mon}}$

The NFA $\mathcal{A}_{\mathsf{mon}}$ accepts interleaved executions (traces) that contain the memory accesses specified by the problematic access pattern of interest. To make the discussion more concrete, we focus on the NFA $\mathcal{A}_{12}$ shown in Fig. 5.2, which tracks the problematic access pattern "$R_1(c); W_2(d); W_2(c); R_1(d)$" for the AS-serializability violation from the program shown in Listing 5.1 (see page 74). For a generic problematic access pattern p, the definition of the NFA $\mathcal{A}_{\mathsf{mon}}$ that recognizes traces containing p follows naturally.

Fig. 5.2 gives a graphical depiction of $\mathcal{A}_{12}$. The initial state is q_1 and the final state is q_7. For a trace to be accepted by $\mathcal{A}_{12}$, it must make a transition through each state q_{1-7}. That is, the states q_{1-7} track the memory accesses that make up problematic access pattern 12. The transition $(q_1, \mathsf{alloc}, q_2)$ models the allocation of the randomly-isolated object. Once the randomly-isolated object has been allocated, $\mathcal{A}_{12}$ ignores reads and writes by following the self-loop on state q_2 until the target EML process Proc begins a unit of work—modeled by the transition $(q_2, [, q_3)$.

$\mathcal{A}_{12}$ makes use of non-determinism. For each state q_i, $3 \leqslant i \leqslant 6$, there is a self-loop labeled with $[R_i W_i$. The symbol $[$ models reentrant calls to units of work. The state does not change because Proc must be executing a unit of work for $\mathcal{A}_{12}$ to be in state q_i. The symbols R_i and W_i denote read and write accesses to any memory location by either thread. The use of non-determinism enables $\mathcal{A}_{12}$ to "guess" which memory accesses are actually involved in problematic access pattern 12. For example, if $\mathcal{A}_{12}$ is in state q_3 and observes the action $R_1(c)$, it can make a transition to state q_4—-the action is part of problematic access pattern 12—or it can follow the self-loop and remain in state q_3—the action is not part of problematic access pattern 12. Non-determinism is required because a thread may perform multiple memory accesses during a unit of work.

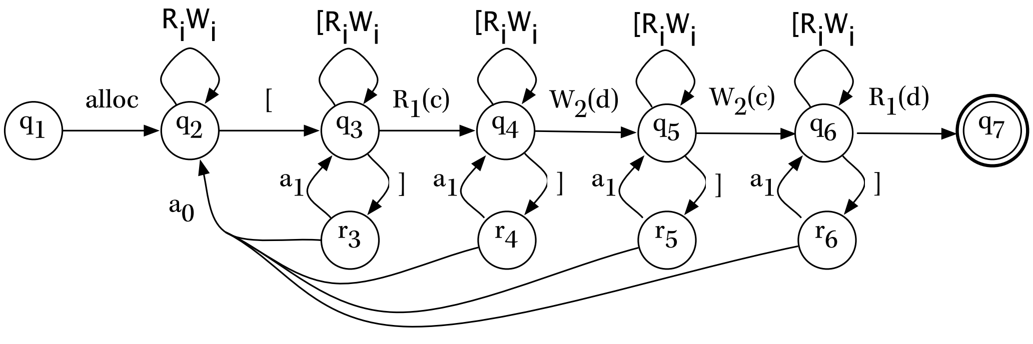

Figure 5.2: The NFA $\mathcal{A}_{12}$ that recognizes traces of interleaved read and write memory accesses containing problematic access pattern 12 for the program shown in Listing 5.1 (see §5.1). The edge labeled alloc denotes allocating the randomly-isolated object. An edge labeled $\mathsf{R}_1(\mathsf{c})$ ($W_2(\mathsf{c})$) denotes a read from (write to) the field `Stack.count` by thread T_1 (T_2). Similarly, edges labeled $\mathsf{R}_1(\mathsf{d})$ and $W_2(\mathsf{d})$ denote accesses to the field `Stack.data`. The self-loops labeled $\mathsf{R}_i W_i$ denote a read or write to any memory location by either thread. The symbols [and] denote Proc beginning and ending a unit of work, respectively. The symbols a_1 and a_0 are used to synchronize with $\mathcal{P}_{\mathsf{unit}}$ to determine the unit-of-work status of Proc. If Proc completed the outermost unit of work, then the state is reset to q_2 by exchanging an a_0 action with $\mathcal{P}_{\mathsf{unit}}$. Otherwise, the state q_i—from which the unit-of-work end action] was witnessed—is restored by exchanging an a_1 action with $\mathcal{P}_{\mathsf{unit}}$.

The states $\mathsf{r}_{3\text{-}6}$ are used to implement a "reset" check. That is, if Proc finishes a unit of work, it will synchronize on the action Proc.unitend—abbreviated by]. Because $\mathcal{A}_{12}$ has only a finite amount of state, it has no way of knowing whether Proc exited the outermost unit of work. Tracking the reentrant depth of calls to unit-of-work methods is the job of $\mathcal{P}_{\mathsf{unit}}$. Thus, $\mathcal{A}_{12}$ must synchronize with $\mathcal{P}_{\mathsf{unit}}$ to determine whether it should reset its state to q_2—Proc completed its outermost unit of work—or return to its previous state q_i from which the action] caused $\mathcal{A}_{12}$ to make a transition to state r_i—Proc exited a nested call to a unit-of-work method.

The definition of an NFA $\mathcal{A}_{\mathsf{mon}}$ for an arbitrary problematic access pattern follows from the definition of $\mathcal{A}_{12}$.

$\mathcal{P}_{\mathsf{mon}}$

The PDS $\mathcal{P}_{\mathsf{mon}}$ is defined from the cross product of $\mathcal{P}_{\mathsf{unit}} = (\mathsf{P}_{\mathsf{unit}}, \Gamma_{\mathsf{unit}}, \mathsf{Lab}_{\mathsf{unit}}, \Delta_{\mathsf{unit}}, c_0)$ and $\mathcal{A}_{\mathsf{mon}} = (Q, \Sigma, \delta, q_1, q_f)$. Specifically, $\mathcal{P}_{\mathsf{mon}} = (Q, \Gamma_{\mathsf{unit}}, \Sigma, \Delta_{\mathsf{mon}}, \langle q_1, 0\rangle)$, where Δ_{mon} is defined as follows:

- For an action $a \in \{\, [,]\,\}$, each rule $\langle p, \gamma\rangle \xhookrightarrow{a} \langle p, u\rangle \in \Delta_{\mathsf{unit}}$, and each transition $(q, a, q') \in \delta$, the rule $\langle q, \gamma\rangle \xhookrightarrow{a} \langle q', u\rangle \in \Delta_{\mathsf{mon}}$. The rules combine the counting of $\mathcal{P}_{\mathsf{unit}}$ with the state transitions of $\mathcal{A}_{\mathsf{mon}}$.

- For an action $a \in \{\, a_1, a_0\,\}$, each rule $\langle p, \gamma\rangle \xhookrightarrow{a} \langle p, u\rangle \in \Delta_{\mathsf{unit}}$, and each transition $(q, a, q') \in \delta$, the rule $\langle q, \gamma\rangle \xhookrightarrow{\tau} \langle q', u\rangle \in \Delta_{\mathsf{mon}}$. Because the actions a_1 and a_0 only implement synchronization operations between $\mathcal{P}_{\mathsf{unit}}$ and $\mathcal{A}_{\mathsf{mon}}$, they are replaced by the special CPDS action τ (i.e., they are not part of an actual trace but are artifacts of our breakdown of the definition of $\mathcal{P}_{\mathsf{mon}}$ into constituents $\mathcal{P}_{\mathsf{unit}}$ and $\mathcal{A}_{\mathsf{mon}}$).

- For each transition $(q, \sigma, q') \in \delta$, where $\sigma \notin \mathsf{Lab}_{\mathsf{unit}}$, Δ_{mon} contains the set of rules: $\{\langle q, \gamma\rangle \xhookrightarrow{\sigma} \langle q', \gamma\rangle \mid \gamma \in \Gamma_{\mathsf{unit}}\}$. These rules account for reads and writes to memory locations, and the allocation of the randomly-isolated object.

At the PDS-language level, the cross product of $\mathcal{P}_{\mathsf{unit}}$ and $\mathcal{A}_{\mathsf{mon}}$ takes their intersection, i.e., $\mathsf{Lang}(\mathcal{P}_{\mathsf{mon}}) = \mathsf{Lang}(\mathcal{P}_{\mathsf{unit}}) \cap \mathsf{Lang}(\mathcal{A}_{\mathsf{mon}})$. Intersection combines the counting of $\mathcal{P}_{\mathsf{unit}}$ with the pattern recognition of $\mathcal{A}_{\mathsf{mon}}$, which produces the desired PDS $\mathcal{P}_{\mathsf{mon}}$ that implements a violation monitor.

CPDS **Query**

Once a CPDS Π has been generated for an EML program EProg, a language-emptiness query is passed to CPDSMC, which requires defining the target set of configurations for each PDS $\mathcal{P}_i$.

- For PDS $\mathcal{P}_l$ that describes EML lock l, the target set of configurations C_l is any configuration: $C_l = \{\langle p, x^* \bot \rangle \mid x \in S_{\mathsf{Procs}}\}$.

- For PDS $\mathcal{P}_x$ that describes EML process $x \in S_{\mathsf{Procs}}$, the target set of configurations C_x is any configuration: $C_x = \{\langle p, u \rangle \mid p \in P_x, u \in \Gamma_x^*\}$.

- For the PDS $\mathcal{P}_{\mathsf{mon}}$ that describes the violation monitor, the target set of configurations C_{mon} is one in which the final control location q_f—denoting that an AS-serializability violation has occurred—has been reached. Specifically, $C_{\mathsf{mon}} = \{\langle q_f, 1^+ 0 \rangle\}$.

Let G be the configuration sets for the PDSs. The language-emptiness query as defined is such that $\mathsf{Lang}(\Pi, G) = \emptyset$ is true *if-and-only-if* the EML program cannot generate a trace accepted by the violation monitor.

5.8 Experiments

Empire is implemented using the WALA (IBM, 2009) program-analysis framework. Random isolation is implemented using WALA's facilities for rewriting the abstract-syntax tree (AST) of a Java program. The default object-sensitive call graph construction and points-to analyses are modified to implement the semantic reinterpretation of "is_ri", as described in §5.4. After rewriting the ASTs, Empire emits an EML program from the input Java program. The EML program is then translated into multiple CPDSs (see §5.7), for which reachability queries are answered using Cpdsmc. Cpdsmc is implemented using the WALi library (Kidd et al., 2009a), and the prefix semiring (see Defn. 4.8 in Ch. 4) is implemented using the BuDDy BDD library (BuDDy, 2004). All experiments were run on a dual-core 3 GHz Pentium Xeon processor with 4 GB of memory.

The goal of the experiments was to determine whether the techniques developed and implemented in Empire could detect both single- and multi-location AS-serializability violations. We evaluated Empire on eight programs from the ConTest suite (Eytani et al., 2007b), which is a set of small benchmarks with

Benchmark	# CPDSs	Viol	OK	OOM	OOT
Account	642	5	78	24	535
AirlineTickets	900	12	882	0	6
PingPong	384	29	349	0	6
ProducerConsumer	512	19	79	37	377
SoftwareVerificationHW	15	0	5	0	10
BugTester	615	0	378	0	237
BuggyProgram	615	0	599	0	16
shop	900	3	0	257	640
Totals	4583	68	2368	318	1827

Table 5.4: Column "Benchmark" specifies the names of the eight ConTest benchmark programs analyzed. Column "# CPDSs" specifies the number of CPDSs generated. Column "Viol" specifies the number of AS-serializability violations detected. Column "OK" specifies the number of CPDS queries that reported no AS-serializability violation. Column "OOM" specifies the number of CPDS queries that exhausted memory (OOM). Column "OOT" specifies the number of CPDS queries that exhausted the 300-second timeout. The horizontal line after row 4 separates the benchmarks that did not contain any synchronization operations after abstraction from those that still did contain synchronization operations.

known concurrency bugs. 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 currently does not support certain Java constructs). When analyzing a program with the user-specified allocation site $\dot{\chi}$ that allocates an object of type T, we used the default assumptions that (i) all fields declared by T are in one atomic set, and (ii) each public method defined by T is a unit of work.

To reduce the size of the generated models, we made minor modifications to the benchmark programs. For the programs analyzed, file I/O is used to output debugging and scheduling information, and to receive input that specifies the number of threads. We removed these operations, and manually unrolled loops that allocate `Thread` objects 2 times. When a benchmark used a shared object of type `java.lang.Object` as a lock, the type was changed to `java.lang.Integer` because our implementation uses *selective* object-sensitivity, for which the use of `java.lang.Object` as a shared lock removes all selectivity and severely degrades performance.[8] The programs `SoftwareVerificationHW` and `shop` define each thread's `run()` method to consist of a loop that repeatedly executes one unit of work. For these programs, the code in the body of the loop was extracted out into its own method so that the default unit-of-work assumptions would be correct. Each modification had no impact on the AS-serializability violations that could occur in a benchmark.

In total, Empire generated 4583 CPDSs. Each query was analyzed by Cpdsmc using a 300-second timeout. Tab. 5.4 presents a summary of the analysis results.[9] The dividing line that separates the first four benchmarks from the latter four benchmarks pertains to lock usage. Each generated EML program consists of a single lock, namely, the lock for the randomly-isolated object.[10] However, for the first four benchmarks, the code does not contain synchronized methods, and hence does not acquire and release the lock associated with the randomly-isolated object. For the latter four benchmarks, the code contains synchronized methods, and the generated EML programs for these four contain lock and unlock operations. In total, Cpdsmc

1. found 68 AS-serializability violations (col. "Viol");

[8] By selective object-sensitivity, we mean that a full object-sensitive call graph is not constructed because doing so exhausts memory resources. Instead, the call graph is only object sensitive with respect to a few key object types: the type T of the specified allocation site, the types of all of T's fields, and the type of all subclasses of threads.

[9] The experimental results differ from Kidd et al. (2009b) because (i) a 300-second timeout was used instead of a 200-second timeout, and (ii) the experiments from Kidd et al. (2009b) used a different version of Cpdsmc that applies the *language strength reduction* transformation (Ch. 6).

[10] Empire can generate multi-lock EML programs when the Java program makes use of global locks that are guaranteed to be unique, such as the lock that is associated with a `Class` object.

2. determined that 2368 queries did not contain an AS-serializability violation (col. “OK”);

3. ran out of memory for 318 of the queries (col. “OOM”);

4. and exhausted the allotted time for 1827 queries (col. “OOT”).

Combining the totals for cols. “Viol” and “OK”, CPDSMC returned a definitive answer for 2436 queries (53%), and exhausted resources for 2145 queries (47%). For the benchmarks where no AS-serializability violations were found, CPDSMC exhausted resources on the queries where a bad configuration was reachable. (We verified reachability of the bad configurations using the decision procedure for finding AS-serializability violations of EML programs that is presented in Ch. 7.)

For each AS-serializability violation reported by EMPIRE, we manually verified whether or not it was an actual AS-serializability violation, and not a false positive due to abstraction. Tab. 5.5 presents a breakdown of the problematic-access-pattern numbers of the AS-serializability violations that were found for each benchmark program. For 5 of the 8 benchmarks listed in Tab. 5.5, EMPIRE found multiple violations. The false positives reported for `PingPong` are due to an over-approximation of a thread’s control flow—exceptional-control paths are allowed in the model that cannot occur during a real execution of the program.

Overall, the experiments showed that EMPIRE is able to detect both single- and multi-location AS-serializability violations. It is interesting to note that for benchmarks where a multi-location AS-serializability violation was found, a single-location AS-serializability violation also occurred. This can be explained by the fact that Java methods typically read and write to a field multiple times, which allows for both single- and multi-location AS-serializability violations to occur.

Program	1	2	3	4	5	6	7	8	9	10	11	12	13	14
account	✓	✓		✓	✓									
airlinesTckts	✓	✓		✓		✓	✓	✓						
PingPong	✓	✓	✓		✓	$\not\checkmark$	$\not\checkmark$	$\not\checkmark$	$\not\checkmark$	$\not\checkmark$	$\not\checkmark$	$\not\checkmark$	$\not\checkmark$	$\not\checkmark$
ProdConsumer	✓	✓	✓	✓	✓						✓	✓	✓	
SoftwareVerificationHW														
BugTester														
BuggyProgram														
shop	✓	✓									✓			

Table 5.5: Marked entries denote violations reported by EMPIRE, with ✓ being a verified violation and $\not\checkmark$ a false positive. Scenarios 6–16 involve two memory locations.

5.9 Related Work

The idea of isolating a distinguished non-summary node that represents the memory location that will be updated during a transition, so that a strong update can be performed on it, has a long history in shape-analysis algorithms (Horwitz et al., 1989; Jones and Muchnick, 1981; Sagiv et al., 2002). When these methods also employ the allocation-site abstraction, each abstract memory configuration will have some bounded number of abstract nodes per allocation site.

Like random-isolation abstraction, *recency abstraction* (Balakrishnan and Reps, 2006) uses no more than *two* abstract blocks per allocation site χ: a non-summary block MRAB[χ], which represents the **m**ost-**r**ecently-**a**llocated **b**lock allocated at χ, and a summary block NMRAB[χ], which represents the **n**on-**m**ost-**r**ecently-**a**llocated **b**locks allocated at χ. As the names indicate, recency abstraction is based on tracking a *temporal* property of a block b: the is-the-most-recent-block-from-$\chi(b)$ property.

With counter abstraction (McMillan, 1999; Pnueli et al., 2002; Yavuz-Kahveci and Bultan, 2002), numeric information is attached to summary objects to characterize the number of concrete objects represented. The information on summary object u of abstract configuration S describes the number of concrete objects that are mapped to u in any concrete configuration that S represents. Counter abstraction has been used to analyze infinite-state systems (McMillan, 1999; Pnueli et al., 2002), as well as in shape analysis (Yavuz-Kahveci and Bultan, 2002).

In contrast to all of the aforementioned work, random-isolation abstraction is based on tracking the properties of a *random* individual, and generalizing from the properties of the randomly chosen individual according to Random-Isolation Principle 2.

Various data-race detection tools were discussed in Ch. 2. We expect that these tools could find the single-location AS-serializability violations that Empire found (see §5.8). However, these tools would not be able to find the multi-location AS-serializability violations that Empire found.

With respect to AS-serializability violation detection, the most closely related work is the dynamic tool for detecting AS-serializability violations developed by Hammer et al. (2008). They also analyzed programs from the ConTest benchmark suite, and their dynamic tool found AS-serializability violations in the programs `Account`, `AirlineTcks`, `BugTester`, `PingPong`, and `SoftwareVerificationHW`. Our tool did not find the AS-serializability violation in `BugTester` because it exhausted the available resources; however, it did find AS-serializability violations in `shop`, which their tool did not. The remaining two benchmarks were not analyzed by their tool.

6 LANGUAGE STRENGTH REDUCTION

6.1 Introduction

Ch. 5 presented Empire, a tool for verifying AS-atomicity of concurrent Java programs. Empire abstracts a concurrent Java program into a program written in the Empire Modeling Language (EML), which consists of a finite set of shared-memory locations, a finite set of locks, and a finite set of concurrently executing processes. Recall that, as in Java, an EML lock is reentrant; i.e., it can be acquired multiple times by the process that owns the lock, but it also must be successively released the same number of times. An EML lock is acquired and released by entering and exiting, respectively, a function that is synchronized on the lock. (Java synchronized blocks are modeled as inlined anonymous functions in EML.)

To verify AS-atomicity of an EML program EProg, Empire compiles EProg into a CPDS Π that contains one PDS for each EML process and each EML lock, and an PDS for the violation monitor. The CPDS Π is then fed to Cpdsmc, which implements a semi-decision procedure that attempts to determine if the intersection of the set of CFLs—the languages of Π's PDSs—is empty or not.

If a language is known to be regular, then Cpdsmc can be directed to treat it as such. Because determining if a CFL is regular is undecidable, we use a simple syntactic heuristic to specify regularity, namely, see whether the rule set Δ of a PDS $\mathcal{P}$ contains only step rules (i.e., $\Delta_0 = \emptyset = \Delta_2$); this ensures that $\mathcal{P}$ defines an NFA. For example, a rule $r = \langle p, \gamma \rangle \xhookrightarrow{a} \langle p', \gamma' \rangle$ would define the transition $\big((p, \gamma), a, (p', \gamma')\big)$ in the transition relation δ of NFA.

The use of regular languages has a direct performance benefit for all of the SDPs defined in Ch. 4. When an NFA $\mathcal{A}$ is used in place of a PDS $\mathcal{P}$,

1. *Precision* increases because Cpdsmc uses the exact language $\mathcal{A}$ in place of an over-approximation of the language of $\mathcal{P}$.

2. *Cost* decreases because (i) Cpdsmc avoids computing the prefix abstractions—

one for each refinement iteration—for $\mathcal{P}$, and (ii) the intersection of the specified regular languages (the languages of $\mathcal{A}$ and other NFAs that replace PDSs) is precomputed once and for all.

This chapter presents a generic technique that we use to reduce the number of CFLs necessary to model an EML program. It is based on the observation that the CFL for an EML lock l can be replaced by a regular language because an EML process's acquisitions and releases of l are synchronized with function calls and returns. We call the process of replacing a CFL by a regular language *language strength reduction*. The end result is that reentrant locks can be replaced by non-reentrant locks *without* sacrificing soundness or precision. Thus, for an EML program with m processes and n locks, applying language strength reduction allows the program to be described by m CFLs and n regular languages, versus $m + n$ CFLs.

As defined in §5.7, the violation monitor $\mathcal{P}_{\mathsf{mon}}$ is a true PDS and not an NFA: it uses its stack to implement a counter. In this case, the stack height represents the number of times the target EML process has entered a unit of work. Just as we observed for EML locks, entering and exiting a unit of work is also synchronized with function calls and returns. Thus, we also apply language strength reduction to the language of the violation monitor. After performing language strength reduction, the transformed violation monitor's language is nearly identical to the finite-state part $\mathcal{A}_{\mathsf{mon}}$ (e.g., $\mathcal{A}_{12}$ in Fig. 5.2). The difference is that non-determinism replaces the use of "reset" states (see §6.5).

For an EML program with set of processes S_{Procs} and set of locks S_{Locks}, language strength reduction proceeds as follows:

1. For each EML process $x \in S_{\mathsf{Procs}}$, the PDS $\mathcal{P}_x$ is defined using the encoding given in §5.7 of Ch. 5.

2. For each EML process $x \in S_{\mathsf{Procs}}$, and for each EML lock $l \in S_{\mathsf{Locks}}$, a *nested-word automaton* $\mathcal{N}_x^l$ (Alur and Madhusudan, 2006) is defined such that it recognizes the same set of executions as $\mathcal{P}_x$. Nested-word automata

(NWA) are formally defined in §6.3, but for this discussion, we can view them as recognizers of CFLs where each word in the CFL is annotated with a relation that captures its parse tree, which, in our setting, is the parsing of matched calls and returns for runs of $\mathcal{P}_x$. The NWA $\mathcal{N}_x^l$ is defined such that it is able to distinguish between an outermost acquisition of l—one that definitely changes the owner of l—and an inner (or reentrant) acquisition of l—one that increments the counter but does not change the owner.

3. An important property of NWAs is that, because the matching relation is exposed, they are closed under intersection. For each EML process $x \in S_{\mathsf{Procs}}$, the NWA $\mathcal{N}_x$ is defined as the intersection of the NWAs $\mathcal{N}_x^l$, $l \in S_{\mathsf{Locks}}$. $\mathcal{N}_x$ is able to distinguish between outermost and inner lock acquisitions and releases by x for all locks $l \in S_{\mathsf{Locks}}$.

4. We define a cross-product-like construction (§6.4) to combine an NWA with a PDS, where the result is a new PDS that simulates both simultaneously. We use the construction to define the PDS $\mathcal{P}_{\mathcal{N}}\langle x \rangle$ from $\mathcal{P}_x$ and $\mathcal{N}_x$. Because $\mathcal{P}_{\mathcal{N}}\langle x \rangle$ simulates both $\mathcal{P}_x$ and $\mathcal{N}_x$, it retains the same set of behaviors of $\mathcal{P}_x$ while being able to distinguish between outermost and inner lock acquisitions and releases.

5. Because $\mathcal{P}_{\mathcal{N}}\langle x \rangle$ can distinguish between outermost and inner lock acquisitions, we remove all inner acquisitions from all runs of $\mathcal{P}_{\mathcal{N}}\langle x \rangle$. This is sound because inner acquisitions do not change the lock owner.

6. The runs of all PDSs $\mathcal{P}_{\mathcal{N}}\langle x \rangle$, $x \in S_{\mathsf{Procs}}$, no longer contain inner (reentrant) lock acquisitions and releases Thus, we no longer require a stack to track nested calls to acquire and release a lock because in the revised model they do not occur. That is, the language that models an EML lock is now *regular*. Thus, the transformed CPDS now contains m (transformed) PDSs, one for each EML process, and n regular languages, one for each lock.

7. Finally, the same technique is applied to the EML process whose unit-of-work status is being monitored by the violation monitor; i.e., we remove inner unit-of-work symbols from its traces. Because the violation monitor $\mathcal{P}_{\mathsf{mon}}$ only required a stack to track the depth of nested calls to unit-of-work methods, its stack is no longer necessary. Thus, strength reduction is also applied to the CFL of the violation monitor defined in §5.7 of Ch. 5 to replace its CFL with a regular language.

We applied the language-strength-reduction transformation to the CPDSs that were generated by EMPIRE for the eight CONTEST benchmark programs discussed in §5.8 of Ch. 5. Overall, the total running time taken by CPDSMC to analyze the transformed CPDSs was 1.8 times faster than analyzing the original CPDSs (see §6.6).

Contributions.

The observation that pushdown automata are closed under intersection when the stacks are synchronized was formalized by Alur and Madhusudan (2004, 2006). They defined nested-word languages, which make stack operations explicit in the words of the language, and nested-word automata (NWA), which accept such languages. They showed that these languages are closed under intersection.

Our approach is similar in spirit to Alur and Madhusudan (2006). We use an NWA $\mathcal{N}$ to model the locking (unit-of-work) behavior of an EML process. We define the nested-word language of a PDS (cf. §6.4) by associating a nested word with every path of the PDS, which makes the stack operations explicit. We give a generic construction that combines $\mathcal{N}$ with a PDS $\mathcal{P}$ to produce another PDS $\mathcal{P}_{\mathcal{N}}$ whose nested-word language is the intersection of the nested-word languages of $\mathcal{P}$ and $\mathcal{N}$.

Language strength reduction requires the ability to distinguish between the lock acquisitions and releases that change the owner of an EML lock $\mathfrak{l}$ and those that do not. We show how to achieve this using the NWA $\mathcal{N}$. We then transfer this ability to the PDS $\mathcal{P}$ for an EML process via the construction of $\mathcal{P}_{\mathcal{N}}$. This enables us to perform language strength reduction for the lock $\mathfrak{l}$ (§6.5), which provides the performance benefits highlighted by items 1 and 2 in §6.1.

The chapter concludes with a construction that combines an *extended weighted pushdown system* (EWPDS) $\mathcal{E}$—an extension to WPDSs that allows for user-defined merge functions to fuse together the weight of the caller with the weight of the callee (Lal et al., 2005)—with an NWA $\mathcal{N}$ to produce another EWPDS $\mathcal{E}_{\mathcal{N}}$. The benefit of this construction is that the EWPDS $\mathcal{E}_{\mathcal{N}}$ models the state transitions of $\mathcal{N}$ via a relational weight domain, which is a *symbolic* encoding of the state transitions of $\mathcal{N}$.

In general, for an EWPDS $\mathcal{E}$ that models some program, computing the COVP value (§6.7) over $\mathcal{E}_{\mathcal{N}}$ captures the set of all behaviors of the program modeled by $\mathcal{E}$ that respect the behaviors described by $\mathcal{N}$. One can view $\mathcal{E}_{\mathcal{N}}$ as the synthesis of several recent threads of research by Alur and Madhusudan (2006); Chaudhuri and Alur (2007) and Lal et al. (2005, 2007, 2008). Compared to standard approaches to property checking, one can simultaneously check properties

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

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

The remainder of the chapter is organized as follows: §6.2 provides an overview. §6.3 presents nested words and nested word automata. §6.4 presents the nested-word language of a PDS and the construction that combines an NWA with an PDS. §6.5 presents the language-strength-reduction transformation. §6.6 describes the experimental evaluation. §6.7 presents the definition of an EWPDS, and the construction that combines an NWA with an EWPDS. §6.8 discusses related work.

6.2 Overview

Fig. 6.1 presents (a textual representation of) an EML program that defines one EML lock l, one shared-memory location v, and one process P0. There are only two possible execution paths of P0, depending on the direction taken at the branch statement in function `testAndSet` on line 13. The execution path that takes the the true branch is given by *Path 1* in Fig. 6.2. Let "$($" and "$)$" denote lock l and unlock l, respectively; "[" and "]" denote unitbegin and unitend, respectively; and R_v and W_v denote reading from and writing to the variable v, respectively. *Path 1* can be described by the word $w_{\text{path}} =$ "$[((R_v)(W_v))]$". The projection of the locking symbols from w_{path} produces the word $w_l =$ "$(()())$". In general, due to recursion, the language that describes the set of possible program behaviors with respect to l is the unbalanced-left matched-parenthesis language shown in Fig. 6.3(a), where "unbalanced-left" denotes that a word in the language may have unmatched left-parenthesis symbols, but not unmatched right-parenthesis symbols.

For *Path 1*, there are two distinct types of lock acquisitions: ownership-changing acquisitions (OC) and non-ownership-changing acquisitions (nOC). The dual also holds for lock releases. With respect to w_l, these two distinct types correspond to outermost parentheses, denoted by "$(_o)_o$", and nested parentheses, denoted by "$(_n)_n$", respectively. Using this notation, w_l can be rewritten as "$(_o(_n)_n(_n)_n)_o$". Fig. 6.3(b) extends this to the language level by distinguishing between the outermost and nested parentheses of Fig. 6.3(a).

Observation 6.1. *With respect to the executions of an EML program, only the OC lock acquisitions and releases enforce mutual exclusion. For a program trace, pro-*

```
lock: l;
var : v;

process P0 {
    synchronized(l) get { read v; }

    synchronized(l) set { write v; }

    synchronized(l) testAndSet {
        lock l;
        get();
        unlock l;
        if( * ) {
          lock l;
          set();
          unlock l;
        }
    }

    main {
        unitbegin;
        lock l;
        testAndSet();
        unlock l;
        unitend;
    }
}
```

Figure 6.1: Example EML program that makes use of reentrant locking.

jecting out the nOC lock acquisitions and releases does not change the set of instructions that are guarded by locks.

Projecting out the nested parentheses for w_l results in “$(_{o})_{o}$”. Performing the projection on the grammar in Fig. 6.3(b) results in a regular language whose grammar is shown in Fig. 6.3(c).

Language strength reduction is a generic technique that allows the use of

Path 1

$$\overbrace{e_{\mathrm{main}} n_{21} n_{22} c_{23}\ \underbrace{e_{\mathrm{testAndSet}} n_{10} c_{11}\ \underbrace{e_{\mathrm{get}} n_5 x_{\mathrm{get}}}_{\texttt{get}}\ n_{12} n_{13} n_{14} c_{15}\ \underbrace{e_{\mathrm{set}} n_7 x_{\mathrm{set}}}_{\texttt{set}}\ n_{16} x_{\mathrm{testAndSet}}}_{\texttt{testAndSet}}\ n_{24} n_{25} x_{\mathrm{main}}}^{\texttt{main}}$$

Figure 6.2: *Path 1* describes the execution path of EML process P0 from Fig. 6.1 that takes the true branch at line 13.

	S	$\rightarrow$	U				
(a)	M	$\rightarrow$	ϵ	$\mid$	$M\,M$	$\mid$	$(\,M\,)$
	U	$\rightarrow$	M	$\mid$	$M\,U$	$\mid$	$(\,U$
	S	$\rightarrow$	U^{o}				
	M^{o}	$\rightarrow$	ϵ	$\mid$	$\mathsf{M}^{\mathsf{o}}\,\mathsf{M}^{\mathsf{o}}$	$\mid$	$(_{\mathsf{o}}\,\mathsf{M}^{\mathsf{n}}\,)_{\mathsf{o}}$
(b)	U^{o}	$\rightarrow$	M^{o}	$\mid$	$\mathsf{M}^{\mathsf{o}}\,\mathsf{U}^{\mathsf{o}}$	$\mid$	$(_{\mathsf{o}}\,\mathsf{U}^{\mathsf{n}}$
	M^{n}	$\rightarrow$	ϵ	$\mid$	$\mathsf{M}^{\mathsf{n}}\,\mathsf{M}^{\mathsf{n}}$	$\mid$	$(_{\mathsf{n}}\,\mathsf{M}^{\mathsf{n}}\,)_{\mathsf{n}}$
	U^{n}	$\rightarrow$	M^{n}	$\mid$	$\mathsf{M}^{\mathsf{n}}\,\mathsf{U}^{\mathsf{n}}$	$\mid$	$(_{\mathsf{n}}\,\mathsf{U}^{\mathsf{n}}$
(c)	S	$\rightarrow$	$(_{\mathsf{o}}\,)_{\mathsf{o}}\,S$	$\mid$	$(_{\mathsf{o}}$	$\mid$	ϵ

Figure 6.3: (a) Grammar for the CFL of a reentrant lock. (b) Grammar that distinguishes between outermost and nested parentheses. (c) Grammar for the regular language of a non-reentrant lock.

the simpler language in Fig. 6.3(c) in place of the language in Fig. 6.3(a). Language strength reduction provides the precision and cost benefits highlighted by items 1 and 2 of §6.1.

Language strength reduction relies on the ability to distinguish between the OC and nOC lock acquisitions of an EML process. In §6.3, we show how this distinction can be captured by an NWA. Having defined the language of Fig. 6.3(b) via an NWA $\mathcal{N}$, we combine it with the PDS $\mathcal{P}$ that represents an EML process. This results in another PDS $\mathcal{P}_{\mathcal{N}}$ on which we then project out all nOC lock acquisitions and releases. The end result is that $\mathcal{P}_{\mathcal{N}}$ uses EML locks in only a *non-reentrant* fashion, which enables each EML lock to be modeled by the regular language shown in Fig. 6.3(c) in CPDSMC. Using the simpler language of Fig. 6.3(c) leads to the speedups reported in §6.6.

6.3 Nested Words

Alur and Madhusudan (2006) define a nested word to be a pair (w, ν), where w is a word $a_1 \ldots a_k$ over a finite alphabet and ν, the *nesting relation*, is a subset of $\{1, 2, \ldots, k\} \times (\{1, 2, \ldots, k\} \cup \{\infty\})$. The nesting relation denotes a set of *properly nested* hierarchical edges of a nested word. For a valid nesting relation, $\nu(i, j)$ implies $i < j$, and for all i', j' such that $\nu(i', j')$ holds and $i < i'$, then either $j < i'$ or $j' < j$. Given ν, i is a *call position* if $\nu(i, j)$ holds for some j, a *return position* if $\nu(k, i)$ holds for some k, and an *internal position* otherwise.

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

Example 6.1. *Path 1* is $\big(\, w, \{\, (1, 22), (4, 19), (7, 10), (14, 17) \,\}\big)$, where w is the list of symbols e_{main} through x_{main} of *Path 1*, and the nesting-relation entries are the positions of w that correspond to the calls to and returns from functions `main`, `testAndSet`, `get`, and `set`, in order.

A set of nested words is *regular* if it can be modeled by a *nested-word automaton* (NWA) (Alur and Madhusudan, 2006). An NWA $\mathcal{N}$ is a tuple $(Q, \Sigma, q_0, \delta, F)$, where Q is a finite set of states, Σ is a finite alphabet, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is a set of final states, and δ is a transition relation that consists of three components:

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

δ_c	δ_r	δ_i
$(q_c, c_{\text{sync}}, \boxtimes)$	$(\boxtimes, q_c, x_{\text{sync}}, q_c)$	(q, σ, q)
(q, c_f, q)	(q, q, x_f, q)	

Figure 6.4: An NWA template for the locking behavior of an EML process.

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

Starting from q_0, an NWA $\mathcal{N}$ reads a nested word *nw* $= (w, \nu)$ from left to right, and performs transitions (possibly non-deterministically) according to the input symbol and the nesting relation. That is, if $\mathcal{N}$ is in state q when reading input symbol σ at position i in *nw*, then if i is an internal or call position, $\mathcal{N}$ makes a transition to q' using $(q, \sigma, q') \in \delta_i$ or $(q, \sigma, q') \in \delta_c$, respectively. Otherwise, i is a return position. Let k be the call predecessor of i, and q_c be the state $\mathcal{N}$ was in just before the transition it made on the k^{th} symbol; then $\mathcal{N}$ uses $(q, q_c, \sigma, q') \in \delta_r$ to make a transition to q'. If, after reading *nw*, $\mathcal{N}$ is in a state $q \in F$, then $\mathcal{N}$ accepts nw (Alur and Madhusudan, 2006).

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

An NWA Template for Lock Behavior.

For an EML lock l and EML process π with set of functions *Sync* synchronized on l and set of functions *Fun* not synchronized on l, the locking behavior of π on l is defined by an NWA $\mathcal{N}\langle\pi\rangle = (Q, \Sigma, q_0, \delta, F)$, where $Q = \{\boxtimes, \square\}$, Σ is the set of control locations of π, $q_0 = \square$, $F = Q$, and δ is defined in Tab. 6.4. The transitions in Tab. 6.4 are instantiated for all $q, q_c \in Q$, $\{c_{\text{sync}}, x_{\text{sync}} \in \Sigma \mid \text{sync} \in \textit{Sync}\}$, $\{c_f, x_f \in \Sigma \mid f \in \textit{Fun}\}$, and $\sigma \in (\Sigma - \{c_{\text{sync}}, x_{\text{sync}}, c_f, x_f \mid \text{sync} \in \textit{Sync}, f \in \textit{Fun}\})$.)

$\mathcal{N}\langle\pi\rangle$ consists of two states: locked ($\boxtimes$) and unlocked ($\square$). The entry to and exit from a function `f` are denoted by e_f and x_f, respectively. When an l-

synchronized function is called, $\mathcal{N}\langle\pi\rangle$ makes a transition to the locked state via the transitions $(q, c_{\text{sync}}, \boxtimes)$. When returning from a function, the state of the caller is restored. For example, the transitions $(\boxtimes, q_c, x_{\text{sync}}, q_c)$ ensure that $\mathcal{N}_\pi\langle\texttt{P0}\rangle$ goes to state q_c of the caller.

Example 6.2. For EML process P0, *Sync* $= \{\texttt{testAndSet}, \texttt{get}, \texttt{set}\}$ and *Fun* $= \{\texttt{main}\}$. The following sequence contains the calls and returns of *Path 1* from Fig. 6.2, and shows the state of $\mathcal{N}\langle\texttt{P0}\rangle$ when processing the nested word *nw* from Ex. 6.1.

$$\underset{\square}{\widehat{}}\, e_{\text{main}}\, \underset{\square}{\widehat{}} \cdots \underset{\square}{\widehat{}}\, c_{23}\, \underset{\boxtimes}{\widehat{}} \cdots \underset{\boxtimes}{\widehat{}}\, c_{11}\, \underset{\boxtimes}{\widehat{}} \cdots \underset{\boxtimes}{\widehat{}}\, x_{\text{get}}\, \underset{\boxtimes}{\widehat{}} \cdots \underset{\boxtimes}{\widehat{}}\, c_{15}\, \underset{\boxtimes}{\widehat{}} \cdots \underset{\boxtimes}{\widehat{}}\, x_{\text{set}}\, \underset{\boxtimes}{\widehat{}} \cdots \underset{\boxtimes}{\widehat{}}\, x_{\text{testAndSet}}\, \underset{\square}{\widehat{}} \cdots \underset{\square}{\widehat{}}\, x_{\text{main}}\, \underset{\square}{\widehat{}}$$

Template Usage.

For EML process P0 from Fig. 6.1, let $\mathcal{P}\langle\texttt{P0}\rangle$ be the PDS that models P0 with the rules shown in Fig. 6.5, and let $\mathcal{N}\langle\texttt{P0}\rangle$ be the NWA that results from instantiating the above template with P0. With respect to the locking behavior of P0, $\mathcal{P}\langle\texttt{P0}\rangle$ cannot distinguish between OC and nOC lock acquisitions and releases, while $\mathcal{N}\langle\texttt{P0}\rangle$ is able to do so via its state space. The transitions $(\square, c_{\text{sync}}, \boxtimes)$ and $(\boxtimes, c_{\text{sync}}, \boxtimes)$ in δ_c are the OC and nOC lock acquisitions, respectively; and transitions $(\boxtimes, \boxtimes, x_{\text{sync}}, \boxtimes)$ and $(\boxtimes, \square, x_{\text{sync}}, \square)$ in δ_r are the nOC and OC lock releases, respectively.

We show how to combine $\mathcal{P}\langle\texttt{P0}\rangle$ and $\mathcal{N}\langle\texttt{P0}\rangle$ in §6.4 to construct another PDS $\mathcal{P}_{\mathcal{N}}\langle\texttt{P0}\rangle$, such that $\mathcal{P}_{\mathcal{N}}\langle\texttt{P0}\rangle$ contains the same behaviors as $\mathcal{P}\langle\texttt{P0}\rangle$, but is able to distinguish between the OC and nOC lock acquisitions and releases. Once such a distinction can be made, we leverage *Observation 1* to remove all nOC lock acquisitions and releases from $\mathcal{P}_{\mathcal{N}}\langle\texttt{P0}\rangle$ (§6.5). This makes it possible to model an EML lock with the trivial language shown in Fig. 6.3(c).

	Rules			Symbol
1	$\langle p, e_{\text{main}} \rangle$	$\xhookrightarrow{\text{skip}}$	$\langle p, n_{21} \rangle$	
2	$\langle p, n_{21} \rangle$	$\xhookrightarrow{\text{unitbegin}}$	$\langle p, n_{22} \rangle$	[
3	$\langle p, n_{22} \rangle$	$\xhookrightarrow{\text{lock } l}$	$\langle p, c_{23} \rangle$	(
4	$\langle p, c_{23} \rangle$	$\xhookrightarrow{\text{skip}}$	$\langle p, e_{\text{testAndSet}}\ n_{24} \rangle$	
5	$\langle p, e_{\text{testAndSet}} \rangle$	$\xhookrightarrow{\text{skip}}$	$\langle p, n_{10} \rangle$	
6	$\langle p, n_{10} \rangle$	$\xhookrightarrow{\text{lock } l}$	$\langle p, c_{11} \rangle$	(
7	$\langle p, c_{11} \rangle$	$\xhookrightarrow{\text{skip}}$	$\langle p, e_{\text{get}}\ n_{12} \rangle$	
8	$\langle p, e_{\text{get}} \rangle$	$\xhookrightarrow{\text{skip}}$	$\langle p, n_{5} \rangle$	
9	$\langle p, n_{5} \rangle$	$\xhookrightarrow{\text{read } v}$	$\langle p, x_{\text{get}} \rangle$	R_v
10	$\langle p, x_{\text{get}} \rangle$	$\xhookrightarrow{\text{skip}}$	$\langle p, \epsilon \rangle$	
11	$\langle p, n_{12} \rangle$	$\xhookrightarrow{\text{unlock } l}$	$\langle p, n_{13} \rangle$	)
12	$\langle p, n_{13} \rangle$	$\xhookrightarrow{\text{skip}}$	$\langle p, n_{14} \rangle$	
13	$\langle p, n_{14} \rangle$	$\xhookrightarrow{\text{lock } l}$	$\langle p, c_{15} \rangle$	(
14	$\langle p, c_{15} \rangle$	$\xhookrightarrow{\text{skip}}$	$\langle p, e_{\text{set}}\ n_{16} \rangle$	
15	$\langle p, e_{\text{set}} \rangle$	$\xhookrightarrow{\text{skip}}$	$\langle p, n_{7} \rangle$	
16	$\langle p, n_{7} \rangle$	$\xhookrightarrow{\text{write } v}$	$\langle p, x_{\text{set}} \rangle$	W_v
17	$\langle p, x_{\text{set}} \rangle$	$\xhookrightarrow{\texttt{skip}}$	$\langle p, \epsilon \rangle$	
18	$\langle p, n_{16} \rangle$	$\xhookrightarrow{\text{unlock } l}$	$\langle p, x_{\text{testAndSet}} \rangle$	)
19	$\langle p, x_{\text{testAndSet}} \rangle$	$\xhookrightarrow{\text{skip}}$	$\langle p, \epsilon \rangle$	
20	$\langle p, n_{24} \rangle$	$\xhookrightarrow{\text{unlock } l}$	$\langle p, n_{25} \rangle$	)
21	$\langle p, n_{25} \rangle$	$\xhookrightarrow{\text{unitend}}$	$\langle p, x_{\text{main}} \rangle$	]
22	$\langle p, x_{\text{main}} \rangle$	$\xhookrightarrow{\text{skip}}$	$\langle p, \epsilon \rangle$	
23	$\langle p, n_{13} \rangle$	$\xhookrightarrow{\text{skip}}$	$\langle p, x_{\text{testAndSet}} \rangle$	

Figure 6.5: PDS rules that encode EML process P0 from Fig. 6.1. PDS stack symbols e_f and x_f denote entry and exit to the function f, respectively, and stack symbols n_i and c_i denote are step and call nodes subscripted by their line number, respectively. The run $[r_1, \ldots, r_{22}]$ corresponds to *Path 1* from Fig. 6.2 in §6.2.

6.4 Combining an NWA with a PDS

We first define the notion of the *nested-word language* (NWL) of a PDS. Defining the NWL of a PDS establishes a relationship between the NWA and PDS formalisms. Additionally, it allows for formal reasoning about the construction that creates a PDS $\mathcal{P}_{\mathcal{N}}$ to simultaneously model a PDS $\mathcal{P}$ and an NWA $\mathcal{N}$.

The Nested-Word Language of a PDS

For a PDS $\mathcal{P} = (\mathsf{P}, \Gamma, \mathsf{Lab}, \Delta, c_0)$, if (w, v) is a nested word in the NWL for $\mathcal{P}$, w consists of the sequence of left-hand-side stack symbols $\gamma_1 \ldots \gamma_j$ for a run $\rho = [r_1, \ldots, r_j]$ of $\mathcal{P}$ from some configuration.[1]

The PDS formalism does not have a natural way of modeling an NWA's ability to inspect the state of the caller when returning (i.e., a natural counterpart to δ_r). The construction that follows uses the PDS stack to achieve the same result. To do so, stack symbols that "remember" the caller's state must be created. These symbols are bookkeeping symbols and should not be considered part of a run of a PDS. Thus, we slightly augment our definition of a PDS so that the stack alphabet Γ is composed of two sets, Γ^α and Γ^β, such that $\Gamma^\alpha \cap \Gamma^\beta = \emptyset$. The two sets distinguish between the "<u>a</u>ctual" and "<u>b</u>ookkeeping" stack symbols. If left unspecified, $\Gamma^\beta = \emptyset$. Additionally, a bookkeeping symbol can only appear on the left-hand side of a step rule.

We define the function $\mathsf{nwpost}(\rho)$ that generates a nested word $nw = (w, v)$ for a run $\rho = [r_1, \ldots, r_j]$. The function $\mathsf{nwpost}(\rho)$ makes use of the helper function $\mathsf{nwpost}[r]((w, v))$ that on input (w, v) produces a new nested word to reflect the execution of the rule r.

[1] Recall that a run is a sequence of PDS rules $[r_1, \ldots, r_j]$, where the first configuration of r_1 is the initial configuration c_0 of $\mathcal{P}$.

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

Using $\mathsf{nwpost}[r]$, $\mathsf{nwpost}([r_1 \dots r_j])$ is defined as follows:[2]

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

Definition 6.3. For a PDS $\mathcal{P}$, $\mathsf{NWLang}(\mathcal{P}) = \{\, \mathsf{nwpost}(\rho) \mid \rho \in \mathsf{Runs}(\mathcal{P}) \,\}$.

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

Construction 1

Construction 1 combines a PDS $\mathcal{P}$ with an NWA $\mathcal{N}$ to produce a new PDS $\mathcal{P}_{\mathcal{N}}$ such that $\mathcal{P}_{\mathcal{N}}$ is able to model both $\mathcal{P}$ and $\mathcal{N}$ simultaneously. That is, the construction ensures that $\mathsf{NWLang}(\mathcal{P}_{\mathcal{N}}) = \mathsf{NWLang}(\mathcal{P}) \cap \mathsf{NWLang}(\mathcal{N})$.

A PDS $\mathcal{P} = (P, \Gamma, \mathsf{Lab}, \Delta, \langle p_0, \gamma_0\rangle)$ and an NWA $\mathcal{N} = (Q, \Sigma, q_0, \delta, F)$, where $\Gamma = \Sigma$, are combined to form a new PDS $\mathcal{P}_{\mathcal{N}} = (P_{\mathcal{N}}, \Gamma_{\mathcal{N}}, \mathsf{Lab}, \Delta_{\mathcal{N}}, \langle (p_0, q_0), \gamma_0\rangle)$, where

[2] $\mathsf{nwpost}[r](nw)$ is not always defined because of *max*; and thus neither is nwpost. However, for a run of a PDS from the initial configuration, both will always be defined.

- $P_{\mathcal{N}} \subseteq (P \times Q) \cup (P \times (Q \times \Gamma))$. The set of control locations $(P \times Q)$ merely combines the control locations P of $\mathcal{P}$ with the states Q of $\mathcal{N}$. The set of control locations $(P \times Q \times \Gamma)$ are used by a pop rule to record the stack symbol $\gamma \in \Gamma$ that was popped off of the stack. Recall that a transition in the return transition relation δ_r is of the form (q_r, q_c, γ, q). The elements q_r and γ are recorded in a control location of the form (p, q_r, γ).

- $\Gamma_{\mathcal{N}} = \Gamma^{\alpha}_{\mathcal{N}} \cup \Gamma^{\beta}_{\mathcal{N}}$, where $\Gamma^{\alpha}_{\mathcal{N}} = \Gamma$ is the stack alphabet of $\mathcal{P}$ (or equivalently, the input alphabet of $\mathcal{N}$), and $\Gamma^{\beta}_{\mathcal{N}} = \Gamma \times Q$ is a set of newly introduced stack symbols that are used to record the state of $\mathcal{N}$ when a function call occurs. Following the discussion above, a pop rule returns to a control location of the form (p, q_r, γ). The push rule of the caller will use a modified return location of the form (γ_r, q_c). When the callee returns (i.e., the pop rule fires), the top of the stack will then contain (γ_r, q_c), and the control location of $\mathcal{P}_{\mathcal{N}}$ will be of the form (p, q_r, γ). All inputs to δ_r are now present (i.e., q_r, q_c, and γ), and $\mathcal{P}_{\mathcal{N}}$ will make a transition to a configuration $\langle (p, q), \gamma_r \rangle$ using $(q_r, q_c, \gamma, q) \in \delta_r$. The rules to implement such a transition are discussed below.

- $\Delta_{\mathcal{N}}$ is defined as follows:

  1. For $r = \langle p, n_1 \rangle \hookrightarrow \langle p', n_2 \rangle \in \Delta_1$ and $t = (q, n_1, q') \in \delta_i$, $\langle (p, q), n_1 \rangle \hookrightarrow \langle (p', q'), n_2 \rangle \in \Delta_{\mathcal{N}}$. These rules implement the standard cross-product construction.
  2. For $r = \langle p, n_c \rangle \hookrightarrow \langle p', e\ r_c \rangle \in \Delta_2$ and $t = (q_c, n_c, q) \in \delta_c$, $\langle (p, q_c), n_c \rangle \hookrightarrow \langle (p', q), e\ (r_c, q_c) \rangle \in \Delta_{\mathcal{N}}$, where $(r_c, q_c) \in \Gamma^{\beta}$. These rules record the state q_c of the caller by pairing it with the return location r_c. Thus, when the callee returns, the state of the caller will be available to $\mathcal{P}_{\mathcal{N}}$ by extracting it from the top-of-stack symbol The need to distinguish between the stack alphabet of $\mathcal{P}$ and the "bookkeeping" symbols introduced to model $\mathcal{N}$ is why we

introduce the new stack symbols Γ^{β} and use the combined stack alphabet $\Gamma = \Gamma^{\alpha} \cup \Gamma^{\beta}$.

3. For $r = \langle p, x \rangle \hookrightarrow \langle p', \epsilon \rangle \in \Delta_0$ and $t = (q_r, q_c, x, q) \in \delta_r$, $\langle (p, q_r), x \rangle \hookrightarrow \langle (p', q_r^x), \epsilon \rangle \in \Delta_{\mathcal{N}}$, where $q_r^x \in \Gamma^{\beta}$. These rules pops the top of the stack and record, via the control location (p', q_r^x), that when making the pop, $\mathcal{N}$ was in state q_r and the top of stack was x. The recorded information is then used by the dispatch rules, defined below, to properly model δ_r of $\mathcal{N}$.
4. For each $p \in P$ and each $r_c \in \Gamma$ that occurs in Δ_2, $\Delta_{\mathcal{N}}$ contains rules of the form $\langle (p, q_r^x), (r_c, q_c) \rangle \hookrightarrow \langle (p, q), r_c \rangle$, where $q_r^x \in \Gamma^{\beta}$ and $(q_r, q_c, x, q) \in \delta_r$. These rules all $\mathcal{P}_{\mathcal{N}}$ to make a transition to the correct return point r_c of $\mathcal{P}$, and also to the correct state q of $\mathcal{N}$. The rules use the control location (p, q_r^x), as well as the return transition relation δ_r, to ensure that $\mathcal{P}$ and $\mathcal{N}$ are properly modeled.

Theorem 6.4. *An NWA $\mathcal{N}$ combined with a PDS $\mathcal{P}$ results in a new PDS $\mathcal{P}_{\mathcal{N}}$ such that $\mathit{NWLang}(\mathcal{P}_{\mathcal{N}}, (p, q)) = \mathit{NWLang}(\mathcal{P}, p) \cap \mathit{NWLang}(\mathcal{N}, q)$, for any $p \in P$ and $q \in Q$.*

Proof. (Sketch) The proof is by induction over the length of a run of the respective PDSs that derive nested words. The full proof can be found in App. A.1. □

6.5 Language Strength Reduction in EMPIRE

Thm. 6.4 shows that the PDS $\mathcal{P}_{\mathcal{N}}$ is able to model both $\mathcal{P}$ and $\mathcal{N}$ simultaneously (for nested words in their intersection). We use this operation in our method for performing language strength reduction on both the language of an EML lock and the language of the violation monitor. As noted in items 1 and 2 from §6.1, replacing a CFL with a regular language improves EMPIRE's performance.

Reducing EML Locks

To illustrate language strength reduction for EML locks we focus on EML process P0 from Fig. 6.1, whose PDS rules are given in Fig. 6.5. The first three steps are as follows:

1. PDS $\mathcal{P}\langle \mathtt{P0} \rangle = (\mathsf{P}, \Gamma, \mathsf{Lab}, \Delta, \langle \mathsf{p}, e_{\mathrm{main}} \rangle)$ is the PDS generated for EML process P0 using the original Empire translation (cf. Ch. 5, §5.7). The rule set Δ is given in Fig. 6.5.

2. NWA $\mathcal{N}$ is generated using the locking template as described in §6.3. (In general, let $\mathsf{S}_{\mathsf{Locks}}$ be the set of locks of the EML program. For each lock $\mathsf{l} \in \mathsf{S}_{\mathsf{Locks}}$, an NWA $\mathcal{N}_{\mathsf{l}}$ is generated using the NWA template with process P0 from §6.3. $\mathcal{N}$ is defined by $\bigcap_{\mathsf{l} \in \mathsf{S}_{\mathsf{Locks}}} \mathcal{N}_{\mathsf{l}}$. The state space Q of $\mathcal{N}$ is equal to $2^{|\mathsf{S}_{\mathsf{Locks}}|}$. That is, $\mathsf{Q} = \{ \boxtimes_{\mathsf{l}}, \square_{\mathsf{l}} \mid \mathsf{l} \in \mathsf{S}_{\mathsf{Locks}} \}$, and each $\mathsf{q} \in \mathsf{Q}$ represents a set of locks that are held.)

3. PDS $\mathcal{P}_{\mathcal{N}}\langle \mathtt{P0} \rangle = (\mathsf{P}_{\mathcal{N}}, \Gamma_{\mathcal{N}}, \mathsf{Lab}, \Delta_{\mathcal{N}}, \langle (\mathsf{p}, \mathsf{q}_0), e_{\mathrm{main}} \rangle)$ is generated from $\mathcal{P}\langle \mathtt{P0} \rangle$ and $\mathcal{N}$. The NWA template from §6.3 has been instantiated for EML process P0, and thus $\mathsf{NWLang}(\mathcal{N}) = \mathsf{NWLang}(\mathcal{P}_{\mathcal{N}}\langle \mathtt{P0} \rangle)$. Hence, $\mathcal{P}_{\mathcal{N}}\langle \mathtt{P0} \rangle$ contains the same behaviors as $\mathcal{P}\langle \mathtt{P0} \rangle$. Additionally, by Thm. 6.4, $\mathcal{P}_{\mathcal{N}}\langle \mathtt{P0} \rangle$ is able to distinguish between OC and nOC lock acquisitions and releases in the same manner as $\mathcal{N}$.

From Fig. 6.3(a) to Fig. 6.3(b)

The translation from P0 to PDS $\mathcal{P}\langle \mathtt{P0} \rangle = (\mathsf{P}, \Gamma, \mathsf{Lab}, \Delta, \mathsf{c}_0)$ generates a single-state PDS, i.e., $\mathsf{P} = \{\mathsf{p}\}$. The rule set Δ is given in Fig. 6.5. To call a synchronized function, the caller first acquires the lock and then calls the function. Upon returning from the callee, the lock is released. This is exemplified by PDS rules 13, 14, and 18 in Fig. 6.5 for calling the synchronized function set. The discussion that follows presents the transformation for the PDS rules that encode

(a)	(b)	(c)
$[((\mathsf{R}_v)(W_v\underline{\Sigma^*}$	$[(_{o}(_{n}\mathsf{R}_v)_{n}(_{n}W_v\underline{\Sigma^*}$	$[(_{o}\mathsf{R}_v W_v)_{o}]$

Figure 6.6: For *Path 1* of $\mathcal{P}_{\mathcal{N}}\langle\mathtt{P0}\rangle$, a prefix bound of 7, and $\rho = [r_1, \ldots, r_{22}]$ from Fig. 6.5, cols. (a) and (b) present $f(\rho)$ before and after distinguishing between OC and nOC lock acquisitions and releases, respectively. Col. (c) presents $f(\rho)$ after removing all nOC lock acquisitions and releases from $\mathcal{P}_{\mathcal{N}}\langle\mathtt{P0}\rangle$. Note that for cols. (a) and (b), the valuation is an approximation, whereas col. (c) is able to describe *Path 1* exactly within the given prefix bound.

the invocation and execution of the synchronized EML function set (rules 13–18 in Fig. 6.5). The EML labels on each rule has been replaced by their corresponding symbol (e.g., lock l with "("). We review PDS rules that acquire l, call set, return from set, and release l. Before doing so, we note that there is a mismatch between Java source code, which has synchronized methods, and EML code, which uses lock l and unlock l statements. The mismatch is because EMPIRE's front end compiles a Java source program into a lower-level format that wraps calls to synchronized methods between monitorenter and monitorexit statements. The EML statements lock l and unlock l then correspond to monitorenter and monitorexit statements, respectively. Due to the mismatch, one should view the "acquire" and "call" rules presented below as defining a single logical transition—invoking a synchronized method—via an acquire followed immediately by a call. Similarly, the "return" and "release" rules (performed in that order) also define a single logical transition—returning from a synchronized method.

acquire $\langle p, n_{14}\rangle \xhookrightarrow{(} \langle p, e_{\text{set}}\, n_{16}\rangle$, rule 13 from Fig. 6.5, acquires EML lock l before calling set.

call $\langle p, c_{15}\rangle \hookrightarrow \langle p, e_{\text{set}}\, n_{16}\rangle$, rule 14 from Fig. 6.5, calls set.

return $\langle p, x_{\text{set}}\rangle \hookrightarrow \langle p, \epsilon\rangle$, rule 17 from Fig. 6.5, returns from set.

release $\langle p, n_{16}\rangle \xhookrightarrow{)} \langle p, x_{\text{testAndSet}}\rangle$, rule 18 from Fig. 6.5, releases EML lock l upon return from `set`.

The PDS $\mathcal{P}_{\mathcal{N}}$ will have the following corresponding rules. The locking symbols "(" and ")" are annotated with the subscripts n and o to denote OC and nOC lock acquisitions and releases, respectively. Moreover, because $\mathcal{P}$ is a single-state PDS, the PDS state p has been omitted for clarity.

acquire $\langle \Box, n_{14}\rangle \xhookrightarrow{(_o} \langle \Box, c_{15}\rangle$ and $\langle \boxtimes, n_{14}\rangle \xhookrightarrow{(_n} \langle \boxtimes, c_{15}\rangle$ correspond to rule 13 listed in item "acquire" above. The lock acquisition is an OC acquisition if NWA is in the not-held state $\Box$. The NWA state *does not* change when acquiring the lock because the NWA state changes only when calling and returning from a synchronized function, which is shown in the "call" rules below. That is, the "acquire" rules defined here and the "call" rules defined next work together to define conceptually a set of single (logical) transitions that perform (i) the acquisition of a lock l, and (ii) the call to the function `set`. As discussed above, this is due to the mismatch between Java source code and EML code.

call $\langle \Box, c_{15}\rangle \hookrightarrow \langle \boxtimes, e_{\text{set}}\ (\Box, n_{16})\rangle$ and $\langle \boxtimes, c_{15}\rangle \hookrightarrow \langle \boxtimes, e_{\text{set}}\ (\boxtimes, n_{16})\rangle$ correspond to rule 14 listed in item "call" above. Because `set` is a synchronized function, the NWA makes a transition to the lock-held state $\boxtimes$ if it was in the lock-not-held state $\Box$. Together with the "acquire" rules defined above, $\mathcal{P}_{\mathcal{N}}\langle\texttt{P0}\rangle$ models both calling a synchronized function and updating the state of the NWA $\mathcal{N}$.

return $\langle \Box, x_{\text{set}}\rangle \hookrightarrow \langle \Box^{x_{\text{set}}}, \epsilon\rangle$ and $\langle \boxtimes, x_{\text{set}}\rangle \hookrightarrow \langle \boxtimes^{x_{\text{set}}}, \epsilon\rangle$ correspond to rule 17 listed in item "return" above. The stack symbol x_{set} has been paired with the PDS state (e.g., $\boxtimes^{x_{\text{set}}}$). This is required so that the "simulate" rules listed next can simulate the δ_r transition relation of $\mathcal{N}$.

simulate $\langle q, (\boxtimes, n_{16})\rangle \hookrightarrow \langle \boxtimes, n_{16}\rangle$ and $\langle q, (\Box, n_{16})\rangle \hookrightarrow \langle \Box, n_{16}\rangle$, where $q \in \{\ \boxtimes^{x_{\text{set}}}, \Box^{x_{\text{set}}}\ \}$, are the rules generated by item 4 in §6.4 to simulate the

δ_r function of $\mathcal{N}$. These rules have the effect of restoring the state of the NWA $\mathcal{N}$ to the state that it was in before the call, which corresponds to how the actual state of a Java program would change when returning from a `synchronized` method m. Specifically, if the lock was held before invoking m, then it remains held when returning from m. Otherwise, the lock was not held when invoking m, and it is released when returning from m. Combining these rules with the "release" rules defined next provides the same functionality for $\mathcal{P}_\mathcal{N}\langle\mathtt{P0}\rangle$.

release $\langle\boxtimes, \mathsf{n}_{16}\rangle \xhookrightarrow{)_\mathsf{n}} \langle\boxtimes, x_{\text{testAndSet}}\rangle$ and $\langle\square, \mathsf{n}_{16}\rangle \xhookrightarrow{)_\mathsf{o}} \langle\square, x_{\text{testAndSet}}\rangle$ correspond to rule 18 listed in item "unlock" above. The NWA state signifies whether the lock l was held before making the call. Hence, the latter rule is an OC lock release because the caller did not hold the lock. Note that the NWA state is not updated in a "release" rule. Updating the NWA state is the job of the "simulate" rules.

We view the additional simulation rules as a non-observable step of $\mathcal{P}_\mathcal{N}$. That is, in the rules listed above, one should consider a simulation rule followed by a lock release rule as one logical step, which explains why the rule $\mathsf{r} = \langle\square, \mathsf{n}_{16}\rangle \xhookrightarrow{)_\mathsf{o}} \langle\square, x_{\text{testAndSet}}\rangle$ releases the lock l, denoted by $)_\mathsf{o}$, even though its from-state is $\square$. In other words, the from-state $\square$ of r is a record that *before* invoking EML function `set`, the caller did not hold the lock.

Performing the annotation on the locking symbols of the rules involves a homomorphism, with respect to lock acquisitions and releases, that maps the language of $\mathcal{P}_\mathcal{N}$ to the language of $\mathcal{P}$ (i.e., from Fig. 6.3(b) to Fig. 6.3(a)). The homomorphism is defined by $[(_\mathsf{o} \mapsto (, (_\mathsf{n} \mapsto (,)_\mathsf{o} \mapsto),)_\mathsf{n} \mapsto)]$. Thus, there is an inverse homomorphism from the language of $\mathcal{P}$ to the language of $\mathcal{P}_\mathcal{N}$.

From Fig. 6.3(b) to Fig. 6.3(c)

Once $\mathcal{P}_\mathcal{N}$ is able to distinguish between OC and nOC lock acquisitions and releases, we leverage *Observation 1* to remove all nOC lock acquisitions and releases. Namely, the locking symbols $(_n$ and $)_n$ are simply replaced by τ, which is the internal action symbol of a CPDS. Performing this transformation induces a homomorphism from the language defined by Fig. 6.3(b) to the language defined byFig. 6.3(c): $[(_n \mapsto \tau,)_n \mapsto \tau]$ and all other symbols map to themselves. This is illustrated by the words shown in Fig. 6.6, cols. (b) and (c). Note that the word shown in column (c) is *not* an approximation like those in columns (a) and (b). This is because the word that describes *Path 1* is shorter after performing language strength reduction.

Removing nested parentheses that denote nOC acquisitions and releases guarantees that all lock acquisitions and releases modeled by $\mathcal{P}_{\mathcal{N}}$ are OC. Thus, all EML locks can now be modeled in the CPDS by a finite-state machine for the trivial language shown in Fig. 6.3(c).

Reducing the Violation Monitor

We also apply language strength reduction to the language of the violation monitor. Just as an NWA can capture the locking behavior of an EML process (cf. the NWA locking template shown in Tab. 6.4), it can also capture the unit-of-work behavior of an EML process. In fact, the NWA locking template shown in Tab. 6.4 can also be used to define the unit-of-work behavior of an EML process. To do so, we replace the set of methods *Sync* with the unit-of-work methods. Thus, the previous discussion that (i) defines the PDS $\mathcal{P}_{\mathcal{N}}\langle \mathtt{P0} \rangle$ from PDS $\mathcal{P}$ and NWA $\mathcal{N}$, and (ii) removes the nOC lock acquisitions and releases already handles removing non-state-changing unit-of-work symbols from the traces of $\mathcal{P}$. That is, we simply treat the target EML process—P0 in the discussion above—as having an extra "unit-of-work" lock whose state is updated when calling and returning from unit-of-work methods.

After applying the transformation, the language of the violation monitor is now regular, i.e., it no longer requires a stack to count the depth of nested calls to unit-of-work methods because all nested unit-of-work symbols have been removed from the traces of the target EML process. The end result is that the violation monitor's language is now regular. For the AS-serializability violation discussed in Chs. 2 and 5, the NFA $\mathcal{A}_{6.7}$ shown in Fig. 6.7 accepts the language of the reduced violation monitor. Comparing $\mathcal{A}_{6.7}$ with the NFA $\mathcal{A}_{12}$—the NFA that defines the finite-state portion of $\mathcal{P}_{\mathsf{mon}}$ in §5.7 on page 94—we see that the "reset" states have been removed. In their place, $\mathcal{A}_{6.7}$ uses non-determinism to "guess" when a violation will occur. That is, when the target EML process begins a unit of work, denoted by the "[" symbol, $\mathcal{A}_{6.7}$ is in state q_2 and can make a transition to state q_3—it has guessed that an AS-serializability violation is about to happen—or follow the self-loop and remain in state q_2—it has guessed that an AS-serializability violation will not occur during this unit of work. If $\mathcal{A}_{6.7}$guesses incorrectly that an AS-serializability violation will occur, then it will become stuck in a state $q_{3\text{–}6}$ when the target EML process exits the unit of work because there is no outgoing edge from these states that is labeled with the symbol "]" (the label Λ that annotates the self-loop on states $q_{3\text{–}6}$ represents $\Sigma \setminus \{\,]\,\}$). Thus, if the target EML process finishes the unit of work and attempts to synchronize with the violation monitor on the action], it will not be able to do so.

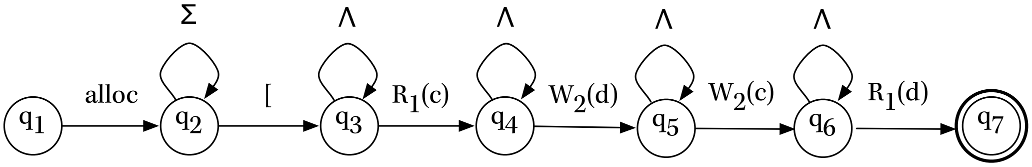

Figure 6.7: The NFA that recognizes the language of the violation monitor from Ch. 5 after language-strength reduction has been performed. Σ denotes the input alphabet, and Λ is defined as $\Sigma \setminus \{]\}$. Once the NFA guesses that a violation will occur by making a transition to state q_3, 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}.

Summary

For both EML locks and the violation monitor, we showed how to (i) construct a PDS $\mathcal{P}_{\mathcal{N}}\langle \mathsf{P0} \rangle$ and then (ii) remove nOC acquisitions and releases. After these steps, the languages of the residual EML lock processes and the violation monitor are regular: in other words, language strength reduction has replaced the original CFLs with regular languages (while each original PDS $\mathcal{P}$ for an EML process has been replaced by $\mathcal{P}_{\mathcal{N}}$, from which nOC acquisitions and releases have been removed). In addition, because the open and close-parenthesis symbols for nOC acquisitions and releases have been removed, a path in an EML process that uses reentrant locking is now described by a shorter word than when nOC acquisitions and releases are observable. This is illustrated in Fig. 6.6. In fact, there is now no cost in the WPDS weight domain (i.e., the prefix abstraction) to model a successive synchronized call, including recursive synchronized functions.

The same holds for the language of the violation monitor—the symbols for nested calls and returns of unit-of-work methods have been removed. As for locks, removing nested symbols allows for the language of the violation monitor to be regular. Moreover, the finite-state portion $\mathcal{A}_{\mathsf{mon}}$ that was used in §5.7 of Ch. 5 is also simpler because the "reset" states are no longer needed. Finally, in some cases, the CPDS model checker can find the same counterexample using a smaller bound k.

6.6 Experiments

We implemented *Construction 1* and the transformations from §6.5 in EMPIRE. We reanalyzed the 4583 transformed CPDSs that were generated for the eight CONTEST benchmark programs used in the experimental evaluation of the techniques from Ch. 5. All experiments were run on the same dual-core 3 GHz Pentium Xeon processor with 4 GB of memory.

The total time taken by CPDSMC to analyze all 4583 transformed CPDSs,

Benchmark	# CPDSs	Viol	OK	OOM	OOT
account	642	5	78	50↑	509↓
airlineTickets	900	12	882	0	6
PingPong	384	29	349	0	6
ProducerConsumer	512	32↑	110↑	85↑	285↓
SoftwareVerificationHW	15	4↑	5	0	6↓
BugTester	615	0	460↑	155↑	0↓
BuggyProgram	615	0	599	0	16
shop	900	6↑	0	839↑	55↓
Totals	4583	88↑	2483↑	1129↑	883↓

Table 6.1: Column "Benchmark" specifies the names of the eight ConTest benchmark programs analyzed. Column "# CPDSs" specifies the number of CPDSs generated. Column "Viol" specifies the number of AS-serializability violations detected. Column "OK" specifies the number of CPDS queries that reported no AS-serializability violation. Column "OOM" specifies the number of CPDS queries that exhausted memory (OOM). Column "OOT" specifies the number of CPDS queries that exhausted the 300-second timeout. The horizontal line after row 4 separates the benchmarks that did not contain any synchronization operations after abstraction from those that still contained synchronization operations. An up arrow (↑) denotes that a table entry is higher when compared to Tab. 5.4. Similarly, a down arrow (↓) denotes that a table entry is lower when compared to Tab. 5.4.

including queries on which it timed out, was around 349,100 seconds. In comparison, the total time taken by Cpdsmc to analyze the untransformed CPDSs—i.e., the total time for the experiments in Ch. 5—was around 589,900 seconds. Comparing the total times, we see that applying the language-strength-reduction transformation provided an overall speedup of around 1.7.

Tab. 6.1 presents a summary table using the same format as in §5.8 from Ch. 5. For comparing the totals, Cpdsmc performs better when analyzing the transformed CPDSs if it (i) finds more violations, (ii) proves more CPDS queries are unreachable, (iii) times out less often, or (iv) runs out of memory instead of timing out. It is better for CPDSMC to exhaust memory resources than time resources because we use a maximum prefix depth of 30 (which roughly equates to 3 GB of memory).[3] Thus, exhausting memory resources is an indication that the CPDS being analyzed is a difficult CPDS because it requires a more precise abstraction than CPDSs for which CPDSMC returned a definitive answer. In addition, when CPDSMC exhausts its available memory resources, it has explored the reachable configuration space of the input CPDS further than if it has exhausted the allotted time. In Tab. 6.1, an arrow next to a table entry denotes the relative change when compared to the corresponding entry in Tab. 5.4 (a table entry with no arrow means that the numbers are the same). Because down arrows (↓) only occur on the timeout column, table entries that are annotated with arrows indicate totals where CPDSMC performed better when analyzing transformed CPDSs (CPDSMC never performed worse when analyzing the transformed models).

If we only consider the four benchmarks whose EML code contains synchronization operations—the bottom four benchmarks on Tab. 6.1—CPDSMC took roughly 92,400 seconds and 298,100 seconds to analyze the transformed and original CPDSs, respectively, which is a total speedup of 3.2. Breaking it down further, if we only consider the benchmarks that contain synchronization operations and CPDSMC returned a definitive answer on *both* the transformed and original models, the total time taken by CPDSMC was roughly 4,400 seconds and 7,700 seconds to analyze the transformed and original CPDSs, respectively, which is a total speedup 1.8.

Fig. 6.8 presents log-log scatter plots that compare the execution times of CPDSMC on the transformed models (y-axis) and the original models (x-axis). The 300-second timeout is denoted by the horizontal and vertical lines that form a box in each plot. The dashed-diagonal line denotes equal running times. Points below and to the right are queries on which CPDSMC was faster on the transformed models. The queries are categorized according to the result returned by CPDSMC when analyzing a transformed model. The plot labeled "OOM" are

[3] The experiments only use 3 GB of memory because the operating system reserves 1 GB for its own use.

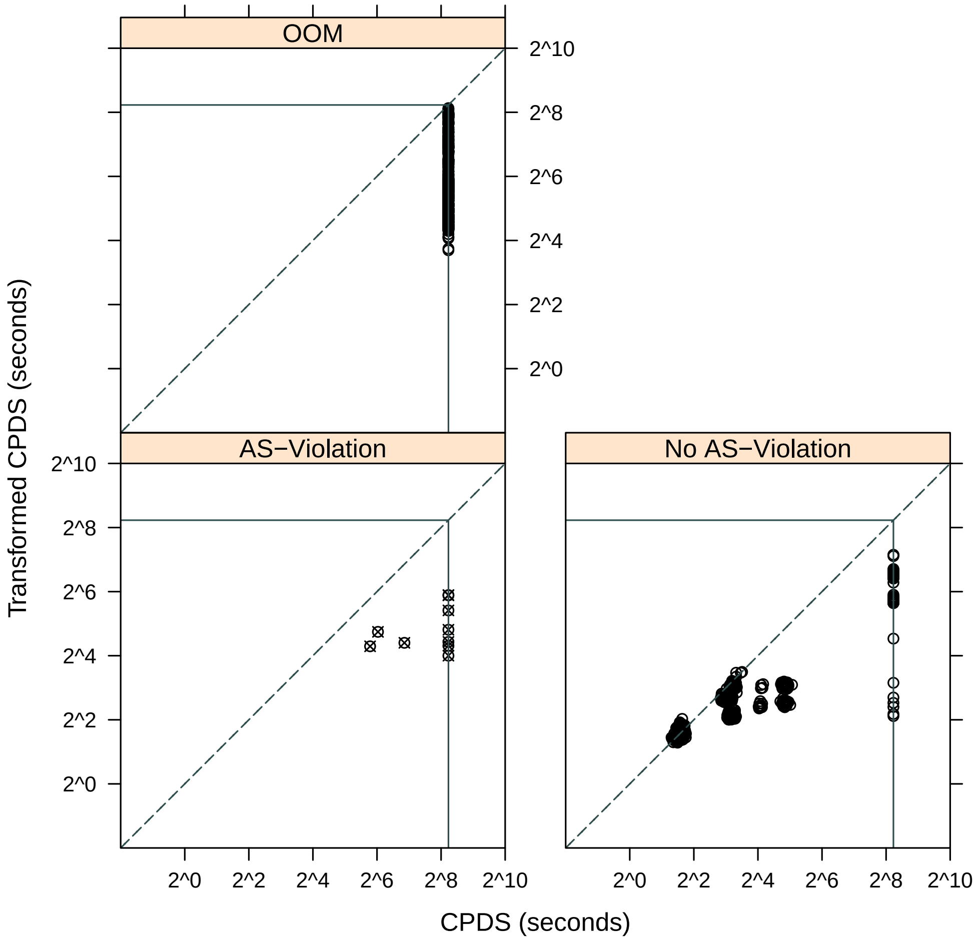

Figure 6.8: Log-log scatter-plots of the Cpdsmc's execution times for queries generated from the four EML programs that contain synchronization operations. The y-axis is the execution time for the transformed CPDSs (y-axis), and the x-axis is the execution time for the original CPDSs using *Individual Multi-step* SDP presented in Ch. 4 (x-axis). The queries are categorized according to the result returned by Cpdsmc on the transformed models. The top plot shows those queries on which Cpdsmc exhausted memory on the transformed models and exhausted the 300-second timeout on the original models. The (lower) left-hand plot shows the queries on which Cpdsmc found an AS-serializability violation. The right-hand plot shows queries on which Cpdsmc found no violation on the transformed models. The 300-second timeout is denoted by the horizontal and vertical lines that form a box in each of the plots. The dashed-diagonal line denotes equal running times: points below and to the right of the dashed lines are runs for which Cpdsmc was faster on the transformed models.

queries on which Cpdsmc ran out of memory. For each of these queries, Cpdsmc exhausted the 300-second timeout when analyzing the original queries. The plot labeled "AS-Violation" shows queries where Cpdsmc found a violation. The vertical band on the right are queries in which Cpdsmc found a violation on the transformed models, but ran out of time on the original models. The plot labeled "No AS-Violation" shows queries on which Cpdsmc reported no violation. The vertical band on the right again marks queries on which Cpdsmc exhausted the 300-second timeout. Overall, the plots show that Cpdsmc, when analyzing the transformed models, was able (i) to find more violations (in examples that contain violations), and (ii) to show that more examples did not contain the AS-serializability violation of interest (in examples free of the specified violations). Moreover, for many queries, when analyzing the transformed models, Cpdsmc exhausted memory resources whereas it exhausted time resources on the untransformed models (plot "OOM"). As explained above, the answer "OOM" is better than "OOT" because it means that Cpdsmc was able to explore more of the state space.

6.7 Combining an NWA with an EWPDS

This section presents a construction that combines an EWPDS $\mathcal{E}$ (defined next) with an NWA $\mathcal{N}$ to produce another EWPDS $\mathcal{E}_{\mathcal{N}}$. With respect to Empire and language strength reduction, $\mathcal{E}_{\mathcal{N}}$ provides a mechanism to simulate *symbolically* the transition relation δ of $\mathcal{N}$. The ability to combine an EWPDS with an NWA is of general interest because of the reasons stated at the end of §6.1, namely it advances the state of the art in property checking of programs. In this case, the property (e.g., a safety property) is expressed as an NWA and the program is modeled as an EWPDS.

Symbolic simulation might also be required to compute a reachability query. Recall that when combining a PDS $\mathcal{P} = (P, \Gamma, \mathsf{Lab}, \Delta, \langle p_0, \gamma_0 \rangle)$ with an NWA $\mathcal{N} = (Q, \Sigma, q_0, \delta, F)$ to produce a PDS $\mathcal{P}_{\mathcal{N}} = (P_{\mathcal{N}}, \Gamma_{\mathcal{N}}, \mathsf{Lab}, \Delta_{\mathcal{N}}, \langle (p_0, q_0), \gamma_0 \rangle$, the state space $P_{\mathcal{N}}$ has size proportional to $|P| \times |Q| \times |\Gamma|$. When the size of $|Q|$ is large—for language strength reduction $|Q| = 2^{S_{\text{Locks}}}$—explicit modeling of the state can cause a simple *post** query to become infeasible because reachability queries on PDSs are quadratic in the number of control locations (e.g., $P_{\mathcal{N}}$) (Schwoon, 2002). Symbolic techniques, such as BDDs (Bryant, 1986), can address the state-space problem because they can provide an exponential space savings. Furthermore, the use of symbolic techniques allows for simulating the transition relation of $\mathcal{N}$ directly instead of the multi-step simulation—i.e., the need to define the "simulation" rules in §6.4—required by the explicit simulation of PDS $\mathcal{P}_{\mathcal{N}}$.

Extended Weighted Pushdown Systems

Extended weighted pushdown systems (EWPDSs) are an extension to WPDSs developed by (Lal et al., 2005).[4] An EWPDS is obtained by augmenting a WPDS with a set of merge functions (Lal et al., 2005). Recall that weights encode the effect that each statement (or PDS rule) has on the data state of the program (cf. Ch. 3). Merge functions are used to fuse the local state of the calling procedure as it existed just before the call with the global state produced by the called procedure. This capability is in similar spirit the an NWA's ability to inspect the state of the caller when modeling a return. With respect to EMPIRE, merge functions can be used to eliminate the lock acquire and release rules that wrap a call to a synchronized function (see items "acquire" and "release" in §6.5) because a single call rule can acquire a lock and use the merge function to release a lock. Before formally defining merge functions and EWPDSs, we briefly review the prefix weight domain defined by Defn. 4.8 in Ch. 4.

Example 6.5. *The Prefix Weight Domain for CPDSs.* For a CFL L over finite alphabet Σ, the prefix abstraction precisely models each word $w \in L$ whose length is less than a bound k. If $|w| \geqslant k$, then w is approximated by the regular lan-

[4] Due to the technical nature of the following material, the reader may want to reacquaint themselves with the definition of WPDSs (Ch. 3), and the definition of the weight domain that implements the prefix abstraction as used by CPDSMC (Ch. 4).

guage $\lfloor w \rfloor^k \Sigma^*$, where $\lfloor w \rfloor^k$ denotes the prefix of w of length k. For two words $w_1 = a_1 \ldots a_i$ and $w_2 = b_1 \ldots b_j$, let $w_1 \bowtie_k w_2$ be the word $\lfloor a_1 \ldots a_i b_1 \ldots b_j \rfloor^k$. We extend $\bowtie_k$ to finite sets in the obvious way. For a finite alphabet Σ and bound k, let D be the powerset of $\bigcup_{0 \leqslant i \leqslant k} \Sigma^i$. The prefix weight domain is defined as $\mathcal{S}_{\alpha_k} = (D, \cup, \bowtie_k, \emptyset, \{\epsilon\})$.

Definition 6.6. A function $m : D \times D \to D$ is a **merge function** with respect to a semiring $\mathcal{S} = (D, \oplus, \otimes, \overline{0}, \overline{1})$ if it satisfies the following properties:

1. **Strictness.** For all $a \in D$, $m(\overline{0}, a) = m(a, \overline{0}) = \overline{0}$.

2. **Distributivity.** The function distributes over $\oplus$. For all $a, b, c \in D$,

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

Example 6.7. *The Prefix Merge Functions of Empire.* EMPIRE could use prefix merge functions of the form $\lambda d_1.\lambda d_2.d_1 \otimes d_2 \otimes d_{\mathrm{const}}$, where d_{const} is either $\overline{1}$ for invoking non-synchronized functions, or $\{\,)\,\}$ for invoking a function that is synchronized on a lock l. Placing the close-parenthesis symbol, corresponding to the release of a lock, inside of a merge function accurately reflects the behavior of returning from a synchronized function.

Definition 6.8. Let $\mathcal{M}$ be the set of all merge functions on weight domain $\mathcal{S}$, and let Δ_2 denote the set of push rules of a PDS $\mathcal{P}$. An **extended weighted pushdown system** is a quadruple $\mathcal{E} = (\mathcal{P}, \mathcal{S}, f, g)$ where $\mathcal{P} = (P, \Gamma, \mathsf{Lab}, \Delta, c_0)$ is a PDS, $\mathcal{S} = (D, \oplus, \otimes, \overline{0}, \overline{1})$ is a weight domain, $f : \Delta \to D$ is a map that assigns a weight to each rule of $\mathcal{P}$, and $g : \Delta_2 \to \mathcal{M}$ assigns a merge function to each rule in Δ_2.

Example 6.9. *An EWPDS for an EML process.* For an EML process π, an EWPDS $\mathcal{E}\langle\pi\rangle = (\mathcal{P}, \mathcal{S}, f, g)$ is generated where $\mathcal{P}$ is defined from EML process π using the technique described in §5.7, $\mathcal{S}$ is the prefix weight domain, and g assigns prefix merge functions to the Δ_2 rules. Fig. 6.9 presents the rules that would encode process P0 from Fig. 6.1. When compared to the PDS rules in Fig. 6.5, one can see that the lock acquire and release rules that wrap a call to a synchronized function are removed. Instead, their functionality is handled by a call rule whose weight acquires a lock and whose merge function releases a lock.[5] For example, the EWPDS rule 10 Fig. 6.9 that models calling the `set` function (i) acquires the lock, (ii) calls `set`, and (iii) releases the lock when `set` returns. In Fig. 6.5, the same behavior required three separate rules, namely, rules 13, 14, and 18.

Run of an EWPDS**.**

A run of an EWPDS $\mathcal{E}$ is simply a run of its underlying PDS. We denote (i) the set of all runs of $\mathcal{E}$ by $\mathsf{Runs}(\mathcal{E})$, (ii) the set of runs ending in configuration c as $\mathsf{Runs}(\mathcal{E}, c)$, and (iii) the set of runs ending in some configuration c from a set of configurations C as $\mathsf{Runs}(\mathcal{E}, C)$. Using f and g, we can associate a value to a run ρ, denoted by $\mathit{val}(\rho)$. To do so, we define the helper functions $\mathit{val}[r]$, *build*, and *flatten*. The function $\mathit{val}[r](z, S)$ takes a weight and a weight-rule stack, and returns a weight and weight-rule stack:

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

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

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

[5] To acquire and release a lock using a single call rule, we would need to modify EML to support `synchronized` functions instead of having explicit lock acquire and release statements.

	Rules	Weight	d_{const}
1	$\langle p, e_{\text{main}} \rangle \xhookrightarrow{\text{skip}} \langle p, n_{21} \rangle$	$\overline{1}$	
2	$\langle p, n_{21} \rangle \xhookrightarrow{\text{unit}} \langle p, c_{23}\ x_{\text{main}} \rangle$	$\{ [\}$	$\{] \}$
3	$\langle p, c_{23} \rangle \xhookrightarrow{\text{sync } l} \langle p, e_{\text{testAndSet}}\ x'_{\text{main}} \rangle$	$\{ (\}$	$\{) \}$
4	$\langle p, e_{\text{testAndSet}} \rangle \xhookrightarrow{\text{skip}} \langle p, c_{11} \rangle$	$\overline{1}$	
5	$\langle p, c_{11} \rangle \xhookrightarrow{\text{sync } l} \langle p, e_{\text{get}}\ n_{13} \rangle$	$\{ (\}$	$\{) \}$
6	$\langle p, e_{\text{get}} \rangle \xhookrightarrow{\text{skip}} \langle p, n_5 \rangle$	$\overline{1}$	
7	$\langle p, n_5 \rangle \xhookrightarrow{\text{read } v} \langle p, x_{\text{get}} \rangle$	$\{R_v\}$	
8	$\langle p, x_{\text{get}} \rangle \xhookrightarrow{\text{skip}} \langle p, \epsilon \rangle$	$\overline{1}$	
9	$\langle p, n_{13} \rangle \xhookrightarrow{\text{skip}} \langle p, c_{15} \rangle$	$\overline{1}$	
10	$\langle p, c_{15} \rangle \xhookrightarrow{\text{sync } l} \langle p, e_{\text{set}}\ x_{\text{testAndSet}} \rangle$	$\{ (\}$	$\{) \}$
11	$\langle p, e_{\text{set}} \rangle \xhookrightarrow{\text{skip}} \langle p, n_7 \rangle$	$\overline{1}$	
12	$\langle p, n_7 \rangle \xhookrightarrow{\text{write } v} \langle p, x_{\text{set}} \rangle$	$\{ W_v \}$	
13	$\langle p, x_{\text{set}} \rangle \xhookrightarrow{\text{skip}} \langle p, \epsilon \rangle$	$\overline{1}$	
14	$\langle p, x_{\text{testAndSet}} \rangle \xhookrightarrow{\text{skip}} \langle p, \epsilon \rangle$	$\overline{1}$	
15	$\langle p, x'_{\text{main}} \rangle \xhookrightarrow{\text{skip}} \langle p, x_{\text{main}} \rangle$	$\overline{1}$	
16	$\langle p, x_{\text{main}} \rangle \xhookrightarrow{\text{skip}} \langle p, \epsilon \rangle$	$\overline{1}$	
17	$\langle p, n_{13} \rangle \xhookrightarrow{\text{skip}} \langle p, x_{\text{testAndSet}} \rangle$	$\overline{1}$	

Figure 6.9: EWPDS rules that encode EML process P0 from Fig. 6.1 (subscripts correspond to the line numbers). Only the constant weight d_{const} is shown for the merge functions. The EML labels "sync l" and "unit" are the hypothetical modifications to EML, and denote a scoped use of lock l and a unit of work, respectively.

The function *flatten*(z, S) "flattens" a weight and weight-rule stack by using the extend ($\otimes$) operation:

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

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

Example 6.10. *Valuation of Path 1.* Using the EWPDS rules of Fig. 6.9, and for a prefix bound $k > 10$, one can verify that $\mathit{val}([r_1, \ldots, r_{14}]) = \{\ [((R_v))(W_v))]\ \}$, which is the set containing only the word given in §6.2 for *Path 1*.

Definition 6.11. For EWPDS $\mathcal{E}$ and a set of configurations C, the *combine-over-all-valid-paths* value $\mathsf{COVP}_{\mathcal{E}}(C)$ is defined as $\bigoplus\{\mathit{val}(\rho) \mid \rho \in \mathsf{Runs}(\mathcal{E}, C)\}$.

The COVP value captures the net effect of all paths leading to a set of configurations. An algorithm for computing COVP is given in Lal et al. (2005).

Example 6.12. *COVP for EML process P0 from Fig. 6.1.* Let $\mathcal{E}\langle \mathtt{P0}\rangle$ be the EWPDS for process P0 with rules given in Fig. 6.9. For a prefix bound $k > 10$, $\mathsf{COVP}_{\mathcal{E}\langle \mathtt{P0}\rangle}(\langle p, x_{\mathrm{main}}\rangle) = \{\ [((R_v)(W_v))]\ ,\ [((R_v))]\ \}$. The first string describes the path that follows the true branch of the `if` statement at line 13 in Fig. 6.1, and the second string describes the path that follows the false branch. Because process P0 has only two valid paths and $k > 10$, the COVP weight precisely describes the behavior of process P0. However, if k was instead the value 8, then the result of the same COVP computation would be $\{\ [((R_v)(W_v)\Sigma^*\ ,\ [((R_v))]\ \}$. Note that the first string has been approximated by an infinite set of strings.

Construction 2

As for the case of combining an NWA with a PDS, we begin by defining the nested-word language (NWL) of an EWPDS. The NWL of an EWPDS $\mathcal{E} = (\mathcal{P}, \mathcal{S}, f, g)$ is merely the NWL of the underlying PDS, with one additional restriction: the run ρ for which a nested word is defined must have a non-zero weight.

Definition 6.13. For an EWPDS $\mathcal{E}$, the nested-word language $\mathsf{NWLang}(\mathcal{E})$ is defined as $\mathsf{NWLang}(\mathcal{E}) = \{\ \mathsf{nwpost}(\rho) \mid \rho \in \mathsf{Runs}(\mathcal{E}) \land \mathit{val}(\rho) \neq \bar{0}\}$.

We will sometimes wish to further restrict $\mathsf{NWLang}(\mathcal{E})$ by an acceptance criterion, which we call φ-*acceptance*.

Definition 6.14. The φ-*accepted* nested-word language for an EWPDS $\mathcal{E}$ and function $\varphi : D \to \mathbb{B}$ is defined as $\mathsf{NWLang}^{\varphi}(\mathcal{E}) = \{\mathsf{nwpost}(\rho) \mid \rho \in \mathsf{Runs}(\mathcal{E}) \wedge \mathit{val}(\rho) \neq \overline{0} \wedge \ \varphi(\mathit{val}(\rho))\}$.

The construction that combines an EWPDS $\mathcal{E}$ with an NWA $\mathcal{N}$ produces another EWPDS $\mathcal{E}_{\mathcal{N}}$. The weight domain of $\mathcal{E}_{\mathcal{N}}$ models the transition relation of $\mathcal{N}$ in addition to the original weight domain of $\mathcal{E}$. This is accomplished via a *relational weight domain.*

Definition 6.15. A *weighted relation* on a set G, with weight domain $\mathcal{S} = (D, \oplus, \otimes, \overline{0}, \overline{1})$, is a function from $(G \times G)$ to D. The composition of two weighted relations R_1 and R_2 is defined as $(R_1; R_2)(g_1, g_3) = \oplus\{w_1 \otimes w_2 \mid \exists g_2 \in G : w_1 = R_1(g_1, g_2), w_2 = R_2(g_2, g_3)\}$. The union of the two weighted relations is defined as $(R_1 \cup R_2)(g_1, g_2) = R_1(g_1, g_2) \oplus R_2(g_1, g_2)$. The identity relation is the function that maps each pair (g, g) to $\overline{1}$ and others to $\overline{0}$. The reflexive transitive closure is defined in terms of these operations, as usual. If R is a weighted relation and $R(g_1, g_2) = z$, then we write $g_1 \xrightarrow{z} g_2 \in R$.

Definition 6.16. If $\mathcal{S}$ is a weight domain with set of weights D and G is a finite set, then the *relational weight domain* on $(G, \mathcal{S})$ is defined as $(2^{G \times G \to D}, \cup, ;, \emptyset, \mathrm{id})$: weights are weighted relations on G, combine is union, extend is weighted relational composition (";"), $\overline{0}$ is the empty relation, and $\overline{1}$ is the weighted identity relation on $(G, \mathcal{S})$.

This weight domain can be encoded symbolically using techniques such as algebraic decision diagrams (Bahar et al., 1993).

The weight domain of $\mathcal{E}_{\mathcal{N}}$ will be a relational weight domain on $(G, \mathcal{S})$, where G encodes the state space of $\mathcal{N}$, and $\mathcal{S}$ is the weight domain of $\mathcal{E}$. Intuitively, for a run ρ of $\mathcal{E}_{\mathcal{N}}$, the valuation $\mathit{val}(\rho)$ in $\mathcal{E}_{\mathcal{N}}$ is a weighted relation R such that if $q_1 \xrightarrow{z} q_2 \in R$, then (i) the valuation $\mathit{val}(\rho)$ in $\mathcal{E}$ must be equal to z, and (ii) starting from state q_1, $\mathcal{N}$ can make a transition to state q_2 on the nested word $\mathsf{nwpost}(\rho)$. We now introduce some notation needed to define how this is accomplished by the construction.

First, for an NWA $\mathcal{N} = (Q, \Sigma, q_0, \delta, F)$, we define $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$. The relational weight domain of $\mathcal{E}_\mathcal{N}$ is over the finite set $Q \times \Sigma_\epsilon$. The pairing of Q with Σ_ϵ is used below to properly model the return relation δ_r of $\mathcal{N}$. We denote an element (q, σ) of this set by q^σ, but omit σ when $\sigma = \epsilon$.

Second, we define the restriction of δ_i to σ, denoted by $\delta_i^{|\sigma}$, to be the relation with $(q_1, q_2) \in \delta_i^{|\sigma}$ iff $(q_1, \sigma, q_2) \in \delta_i$. Note that by representing (q_1, q_2) as $(q_1^\epsilon, q_2^\epsilon)$, $\delta_i^{|\sigma}$ can be embedded into $(Q \times \Sigma_\epsilon) \times (Q \times \Sigma_\epsilon)$ using only states in which $q \in Q$ is paired with ϵ (i.e., q^ϵ). Henceforth, we abuse notation and use $\delta_i^{|\sigma}$ to mean the version that is embedded in $(Q \times \Sigma_\epsilon) \times (Q \times \Sigma_\epsilon)$. We define $\delta_c^{|\sigma}$ similarly. $\delta_i^{|\sigma}$ and $\delta_c^{|\sigma}$ will be the relational part of the weights that annotate step and push rules in $\mathcal{E}_\mathcal{N}$. By restricting δ_i (δ_c) to σ, a run of $\mathcal{E}_\mathcal{N}$ enforces that $\mathcal{E}$ and $\mathcal{N}$ are kept in lock step (see *Construction 1*).

Third, we define the function *expand*(σ), which takes as input a symbol $\sigma \in \Sigma$ and generates the relation $\{(q^\epsilon, q^\sigma) \mid q \in Q\}$. This is used to pass the return location to $\mathcal{E}_\mathcal{N}$'s merge functions, which is needed for properly modeling the return relation δ_r of $\mathcal{N}$.

Fourth, we define $\hat{\delta}$ so that $(q_r^\sigma, q_c, q) \in \hat{\delta}$ iff $(q_r, q_c, \sigma, q) \in \delta_r$. Notice that $\hat{\delta}$ combines the input symbol σ used in δ_r with the return state. This is used by $\mathcal{E}_\mathcal{N}$'s merge functions to receive the return location passed via *expand*.

Construction 2. The combination of an EWPDS $\mathcal{E} = (\mathcal{P}, \mathcal{S}, f, g)$ and an NWA $\mathcal{N} = (Q, \Sigma, q_0, \delta, F)$ is modeled by an EWPDS $\mathcal{E}_\mathcal{N}$ that has the same underlying PDS as $\mathcal{E}$, but with a new weight domain and new assignments of weights and merge functions to rules: $\mathcal{E}_\mathcal{N} = (\mathcal{P}, \mathcal{S}_\mathcal{N}, f_\mathcal{N}, g_\mathcal{N})$, where $\mathcal{S}_\mathcal{N} = (D_\mathcal{N}, \oplus_\mathcal{N}, \otimes_\mathcal{N}, \bar{0}_\mathcal{N}, \bar{1}_\mathcal{N})$ is the relational weight domain on the set $Q \times \Sigma_\epsilon$ and weight domain $\mathcal{S}$, and $f_\mathcal{N}$ and $g_\mathcal{N}$ are defined as follows:

1. For step rule $r = \langle p, n_1 \rangle \hookrightarrow \langle p', n_2 \rangle \in \Delta$, $f_\mathcal{N}(r) = \{q_1 \xrightarrow{f(r)} q_2 \mid (q_1, q_2) \in \delta_i^{|n_1}\}$.

2. For push rule $r = \langle p, n_c \rangle \hookrightarrow \langle p', e\, r_c \rangle \in \Delta$, $f_\mathcal{N}(r) = \{q_1 \xrightarrow{f(r)} q_2 \mid (q_1, q_2) \in \delta_c^{|n_c}\}$ and $g_\mathcal{N}(r)(w_c, w_x) = \{q_1 \xrightarrow{z} q_2\}$, where q_1, q_2, and z are defined

by:

$$z \;=\; \oplus \left\{ g(r)(z_1, z_2) \mid \exists a, b : \left(\begin{array}{lll} & q_1 \xrightarrow{z_1} a & \in w_c \\ \wedge & a \xrightarrow{z_2} b & \in (f_{\mathcal{N}}(r) \otimes w_x) \\ \wedge & \hat{\delta}(b, a, q_2) & \end{array} \right) \right\} \tag{6.1}$$

3. For pop rule $r = \langle p, x \rangle \hookrightarrow \langle p', \epsilon \rangle \in \Delta$, $f_{\mathcal{N}}(r) = \{q \xrightarrow{f(r)} q^x \mid (q, q^x) \in$ *expand*$(x)\}$.

The properties of *Construction 2* are that (i) $\mathcal{E}_{\mathcal{N}}$'s nested-word language is the intersection of those of $\mathcal{E}$ and $\mathcal{N}$, and (ii) the behaviors of $\mathcal{E}_{\mathcal{N}}$ (summarized by its COVP values) are those of $\mathcal{E}$ restricted by $\mathcal{N}$. Formally, these are captured by Thm. 6.17 and Cor. 6.18.

Theorem 6.17. *An NWA $\mathcal{N}$ combined with an EWPDS $\mathcal{E}$ results in an EWPDS $\mathcal{E}_{\mathcal{N}}$ such that $\mathit{NWLang}^{\varphi}(\mathcal{E}_{\mathcal{N}}) = \mathit{NWLang}(\mathcal{N}, Q) \cap \mathit{NWLang}(\mathcal{E})$, where for a run ρ of $\mathcal{E}_{\mathcal{N}}$ with $z = \mathit{val}(\rho)$, $\varphi(z) = \exists q \in Q : q_0 \xrightarrow{y} q \in z$, and $y \neq \overline{0}$.*

Proof. The proof is given in App. A.2. □

Corollary 6.18. *An NWA $\mathcal{N}$ combined with an EWPDS $\mathcal{E}$ results in an EWPDS $\mathcal{E}_{\mathcal{N}}$ such that $\mathit{COVP}_{\mathcal{E}_{\mathcal{N}}}(C) = \bigoplus\{\mathit{val}(\rho) \mid \rho \in \mathit{Runs}(\mathcal{E}, C), \mathit{nwpost}(\rho) \in \mathit{NWLang}(A, Q)\}$.*

Complexity of $\mathcal{E}_{\mathcal{N}}$ versus $\mathcal{E}$. The complexity of computing COVP on an EWPDS is proportional to the *height* of the weight domain, which is defined to be the length of the longest descending chain in the domain.[6] If H is the height of the weight domain of $\mathcal{E}$, then the height of the weight domain of $\mathcal{E}_{\mathcal{N}}$ is $H|Q|^2$, where Q is the set of states of $\mathcal{N}$. Because $\mathcal{E}$ and $\mathcal{E}_{\mathcal{N}}$ have the same PDS, the complexity of computing COVP on $\mathcal{E}_{\mathcal{N}}$ only increases by a factor of $|Q|^2$, albeit for language strength reduction $|Q| = 2^{S_{\mathsf{Locks}}}$.

[6] EWPDSs can also be used when the height is unbounded, provided there are no infinite descending chains. To simplify the discussion of complexity, we assume the height to be finite.

Summary

We have completed the discussion of *Construction 2*. The ability to combine an EWPDS with an NWA advances the state of the art for property checking programs (described in §6.1 and explored in more detail by (Kidd et al., 2007)). Empire does not use this construction because the language-strength-reduction transformation had the beautiful side effect of transforming the program model so that the model-checking problem is *decidable*, which is discussed in Ch. 7.

6.8 Related Work

Alur and Madhusudan (2004, 2006) introduced the concept of an NWA. For program verification, they showed that a property specification and a program can be modeled by NWAs, and that verification can be solved by taking their intersection. Our work extends this result to property checking where the program is specified by an EWPDS and the property by an NWA. Because EWPDSs allow programs to be abstracted using more than just predicate-abstraction domains (i.e., abstract programs can be more than just Boolean programs), our work has broadened the class of program abstractions for which one can use an NWA as the property specification.

Chaudhuri and Alur (2007) instrument a C program with an NWA that defines a property specification. This approach diffuses the NWA throughout the program proper. Our approach combines the NWA with an EWPDS, but keeps the NWA separated by modeling it using weights. This is beneficial for reporting error-paths back to a user when model checking a C program because the internals of the NWA are not exposed in the error-path. Additionally, by keeping the NWA separated in the weight domain, one can use symbolic encoding of weights (Schwoon, 2002) for handling the potentially exponential size of the NWA.

Kahlon et al. (2005a,b); Kahlon and Gupta (2007) analyze concurrent recursive programs that use nested locking, where nested locking means that all locks are released in the opposite order in which they are acquired. Their locks, however, are not reentrant and are not syntactically scoped. If one enforces syntactically scoped locks, then one can apply our techniques for language strength reduction to model a program with reentrant locks using only non-reentrant locks. This would produce a model to which their model-checking algorithms could be applied.

7 A DECISION PROCEDURE

In this chapter, we combine the elements developed in previous chapters to show that the problem of checking AS-atomicity of an EML program is actually decidable. More precisely, we show that AS-serializability violation detection for a language with (i) reentrant locks, (ii) an unbounded number of context switches, and (iii) an unbounded number of lock acquisitions is decidable. (Actually, we show decidability for a model-checking formalism that can be used to encode AS-serializability; however, for the purposes of this introduction, we will provide intuition in terms of AS-serializability.) Because AS-serializability violations can be completely characterized by the fourteen problematic access patterns (see §2.3 of Ch. 2), this result shows that AS-atomicity verification is also decidable.

What are the Difficulties?

In Chs. 5 and 6, we addressed AS-serializability violation detection by encoding the problem as a CPDS model-checking problem. The results of the present chapter 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 only able to obtain incomplete answers for 49% of the CPDS queries posed in the experimental evaluation of Ch. 5 (see §5.8).

We discovered that the problem was decidable only after working on the problem for several years (and developing the methods discussed in Chs. 5 and 6). 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. Our initial encoding of a counter (see Defn. 4.15 in §4.6) uses the

PDS stack to count in unary the value of the counter.

2. Like EML locks, units of work are also reentrant, and hence also require a counter to track the depth of nested calls to unit-of-work methods. The counter to track the depth of nested calls to unit-of-work methods was also encoded as a PDS that uses its stack to count in unary.

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. 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 PDS transitions with respect to locking semantics, induces an apparent tight coupling between the PDSs. That is, we at first thought that the PDSs needed to synchronize their locking behaviors with the lock-PDSs, which causes synchronization between the individual PDSs, and thus a tight coupling.

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 that was presented in Ch. 6 provides a mechanism to replace reentrant locks with non-reentrant locks while still being able to explore the entire state space. That is, precision is not lost because the transformed model contains the same set of behaviors as the original model.

2. The language-strength-reduction transformation also provides a mechanism to eliminate the need to model nested calls to unit-of-work methods,

which reduces the language of a violation monitor from one that is context-free to one that is regular. Moreover, for each problematic access pattern, the NFA that recognizes traces that violate the pattern always has a special form (discussed in §7.1).

3. The techniques presented in this chapter 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), and then a post-processing step determines whether there exists an interleaved execution that satisfies the constraints computed for each PDS.

Contributions

This chapter makes the following contributions:

- We define a decision procedure for verifying AS-atomicity of an EML program. The decision procedure handles (i) reentrant locks (via language strength reduction), (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-atomicity of an EML program also verifies AS-atomicity of the concurrent Java program.
- The decision procedure is *modular*: each PDS is analyzed independently with respect to the violation monitor—the NFA that recognizes traces containing an AS-serializability violation—and then a single compatibil-

ity check is performed that ties together the results obtained from the different PDSs.

- We leverage the special form of the NFA for a violation monitor 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 the EML programs analyzed in Ch. 5. The decision procedure was *34* times faster overall (i.e., the total running time for all queries) when compared to the overall running time of CPDSMC on the corresponding set of queries. Moreover, for each query where CPDSMC timed out, which was roughly 49% of the queries, the decision procedure actually performed more work because it explored the *entire* state space.

The rest of the chapter is organized as follows: §7.1 presents a more detailed overview of the steps that were required to obtain the result that AS-serializability-violation detection is decidable. §7.2 defines multi-PDSs and IPAs. §7.3 reviews a decomposition result due to Kahlon and Gupta. §7.4 presents lock histories. §7.5 presents the decision procedure for a 2-PDS. §7.6 gives a detailed comparison with a certain decision procedure of Kahlon and Gupta. §7.7 presents a symbolic implementation of our 2-PDS decision procedure. §7.8 generalizes the 2-PDS decision procedure to handle N-PDSs. §7.9 presents experimental results. §7.10 describes related work.

7.1 The Road to Decidability

This section provides insight into how the problem can be transformed to finesse the features that make it appear to be undecidable.

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 CPDSs, global synchronization is a communicating action. For the concurrent-PDSs of Qadeer and Rehof (2005) used for context-bounded model checking, a global synchronization is a context switch. Both formalisms artificially bound the number of global synchronizations to answer a bounded-reachability query: CPDSs with the prefix abstraction and concurrent-PDSs with context bounding.

For AS-serializability-violation detection, a global synchronization occurs when the violation monitor updates its state (locks will be discussed shortly). For the Java program from Ch. 5 shown in Listing 5.1 on page 74, there are two threads, T_1 and T_2, and the atomic set consists of fields `count` (c) and `data` (d). The instantiation of problematic access pattern 12 is defined as "$\mathsf{R}_1(\mathsf{c}); \mathcal{W}_2(\mathsf{d}); \mathcal{W}_2(\mathsf{c}); \mathsf{R}_1(\mathsf{d})$". Before applying the language-strength-reduction transformation from Ch. 6, the number of global synchronizations is unbounded because the violation monitor, whose finite-state portion is repeated in Fig. 7.1, is required to count the reentrant unit-of-work depth for thread T_1 using its stack. Thus, the violation monitor, or rather the PDS $\mathcal{P}_{\mathsf{mon}}$ that implements the violation monitor, could be forced to reset its state when T_1 completes a unit of work and the problematic access pattern has not been observed.

After applying the language-strength-reduction transformation, the language of the violation monitor is much simpler because reentrant units of work have been eliminated and thus counting is no longer required. In fact, the language is now *regular*. The non-deterministic finite automaton (NFA) $\mathcal{A}_{7.2}$ that recognizes the instantiation of problematic access pattern 12 is shown in Fig. 7.2. The absence of counting is reflected in the fact that $\mathcal{A}_{7.2}$ does not contain the "reset" states r_{3-6} needed by the PDS $\mathcal{P}_{\mathsf{mon}}$.

$\mathcal{A}_{7.2}$, and in general all NFAs that recognize the language of a reformulated violation monitor after language-strength reduction, has a special form—the

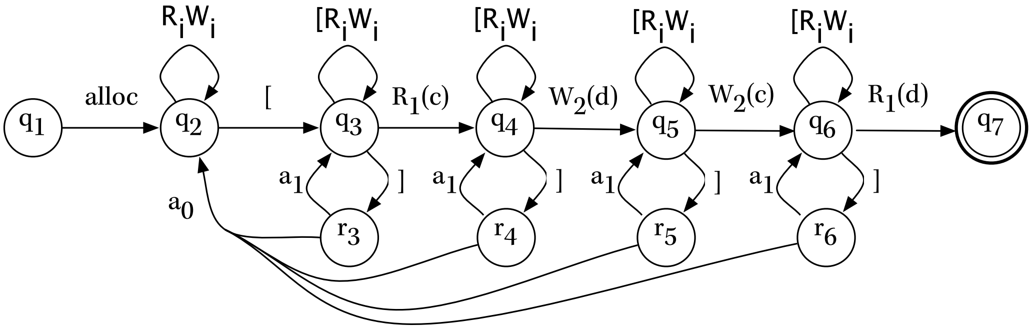

Figure 7.1: The state transitions of the PDS $\mathcal{P}_{\mathsf{mon}}$ from Ch. 5. The dashed lines denote state transitions that require stack inspection. If the stack is empty, the state is "reset" to q_2. Otherwise, the top of the stack will contain the unit-of-work marker $\top$, and the state is restored from r_i to state q_i.

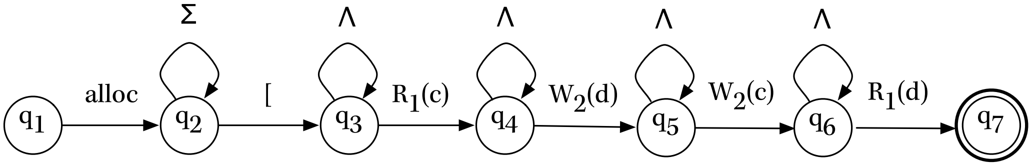

Figure 7.2: The NFA $\mathcal{A}_{7.2}$ that recognizes the language of the violation monitor from Ch. 5 after language-strength reduction has been performed. Σ denotes the input alphabet of $\mathcal{A}_{7.2}$, and Λ is defined as $\Sigma \setminus \{]\}$. Once $\mathcal{A}_{7.2}$ guesses that a violation 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}.

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*. The special form of IPAs provides a bound on the number of global synchronizations. That is, 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.

Still ignoring locks and focusing on the two-threaded Java program described above, the essence of the decision procedure is to ask for reachability of a *sequence* of global configurations, where a global configuration g is a tuple of (single-PDS) configurations (c_1, c_2), one for each PDS $\mathcal{P}_1$ and $\mathcal{P}_2$, respectively. The sequence of configurations comes directly from an IPA. For IPA $\mathcal{A}_{7.2}$, the first global configuration in the sequence is one in which $\mathcal{P}_1$ causes $\mathcal{A}_{7.2}$ to transition from state q_1 to state q_2. The rest of the global configurations model the rest of the required phase transitions that cause $\mathcal{A}_{7.2}$ to reach its accepting state. If such a sequence of configurations exists, then it is possible to drive $\mathcal{A}_{7.2}$ to its accepting state. Driving $\mathcal{A}_{7.2}$ to its accepting state shows that the EML program contains an AS-serializability violation, namely, the one defined by problematic access pattern 12.

Accounting for Locks

Kahlon et al. (2005a) presents a decision procedure for checking reachability of a set of global configurations of a *multi*-PDS, where a multi-PDS consists of a set of PDSs $\mathcal{P}_1, \ldots, \mathcal{P}_n$ that synchronize on a finite set of *non-reentrant* nested locks S_{Locks} of size $\mathcal{L}$ (i.e., $\mathcal{L} = |S_{\mathsf{Locks}}|$). For multi-PDSs, a global configuration is a tuple $(c_1, \ldots, c_n)$ of single-PDS configurations. 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.

Recall that after applying the language-strength reduction transformation, reentrant locks can be modeled by non-reentrant locks. Thus, translating EML into a multi-PDS instead of a CPDS as was performed in Ch. 6, we can use the techniques of Kahlon et al. (2005a) to check for reachability of a single set of global configurations. However, we require the ability to check for reachability of a bounded sequence of global configurations.

To check 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 §7.4. The first technique, which is the focus of this chapter, 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. Moreover, tupling establishes a *correlation* between the lock histories that arise at each global configuration in the desired sequence. 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 $\mathcal{A}_{7.2}$, the tuple would have seven lock histories, one for each state q_i, $1 \leqslant i \leqslant 7$.

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 $\mathcal{A}_{7.2}$, the chain of reachability queries would begin by computing a *set of sets* of global configurations that are compatible (via lock histories) such that $\mathcal{A}_{7.2}$ performs a transition from state q_1 to state q_2.[1] 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. A comparison between the chained-reachability approach of Kahlon

[1] The algorithm presented in Kahlon and Gupta (2007) is stated incorrectly and uses only sets of global configurations. We will discuss this error in more detail in §7.6 where we present a detailed comparison between our decision procedure and the one of Kahlon and Gupta (2007).

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

and Gupta (2007) and our approach is given in Tab. 7.1. We highlight two points:

1. Our tupling approach avoids an exponential in the number of locks $\mathcal{L}$ when compared to the chaining approach of Kahlon and Gupta (2007). (See the rightmost column in Tab. 7.1.)

2. Our tupling approach 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 §7.7.

Even though our decision procedure was designed for verifying AS-atomicity of EML programs, it is of general application when the program can be modeled as a multi-PDS and the property of interest specified as an arbitrary IPA, i.e., one whose input alphabet is not specific to verifying AS-atomicity. Thus, the remainder of the chapter is couched in terms of verifying properties specified by an IPA. Where appropriate, discussion has been added to establish the link between the generic decision procedure and the IPAs that arise for verifying AS-atomicity of EML programs.

7.2 Program Model and Property Specifications

A *multi-PDS* consists of a finite number of PDSs $\mathcal{P}_1, \ldots, \mathcal{P}_n$—where each PDS $\mathcal{P}_j = (P_j, \Gamma_j, \mathsf{Lab}_j, \Delta_j, c_0^j)$[2]—that synchronize via a finite set of locks $S_{\mathsf{Locks}} = \{l_1, \ldots, l_{\mathcal{L}}\}$ (i.e., $\mathcal{L} = |S_{\mathsf{Locks}}|$). The actions Lab of each PDS consist of lock-acquires ("$(_i$") and releases ("$)_i$") for $1 \leqslant i \leqslant \mathcal{L}$, plus symbols from Σ, a finite alphabet of non-parenthesis symbols.

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

The choice of what symbols (actions) appear in Σ depends on the intended application. For the target application of verifying AS-atomicity, 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).

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

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

[2] PDSs are formally defined in Ch. 3.

For $1 < j \leqslant n$, a global configuration $(c_1, \ldots, c_j, \ldots, c_n, o_1, \ldots, o_{\mathcal{L}})$ can make a transition to $(c_1, \ldots, c'_j, \ldots, c_n, o'_1, \ldots, o'_{\mathcal{L}})$ in a similar fashion.

A program property is specified as an indexed phase automaton.

Definition 7.1. 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. 7.1 ensures that the only loops in an IPA are self-loops on states. We assume that for every x, $1 \leqslant x < |Q|$, 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 7.2 (Bounded Phase Transition). *Let $\mathcal{A} = (Q, \mathit{Id}, \Sigma, \delta)$ be an IPA. 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 §7.8 we only consider 2-PDSs, and fix $\Pi = (\mathcal{P}_1, \mathcal{P}_2, S_{\mathsf{Locks}}, \Sigma)$ and $\mathcal{A} = (Q, \mathit{Id}, \Sigma, \delta)$. Given Π and $\mathcal{A}$, the model-checking problem of interest is to determine if there is an execution that begins at the initial global configuration g_0 that drives $\mathcal{A}$ to its final state.

§7.8 shows how to generalize the techniques to multi-PDSs. The implementation, discussed in §7.7, is for the general case.

7.3 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 $g = (c_1, c_2, o_1, \ldots, o_2)$. Let LocksHeld$(\mathcal{P}_k, g)$, $k \in \{1, 2\}$, denote $\{l_i \mid o_i = k\}$; 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 ρ_k.

Before the execution of ρ_k, PDS $\mathcal{P}_k$ has a (possibly empty) set of initially-held locks, i.e., the set 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 $g = (c_1, c_2, o_1, \ldots, o_{\mathcal{L}})$, 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 $g' = (c'_1, c'_2, o'_1, \ldots, o'_{\mathcal{L}})$. 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 $\mathcal{P}_2$.

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 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: $\mathcal{P}_k$ 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. 7.3.

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. 7.3 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. 7.3, 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 $\mathcal{P}_1$.

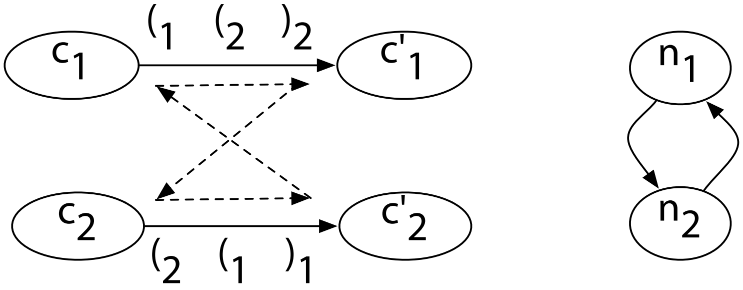

Figure 7.3: Individual executions of $\mathcal{P}_1$ and $\mathcal{P}_2$ from configurations c_1 and c_2 to configurations c_1' and c_2', respectively. The symbols $(_i$ and $)_i$ denote acquiring and releasing lock l_i, respectively. The dashed arrows denote the sequential and inter-PDS locking dependences 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.

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 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 7.3. *(**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 $\Pi = (\mathcal{P}_1, \mathcal{P}_2, \mathit{Locks}, \Sigma)$, there* does not *exist an interleaving of rule sequences ρ_1 and ρ_2 from global configuration $g = (c_1, c_2, o_1, \ldots, o_{\mathcal{L}})$ to global configuration $g' = (c'_1, c'_2, o'_1, \ldots, o'_{\mathcal{L}})$ 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 a lock.*
2. *LocksHeld$(\mathcal{P}_1, g') \cap$ LocksHeld$(\mathcal{P}_2, g') \neq \emptyset$: $\mathcal{P}_1$ and $\mathcal{P}_2$ both hold a lock.*
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.

7.4 Extracting Information from PDS Rule Sequences

To employ Thm. 7.3, 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 "$)($".[3]

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

$$\mathit{matched} \;\rightarrow\; \epsilon \;\mid\; \sigma\ \mathit{matched} \;\mid\; (_i\ \mathit{matched}\ )_i\ \mathit{matched} \quad [\text{for } 1 \leqslant i \leqslant \mathcal{L}]$$

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 $\mathsf{Lang}(\mathit{matched})$, which are languages of partially-matched parentheses. In particular,

- The words in the language of suffixes of $\mathsf{Lang}(\mathit{matched})$ 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 $\mathsf{Lang}(\mathit{matched})$ 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.

[3] 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.

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

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

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

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

Example 7.4. 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.

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

Let $w_k \in \mathsf{Lang}(\mathit{suffixPrefix})$ be the word formed by concatenating the action symbols of the rule sequence ρ_k. One can see that to use Thm. 7.3, we merely need to extract the relevant information from w_k. That is, items 3 and 4 require extracting (or recording) information from the *unbalR* and *unbalL* portions of w_k, respectively; item 5 requires extracting information from the *matched* portion of w_k; and items 1 and 2 require extracting information from the initial and final parse configurations of w_k.

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

- The *acquisition history* (AH) (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.[4]

- The *release history* (RH) (Kahlon and Gupta, 2007) of an initially-held lock l_i is the union of the set $\{l_i\}$ with the set of locks that are released (or acquired and released) before the initial release of l_i.

- The *release set* (R) is the set of initially-released locks.

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

- The *use set* (U) is the set of locks that form the *matched* part of w_k.
- The *acquisition set* (A) is the set of finally-acquired locks.
- The *held-throughout set* (HT) is the set of initially-held locks that are not released.

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, $1 \leqslant i \leqslant \mathcal{L}$.
- $\widehat{\mathsf{AH}}$ is a tuple of $\mathcal{L}$ acquisition histories, one for each lock l_i, $1 \leqslant i \leqslant \mathcal{L}$.

Let $\rho = [r_1, \ldots, r_n]$ be a rule sequence that drives a PDS from some starting configuration to an ending configuration, and let $\mathcal{I}$ be the set of locks held at the beginning of ρ. We define an abstraction function $\eta(\rho, \mathcal{I})$ from rule sequences and initially-held locks to lock histories; $\eta(\rho, \mathcal{I})$ uses an auxiliary function, post_η, which tracks R, $\widehat{\mathsf{RH}}$, U, $\widehat{\mathsf{AH}}$, A, and HT for each successively longer prefix.

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

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

Example 7.5. Suppose that ρ is a rule sequence whose labels spell out the string from Example 7.4, and $\mathcal{I} = \{1, 3, 7, 9\}$. Then $\eta(\rho, \mathcal{I})$ returns the following lock history (only lock indices are written):

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

Remark 7.6. R and A are included above only for clarity; they can be recovered from $\widehat{RH}$ and $\widehat{AH}$, as follows: $R = \{i \mid \widehat{RH}[i] \neq \emptyset\}$ and $A = \{i \mid \widehat{AH}[i] \neq \emptyset\}$. In addition, from $LH = (R, \widehat{RH}, U, \widehat{AH}, A, HT)$, it is easy to see that the set $\mathcal{I}$ of initially-held locks is equal to $(R \cup HT)$, and the set of finally-held locks is equal to $(A \cup HT)$.

Definition 7.7. Lock histories $LH_1 = (R_1, \widehat{RH}_1, U_1, \widehat{AH}_1, A_1, HT_1)$ and $LH_2 = (R_2, \widehat{RH}_2, U_2, \widehat{AH}_2, A_2, HT_2)$ are *compatible*, denoted by $\mathsf{Compatible}(LH_1, LH_2)$, iff all of the following five conditions hold:

1.$(R_1 \cup HT_1) \cap (R_2 \cup HT_2) = \emptyset$ 2.$(A_1 \cup HT_1) \cap (A_2 \cup HT_2) = \emptyset$

3. $\nexists\, i, j \,.\, l_j \in \widehat{AH}_1[i] \wedge l_i \in \widehat{AH}_2[j]$ 4. $\nexists\, i, j \,.\, l_j \in \widehat{RH}_1[i] \wedge l_i \in \widehat{RH}_2[j]$

5.$(A_1 \cup U_1) \cap HT_2 = \emptyset \wedge (A_2 \cup U_2) \cap HT_1 = \emptyset$

Each conjunct verifies the absence of the corresponding incompatibility condition from Thm. 7.3: 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.

7.5 The Decision Procedure

As noted in §7.3, the decision procedure analyzes the PDSs independently. This decoupling of the PDSs has two consequences.

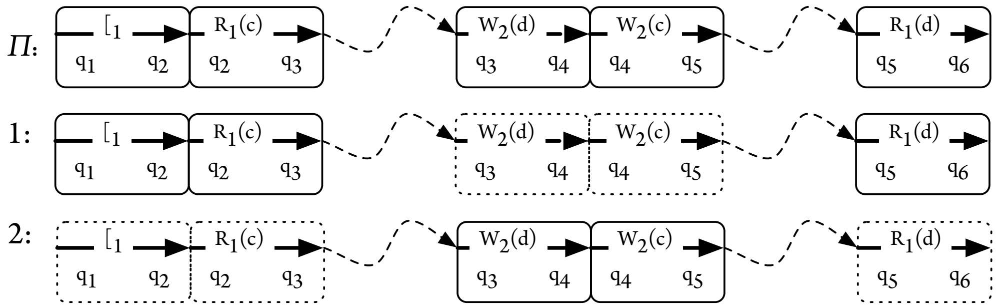

Figure 7.4: Π: a bad interleaving that is recognized by $\mathcal{A}_{7.2}$ (see 145), showing only the actions that cause a phase transition. 1: the same interleaving from Thread 1's point of view. The dashed boxes show where Thread 1 guesses that Thread 2 causes a phase transition. 2: the same but from Thread 2's point of view and with the appropriate guesses.

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. 7.4. The interleaving labeled "Π" is an example interleaved execution that is accepted by the IPA $\mathcal{A}_{7.2}$ from §7.1. In Fig. 7.4, only the PDS actions that cause phase transitions are show. 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 $W_w(\mathtt{d})$ actions, $\mathcal{P}_2$ must hold the lock associated with the randomly-isolated object because all writes to `Stack.data` occur in a `synchronized` method. If $\mathcal{P}_1$ guesses that $\mathcal{P}_2$ performs the $W_2(\mathtt{d})$ action at a point when it currently holds the lock associated with the randomly-isolated object, 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.

Combining a PDS with a 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 $\mathcal{A}$. Our approach is to combine $\mathcal{A}$ and $\mathcal{P}_1$ to define a new PDS $\mathcal{P}_1^{\mathcal{A}}$ 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 $|\mathsf{Q}| - 1$ phase transitions. Hence, a valid interleaved execution must be able to reach $|\mathsf{Q}|$ global configurations, one for each of the $|\mathsf{Q}|$ phases.

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 $|\mathsf{Q}| - 1$ phase transitions, and hence our algorithm keeps track of $|\mathsf{Q}|$ lock histories, one for each phase. This is accomplished by encoding into the state space of $\mathcal{P}_1^{\mathcal{A}}$ a *tuple* of $|\mathsf{Q}|$ 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 $\mathcal{P}_1^{\mathcal{A}}$ 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 $\widehat{\mathcal{LH}} = \mathcal{LH}^{|\mathsf{Q}|}$ denote the set of all tuples of lock histories of length $|\mathsf{Q}|$. We denote a typical lock history by LH, and a typical tuple of lock histories by $\widehat{\mathsf{LH}}$. $\widehat{\mathsf{LH}}[\mathsf{i}]$ denotes the i^{th} component of $\widehat{\mathsf{LH}}$.

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

Let $\mathcal{P}_i = (P_i, \mathsf{Lab}_i, \Gamma_i, \Delta_i, \langle p_0, \gamma_0 \rangle)$ 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 $P_i^{A} \subseteq P_i \times Q \times \widehat{\mathcal{LH}}$. 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 for 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.

Checking Path Compatibility

For a generated PDS $\mathcal{P}_k^{\mathcal{A}}$, we are interested in the set of paths that begin in the initial configuration $\langle p_0^{\mathcal{A}}, \gamma_0 \rangle$ and drive $\mathcal{A}$ to its 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 $\mathcal{P}_1^{\mathcal{A}}$ and $\mathcal{P}_2^{\mathcal{A}}$, respectively. To determine if there exists a compatible scheduling for ρ_1 and ρ_2, we use Thm. 7.3 on each component of the lock-history tuples $\widehat{\mathsf{LH}}_1$ and $\widehat{\mathsf{LH}}_2$ from the ending configurations of ρ_1 and ρ_2:

$$\mathsf{Compatible}(\widehat{\mathsf{LH}}_1, \widehat{\mathsf{LH}}_2) \Longleftrightarrow \bigwedge_{i=1}^{|Q|} \mathsf{Compatible}(\widehat{\mathsf{LH}}_1[i], \widehat{\mathsf{LH}}_2[i]). \tag{7.1}$$

Due to recursion, $\mathcal{P}_1^{\mathcal{A}}$ and $\mathcal{P}_2^{\mathcal{A}}$ could each have an infinite number of such paths. However, each path is abstracted as a tuple of lock histories $\widehat{\mathsf{LH}}$, and there are only a finite number of tuples in $\widehat{\mathcal{LH}}$; thus, we only have to check a finite number of $(\widehat{\mathsf{LH}}_1, \widehat{\mathsf{LH}}_2)$ pairs. For each PDS $\mathcal{P}^{\mathcal{A}} = (P^{\mathcal{A}}, \mathsf{Lab}, \Gamma, \Delta, c_0^{\mathcal{A}})$, the set of relevant $\widehat{\mathsf{LH}}$ tuples are found in the $\mathcal{P}_k^{\mathcal{A}}$-automaton $\mathcal{A}^k_{post^*}$ that results

Algorithm 7.1: 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 $\mathit{Compatible}(\widehat{LH}_1, \widehat{LH}_2)$ then
5             return true;
6 return false;
```

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. 7.1 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 7.8. *For 2-PDS $\Pi = (\mathcal{P}_1, \mathcal{P}_2, \mathit{Locks}, \Sigma)$ and IPA $\mathcal{A}$, there exists an execution of Π that drives $\mathcal{A}$ to its accepting state iff Alg. 7.1 returns true.*

Sketch. The proof builds on Thm. 7.3 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. 7.3. Because each phase can be scheduled, there exists a scheduling for runs ρ_1 and ρ_2. See App. A.3 for the entire proof. □

7.6 Comparison

We now present a more detailed comparison of the decision procedure from §7.5 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 (§7.5), 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 $\mathsf{Compatible}$ relation—i.e., that $\mathsf{Compatible}(LH_1, LH_2)$ 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 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 $\mathsf{Compatible}$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., $\mathsf{Compatible}(\mathsf{LH}_1, \mathsf{LH}_2)$, where LH_1 is the lock history component of the control location of configuration c_1). To maintain the invariant, after computing the individual reachability query on each automaton C_i (e.g., $\mathit{pre}^*(C_i)$), 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 Π 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 Π must be expanded to include the states of $\mathcal{A}$. To do so, one would need to perform the cross product of $\mathcal{P}_i$, $1 \leqslant i \leqslant 2$, 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:

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

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., $\forall LH_1, LH_2 \in \mathcal{LH} : \mathsf{Compatible}(LH_1, LH_2)$), the query of interest is then whether any of the following configuration pairs are reachable from the initial configuration:

$$\{(\langle(p_1, q_{|Q|}, LH_1), u_1\rangle, \langle(p_2, q_{|Q|}, LH_2), u_2\rangle) \mid p_1 \in P_1, u_1 \in \Gamma_1^*, p_2 \in P_2, u_2 \in \Gamma_2^*\}.$$

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).

By moving from lock histories to *tuples* of lock histories, our decision procedure that is presented in §7.5 does not require multiple reachability queries. Thus, we do 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. 7.2 repeats the comparison table from the beginning of the chapter to emphasize that worst-case running time of our algorithm has one less exponential factor when compared to the (corrected) Kahlon-Gupta algorithm. Specifically, see the rightmost column.

	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 7.2: 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).

7.7 A Symbolic Implementation

Alg. 7.1 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. 7.1, 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 $Q \times \mathcal{LH}$, 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.

We explain the improved approach by defining a suitable weight domain for use with a WPDS.[5]

[5] To remind the reader, a WPDS $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$ is a PDS $\mathcal{P} = (P, \mathsf{Lab}, \Gamma, \Delta, c_0)$ augmented with a bounded idempotent semiring $\mathcal{S} = (D, \otimes, \oplus, \bar{1}, \bar{0})$, and a function $f : \Delta \to D$ that assigns a semiring element $d \in D$ to each rule $r \in \Delta$. The result of a WPDS *post** computation is a weighted automaton $\mathcal{A}_{post^*}$. For the discussion that follows, $\mathcal{A}_{post^*}$ is a function from a

Definition 7.9. 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}:

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

Definition 7.10. 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

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

A *monomial* is written as $c_i z^i$ (all other coefficients are understood to be $\emptyset$); a monomial $c_0 z^0$ denotes a *constant*. The *multi-arity relational weight domain over* S *and* θ is defined by $(\mathcal{RFPS}[S, \theta], \times, +, \mathit{Id}, \emptyset)$, 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 $j \geqslant \theta$ 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 7.11. A multi-arity relational weight domain over S and θ, as defined in Defn. 7.10, meets the requirements of a bounded idempotent semiring (Defn. 3.9) because of (i) the properties of polynomial addition and truncated polynomial multiplication, (ii) the fact that the set of all relations of finite arity $\geqslant 2$ and the operation of generalized relational composition defined in Defn. 7.9 (";") 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 $\mathcal{A} =$

regular set of configurations C to the combine-over-all-paths (COVP) from c_0 to all $c \in C$; i.e., $\mathcal{A}_{post^*}(C) = \bigoplus\{v \mid \exists c \in C : c_0 \xrightarrow{r_1 \ldots r_n} c, v = f(r_1) \otimes \ldots \otimes f(r_n)\}$, where $r_1 \ldots r_n$ is a sequence of rules that transforms c_0 into c. See §3.2 of Ch. 3 for the formal definitions.

(Q, Id, Σ, δ). The construction is similar to that in §7.5, i.e., a cross product is performed that pairs the control states of $\mathcal{P}_i$ with the state space of $\mathcal{A}$. The difference is that the lock-history tuples are removed from the control state, and instead are modeled by $\mathcal{S}$, the multi-arity relational weight domain over the finite set $\mathcal{LH}$ and $\theta = |Q|$. We define $\mathcal{P}_i^{\mathcal{W}} = (P_i \times Q, \emptyset, \Gamma_i, \Delta_i^{\mathcal{W}}, \langle (p_0, q_1), \gamma_0 \rangle)$, where $\Delta_i^{\mathcal{W}}$ and f are defined as follows:

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

   (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 $r = \langle (p, q_x), \gamma \rangle \hookrightarrow \langle (p', q_x), u \rangle \in \Delta_i^{\mathcal{W}}$.

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

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

A multi-arity relational weight domain is parameterized by the quantity θ—the maximum number of phases of interest—which we have picked to be $|Q|$. We must argue that weight operations performed during model checking do not cause this threshold to be exceeded. For configuration $\langle (p, q_x), u \rangle$ to be reachable from the initial configuration $\langle (p_0, q_1), \gamma_0 \rangle$ of some WPDS $\mathcal{W}_i$, 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 $\mathcal{A}_{post^*}(\{\langle (p, q_x), u \rangle\})$

Algorithm 7.2: 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.
let $\mathcal{A}^1_{post^*} \leftarrow post^*_{\mathcal{W}_1}$; let $\mathcal{A}^2_{post^*} \leftarrow post^*_{\mathcal{W}_2}$;
let $c^1_{|Q|-1} z^{|Q|-1} = \mathcal{A}^1_{post^*}\left(\{\langle(p_1, q_{|Q|}), u\rangle \mid p_1 \in P_1 \wedge u \in \Gamma_1^*\}\right)$;
let $c^2_{|Q|-1} z^{|Q|-1} = \mathcal{A}^2_{post^*}\left(\{\langle(p_2, q_{|Q|}), u\rangle \mid p_2 \in P_2 \wedge u \in \Gamma_2^*\}\right)$;
return
    $\exists \langle LH_0, \widehat{LH}_1\rangle \in c^1_{|Q|-1}, \langle LH_0, \widehat{LH}_2\rangle \in c^2_{|Q|-1} : Compatible(\widehat{LH}_1, \widehat{LH}_2)$;
```

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. 7.2 states the algorithm for solving the multi-PDS model-checking problem for IPAs. Note that the final step of Alg. 7.2 can be performed with a single BDD operation.

Theorem 7.12. *For 2-PDS $\Pi = (\mathcal{P}_1, \mathcal{P}_2, Locks, \Sigma)$ and IPA $\mathcal{A}$, there exists an execution of Π that drives $\mathcal{A}$ to the accepting state iff Alg. 7.2 returns true.*

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. 7.2 computes the same result as Alg. 7.1. The proof then reduces to the proof of correctness for Alg. 7.1, which is given in App. A.3. The full proof of simulation is given in App. A.4. □

7.8 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 §7.5 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, to 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 Thm. 7.3 (page 154). The extension of items 1, 2, and 5 to N lock-history tuples is straightforward.

1. LocksHeld$(\mathcal{P}_1, g) \cap \ldots \cap$ LocksHeld$(\mathcal{P}_N, g) \neq \emptyset$.

2. LocksHeld$(\mathcal{P}_1, g') \cap \ldots \cap$ 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 3 and 4 define incompatibility to be a cycle of length two in the acquisition and release histories, respectively. For example, consider a 3-PDS with three (or more) locks. The absence of a cycle of length three in an 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. Hence, the generalized condition requires checking for a cycle in the acquisition and release histories that has a length anywhere from two to N. We use the notation $\mathsf{Compatible}(\mathsf{LH}_1, \ldots, \mathsf{LH}_N)$ to denote the generalized check. Then Alg. 7.1

is modified to contain N **foreach** loops, and the compatibility check at line 4 is replaced with $\mathsf{Compatible}(\widehat{\mathsf{LH}}_1, \ldots, \widehat{\mathsf{LH}}_\mathsf{N})$.

Similarly, Alg. 7.2 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

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

As in Alg. 7.2, the compatibility check can be performed via a single BDD operation by defining the N-way compatibility relation.

7.9 Experiments

We modified the Empire back end to generate muli-PDSs instead of CPDSs, and implemented the symbolic decision procedure presented in Alg. 7.2 of §7.7 in a tool called Ipamc. 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 multi-arity relational weight domain is included in WALi starting with release 3.0.)

Benchmark	# CPDSs	Viol	OK	OOM	OOT
SoftwareVerificationHW	15	6↑	9↑	0	0↓
BugTester	615	0	615↑	0↓	0
BuggyProgram	615	16↑	599	0	0
shop	900	27↑	873↑	0↓	0↓
Totals	4583	49↑	2096↑	0↓	0↓

Table 7.3: Analysis summaries of the four benchmark programs that contain locking operations. The annotations "↑" and "↓" show the relative change with respect to the analysis summaries presented in §6.6 of Ch. 6.

We reanalyzed the four EML programs that contained synchronization operations—SoftwareVerificationHW, BugTester, BuggyProgram, and shop—

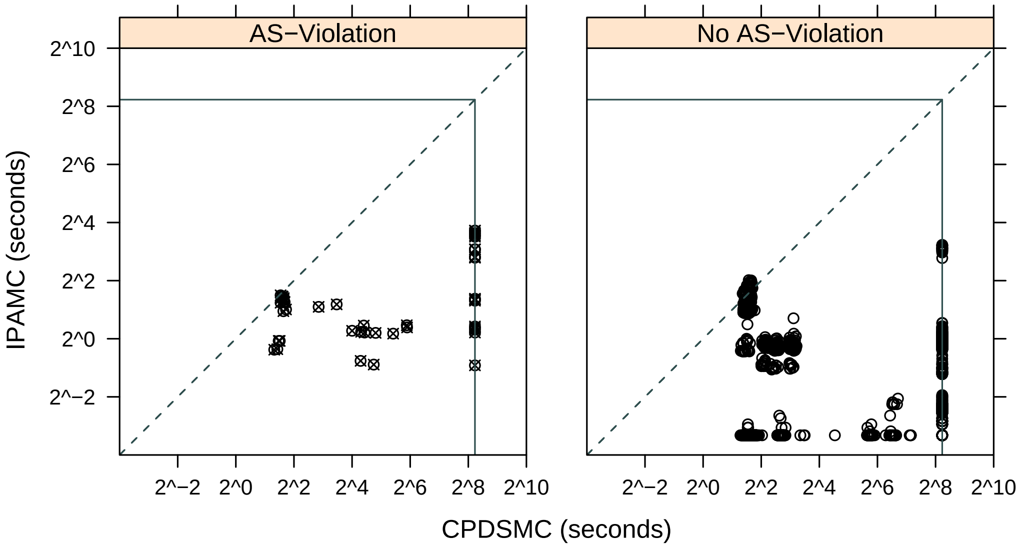

Figure 7.5: Log-log scatter-plots of the execution times of Ipamc (y-axis) versus Cpdsmc (x-axis). The left-hand graph shows the 49 queries for which Ipamc reported an AS-serializability violation; the right-hand graph shows the 2,096 queries for which Ipamc verified correctness. The dashed lines denote 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 minimal reported time is 0.1 second.

from §6.6. All experiments were run on a dual-core 3 GHz Pentium Xeon processor with 4 GB of memory.

The purpose of the experiment was to determine the answer to the queries on which Cpdsmc exhausted resources (roughly 50%), and to compare the performance of the symbolic-decision procedure Alg. 7.2) implemented in Ipamc to the *Individual Multi-step* SDP that is implemented in Cpdsmc. The analysis summaries are shown in Tab. 7.3. As expected, the decision procedure returned a definitive answer for all queries. For the benchmark `BugTester`, no AS-serializability violations were found. Upon further inspection of the benchmark's code, the allocation site that was specified allocates an object on which there cannot be an AS-serializability violation. From the analysis of `BugTester`, and via the use of Ipamc as the back end of Empire, we have shown that Empire is able to verify AS-atomicity for objects allocated at a specified allocation site. The ability to prove partial correctness—AS-atomicity for objects allocated at a specific allocation site—even for programs that in fact do contain bugs, is a benefit of the modular approach taken to verifying AS-atomicity.

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 Chs. 5 and 6). (The third possibility is that it times out.) Fig. 7.5 presents log-log scatter plots of the execution times of Ipamc (y-axis) versus Cpdsmc, where Cpdsmc used the SDPd presented in Ch. 4 (x-axis). The "inner box" in each scatter plot marks the timeout threshold of 300 seconds. 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 that Cpdsmc. Because almost every point is below and to the right of the dashed-diagonal line in Fig. 7.5, we can see that Ipamc performed better than Cpdsmc on nearly every query.

Tab. 7.4 presents a comparison of the total time to execute all queries. The total times are partitioned according to 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 on about 50% of the queries—both for the ones for which Ipamc reported an AS-serializability violation (37 timeouts out of 47 queries), as well as the ones for which Ipamc verified correctness (1,034 timeouts out of 2,145 queries).

Tab. 7.5 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

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

using CPDSMC because CPDSMC exhausted the available resources.[6] The number of additional AS-serializability violations detected clearly shows the benefit of using a decision procedure over a semi-decision procedure.

7.10 Related Work

This chapter introduces a different technique than that used by Kahlon and Gupta (2007). To decide the model-checking problem for IPAs, one needs to check pairwise reachability of *multiple* global configurations in succession. Our algorithm uses WPDS weights that are sets of lock-history tuples, whereas Kahlon and Gupta use sets of pairs of configuration automata.

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

[6] We have not manually verified that the additional AS-serializability violations found using IPAMC are actual bugs or false positives.

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 7.5: 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.

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

Table 7.6: Related work on LTL/atomicity checking and context-bounded model checking (CBMC). 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*

on intermediate results, it avoids enumerating compatible configuration pairs, thereby enabling BDD-based symbolic representations to be used throughout.

This chapter advocates the use of *tupling* versus *splitting* for maintaining correlations between intermediate configurations of a multi-configuration reachability query, i.e., a reachability query that must determine reachability of a sequence of intermediate global configurations of a multi-PDS. Besides the (corrected) Kahlon and Gupta (2007) algorithm, the algorithm for context-bounded model checking by Qadeer and Rehof (2005) also requires splitting to maintain correlations. In their case, they require splitting to enumerate the global state space at a context switch, which, in essence, amounts to using sets of automaton-pairs. Similar to our approach, 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.[7] Tab. 7.6 presents a comparison of the four approaches. In general, the use of tupling is preferred because:

1. Tupling permits an exponential-cost enumeration step to be avoided (and thus have better worst-case asymptotic cost).

2. Tupling enables symbolic techniques to be employed, which has the ad-

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

> ditional benefit of allowing the remaining exponential factors to be overcome in practice.

Comparing context-bounded model checking as defined by Qadeer and Rehof (2005); Lal et al. (2008) with our model-checking problem, we bound the number of phases, but permit an unbounded number of context switches and an unbounded number of lock acquisitions and releases by each PDS. The decision procedures from §7.5 and §7.7 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.

8 CONCLUDING REMARKS

Commodity processors currently have two, four, or eight cores. Commerical chip manufacturers, such as Intel®, have stated that the number of cores will not only continue to rise, but will rise at a rapid pace. The trend to higher numbers of cores is evident from Intel's *Tera-scale Computing Research Program*:

> *The Intel® Tera-scale Computing Research Program is a worldwide effort to advance computing technology for the next decade. By scaling multi-core architectures to 10s to 100s of cores and embracing a shift to parallel programming, we aim to improve performance, increase energy-efficiency, and make future applications more compelling and immersive.*
>
> — Intel Tera-scale Computing Research Program (2009)

While the shift to multi-core processors provides computing power heretofore unseen in the consumer and server markets, to take advantage of this new computing power requires the use of concurrent programming—i.e., programs will consist of multiple threads executing in parallel to accomplish a desired task.

Writing a correct concurrent program is notoriously more difficult than writing its sequential counterpart. Indeed, data-consistency errors—a type of programming error that does not occur in sequential programs—can arise in a concurrent program because of non-deterministic interference from concurrently executing threads.

A major focus of my dissertation research has been the development of techniques to assist a programmer in reasoning about the data consistency of a program. Specifically, the techniques that I developed, and that are presented in the dissertation, check that a concurrent Java program has the AS-atomicity property. The dissertation has shown that software model checking is a feasible approach for verifying AS-atomicity of concurrent programs. Specifically, the major result was to show that the problem is decidable for recursive concurrent EML programs with (i) reentrant locks, (ii) recursive unit-of-work methods, and (iii) a property specified as an *indexed phase automaton* (IPA).

We were able to obtain this decidability result only after working on the problem for several years, and developing both a novel program abstraction and several techniques for transforming program models to address the sources of unboundedness that remain present in an abstract program (i.e., an EML program).

A central theme of the dissertation is overcoming sources of *unboundedness*. We say that a resource requirement for programs written in a language L is *unbounded* if there is no *a priori* bound on the amount of the resource that may be used by an L program. A concurrent Java program can have (1) an unbounded memory requirement, (2) an unbounded-size stack (due to recursion), (3) an unbounded set of values that each memory location can hold, (4) an unbounded single-threaded execution trace (due to looping and recursion), (5) an unbounded number of threads, (6) an unbounded number of lock acquisitions and releases (due to unbounded execution traces), (7) an unbounded number of times that a particular thread may reacquire a reentrant lock (due to recursion), (8) an unbounded number of nested calls to unit-of-work methods (due to recursion), and (9) an unbounded number of execution interleavings.

We were able to account for some of the sources of unboundedness through the use of abstraction.

Ch. 5 presented *random isolation*, a program abstraction that is used to reason about the locking behavior of programs that use dynamic memory allocation. Random-isolation abstraction, like allocation-site abstraction, bounds the number of abstract objects manipulated by an abstracted program $\mathsf{Prog}^\sharp$ by associating abstract objects to $\mathsf{Prog}^\sharp$'s allocation sites, where an abstract object over-approximates the set of concrete objects that can be allocated at a site in the original program Prog. However, unlike allocation-site abstraction, random-isolation abstraction associates *two* abstract objects to a specified allocation site.[1] We use *isolation* to ensure that one of the abstract objects represents a singleton set of concrete objects so that strong updates can be performed on that abstract object's state. The ability to perform strong updates is crucial for the analysis of concurrent programs that use lock-based synchronization because it enables a model checker to reason precisely about the state of the lock that is associated with the randomly-isolated object. We use *randomness* to determine when the randomly-isolated object is allocated, which provides a mechanism for generalizing a proof that a property holds for the randomly-isolated object to a proof that the property holds for *all* concrete objects that can be allocated at the specified site.

To bound the values that a memory location can hold, we observed that, for AS-atomicity verification, one is only concerned with read and write accesses to shared-memory locations. Thus, we further abstract the program by removing *values* from memory locations, and only reason about accesses to them by the individual threads.

After applying these abstractions to generate an abstract program $\mathsf{Prog}^\sharp$ that over-approximates the set of behaviors of a concurrent Java program Prog, some of the sources of unboundedness have been removed. After applying random-isolation abstraction, the unbounded number of objects that can be allocated by Prog are represented by a finite number of abstract objects that can be allocated in $\mathsf{Prog}^\sharp$. We abstracted the unbounded set of values that a memory location can hold. Finally, we bound the number of threads in $\mathsf{Prog}^\sharp$ to remove the unbounded number of threads that can be created.

However, many sources of unboundedness remain, namely, (1) $\mathsf{Prog}^\sharp$ contains reentrant locks; (2) each thread can define recursive methods, which causes the size of each local stack to be unbounded; (3) unit-of-work methods can also be recursive, which requires the violation monitor to use an infinite-state counter to track the depth of nested calls to unit-of-work methods; and (4) the number of interleaved executions is unbounded. Because of these remaining sources of un-

[1] Random isolation can, in general, be applied to every allocation site. We have not explored this broader use of random isolation.

boundedness, we at first believed that verifying AS-atomicity for $\mathsf{Prog}^\sharp$—or the EML program that is generated from $\mathsf{Prog}^\sharp$—is, in general, undecidable. Thus, in Ch. 5 we applied CPDS model checking (Chaki et al., 2006) to the problem of verifying AS-atomicity, and showed that it was useful in practice. With respect to CPDS model checking, Ch. 4 presented several (unpublished) variations on the original semi-decision procedure, and, by analyzing a model of a real Windows Bluetooth driver, showed that these techniques provide improved performance in practice.

Ch. 6 presented a technique that we call *language strength reduction*, which enables the context-free languages of an EML lock and the violation monitor to be replaced with regular languages. To do so, we defined the nested-word language of a pushdown system (PDS), and gave a construction that combined a nested-word automaton (NWA) with a PDS. The result is a new PDS $\mathcal{P}_\mathcal{N}$ that (i) retains the same set of behaviors of the original PDS, and (ii) is able to distinguish between ownership-changing (OC) and non-ownership-changing (nOC) lock acquisitions and releases (and also outermost and inner calls to unit-of-work methods). Because $\mathcal{P}_\mathcal{N}$ can make these distinctions, we could then remove from $\mathcal{P}_\mathcal{N}$ all nOC lock acquisitions and releases, as well as actions that model calls to and returns from inner unit-of-work methods. The overall effect is that the PDSs that model EML locks and the violation monitor no longer require a stack to count their respective nesting depths, which allowed their languages to be defined by non-deterministic finite automata (NFAs)—i.e., strength reduction was applied to their languages. Language strength reduction eliminated two sources of unboundedness, and in doing so, produced an overall speedup of 1.8 when analyzing CPDSs generated by EMPIRE.

Another important benefit of language strength reduction is that the NFA $\mathcal{A}$ that recognizes the language of the violation monitor always has a special form, namely, the only loops in $\mathcal{A}$ are self-loops on states, which induces a total order of the states of $\mathcal{A}$. We call such an NFA an indexed-phase automaton (IPA). IPAs enjoy the bounded-phase-transition property, which means that there is a bound on the number of global synchronizations due to memory accesses between the PDSs that model EML processes.

Even with these transformations to the model, the remaining PDSs of the model still had, at first sight, a tight coupling because they need to synchronize with each other on their use of locks. Nevertheless, it turned out that the remaining PDSs are not that tightly coupled. We used *lock histories* to summarize the constraints that an individual PDS's execution places on an interleaved execution. Lock histories enable the PDSs that model a concurrent program to be analyzed independently from each other. After analyzing each PDS independently, the constraints that are generated for the PDSs are then checked for compatibility. In Thm. 7.8 we proved that there exists a set of lock histories (one lock history per PDS) that satisfies the Compatible relation if-and-only-if there exists an interleaved execution of the multi-PDS that can drive the NFA of the transformed violation monitor to its accepting state—i.e., that an AS-serializability violation can occur. Thus, by Thm. 7.8, we were able to show that verifying AS-atomicity of an EML program is, in fact, *decidable*.

Our decidability result is related to work by Kahlon and Gupta (2007), who showed that, for certain fragments of (indexed) LTL, model checking multi-PDSs is decidable. Their decision procedure—as it would be applied to AS-atomicity verification—also uses lock histories. Our work differs from the (corrected) Kahlon and Gupta decision procedure because it uses *tupling* rather than *splitting* to answer the sequence of reachability queries that arise during the model-checking process. The use of tupling (i) eliminated an exponential factor in the analysis cost, and (ii) isolates another exponential factor in the PDS control locations, for which symbolic techniques can be used to overcome this cost in practice.

Moving forward, there are various directions that one could take as follow-on work to the dissertation. A natural extension of the work on the decision procedure presented in Ch. 7 would be to determine whether our tupling approach can be extended to handle properties specified in the fragments of (indexed)

LTL that Kahlon and Gupta proved were decidable (Kahlon and Gupta, 2007). Such an extension would be an important result because, as we discussed in §7.6 of Ch. 7, the use of splitting in the Kahlon-Gupta decision procedure(s) appears to be the obstacle to creating an implementation that is efficient enough to be used in practice. As a teaser, we know how to relax property specifications from the linear form of an IPA to a directed-graph form that is constrained so that the only loops are self-loops on nodes in the graph. The research opportunity is to show that a property specified in a decidable fragment of (indexed) LTL can be encoded in the relaxed-graph-form IPA.

Another extension is to relax the finite-thread restriction of EML programs. The work on *dynamic-pushdown networks* (DPNs) (Bouajjani et al., 2005)—an extension of multi-PDSs to include thread creation—by Lammich et al. (2009) looks like a promising approach. It is interesting to note that the tupling-versus-splitting issue arises for DPNs as well. Lammich et al. (2009) use chained reachability queries to prove properties for DPNs. Though they do not have an explicit step that splits the automata that result from intermediate queries, they do require a rather expensive intersection operation that, in essence, amounts to splitting. At present, we do not fully understand how tupling would work with dynamic thread creation. That is, tupling is a mechanism to maintain necessary correlations between intermediate configurations that are part of a sequence of reachable configurations of a *single* PDS. For DPNs, one must be able to reason about not only the intermediate configurations of a single PDS, but also any PDSs that it may dynamically create. Thus, it would appear that another operation is required to join the tuples of child PDSs with a parent PDS.

Beyond extending Empire, the possibilities are endless. We know for certain that concurrent programming, and, in particular, the analysis of concurrent programs will remain an important area of research for the foreseeable future. With respect to the analysis of concurrent programs, we have shown that one can sometimes finesse multiple sources of unboundedness as a way of establishing decidability. In particular, for software model checking using multi-PDSs, there is a chance of a problem being decidable if it can be shown that only a *bounded* number of global synchronizations is actually necessary.

The path that I have taken in my dissertation research to obtain a decision procedure provides insight into both of these issues, namely, that apparently undecidable problems can sometimes be massaged or transformed to (1) remove sources of unboundedness, and (2) bound the number of global synchronizations, at which point a decision procedure can be obtained.

REFERENCES

Alur, Rajeev, and P. Madhusudan. 2004. Visibly pushdown languages. In *STOC*.

———. 2006. Adding nesting structure to words. In *DLT*.

Artho, Cyrille, Klaus Havelund, and Armin Biere. 2003. High-level data races. *Journal on Software Testing, Verification and Reliability (STVR)* 13(4).

Bahar, R.I., E.A. Frohm, C.M. Gaona, G.D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. 1993. Algebraic decision diagrams and their applications. In *cad*, 188–191.

Balakrishnan, G., and T. Reps. 2006. Recency-abstraction for heap-allocated storage. In *SAS*.

Bernstein, P., V. Hadzilacos, and N. Goodman. 1987. *Concurrency control and recovery in database systems*. Addison-Wesley.

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

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

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

Bryant, R.E. 1986. Graph-based algorithms for Boolean function manipulation. *TOC* 677–691.

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

Chaki, Sagar, Edmund Clarke, Nicholas Kidd, Thomas Reps, and Tayssir Touili. 2006. Verifying concurrent message-passing C programs with recursive calls. In *TACAS*, 334–349.

Chaudhuri, S., and R. Alur. 2007. Instrumenting C programs with nested word monitors. In *SPIN*.

Clarke, Edmund M., and E. Allen Emerson. 1982. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of programs, workshop*, 52–71.

Cousot, P., and R. Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 238–252.

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

———. 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 P. Wolper. 1997. A direct symbolic approach to model checking pushdown systems. *Elec. Notes in Theor. Comp. Sci.* 9.

Flanagan, Cormac, and Stephen N Freund. 2004. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL*, 256–267.

———. 2008. Velodrome: A sound and complete dynamic analysis for atomicity. In *POPL*.

Flanagan, Cormac, and Shaz Qadeer. 2003. A type and effect system for atomicity. In *PLDI*.

Hammer, C., J. Dolby, M. Vaziri, and F. Tip. 2008. Dynamic detection of atomic-set serializability violations. In *ICSE*.

Harrison, M. A. 1978. *Introduction to formal language theory*. Boston, MA, USA.

Horwitz, S., P. Pfeiffer, and T. Reps. 1989. Dependence analysis for pointer variables. In *PLDI*.

IBM. 2009. Watson Libraries for Analysis (WALA). http://wala.sourceforge.net/wiki/index.php.

Intel Tera-scale Computing Research Program. 2009. Tera-scale computing research program. [Online; accessed 24-July-2009].

Jones, N.D., and S.S. Muchnick. 1981. Flow analysis and optimization of Lisp-like structures. In *Program flow analysis: Theory and applications*. Prentice-Hall.

———. 1982. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *POPL*.

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

———. 2009. Personal communication.

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

Kahlon, V., Y. Yang, S. Sankaranarayan, and A. Gupta. 2005b. Fast and accurate static data-race detection for concurrent programs. In *CAV*.

Kidd, Nicholas. 2009. The bluetooth driver models. http://pages.cs.wisc.edu/~kidd/bluetooth.

Kidd, Nicholas, Akash Lal, and Thomas Reps. 2007. Advanced queries for property checking. Tech. Rep. 1621, Univ. of Wisconsin. Available at http://www.cs.wisc.edu/wpis/abstracts/tr1621.abs.html.

———. 2008. Language strength reduction. In *SAS*, 283–298.

———. 2009a. WALi: The Weighted Automaton Library. http://www.cs.wisc.edu/wpis/wpds/download.php.

Kidd, Nicholas, Thomas Reps, Julian Dolby, and Mandana Vaziri. 2009b. Finding concurrency-related bugs using random isolation. In *VMCAI*, 198–213.

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

Lal, A., and Thomas Reps. 2008. Reducing concurrent analysis under a context bound to sequential analysis. In *CAV*.

Lal, Akash, Nicholas Kidd, Thomas Reps, and Tayssir Touili. 2007. Abstract error projection. In *SAS*.

Lal, Akash, and Thomas Reps. 2006. Improving pushdown system model checking. In *CAV*.

Lal, Akash, Thomas Reps, and Gogul Balakrishnan. 2005. Extended weighted pushdown systems. In *CAV*.

Lal, Akash, Tayssir Touili, Nicholas Kidd, and Thomas Reps. 2008. Interprocedural analysis of concurrent programs under a context bound. In *TACAS*, 282–298.

Lammich, Peter, Markus Müller-Olm, and Alexander Wenner. 2009. Predecessor sets of dynamic pushdown networks with tree-regular constraints. In *CAV*. To appear.

Lipton, Richard J. 1975. Reduction: a method of proving properties of parallel programs. *Commun. ACM* 18(12):717–721.

Lu, Shan, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from mistakes—a comprehensive study on real world concurrency bug characteristics. In *ASPLOS*.

Lu, Shan, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. 2006. AVIO: Detecting atomicity violations via access interleaving invariants. In *ASPLOS*.

McMillan, K.L. 1999. Verification of infinite state systems by compositional model checking. In *CHARME*, 219–234.

Milanova, Ana, Atanas Rountev, and Barbara G. Ryder. 2005. Parameterized object sensitivity for points-to analysis for Java. *TOSEM* 14(1).

Naik, Mayur, and Alex Aiken. 2007. Conditional must not aliasing for static race detection. In *POPL*, 327–338.

Nederhof, Mark-Jan. 1999. Practical experiments with regular approximation of context-free languages. *CoRR*.

Papadimitriou, C. 1986. *The theory of database concurrency control*. Computer Science Press.

Pnueli, A., J. Xu, and L. Zuck. 2002. Liveness with (0, 1, ∞)-counter abstraction. In *CAV*.

Pratikakis, Polyvios, Jeffrey S. Foster, and Michael Hicks. 2006. LOCKSMITH: Context-sensitive correlation analysis for race detection. In *PLDI*, 320–331. ACM Press.

Qadeer, S., S. K. Rajamani, and J. Rehof. 2004. Summarizing procedures in concurrent programs. In *POPL*, 245–255.

Qadeer, S., and D. Wu. 2004. KISS: Keep It Simple and Sequential. In *PLDI*.

Qadeer, Shaz, and Jakob Rehof. 2005. Context-bounded model checking of concurrent software. In *TACAS*.

Queille, Jean-Pierre, and Joseph Sifakis. 1982. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th colloquium on international symposium on programming*, 337–351. Springer-Verlag.

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

Reps, Thomas. 1998. Program analysis via graph reachability. *Inf. and Softw. Tech.* 40.

Reps, Thomas, Akash Lal, and Nicholas Kidd. 2007. Program analysis using weighted pushdown systems. In *FSTTCS*.

Reps, Thomas, Stefan Schwoon, and Somesh Jha. 2003. Weighted pushdown systems and their application to interprocedural dataflow analysis. In *SAS*.

Reps, Thomas, Stefan Schwoon, Somesh Jha, and David Melski. 2005. Weighted pushdown systems and their application to interprocedural dataflow analysis. *SCP* 58.

Sagiv, M., T. Reps, and R. Wilhelm. 2002. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems* 24(3).

Savage, Stefan, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas E. Anderson. 1997. Eraser: A dynamic data race detector for multithreaded programs. *Theoretical Computer Science* 15(4):391–411.

Schwoon, Stefan. 2002. Model-checking pushdown systems. Ph.D. thesis, TUM.

Vaziri, Mandana, Frank Tip, and Julian Dolby. 2006. Associating synchronization constraints with data in an object-oriented language. In *POPL*, 334–345.

Visser, W. 2009. Personal communication.

Wang, Liqiang, and Scott D. Stoller. 2006. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *PPoPP*, 137–146.

Wikipedia. 2009. Northeast blackout of 2003. [Online; accessed 1-June-2009].

Xu, Min, Rastislav Bodík, and Mark D. Hill. 2005. A serializability violation detector for shared-memory server programs. In *PLDI*.

Yavuz-Kahveci, T., and T. Bultan. 2002. Automated verification of concurrent linked lists with counters. In *SAS*, 69–84.

A APPENDIX

A.1 Proof of Thm. 6.4

Proof. The proof is organized as follows:

1. $\mathsf{NWLang}(\mathcal{P}_{\mathcal{N}}, (p', q')) \subseteq \mathsf{NWLang}(\mathcal{P}, p')$ by Lem. A.4;
2. $\mathsf{NWLang}(\mathcal{P}_{\mathcal{N}}, (p', q')) \subseteq \mathsf{NWLang}(\mathcal{N}, q')$ by Lem. A.5; and
3. $\mathsf{NWLang}(\mathcal{P}_{\mathcal{N}}, (p', q')) \supseteq \mathsf{NWLang}(\mathcal{P}, p') \cap \mathsf{NWLang}(\mathcal{N}, q')$ by Lem. A.6.

□

Corollary A.1. $\mathit{NWLang}(\mathcal{P}_{\mathcal{N}}, P \times Q) = \mathit{NWLang}(\mathcal{P}, P) \cap \mathit{NWLang}(\mathcal{N}, Q)$

Proof.

$$\begin{aligned}
\mathsf{NWLang}(\mathcal{P}, P) \cap \mathsf{NWLang}(\mathcal{N}, Q) &= \bigcup_{p \in P} \mathsf{NWLang}(\mathcal{P}, p) \cap \bigcup_{q \in Q} \mathsf{NWLang}(\mathcal{N}, q) \\
&= \bigcup_{p \in P, q \in Q} \mathsf{NWLang}(\mathcal{P}, p) \cap \mathsf{NWLang}(\mathcal{N}, q) \\
&= \mathsf{NWLang}(\mathcal{P}_{\mathcal{N}}, P \times Q)
\end{aligned}$$

Notice that the last step above follows from Thm. 6.4.

□

When discussing the nested-word languages for $\mathcal{P}$ and $\mathcal{P}_{\mathcal{N}}$, we will always be starting from configuration $\langle p_0, e_{\text{main}} \rangle$ and $\langle (p_0, q_0), e_{\text{main}} \rangle$, respectively. In addition, because of the spitting of Γ into Γ^{α} and Γ^{β}, in the inductive steps of the proof we will use the *effective length* of a run defined as follows:

Definition A.1. The *effective length* of a run $\rho = [r_1, \ldots, r_j]$ is equal to the length of the word component w of the nested word $nw = (w, v) = \mathsf{nwpost}(\rho)$. For a run ρ, we denote its effective length by *EffLen*(ρ).

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

We first present the proofs for Lemmas A.2 and A.3. Lemmas A.2 and A.3 are helper lemmas that aid in proving Lemmas A.4, A.5, and A.6.

Lemma A.2. *For a run $[r_1, \ldots, r_j]$ of $\mathcal{P}_\mathcal{N}$ that generates a nested word $nw \in \mathit{NWLang}(\mathcal{P}_\mathcal{N}, (p', q'))$, the rule r_j is either a step rule or a push rule from $\Delta_\mathcal{N}$, but not a pop rule.*

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

□

Lemma A.3. *For a nested word $nw = (w, v) \in \mathit{NWLang}(\mathcal{P}, p') \cap \mathit{NWLang}(\mathcal{N}, q')$, the length j of the run $\rho = [r_1, \ldots, r_j]$ of $\mathcal{P}$ that generates nw is equal to $|w|$.*

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

□

Lemma A.4. $\mathit{NWLang}(\mathcal{P}_\mathcal{N}, (p', q')) \subseteq \mathit{NWLang}(\mathcal{P}, p')$.

Proof. We must show that $\mathcal{P}_\mathcal{N}$ is a subset of $\mathcal{P}$; i.e., that *Construction 1* did not introduce behaviors or runs that were not originally a part of $\mathcal{P}$. The intuition on which the proof is based is that *Construction 1* produces a PDS whose set of runs is a restriction of the set of runs of $\mathcal{P}$. Thus, we prove Lem. A.4 by providing a function that maps each run $\rho_{\mathcal{N}}$ of $\mathcal{P}_{\mathcal{N}}$ to a run ρ of $\mathcal{P}$ such that $\mathsf{nwpost}(\rho_{\mathcal{N}}) = \mathsf{nwpost}(\rho)$. For a rule $r \in \Delta$ and transition $t \in \delta$, we use the constructor $\kappa(r, t)$ to denote the set of rules in $\Delta_{\mathcal{N}}$ that are generated by *Construction 1* (see §6.4). The proof makes use of the deconstructor $\kappa_{\Delta}^{-1} : \Delta_{\mathcal{N}} \to \Delta$, defined as follows:

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

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

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

For a rule $r \in \Delta_{\mathcal{N}}$ and nested word $nw = (w, v)$, we show by a case analysis on the form of the rule r that $\mathsf{nwpost}[r](nw) = \mathsf{nwpost}[\kappa_{\Delta}^{-1}(r)](nw)$.

1. $r = \langle (p, q), n_1 \rangle \hookrightarrow \langle (p', q'), n_2 \rangle$, and $n_1 \in \Gamma_{\mathcal{N}}^{\alpha}$. By definition, $\kappa_{\Delta}^{-1}(r) = \langle p, n_1 \rangle \hookrightarrow \langle p', n_2 \rangle$. Both $\mathsf{nwpost}[r](nw)$ and $\mathsf{nwpost}[\kappa_{\Delta}^{-1}(r)](nw)$ append the symbol n_1 to the nested word nw, and neither affect the nesting relation.

2. $r = \langle (p, q), n_1 \rangle \hookrightarrow \langle (p', q'), n_2 \rangle$, and $n_1 \in \Gamma_{\mathcal{N}}^{\beta}$. Because $n_1 \in \Gamma_{\mathcal{N}}^{\beta}$, $\mathsf{nwpost}[r](nw) = nw$. By definition, $\kappa_{\Delta}^{-1}(r) = \epsilon$, and $\mathsf{nwpost}[\kappa_{\Delta}^{-1}(r)](nw) = nw$.

3. $r = \langle (p, q_c), n_c \rangle \hookrightarrow \langle (p', q'), e\ (r_c, q_c) \rangle$. By definition, $\kappa_{\Delta}^{-1}(r) = \langle p, n_c \rangle \hookrightarrow \langle p', e\, r_c \rangle$. Both $\mathsf{nwpost}[r](nw)$ and $\mathsf{nwpost}[\kappa_{\Delta}^{-1}(r)](nw)$ append n_c to w and add $\langle (|w \cdot n_c|), \infty \rangle$ to v.

4. $r = \langle (p, q_r), x \rangle \hookrightarrow \langle (p', q_r^x), \epsilon \rangle$. By definition, $\kappa_{\Delta}^{-1}(r) = \langle p, x \rangle \hookrightarrow \langle p', \epsilon \rangle$. Let $i = max(\{k \mid \langle k, \infty \rangle \in v\})$. Both $\mathsf{nwpost}[r](nw)$ and

$\mathsf{nwpost}[\kappa_\Delta^{-1}(r_j)](nw)$ append x to w, remove $\langle i, \infty \rangle$ from v, and add $\langle i, |w \cdot x| \rangle$ to v.

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

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

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

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

1. $j = 0$. By definition, $[] \in \mathsf{Runs}(\mathcal{P})$.

2. Each rule of the run is such that its left-hand-side stack symbol $\gamma \in \Gamma_\mathcal{N}^\beta$. By the definition of *Construction 1*, each such rule must be preceded by a pop rule, and the left-hand-side stack symbol of every pop rule in $\Delta_\mathcal{N}$ is a member of $\Gamma_\mathcal{N}^\alpha$. Thus, this case cannot arise.

Inductive step: Let k be the effective length of the run. Let $[r_1, \ldots, r_{i-1}]$ be a prefix of the run such that *EffLen*$([r_1, \ldots, r_{i-1}]) = k - 1$, and $\mathsf{nwpost}([r_1, \ldots, r_{i-1}]) \in \mathsf{NWLang}(\mathcal{P}_\mathcal{N}, (p, q))$. By the inductive hypothesis we can assume that $[\kappa_\Delta^{-1}(r_1), \ldots, \kappa_\Delta^{-1}(r_{i-1})]$ is a run of $\mathcal{P}$ and that $\mathsf{nwpost}([\kappa_\Delta^{-1}(r_1), \ldots, \kappa_\Delta^{-1}(r_{i-1})] \in \mathsf{NWLang}(\mathcal{P}, p)$. Note that r_{i-1} cannot be a pop rule by Lem. A.2. We perform a case analysis on the suffix $[r_i, \ldots, r_j]$ of the run to prove Lem. A.4. In each case, we assume that the prefix $[r_1, \ldots, r_{i-1}]$ transforms the configuration $\langle (p_0, q_0), e_{\text{main}} \rangle$ to a configuration $\langle (p, q), \gamma\, u \rangle$ and the prefix $[\kappa_\Delta^{-1}(r_1), \ldots, \kappa_\Delta^{-1}(r_{i-1})]$ transforms the configuration $\langle p_0, e_{\text{main}} \rangle$ to the configuration $\langle p, \gamma u' \rangle$. Note that u and u' are related. Namely, for each stack symbol $(r_c, q_c) \in u$, u' has the corresponding stack symbol r_c.

1. The suffix $[r_i, \dots, r_j]$ has length zero. This case invalidates the inductive assumptions because it does not increase the effective length of the run by 1.

2. The suffix $[r_i, \dots, r_j]$ has length one. In this case, $r_i = r_j$ and there are four possible forms that the rule r_j can have.

   a) $r_j = \langle (p, q), \gamma \rangle \hookrightarrow \langle (p', q'), \gamma' \rangle$, and $\gamma \in \Gamma^{\alpha}_{\mathcal{N}}$. The configuration $\langle (p, q), \gamma\, u \rangle$ is transformed to the configuration $\langle (p', q'), \gamma'\, u \rangle$. By definition, $\kappa^{-1}_{\Delta}(r_j) = \langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \rangle$, and the configuration $\langle p, \gamma\, u' \rangle$ is transformed to the configuration $\langle p', \gamma'\, u' \rangle$.

   b) $r_j = \langle (p, q_c), \gamma \rangle \hookrightarrow \langle (p', q'), e\, (r_c, q_c) \rangle$. The configuration $\langle (p, q), \gamma\, u \rangle$ is transformed to the configuration $\langle (p', q'), e\, (r_c, q_c)\, u \rangle$. By definition, $\kappa^{-1}_{\Delta}(r_j) = \langle p, \gamma \rangle \hookrightarrow \langle p', e\, r_c \rangle$, and the configuration $\langle p, \gamma\, u' \rangle$ is transformed to the configuration $\langle p', e\, r_c\, u' \rangle$.

   c) $r_j = \langle (p, q), \gamma \rangle \hookrightarrow \langle (p, q'), \gamma' \rangle$, and $\gamma' \in \Gamma^{\beta}_{\mathcal{N}}$. This case is not valid because the effective length of the run $[r_1, \dots, r_j]$ is $k - 1$.

   d) $r_j = \langle (p, q_r), \gamma \rangle \hookrightarrow \langle (p', q^x_r), \epsilon \rangle$. This case is not valid because a run cannot end with a pop rule by Lem. A.2.

3. The suffix $[r_i, \dots, r_j]$ has length two. In this case, one of the rules must have its left-hand-side stack symbol in $\Gamma^{\alpha}_{\mathcal{N}}$ and the other in $\Gamma^{\beta}_{\mathcal{N}}$. The only rules whose left-hand-side stack symbols are in $\Gamma^{\beta}_{\mathcal{N}}$ are of the form $\langle (p', q^x_r), (r_c, q_c) \rangle \hookrightarrow \langle (p', q'), r_c \rangle$. For a run of $\mathcal{P}_{\mathcal{N}}$, a rule of this form must be immediately preceded by a pop rule. This follows from *Construction 1*. Because r_{i-1} is not a pop rule, we only need to consider the following case:

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

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

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

Applying the suffix to the configuration $\langle p, x\, r_c\, u'''\rangle$ results in the configuration $\langle p, r_c\, u'''\rangle$.

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

$\square$

Lemma A.5. *NWLang*$(\mathcal{P}_\mathcal{N}, (p', q')) \subseteq$ *NWLang*$(\mathcal{N}, q')$.

Proof. The proof of Lem. A.5 is similar to the proof of Lem. A.4. Specifically, we provide a function such that for a run $[r_1, \ldots, r_j]$ of $\mathcal{P}_\mathcal{N}$, the function defines a run $[t_1, \ldots, t_j]$ of $\mathcal{N}$, where a run of $\mathcal{N}$ is a sequence of transitions that $\mathcal{N}$ can use to read a nested word NW. The proof makes use of the deconstructor $\kappa_\delta^{-1} : \Delta_\mathcal{N} \to \delta$, defined as follows:

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

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

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

Let $nw \in \mathsf{NWLang}(\mathcal{P}_\mathcal{N}, (p', q'))$ be the nested word (w, v). By the definition of $\mathsf{NWLang}(\mathcal{P}_\mathcal{N}, (p', q'))$, there exists a run $[r_1, \ldots, r_j]$ of $\mathcal{P}_\mathcal{N}$ that generates nw. We show that $\mathcal{N}$ can read nw and end in state q' using the run $\kappa_\delta^{-1}([r_1, \ldots, r_j])$. The proof is by induction on the effective length of the run. The only cases that need to be considered are exactly the cases enumerated in the proof of Lem. A.4. Hence, we omit the invalid cases instead of restating why they are invalid.

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

Inductive step: Let k be the effective length of the run. We assume that Lem. A.5 holds for the prefix $[r_1, \ldots, r_{i-1}]$ of the run whose effective length is $k - 1$. We perform a case analysis on the suffix $[r_i, \ldots, r_j]$ of the run to prove Lem. A.5. In each case, we assume that the prefix transforms the configuration $\langle (p_0, q_0), e_{\text{main}} \rangle$ to some configuration $\langle (p, q), \gamma u \rangle$.

1. The suffix $[r_i, \ldots, r_j]$ has length one. In this case, $r_i = r_j$ and there are two possible forms that the rule r_j can have such that it is a valid prefix and suffix of the run.

   a) $r_j = \langle (p, q), \gamma \rangle \hookrightarrow \langle (p', q'), \gamma' \rangle, \gamma \in \Gamma_\mathcal{N}^\alpha$. $\mathcal{N}$ can make a transition from state q to state q' when reading input symbol γ via $\kappa_\delta^{-1}(r_j)$.

   b) $r_j = \langle (p, q), \gamma \rangle \hookrightarrow \langle (p', q'), \gamma' \gamma'' \rangle$. $\mathcal{N}$ can make a transition from state q to state q' when reading input symbol γ via $\kappa_\delta^{-1}(r_j)$.

2. The suffix $[r_i, \dots, r_j]$ has length two and is of the form:

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

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

□

Lemma A.6. *NWLang*$(\mathcal{P}_\mathcal{N}, (p', q')) \supseteq$ *NWLang*$(\mathcal{P}, p') \cap$ *NWLang*$(\mathcal{N}, q')$

Proof. Let $nw \in \mathsf{NWLang}(\mathcal{P}, p') \cap \mathsf{NWLang}(\mathcal{N}, q')$ be the nested word (w, v). By the definition of $\mathsf{NWLang}(\mathcal{P}, p')$, there exists a run $[r_1, \dots, r_j]$ of $\mathcal{P}$ from $\langle p_0, e_{\mathrm{main}}\rangle$ such that $\mathsf{nwpost}([r_1, \dots, r_j]) = nw$. From Lem. A.3, we know that the length j of the run is equal to $|w|$. From the definition of $\mathsf{NWLang}(\mathcal{N}, q')$, there exists a run $[t_1, \dots, t_m]$ of $\mathcal{N}$ that reads nw, and leaves $\mathcal{N}$ in state $q' \in Q$. By the definition of NWAs, the number of transitions $m = |w|$ (each transition reads exactly one character from the input string). Thus, $j = m = |w|$. We show that for nw, there exists a run of $\mathcal{P}_\mathcal{N}$ that simulates both $\mathcal{P}$ and $\mathcal{N}$. The proof is via induction on the length j of the runs of $\mathcal{P}$ and $\mathcal{N}$.

Base case: If $j = 0$, then $nw = (\epsilon, \emptyset)$, and $nw = \mathsf{nwpost}([])$. By definition, $nw \in \mathsf{NWLang}(\mathcal{P}_\mathcal{N}, (p_0, q_0))$.

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

1. $r_j \in \Delta_1$ and $t_j \in \delta_i$. $\mathcal{P}_\mathcal{N}$ simulates the steps of $\mathcal{P}$ and $\mathcal{N}$ by the rule r_j and transition t_j, respectively, via the rule $\kappa(r_j, t_j) \in \Delta_\mathcal{N}$.

2. $r_j \in \Delta_2$ and $t_j \in \delta_c$. $\mathcal{P}_\mathcal{N}$ simulates the steps of $\mathcal{P}$ and $\mathcal{N}$ by r_j and t_j, respectively, via the rule $\kappa(r_j, t_j) \in \Delta_\mathcal{N}$.

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

□

A.2 Proof of Thm. 6.17

Proof. The proof is organized as follows:

1. $\mathsf{NWLang}^{\varphi}(\mathcal{E}_\mathcal{N}) \subseteq \mathsf{NWLang}(\mathcal{E})$ by Lem. A.11;

2. $\mathsf{NWLang}^{\varphi}(\mathcal{E}_\mathcal{N}) \subseteq \mathsf{NWLang}(\mathcal{N}), Q)$ by Lem. A.12; and

3. $\mathsf{NWLang}^{\varphi}(\mathcal{E}_\mathcal{N}) \supseteq \mathsf{NWLang}(\mathcal{N}, Q) \cap \mathsf{NWLang}(\mathcal{E})$ by Lem. A.13.

□

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

We first prove properties of the functions *build*,*flatten*, and ν for a run $\rho = [r_1, \dots, r_j]$ that are important for proving Thm. 6.17.

Lemma A.7. *flatten*$(z \otimes z', S) =$*flatten*$(z, S) \otimes z'$.

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

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

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

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

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

□

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

Proof. Let $(z', S') =$ *build*$([r_1, \dots, r_{j-1}])$. From Lem. A.7 and the definitions of

$\nu[r]$, *build*, and *flatten*, the following holds:

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

□

Lemma A.9. *For a run* $[r_1, \ldots, r_j]$*, if* r_j *is a call rule, then* $\nu([r_1, \ldots, r_j]) = \nu([r_1, \ldots, r_{j-1}]) \otimes f(r_j)$.

Proof. Let $(z', S') = \mathit{build}([r_1, \ldots, r_{j-1}])$. From Lem. A.7 and the definitions of $\nu[r]$, *build*, and ν, the following holds:

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

□

Lemma A.10. *Let $\rho = [r_1, \ldots, r_j]$ be a run of $\mathcal{E}$ and $\mathcal{E}_{\mathcal{N}}$ such that: $nw = \mathit{nwpost}(\rho)$, $nw \in \mathit{NWLang}(\mathcal{E})$, $nw \in \mathit{NWLang}(\mathcal{E}_{\mathcal{N}})$, $y = \nu_{\mathcal{E}}(\rho)$, and $z = \nu_{\mathcal{E}_{\mathcal{N}}}(\rho)$. If $y \neq \bar{0}$, then for all $q \xrightarrow{y'} q' \in z$ such that $y' \neq \bar{0}$, $y' = y$. Otherwise if $y = \bar{0}$, then $z = \bar{0}_{\mathcal{N}}$.*

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

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

Inductive step: We assume that Lem. A.10 holds for $\rho' = [r_1, \ldots, r_{j-1}]$, and prove that it holds for ρ by a case analysis of r_j. By our assumption, we have the following:

- $\nu_{\mathcal{E}}(\rho') = y'$.
- $\nu_{\mathcal{E}_{\mathcal{N}}}(\rho') = z'$, and for all $q \xrightarrow{y''} q' \in z'$ where $y'' \neq \bar{0}$, $y'' = y'$.

Let $\nu_{\mathcal{E}}(\rho) = y$ and $\nu_{\mathcal{E}_{\mathcal{N}}}(\rho) = z$.

1. $r_j = \langle p, n_1 \rangle \hookrightarrow \langle p', n_2 \rangle$. By Lem. A.8, $y = y' \otimes f(r_j)$. By definition, $f_A(r_j) = \{q' \xrightarrow{f(r_j)} q'' \mid q' \to q'' \in \delta_i^{|n_1}\}$. From Lem. A.8, the following holds:

$$
\begin{aligned}
\nu_{\mathcal{E}_{\mathcal{N}}}(\rho) &= \nu_{\mathcal{E}_{\mathcal{N}}}([r_1, \ldots, r_{j-1}, r_j]) \\
&= \nu_{\mathcal{E}_{\mathcal{N}}}([r_1, \ldots, r_{j-1}]) \otimes f_{\mathcal{N}}(r_j) \\
&= \nu_{\mathcal{E}_{\mathcal{N}}}(\rho') \otimes f_{\mathcal{N}}(r_j) \\
&= z' \otimes f_{\mathcal{N}}(r_j) \\
&= z' \otimes \{q' \xrightarrow{f(r_j)} q'' \mid q' \to q'' \in \delta_i^{|n_1}\} \\
&= \{q \xrightarrow{y' \otimes f(r_j)} q'' \mid q \xrightarrow{y'} q' \in z' \wedge q' \xrightarrow{f(r_j)} q'' \in f_{\mathcal{N}}(r_j)\} \\
&= \{q \xrightarrow{y} q'' \mid q \xrightarrow{y'} q' \in z' \wedge q' \xrightarrow{f(r_j)} q'' \in f_{\mathcal{N}}(r_j)\}
\end{aligned}
$$

2. $r_j = \langle p, n_c \rangle \hookrightarrow \langle p', e\ r_c \rangle$. The same argument as above applies here, simply replace Lem. A.8 with Lem. A.9 and $f_A(r_j) = \{q' \xrightarrow{f(r_j)} q'' \mid q' \to q'' \in \delta_c^{|n_c}\}$.

3. $r_j = \langle p, x \rangle \hookrightarrow \langle p', \epsilon \rangle$. Because nw is a nested word and by the definition of nwpost (i.e., nwpost is undefined for the case where a pop rule does not have a matching push rule), there must exist a matching push rule r_c for r_j in the run. Thus, with respect to $\nu_{\mathcal{E}}$, the following holds:

$$\begin{aligned}
\nu_{\mathcal{E}}([r_1, \ldots, r_{j-1}]) &= \mathit{flatten}_{\mathcal{E}}(\mathit{build}_{\mathcal{E}}([r_1, \ldots, r_{j-1}])) \\
&= \mathit{flatten}_{\mathcal{E}}(y_x, (y_c, r_c) || S) \\
&= \mathit{flatten}_{\mathcal{E}}(y_c \otimes f(r_c) \otimes y_x, S) \\
&= \mathit{flatten}_{\mathcal{E}}(\overline{1} \otimes (y_c \otimes f(r_c) \otimes y_x), S) \\
&= \mathit{flatten}_{\mathcal{E}}(\overline{1}, S) \otimes (y_c \otimes f(r_c) \otimes y_x) \qquad \text{(A.1)} \\
\nu_{\mathcal{E}}([r_1, \ldots, r_j]) &= \mathit{flatten}_{\mathcal{E}}(\mathit{build}_{\mathcal{E}}([r_1, \ldots, r_j])) \\
&= \mathit{flatten}_{\mathcal{E}}(\nu_{\mathcal{E}}[r_j](\mathit{build}_{\mathcal{E}}([r_1, \ldots, r_{j-1}]))) \\
&= \mathit{flatten}_{\mathcal{E}}(\nu_{\mathcal{E}}[r_j](y_x, (y_c, r_c) || S)) \\
&= \mathit{flatten}_{\mathcal{E}}(g(r_c)(y_c, f(r_c) \otimes y_x \otimes f(r_j)), S) \\
&= \mathit{flatten}_{\mathcal{E}}(\overline{1} \otimes (g(r_c)(y_c, f(r_c) \otimes y_x \otimes f(r_j))), S) \\
&= \mathit{flatten}_{\mathcal{E}}(\overline{1}, S) \otimes g(r_c)(y_c, f(r_c) \otimes y_x \otimes f(r_j)) \\
& \qquad \text{(A.2)}
\end{aligned}$$

Similar for $\nu_{\mathcal{E}_\mathcal{N}}$, the following holds:

$$\begin{aligned}
\nu_{\mathcal{E}_\mathcal{N}}([r_1,\dots,r_{j-1}]) &= \mathit{flatten}_{\mathcal{E}_\mathcal{N}}(\mathit{build}_{\mathcal{E}_\mathcal{N}}([r_1,\dots,r_{j-1}])) \\
&= \mathit{flatten}_{\mathcal{E}_\mathcal{N}}(z_x,(z_c,r_c)\|S) \\
&= \mathit{flatten}_{\mathcal{E}_\mathcal{N}}(z_c \otimes f_\mathcal{N}(r_c) \otimes z_x, S) \\
&= \mathit{flatten}_{\mathcal{E}_\mathcal{N}}(\bar{1}_\mathcal{N} \otimes (z_c \otimes f_\mathcal{N}(r_c) \otimes z_x), S) \\
&= \mathit{flatten}_{\mathcal{E}_\mathcal{N}}(\bar{1}_\mathcal{N}, S) \otimes (z_c \otimes f_\mathcal{N}(r_c) \otimes z_x)
\end{aligned} \tag{A.3}$$

$$\begin{aligned}
\nu_{\mathcal{E}_\mathcal{N}}([r_1,\dots,r_j]) &= \\
&= \mathit{flatten}_{\mathcal{E}_\mathcal{N}}(\mathit{build}_{\mathcal{E}_\mathcal{N}}([r_1,\dots,r_j])) \\
&= \mathit{flatten}_{\mathcal{E}_\mathcal{N}}(\nu_{\mathcal{E}_\mathcal{N}}[r_j](\mathit{build}_{\mathcal{E}_\mathcal{N}}([r_1,\dots,r_{j-1}]))) \\
&= \mathit{flatten}_{\mathcal{E}_\mathcal{N}}(\nu_{\mathcal{E}_\mathcal{N}}[r_j](z_x,(z_c,r_c)\|S)) \\
&= \mathit{flatten}_{\mathcal{E}_\mathcal{N}}(g_\mathcal{N}(r_c)(z_c, f_\mathcal{N}(r_c) \otimes z_x \otimes f_\mathcal{N}(r_j)), S) \\
&= \mathit{flatten}_{\mathcal{E}_\mathcal{N}}(\bar{1}_\mathcal{N} \otimes (g_\mathcal{N}(r_c)(z_c, f_\mathcal{N}(r_c) \otimes z_x \otimes f_\mathcal{N}(r_j))), S) \\
&= \mathit{flatten}_{\mathcal{E}_\mathcal{N}}(\bar{1}_\mathcal{N}, S) \otimes g_\mathcal{N}(r_c)(z_c, f_\mathcal{N}(r_c) \otimes z_x \otimes f_\mathcal{N}(r_j))
\end{aligned} \tag{A.4}$$

Because both Eqns. (A.1) and (A.2) contain $\mathit{flatten}_\mathcal{E}(\bar{1}, S)$ on the left-hand side of the extend ($\otimes$), it contributes the same value in both cases. We denote the right-hand sides of the extend of Eqns. (A.1) and (A.2) by y_{j-1} and y_j, respectively. Likewise, both Eqns. (A.3) and (A.4) contain $\mathit{flatten}_{\mathcal{E}_\mathcal{N}}(\bar{1}_\mathcal{N}, S)$ on the left-hand side of the extend, and thus it contributes the same value in both cases. Similarly, we denote the right-hand side of the extend of Eqns. (A.3) and (A.4) by z_{j-1} and z_j, respectively. From our assumptions, we have the following:

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

where $y_{j-1} = y_c \otimes f(r_c) \otimes y_x$. Notice that the right-hand side of the extend in Eqn. (A.1) annotates the tuples in z_{j-1}. From the definition of $g_{\mathcal{N}}$ (Eqn. (6.1)), the following holds:

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

where $y_j = g(r_c)(y_c, f(r_c) \otimes y_x \otimes f(r_j))$. Notice that the right-hand side of Eqn. (A.2) annotates the tuples in z_j. Thus, we have proved the inductive step.

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

Lemma A.11. *NWLang*$^{\varphi}(\mathcal{E}_{\mathcal{N}}) \subseteq$ *NWLang*$(\mathcal{E})$.

Proof. We prove Lem. A.11 by showing that $\mathsf{NWLang}(\mathcal{E}_{\mathcal{N}}) \subseteq \mathsf{NWLang}(\mathcal{E})$. Because both $\mathcal{E}_{\mathcal{N}}$ and $\mathcal{E}$ have the same underlying PDS $\mathcal{P}$, a run of $\mathcal{E}_{\mathcal{N}}$ is a run of $\mathcal{E}$. Specifically, for a run $\rho = [r_1, \ldots, r_j]$, $\rho \in \mathsf{Runs}(\mathcal{E}_{\mathcal{N}})$ and $\rho \in \mathsf{Runs}(\mathcal{E})$. The NWLang for an EWPDS is defined in terms of runs and the weighted valuation for a run. Thus, we need only show that $\nu_{\mathcal{E}}(\rho) = \bar{0} \implies \nu_{\mathcal{E}_{\mathcal{N}}}(\rho) = \bar{0}_{\mathcal{N}}$, which follows from Lem. A.10. From Defn. 6.14, $\mathsf{NWLang}^{\varphi}(\mathcal{E}_{\mathcal{N}}) \subseteq \mathsf{NWLang}(\mathcal{E}_{\mathcal{N}})$, which proves Lem. A.11.

□

Lemma A.12. *NWLang*$^{\varphi}(\mathcal{E}_{\mathcal{N}}) \subseteq$ *NWLang*$(\mathcal{N}, Q)$.

Proof. For a nested word $nw \in \mathsf{NWLang}(\mathcal{E}_{\mathcal{N}})$, let $[r_1, \ldots, r_j]$ be a run that generates nw, and let $z = \nu_{\mathcal{E}_{\mathcal{N}}}([r_1, \ldots, r_j])$ be the weighted valuation of the run. We show that for each $q_0 \xrightarrow{y} q \in z$ such that $y \neq \overline{0}$, $nw \in \mathsf{NWLang}(\mathcal{N}, q)$. Lem. A.12 follows from this.

Lem. A.10 proves that for a run $\rho = [r_1, \ldots, r_j]$ of $\mathcal{E}_{\mathcal{N}}$, the weighted valuation $z = \nu_{\mathcal{E}_{\mathcal{N}}}(\rho)$ of the run conceptually consists of two parts. The first part is the relational part of the weighted relation, which models the NWA $\mathcal{N}$. The second part is the weighted part of weighted relations, which models $\mathcal{E}$. We take advantage of this fact by observing that if we mask off the second part, then the weight domain resembles an ordinary relational weight domain. In fact, relations are a degenerate form of weighted relations, where the weight domain is the Boolean semiring $(\{\overline{1}_{\mathbb{B}}, \overline{0}_{\mathbb{B}}\}, \oplus_{\mathbb{B}}, \otimes_{\mathbb{B}}, \overline{0}_{\mathbb{B}}, \overline{1}_{\mathbb{B}})$. From this observation, we take the following approach. First, we define an operation called *reduce*, which performs the masking referred to above, and allows us to reason bout (essentially) unweighted runs of $\mathcal{E}_{\mathcal{N}}$. Second, we show that the nested-word language of the reduced $\mathcal{E}_{\mathcal{N}}$ is a subset of $\mathsf{NWLang}(\mathcal{N}, Q)$. Because the "reduce" operation can only make $\mathsf{NWLang}(\mathcal{E}_{\mathcal{N}})$ larger, and because $\mathsf{NWLang}^{\varphi}(\mathcal{E}_{\mathcal{N}})$ can only be a subset, we complete the proof.

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

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

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

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

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

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

For an EWPDS $\mathcal{E} = (\mathcal{P}, \mathcal{S}, \mathsf{f}, \mathsf{g})$, the operation *reduce* that produces defines a new EWPDS $\mathcal{E}_{\mathbb{B}} = (\mathcal{P}, \mathcal{S}_{\mathbb{B}}, \mathsf{f}_{\mathbb{B}}, \mathsf{g}_{\mathbb{B}})$, where $\mathsf{f}_{\mathbb{B}}(r) = \alpha_{\mathbb{B}}(\mathsf{f}(r))$, and $\mathsf{g}_{\mathbb{B}}(r) = \lambda w_1.\lambda w_2.w_1 \otimes_{\mathbb{B}} w_2$. Interestingly, $\mathcal{P} = \mathcal{E}_{\mathbb{B}}$, and thus $\mathsf{NWLang}(\mathcal{P}) = \mathsf{NWLang}(\mathcal{E}_{\mathbb{B}})$. This is easy to see as all runs of $\mathcal{E}_{\mathbb{B}}$ have the weight $\overline{1}_{\mathbb{B}}$, which simply indicates reachability. This is precisely what a run of $\mathcal{P}$ represents. Notice that $\mathsf{NWLang}(\mathcal{P}) \supseteq \mathsf{NWLang}(\mathcal{E})$ because of the argument made above. Let $\mathcal{E}^{\mathbb{B}}_{\mathcal{N}}$ be the EWPDS that is generated by *Construction 2* from $\mathcal{E}_{\mathbb{B}}$ and $\mathcal{N}$. Because of $\mathcal{E}_{\mathbb{B}}$ over-approximates $\mathcal{E}$, we have the following: $\mathsf{NWLang}(\mathcal{E}_{\mathcal{N}}) \subseteq \mathsf{NWLang}(\mathcal{E}^{\mathbb{B}}_{\mathcal{N}})$. Thus, proving that $\mathsf{NWLang}(\mathcal{E}^{\mathbb{B}}_{\mathcal{N}}) \subseteq \mathsf{NWLang}(\mathcal{N}, Q)$ is also a proof that $\mathsf{NWLang}(\mathcal{E}_{\mathcal{N}}) \subseteq \mathsf{NWLang}(\mathcal{N}, Q)$, which completes the proof of this lemma. Because all weights that annotate a weighted relation of $\mathcal{E}^{\mathbb{B}}_{\mathcal{N}}$ are the one weight $\overline{1}_{\mathbb{B}}$, we omit the discussion of the weights of $\mathcal{E}^{\mathbb{B}}_{\mathcal{N}}$ from the rest of the proof

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

$\mathsf{NWLang}(\mathcal{N}, q)$ is a member of $\mathsf{NWLang}(\mathcal{N}, q_0)$.

Inductive step: We assume that for length $j-1$, $nw' = \mathsf{nwpost}([r_1, \ldots, r_{j-1}])$, $z' = \nu([r_1, \ldots, r_{j-1}])$, and for each $q_0 \to q' \in z'$, $nw' \in \mathsf{NWLang}(\mathcal{N}, q')$. We now consider the possible forms of rule r_j.

1. $r_j = \langle p, n_1 \rangle \hookrightarrow \langle p', n_2 \rangle$. From Lem. A.8, $z = z' \otimes f(r_j)$. From *Construction 2*, $f(r_j) = \delta_i^{|n_1}$. By the definition of $\delta_i^{|n_1}$, for each $q' \to q \in f(r_j)$, the NWA $\mathcal{N}$ can make a transition from state q' to state q when reading input symbol n_1. By the definition of $z' \otimes f(r_j)$, for each $q_0 \to q' \in z'$ and $q' \to q \in f(r_j)$, the weight z contains the tuple $q_0 \to q$. Thus, $nw \in \mathsf{NWLang}(\mathcal{N}, q)$

2. $r_j = \langle p, n_c \rangle \hookrightarrow \langle p', e\, r_c \rangle$. From Lem. A.9, $z = z' \otimes f(r_j)$. From *Construction 2*, $f(r_j) = \delta_c^{|n_c}$. By the definition of $\delta_c^{|n_c}$, for each $q' \to q \in f(r_j)$, the NWA $\mathcal{N}$ can make a transition from state q' to state q when reading input symbol n_c. By the definition of $z' \otimes f(r_j)$, for each $q_0 \to q' \in z'$ and $q' \to q \in f(r_j)$, the weight z contains the tuple $q_0 \to q$. Thus $nw \in \mathsf{NWLang}(\mathcal{N}, q)$.

3. $r_j = \langle p, x \rangle \hookrightarrow \langle p', \epsilon \rangle$. Because nw is a nested word and by the definition of nwpost (i.e., nwpost is undefined for the case where a pop rule does not have a matching push rule), there must exist a matching push rule r_c for r_j in the run. Therefore, *build*$([r_1, \ldots, r_{j-1}])$ must return a pair of the form $(z_x, (z_c, r_c)||S')$. From our assumptions, the following must hold:

$$\begin{aligned} z' &= \nu([r_1, \ldots, r_{j-1}]) \\ &= \textit{flatten}(\textit{build}([r_1, \ldots, r_{j-1}])) \\ &= \textit{flatten}(z_x, (z_c, r_c)||S') \\ &= \textit{flatten}((z_c \otimes f(r_c) \otimes z_x), S') \\ &= \textit{flatten}(\{q'' \to q' \mid \exists a : z_c(q'', a) \wedge (f(r_c) \otimes z_x)(a, q')\}, S') \end{aligned}$$

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

$$\begin{aligned} z &= \nu([r_1, \ldots, r_{j-1}, r_j]) \\ &= \textit{flatten}(\textit{build}([r_1, \ldots, r_{j-1}, r_j])) \\ &= \textit{flatten}(\nu[r_j](\textit{build}([r_1, \ldots, r_{j-1}]))) \\ &= \textit{flatten}(\nu[r_j]((z_x, (z_c, r_c) \| S'))) \\ &= \textit{flatten}(g(r_c)(z_c, f(r_c) \otimes z_x \otimes f(r_j)), S') \\ &= \textit{flatten}(\{g'' \to q \mid \exists a, b : \underline{z_c(q'', a) \wedge (f(r_c) \otimes z_x)(a, q')} \\ &\qquad\qquad \wedge f(r_j)(q', b) \wedge \hat{\delta}(b, a, q)\}, S') \end{aligned}$$

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

□

Lemma A.13. $\mathit{NWLang}^{\varphi}(\mathcal{E}_{\mathcal{N}}) \supseteq \mathit{NWLang}(\mathcal{N}, Q) \cap \mathit{NWLang}(\mathcal{E})$

Proof. Let $nw = (w, \nu)$ be a nested word in $\mathsf{NWLang}(\mathcal{N}, Q) \cap \mathsf{NWLang}(\mathcal{E})$ such that $|w| = j$. From the definition of $\mathsf{NWLang}(\mathcal{N}, Q)$, there must exist a run $[t_1, \ldots, t_j]$ that $\mathcal{N}$ can use to read nw. For the PDS component $\mathcal{P}$ of $\mathcal{E}$ and $\mathcal{E}_{\mathcal{N}}$, $\Gamma^{\beta} = \emptyset$. From the definition of $\mathsf{NWLang}(\mathcal{E})$ and by Lem. A.3, there must exist a run $[r_1, \ldots, r_j]$ of $\mathcal{E}$ such that $\mathsf{nwpost}([r_1, \ldots, r_j]) = nw$. The proof is a simulation proof and is performed by induction on the length j. That is, we show that there exists a run of $\mathcal{E}_{\mathcal{N}}$ such that the run simultaneously models the runs $[t_1, \ldots, t_j]$ of $\mathcal{N}$ and $[r_1, \ldots, r_j]$ of $\mathcal{E}$.

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

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

- $nw' = \mathsf{nwpost}([r_1, \ldots, r_{j-1}]) \in \mathsf{NWLang}(\mathcal{E}) \cap \mathsf{NWLang}(\mathcal{N}, Q)$.
- $y' = \nu_{\mathcal{E}}([r_1, \ldots, r_{j-1}]) \neq \bar{0}$.
- $y = \nu_{\mathcal{E}}([r_1, \ldots, r_j]) \neq \bar{0}$.
- $z' = \nu_{\mathcal{E}_{\mathcal{N}}}([r_1, \ldots, r_{j-1}]) \neq \bar{0}_{\mathcal{N}}$.
- $z = \nu_{\mathcal{E}_{\mathcal{N}}}([r_1, \ldots, r_j]) \neq \bar{0}_{\mathcal{N}}$.
- q' be the state of $\mathcal{N}$ after making $j - 1$ transitions.
- q be the state of $\mathcal{N}$ after making j transitions.

Because $\mathcal{E}_{\mathcal{N}}$ and $\mathcal{E}$ have the same underlying PDS, we know that any run of $\mathcal{E}$ is a run of $\mathcal{E}_{\mathcal{N}}$. We now need to show that for each $q_0 \xrightarrow{y'} q' \in z'$ such that $y' \neq \bar{0}$, we have $q_0 \xrightarrow{y} q \in z$, where q is the state that $\mathcal{N}$ is in after making j transitions. This follows from Lem. A.10.

□

A.3 Proof of Thm. 7.8

Proof. From Thm. 7.3, we know that for paths π_1 and π_2 with rule sequences ρ_1 and ρ_2 from PDSs $\mathcal{P}_1$ and $\mathcal{P}_2$, respectively, where π_1 and π_2 begin with a disjoint set of initially held locks $\mathcal{I}_1$ and $\mathcal{I}_2$, there exists a compatible scheduling of π_1 and π_2 iff $\mathsf{Compatible}(\eta(\rho_1, \mathcal{I}_1), \eta(\rho_2, \mathcal{I}_2))$.

If Alg. 7.1 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) for a path π_1 (π_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 $\mathsf{Compatible}(\widehat{\mathsf{LH}}_1, \widehat{\mathsf{LH}}_2)$. Because of the Decomposition Theorem and the definition of $\mathsf{Compatible}(\widehat{\mathsf{LH}}_1, \widehat{\mathsf{LH}}_2)$, 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 $\mathcal{N}$ is in its accepting state.

If Alg. 7.1 returns false, then there does not exist two tuples of locks histories. This occurs if either one (or both) of the PDSs does not have a path that can drive the PDS $\mathcal{N}$ to the accepting state, and thus it is not possible for an interleaved execution of Π to drive IPA $\mathcal{N}$ to the accepting state. Otherwise, there must not exist two tuples that are in the Compatible relation. From the definition of Compatible, there must be some phase such that the lock histories are incompatible, and thus no interleaved execution exists.

The generalization to an arbitrary number of PDSs proceeds similarly.

□

A.4 Proof of Thm. 7.12

Proof. The proof proceeds as follows: (i) show by induction that Alg. 7.1 and Alg. 7.2 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. $\mathcal{P} = (\mathsf{P}, \mathsf{Lab}, \Gamma, \Delta, c_0)$ is a PDS

2. $\mathcal{N} = (Q, \mathit{Id}, \Sigma, \delta)$ is a IPA
3. $\mathcal{P}^{\mathcal{N}} = \big(P^{\mathcal{N}}, \emptyset, \Gamma, \Delta^{\mathcal{N}}, \langle (p_0, q_1, \widehat{\mathsf{LH}}_0), \gamma_0 \rangle\big)$ is the (unlabeled) PDS that results from combining $\mathcal{P}$ with $\mathcal{N}$ as defined in §7.5
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 $\mathcal{N}$ as defined in §7.7
5. $\rho^{\mathcal{P}} = [r_1^{\mathcal{P}}, \ldots, r_n^{\mathcal{P}}]$ is a rule sequence from $\mathcal{P}$
6. $\rho^{\mathcal{N}} = [r_1^{\mathcal{N}}, \ldots, r_n^{\mathcal{N}}]$ is a rule sequence from $\mathcal{P}^{\mathcal{N}}$
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 $\mathsf{inflate}$ 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 $\rho^{\mathcal{N}}$ of $\mathcal{P}^{\mathcal{N}}$ 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 $\mathsf{inflate}$ "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 $\mathcal{P}^{\mathcal{N}}$ only modifies the lock-history tuple entries for the phases that it has been in or is currently executing in. The function $\mathsf{deflate}$ simply undoes the result of $\mathsf{inflate}$, i.e., $c_{x-1} z^{x-1} = \mathsf{deflate}(\mathsf{inflate}(c_{x-1} z^{x-1}, x), x)$.

Let $c_0^{\mathcal{N}} \Rightarrow^{\rho^{\mathcal{N}}} c^{\mathcal{N}}$ denote that $\mathcal{P}^{\mathcal{N}}$ makes a transition to a configuration $c^{\mathcal{N}}$ from configuration $c_0^{\mathcal{N}}$ when executing rule sequence $\rho^{\mathcal{N}}$. 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:

$$c_0^{\mathcal{N}} \Rightarrow^{\rho^{\mathcal{N}}} \langle (p, q_x, \widehat{\mathsf{LH}}), u \rangle \Leftrightarrow c_0^{\mathcal{W}} \Rightarrow^{\rho^{\mathcal{W}}} \langle (p, q_x), u \rangle \wedge \langle \mathsf{LH}_0, \widehat{\mathsf{LH}} \rangle \in \mathsf{inflate}(\mathit{val}(\rho^{\mathcal{W}}), x).$$

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

Show ⇒.
For rule sequence $\rho^{\mathcal{N}} = [r_1^{\mathcal{N}}, \ldots, r_n^{\mathcal{N}}]$, assume that $c_0^{\mathcal{N}} \Rightarrow^{\rho^{\mathcal{N}}} \langle (p, q_x, \widehat{\mathsf{LH}}), u \rangle$. 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 $\mathcal{P}^{\mathcal{N}}$ that occurs when firing a rule sequence of $\mathcal{P}^{\mathcal{N}}$.

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

  For the base case, there is only one rule: $r_1^{\mathcal{N}} = \langle (p_0, q_1, \widehat{\mathsf{LH}}_0), \gamma_0 \rangle \hookrightarrow \langle (p, q_x, \widehat{\mathsf{LH}}), u \rangle$. From the definition of $\mathcal{P}^{\mathcal{N}}$, there must be the rule $r_1^{\mathcal{P}} = \langle p_0, \gamma_0 \rangle \overset{a}{\hookrightarrow} \langle p, u \rangle$ 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 $r_1^{\mathcal{N}}$ 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) $\widehat{\mathsf{LH}} = \widehat{\mathsf{LH}}_0[1 \mapsto \mathsf{post}(\widehat{\mathsf{LH}}_0[1], a)] = \langle \mathsf{post}(\mathsf{LH}_0, a), \mathsf{LH}_0^{|Q|-1} \rangle$
     b) $f(r_1^{\mathcal{W}}) = \{ \langle \mathsf{LH}, \mathsf{post}(\mathsf{LH}, a) \rangle \mid \mathsf{LH} \in \mathcal{LH} \} z^0$
     c) $\mathsf{inflate}(\mathit{val}([r_1^{\mathcal{W}}]), 1) = \{ \langle \mathsf{LH}, \mathsf{post}(\mathsf{LH}, a), \mathsf{LH}_0^{|Q|-1} \rangle \mid \mathsf{LH} \in \mathcal{LH} \} z^{|Q|-1}$
     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 $\rho_n^{\mathcal{N}} = [r_1^{\mathcal{N}}, \ldots, r_{n-1}^{\mathcal{N}}, r_n^{\mathcal{N}}]$, and assume that for the first $n - 1$ rules of the sequence, $c_0^{\mathcal{N}} \Rightarrow^{\rho_{n-1}^{\mathcal{N}}} \langle (p, q_x, \widehat{\mathsf{LH}}), \gamma u \rangle$. 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 $r_n^{\mathcal{N}} = \langle (p, q_x, \widehat{\mathsf{LH}}), \gamma \rangle \hookrightarrow \langle (p', q_y, \widehat{\mathsf{LH}}'), u' \rangle$, then $c_0^{\mathcal{N}} \Rightarrow^{\rho_n^{\mathcal{N}}} \langle (p', q_y, \widehat{\mathsf{LH}}'), u'u \rangle$. From the definition of $\mathcal{P}^{\mathcal{N}}$, there must exist a rule $\langle p, \gamma \rangle \overset{a}{\hookrightarrow} \langle p', u' \rangle \in \Delta$ 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 $r_n^{\mathcal{N}} = \langle (p, q_x, \widehat{\mathsf{LH}}), \gamma \rangle \hookrightarrow \langle (p', q_y, \widehat{\mathsf{LH}}'), u' \rangle$.

1. If $x = y$, then
   a) $\widehat{\mathsf{LH}}' = \widehat{\mathsf{LH}}[x \mapsto \mathsf{post}(\widehat{\mathsf{LH}}[x], a)] =$
   $\langle \mathsf{LH}^1, \ldots, \mathsf{post}(\mathsf{LH}^x, a), \mathsf{LH}_0^{x+1}, \ldots, \mathsf{LH}_0^{|Q|} \rangle$
   b) $f(r_n^{\mathcal{W}}) = \big\{ \langle \mathsf{LH}, \mathsf{post}(\mathsf{LH}, a) \rangle \mid \mathsf{LH} \in \mathcal{LH} \big\} z^0$
   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) $\langle \mathsf{LH}_0, \mathsf{LH}^1, \ldots, \mathsf{post}(\mathsf{LH}^x, a) \rangle \in \mathsf{deflate}(\mathsf{inflate}(\mathit{val}(\rho_{n-1}^{\mathcal{W}}), x), x) \otimes f(r_n^{\mathcal{W}})$

e) $\mathsf{inflate}(\mathit{val}(\rho^{\mathcal{W}}_{n}), x) = \mathsf{inflate}(\mathit{val}(\rho^{\mathcal{W}}_{n-1}) \otimes f(r^{\mathcal{W}}_{1}), x)$

f) $\langle \mathsf{LH}_0, \widehat{\mathsf{LH}}' \rangle \in \mathsf{inflate}(\mathit{val}(\rho^{\mathcal{W}}_{n}), x)$

2. Otherwise $y = x + 1$, and

a) $\widehat{\mathsf{LH}}' = \widehat{\mathsf{LH}}[y \mapsto \mathsf{ptrans}(\widehat{\mathsf{LH}}[x])] =$
$\langle \mathsf{LH}^1, \ldots, \mathsf{LH}^x, \mathsf{ptrans}(\mathsf{LH}^x), \ldots, \mathsf{LH}^{|Q|}_0 \rangle$

b) $f(r^{\mathcal{W}}_{n}) = \{\langle \mathsf{LH}, \mathsf{LH}, \mathsf{ptrans}(\mathsf{LH}) \rangle \mid \mathsf{LH} \in \mathcal{LH}\} z^1$.

c) $\langle \mathsf{LH}_0, \mathsf{LH}^1, \ldots, \mathsf{LH}^x \rangle \in \mathsf{deflate}(\mathsf{inflate}(\mathit{val}(\rho^{\mathcal{W}}_{n-1}), x), x)$, by the induction hypothesis

d) $\langle \mathsf{LH}_0, \mathsf{LH}^1, \ldots, \mathsf{LH}^x, \mathsf{ptrans}(\mathsf{LH}^x) \rangle \in$
$\mathsf{deflate}(\mathsf{inflate}(\mathit{val}(\rho^{\mathcal{W}}_{n-1}), x), x) \otimes f(r^{\mathcal{W}}_{n})$

e) $\mathsf{inflate}(\mathit{val}(\rho^{\mathcal{W}}_{n}), y) = \mathsf{inflate}(\mathit{val}(\rho^{\mathcal{W}}_{n-1}) \otimes f(r^{\mathcal{W}}_{1}), y)$

f) $\langle \mathsf{LH}_0, \widehat{\mathsf{LH}}' \rangle \in \mathsf{inflate}(\mathit{val}(\rho^{\mathcal{W}}_{n}), y)$

Show $\Leftarrow$.
For a rule sequence $\rho^{\mathcal{W}}_{n} = [r^{\mathcal{W}}_{1}, \ldots, r^{\mathcal{W}}_{n}]$, assume that $c^{\mathcal{W}}_{0} \Rightarrow^{\rho^{\mathcal{W}}} \langle (p, q_x), u \rangle$ and that $\langle \mathsf{LH}_0, \widehat{\mathsf{LH}} \rangle \in \mathsf{inflate}(\mathit{val}(\rho^{\mathcal{W}}_{n}), x)$. We show how to construct a rule sequence $\rho^{\mathcal{N}} = [r^{\mathcal{N}}_{1}, \ldots, r^{\mathcal{N}}_{n}]$ such that $c^{\mathcal{N}}_{0} \Rightarrow^{\rho^{\mathcal{N}}} \langle (p, q_x, \widehat{\mathsf{LH}}), u \rangle$. 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: $r^{\mathcal{W}}_{1} = \langle (p_0, q_1), \gamma_0 \rangle \hookrightarrow \langle (p, q_x), u \rangle$. From the definition of $\mathcal{W}$, there must exist the rule $r^{\mathcal{P}}_{1} = \langle p_0, \gamma_0 \rangle \overset{a}{\hookrightarrow} \langle p, u \rangle \in \Delta$, and a transition $(q_1, i, a, q_x) \in \delta$. Thus, by the definition of $\mathcal{P}^{\mathcal{N}}$, there must be a rule $r^{\mathcal{N}}_{1} = \langle (p_0, q_1, \widehat{\mathsf{LH}}_0), \gamma_0 \rangle \hookrightarrow \langle (p, q_x, \widehat{\mathsf{LH}}'), u \rangle$. We perform a case analysis on $r^{\mathcal{W}}_{1}$ to show that $\langle \mathsf{LH}_0, \widehat{\mathsf{LH}} \rangle \in \mathsf{inflate}(\mathit{val}([r^{\mathcal{W}}_{1}]), x) \Rightarrow \widehat{\mathsf{LH}}' = \widehat{\mathsf{LH}}$.

1. If $x = 1$, then

a) $f(r^{\mathcal{W}}_{1}) = \{\langle \mathsf{LH}, \mathsf{post}(\mathsf{LH}, a) \rangle \mid \mathsf{LH} \in \mathcal{LH}\} z^0$

b) $\langle \mathsf{LH}_0, \mathsf{post}(\mathsf{LH}_0, a), \mathsf{LH}_0^{Q-1} \rangle \in \mathsf{inflate}(\mathit{val}([r_1^{\mathcal{W}}]), x)$

c) $\widehat{\mathsf{LH}}' = \widehat{\mathsf{LH}}_0[1 \mapsto \mathsf{post}(\widehat{\mathsf{LH}}_0[1], a)] = \langle \mathsf{post}(\mathsf{LH}_0, a), \mathsf{LH}_0^{|Q|-1} \rangle$.

d) $\widehat{\mathsf{LH}}' = \widehat{\mathsf{LH}}$

2. Otherwise $x = 2$, then

a) $f(r_1^{\mathcal{W}}) = c_1 z^1 = \big\{ \langle \mathsf{LH}, \mathsf{LH}, \mathsf{ptrans}(\mathsf{LH}) \rangle \mid \mathsf{LH} \in \mathcal{LH} \big\} z^1$.

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 $\rho_n^{\mathcal{W}} = [r_1^{\mathcal{W}}, \dots, r_{n-1}^{\mathcal{W}}, r_n^{\mathcal{W}}]$, 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 $\rho_{n-1}^{\mathcal{N}} = [r_1^{\mathcal{N}}, \dots, r_{n-1}^{\mathcal{N}}]$ such that $c_0^{\mathcal{N}} \Rightarrow^{\rho_{n-1}^{\mathcal{N}}} \langle (p, q_x, \widehat{\mathsf{LH}}), \gamma u \rangle$. Furthermore, let $\widehat{\mathsf{LH}} = \langle \mathsf{LH}^1, \dots, \mathsf{LH}^x, \mathsf{LH}_0^{x+1}, \dots, \mathsf{LH}_0^{|Q|} \rangle$; then $\langle \mathsf{LH}_0, \mathsf{LH}^1, \dots, \mathsf{LH}^x \rangle \in \mathit{val}(\rho_{n-1}^{\mathcal{W}})$. 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 $\mathcal{P}^{\mathcal{N}}$, there must exist a rule $r_n^{\mathcal{N}} = \langle (p, q_x, \widehat{\mathsf{LH}}), \gamma \rangle \hookrightarrow \langle (p', q_y, \widehat{\mathsf{LH}}'), u' \rangle \in \Delta^{\mathcal{N}}$, and $c_0^{\mathcal{N}} \Rightarrow^{\rho_n^{\mathcal{W}}} \langle (p', q_y, \widehat{\mathsf{LH}}'), u'u \rangle$. 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) $f(r_n^{\mathcal{W}}) = \big\{ \langle \mathsf{LH}, \mathsf{post}(\mathsf{LH}, a) \rangle \mid \mathsf{LH} \in \mathcal{LH} \big\} z^0$

b) $\langle \mathsf{LH}_0, \mathsf{LH}^1, \dots, \mathsf{LH}^x \rangle \in \mathit{val}(\rho_{n-1}^{\mathcal{W}})$, by the induction hypothesis

c) $\langle \mathsf{LH}_0, \mathsf{LH}^1, \dots, \mathsf{post}(\mathsf{LH}^x, a) \rangle \in \mathit{val}(\rho_{n-1}^{\mathcal{W}}) \otimes f(r_n^{\mathcal{W}})$

d) $\langle \mathsf{LH}_0, \mathsf{LH}^1, \dots, \mathsf{post}(\mathsf{LH}^x, a) \rangle \in \mathit{val}(\rho_n^{\mathcal{W}})$

e) $\langle \mathsf{LH}_0, \mathsf{LH}^1, \ldots, \mathsf{post}(\mathsf{LH}^x, a), \mathsf{LH}_0^{x+1}, \ldots, \mathsf{LH}_0^{|Q|} \rangle \in \mathsf{inflate}(val(\rho_n^{\mathcal{W}}), x)$

f) $\widehat{\mathsf{LH}}' = \widehat{\mathsf{LH}}[x \mapsto \mathsf{post}(\widehat{\mathsf{LH}}[x], a)] =$
$\langle \mathsf{LH}^1, \ldots, \mathsf{post}(\mathsf{LH}^x, a), \mathsf{LH}_0^{x+1}, \ldots, \mathsf{LH}_0^{|Q|} \rangle$

g) $\widehat{\mathsf{LH}}' = \widehat{\mathsf{LH}}$

2. Otherwise $y = x + 1$, and

a) $\mathsf{f}(r_n^{\mathcal{W}}) = c_1 z^1 = \big\{ \langle \mathsf{LH}, \mathsf{LH}, \mathsf{ptrans}(\mathsf{LH}) \rangle \mid \mathsf{LH} \in \mathcal{LH} \big\} z^1$

b) $\langle \mathsf{LH}_0, \mathsf{LH}^1, \ldots, \mathsf{LH}^x \rangle \in val(\rho_{n-1}^{\mathcal{W}})$, by the induction hypothesis

c) $\langle \mathsf{LH}_0, \mathsf{LH}^1, \ldots, \mathsf{LH}^x, \mathsf{ptrans}(\mathsf{LH}^x) \rangle \in val(\rho_{n-1}^{\mathcal{W}}) \otimes \mathsf{f}(r_n^{\mathcal{W}})$

d) $\langle \mathsf{LH}_0, \mathsf{LH}^1, \ldots, \mathsf{LH}^x, \mathsf{ptrans}(\mathsf{LH}^x) \rangle \in val(\rho_n^{\mathcal{W}})$

e) $\langle \mathsf{LH}_0, \mathsf{LH}^1, \ldots, \mathsf{LH}^x, \mathsf{ptrans}(\mathsf{LH}^x), \ldots, \mathsf{LH}_0^{|Q|} \rangle \in \mathsf{inflate}(val(\rho_n^{\mathcal{W}}), x)$

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 $\mathcal{P}^{\mathcal{N}}$, 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. 7.2 computes the same result as Alg. 7.1, and thus completes the proof of correctness.

□