Informatik

Lock-Sensitive Analysis of Parallel Programs

Inaugural-Dissertation
zur Erlangung des Doktorgrades
der Naturwissenschaften im Fachbereich
Mathematik und Informatik
der Mathematisch-Naturwissenschaftlichen Fakultät
der Westfälischen Wilhelms-Universität Münster

vorgelegt von
Peter Lammich
aus Freiburg im Breisgau
– 2011 –

Dekan:	Prof. Dr. Matthias Löwe
Erster Gutachter:	Prof. Dr. Markus Müller-Olm
Zweiter Gutachter:	Prof. Dr. Helmut Seidl
Tag der mündlichen Prüfung:	28.6.2011
Tag der Promotion:	28.6.2011

Abstract

We present a model-checking algorithm for dynamic pushdown networks with monitors (Monitor-DPNs). Monitor-DPNs are a model for parallel programs with recursive procedures, thread creation, and mutual exclusion via locks that are bound to syntactic blocks in the program (monitors). We consider predecessor set computation, with which many interesting properties can be expressed, among them race-conditions, bitvector-analysis, and the $(\mathsf{EF}, \mathsf{EX})$-fragment of the branching time logic CTL.

Our algorithm is based on *acquisition structures*, which are a finite-state abstraction of executions. An acquisition structure contains enough information to decide whether an execution is feasible w.r.t. the semantics of locks. By encoding the acquisition structures into the control-states of a Monitor-DPN, we reduce lock-sensitive predecessor set computation to lock-*insensitive* predecessor set computation, for which efficient algorithms are known.

For fixed-size, negation-free $(\mathsf{EF}, \mathsf{EX})$-formulas, which are sufficient to describe most interesting properties, our algorithm requires exponential time in the number of locks, but only polynomial time in the program size. To justify the exponential complexity, we show that checking most interesting properties is NP-hard. We also show that the model-checking problem for fixed-size, negation-free $(\mathsf{EF}, \mathsf{EX})$-formulas is NP-complete, while slightly more general problems are PSPACE-hard.

Contents

1 Introduction

Computer programming is prone to errors. Due to Rice's theorem [107], there exists no algorithm that automatically proves that a program meets its specification. In practice, several methods are applied to verify program correctness. The most widely used method is testing, i.e., running the program on some test input, and verify that it yields the expected output. Testing is inherently incomplete, as it only covers a finite subset of the possible executions. On the other hand, testing is simple and requires no knowledge of formal methods, which explains its widespread use. Another method is to rigorously prove that a program meets its specification. This requires the use of sophisticated formal methods. Moreover, writing specifications and proofs is tedious and error-prone, in particular for large programs. While interactive theorem provers and automated proof checkers help to avoid errors in proofs, they require even more rigorous, machine readable formalizations, which tend to be hardly understandable by humans.

Automatic verification methods lie in between testing and complete correctness proofs. The goal is to automatically verify absence of certain errors, like buffer-overflows or null-pointer accesses. Due to Rice's theorem, such methods must be incomplete, i.e., they may fail to verify correct programs, or unsound, i.e., they may fail to detect errors. However, one tries to design methods that succeed for typical programs.

Due to the widespread availability of concurrent hardware, like multicore processors, concurrent programs are getting more and more important, and almost all modern programming languages support concurrent programming. However, concurrent programming is prone to subtle errors like race-conditions and deadlocks that do not occur in sequential programs. Even worse, execution of concurrent programs is nondeterministic, as it depends on the order in which the steps of the parallel processes are executed. Thus, there are concurrency related errors that manifest themselves only at a very low probability, depending on external parameters like the scheduler and the processor speed. Such errors are likely to be missed by testing. On the other hand, formal methods for proving correctness of concurrent programs are even more complex than those for sequential programs. This increases the need for automatic verification methods for concurrent programs.

Example 1.1. *As an example, consider the following Java [47] program:*

```
public class Example implements Runnable {
  private static void write_terminal(String s) {
    for (int i=0;i<s.length();++i) {
      System.out.print(s.charAt(i));
    }
    System.out.flush();
  }

  private synchronized void write(String s) {
    write_terminal(s);
  }

  public void run() {
    write("Hello");
  }

  public static void main(String args[]) {
    new Thread(new Example()).start();
    new Thread(new Example()).start();
  }
}
```

The program creates two threads, and each thread invokes the write() *method to print the string "Hello". To synchronize the concurrent access to the printer, the* write() *method is marked as synchronized. Probably, the intention of the programmer was that the program outputs the string "HelloHello". And indeed, running the program on the author's machine 50 times yields the correct output each time. However, the program contains a subtle error: The semantics of the* ***synchronized****-keyword is to synchronize on the monitor of the object on that the method is invoked. In our case, this is the thread object. Thus, the* write() *methods of the two threads are synchronized on two different objects, and there is actually no mutual exclusion! The reason why this error does not manifest itself depends on the scheduler that uses timeslicing to switch between the threads. As the time required to complete the call of* write() *is small in relation to the duration of a timeslice, a preemption inside the* write *method is very unlikely. However, this behavior may be different on other architectures or operating systems, such that the error may manifest itself with a higher probability there. We will resume this example in Section 7.4, where we show how the analysis developed in this thesis detects the error.*

Automatic verification methods often work in two phases: In the first phase, the program to be verified is abstracted to an *abstract model*, on which the verification problem is decidable. The abstraction is usually sound, i.e., behaviors of the program correspond to behaviors of the abstract model. However, the abstraction is not precise in general, i.e., the abstract model may have *spurious behaviors* that do not correspond to actual behaviors of the program.

The second phase then decides the verification problem on the abstract model. If the second phase succeeds, the program is proved correct (i.e., free of the errors the analysis checks for), because all behaviors of the abstract model—that are a superset of the behaviors of the concrete model—are proved correct. If, however, the second phase fails, the abstract model has an erroneous behavior. This behavior may either correspond to a behavior of the program, or it may be spurious. In the former case, the program really has an error. In the latter case, the program may be correct or not. If the decision procedure used in the second phase is able to produce a description of the erroneous behavior of the abstract model, this behavior can be projected on the program and tested for feasibility by simulating the program. If the behavior is indeed feasible, a definite error has been detected. Otherwise, information about the failed simulation may be used to refine the abstraction, and the whole process is iterated with the refined abstraction. This scheme is commonly referred to as *counter example guided abstraction refinement* (CEGAR) [25].

This thesis is focused on the second phase of automatic verification, i.e., deciding verification problems on the abstract model. Our model, Monitor-DPN, is an infinite-state model with unboundedly deep stacks and unboundedly many threads that may be created dynamically. Additionally, synchronization between threads via reentrant monitors is supported. What is not supported, and thus has to be abstracted away in the first phase, are infinite-state data structures on the heap, shared memory, and other synchronization mechanisms like wait/notify and join. With the exception of join, support of any of these concepts leads to undecidability of almost all interesting properties.

Our method is based on predecessor sets. For a set of configurations C, the predecessor set of C consists of all configurations c' such that there exists a sequence of transitions from c' to some configuration $c \in C$. Many interesting properties can be expressed with predecessor sets, among them absence of race-conditions and the $(\mathsf{EF}, \mathsf{EX})$-fragment of the branching time logic CTL [27]. Moreover, our technique can be applied to increase the precision of bounded model-checking, where the objective is finding errors rather than proving their absence.

In the remainder of this chapter, we discuss related work and the contributions of this thesis. Finally, we sketch the outline of this thesis.

1.1 Related Work

Testing of programs has been done since the early days of computers (cf. [45] for an overview). Methods for rigorous correctness proofs of programs are usually based on the work of Floyd [42] and Hoare [50] (Hoare-logic), with extensions to handle concurrent programs (cf. [53, 95]) and recursive programs (cf. [5] for an overview). However, those methods are quite complex and tedious for large programs. Up to now, complete formal verification of programs with interactive theorem provers and Floyd-Hoare style logic (e.g. [64, 118]) is mostly restricted to academia.

Model-Checking Model-checking was developed and pushed forward by Clarke and Emerson [26] and Queille and Sifakis [103] out of the need for automatic methods to verify concurrent programs [24]. The basic idea is to automatically verify a system specified by an abstract model against a specification in temporal logic. An overview of the history of model-checking can be found in [24].

The biggest obstacle for model-checking is state-space explosion: The state-space of a concurrent program has exponential size in the number of concurrently executed processes. This limits the explicit exploration of the state-space to rather small programs. The state-space explosion problem has been tackled with different techniques, among them symbolic model-checking [20], where sets of states are represented by binary decision diagrams, and partial order reduction techniques [46, 97, 114], which reduce the number of states that need to be explored by exploiting that steps of concurrent processes are often independent.

Model-checking can also be applied to infinite-state systems. For example, Bouajjani et al. [13] model-check branching and linear time logics on pushdown processes. However, regarding concurrent infinite-state systems, model-checking tends to be undecidable. Mayr [86] provides a classification of infinite-state systems and obtains decidability and undecidability results for various temporal logics. Among other results, Mayr [85, 86] shows that the $(\mathsf{EF}, \mathsf{EX})$-fragment of CTL (referred to as *EF-logic*) is decidable for models with parallel procedure calls.[1]

Another technique to cope with state-space explosion or infinite-state systems is bounded model-checking, where the state-space is only explored up to a certain search depth [9, 10]. While this allows the application of alternative symbolic techniques like SAT-solvers, bounding the search depth no longer allows to prove the system correct. One can only find errors, hoping that all errors manifest themselves within a tractable search depth. The idea of bounded model-checking can also be combined with precise techniques developed for pushdown systems. Qadeer and Rehof [101] apply bounded model-checking to recursive, concurrent Boolean programs, where the bound is not the number of steps, but the number of context switches. Their technique is improved and generalized to DPNs by Bouajjani et al. [15].

[1]Presented in a process-algebraic framework, called PA- and PAD-processes.

Predecessor Sets A well-known result by Büchi [19] is that the set of reachable configurations of a pushdown system is regular. This has been extended by Caucal [22, 23], who characterizes pushdown systems as rational transducers. This result implies that predecessor and successor sets of regular sets of configurations are regular again and can be computed. Bouajjani and Maler [11], Bouajjani et al. [13], and Finkel et al. [41] apply predecessor and successor set computation to decide temporal logics for pushdown systems. Successor and predecessor set computation has been generalized to PA-processes by Lugiez and Schnoebelen [82, 83]. While most branching time temporal logics are undecidable for PA-processes, EF-logic can be naturally decided via predecessor set computation. Thus, the algorithm of Lugiez and Schnoebelen [82, 83] is a succinct alternative to the more complicated tableaux based method of Mayr [85, 86].

Dynamic Pushdown Networks Intuitively, PA-processes support recursion and concurrency by allowing procedures to be called in parallel. After a parallel call, the procedures run in parallel, and the call returns if all called procedures have returned. However, concurrency in programming languages like Java [47] works differently: Once a thread is created, it runs until termination and does not synchronize with its creator (unless the creator invokes a join-statement). In particular, a thread may survive the procedure that created it. This type of concurrency is called *dynamic thread creation* and cannot be modeled by PA-processes [16]. Bouajjani et al. [16, 17] propose dynamic pushdown networks (DPNs), which support dynamic thread creation. They show that predecessor set computation for DPNs can be done in polynomial time and preserves regularity. A DPN is a pushdown system where rules may create (spawn) new threads as a side effect. The spawned threads have their own stacks and run in parallel with all other threads in the system. A disadvantage is that DPNs do not support joining of threads as PA-processes do: Once a thread is spawned, it runs completely independent from its creator. Thus, the expressive power of DPNs and PA-processes are incomparable. Hence, Bouajjani et al. [16, 17] propose *Constrained DPNs* (CDPNs) that are strictly more general than DPNs and PA-processes. They show that predecessor sets for CDPNs still preserve regularity and are effectively computable.

More recently, we have explored an alternative technique that computes successor sets of DPNs [44, 74]. It is based on characterizing the set of executions of a DPN as a tree automaton, and may also be applicable to CDPNs.

Communication between Threads In DPNs and CDPNs, the communication between threads is rather limited: In DPNs, the only communication between threads is the causal dependence of a thread on the step that created it. In CDPNs, threads may additionally observe their children by *stable constraints*, e.g. observe termination or progress of spawned threads. However, real programming languages support more powerful communication between threads, like shared memory or message-passing. Unfortunately, most interesting properties are undecidable for concurrent pushdown systems with shared global state or other synchronization mechanisms like rendezvous-communication and message-passing [104].

Besides shared memory, real programming languages also use locks and monitors to synchronize the access to shared resources. Intuitively, a *lock* is an object that can be *acquired* by at most one thread at the same time. If a thread wants to acquire a lock that is already acquired by another thread, it has to wait until the other thread *releases* the lock. Monitors (cf. [51]) are a special discipline of using locks: A *monitor* is a syntactic block in the program text that is associated with a lock. The lock is acquired upon entering the monitor and released upon leaving the monitor. For example, the synchronized methods of Java [47] implement monitors.

Monitors enforce locks to be used in a *well-nested* fashion, i.e., a thread can only release the last lock that it has acquired and not yet released. For well-nested locking, each thread can be thought of having a stack of locks (*lockstack*). An acquisition pushes the acquired lock onto the lockstack, and a release pops the topmost lock from the stack. For monitors, locks are acquired and released inside the same procedure. Thus, the lockstack is synchronized with the *callstack*, which stores the return addresses.

In real programming languages, locks are often reentrant, i.e., a thread is allowed to re-acquire a lock that it already possesses. Each acquisition-operation increments a counter for the lock and each release-operation decrements the counter. The thread releases the lock if the counter falls to zero.

When proving absence of errors like race-conditions, it is essential not to abstract away from locks, as they are typically used to avoid such errors. Kahlon et al. [57] have shown that reachability analysis for parallel pushdown systems with well-nested, non-reentrant locks remains decidable. They extended their results to checking fragments of LTL and CTL [55, 56], and to systems that satisfy the *bounded lock-chain* criterion [54], a generalization of well-nested locks. Note that reachability properties w.r.t. arbitrarily non-well-nested locks are undecidable [57].

Their model, called *parallel pushdown systems* (PPDS), consists of a fixed number of pushdown processes that run completely independent, i.e., there are no parallel procedure calls like in PA-processes, nor thread creations like in DPNs. They decide pairwise reachability properties (viz. logics with double-indexed atomic propositions in [55, 56]).

Preceding Results In this paragraph, we describe our own results that are related to this thesis. The main objective of our research is to extend DPNs with more powerful communication mechanisms, such that interesting properties, like absence of race-conditions, remain decidable. Considering the tight undecidability boundary [104], mutual exclusion via locks is a natural candidate for such an extension. Leveraging the *acquisition history* methods of [57], we first developed an analysis for pairwise reachability of interprocedural flowgraphs with dynamic thread creation and reentrant monitors [78]. This model differs from DPNs in that a procedure may not pass a return value to its caller.

A main observation of [78] is that, as far as reachability is concerned, an execution can always be reordered such that steps inside a monitor are scheduled atomically. This is used to prune irrelevant steps from the execution. For pairwise reachability, the pruned execution contains no more than two threads that are running simultaneously.

While [57] deals with non-reentrant, well-nested locks, our methods cover reentrant monitors. These two models are incomparable. Unfortunately, we have no results for the natural generalization of both models, i.e., well-nested, reentrant locks (cf. Section 8.1). Another difference between [57] and our method in [78] is that [57] is based on successor set computation for pushdown automata, while we use fixed point computation and an abstract interpretation framework [31, 32].

Our technique of reordering schedules implies a cut-off: For a specific execution, we call the acquisition of a lock *final* if this lock is not released during the remainder of the execution. Obviously, the number of non-reentrant final acquisitions in any execution is bounded by the number of locks. Symmetrically, a release of a lock that is not acquired during the prefix of the execution is called *initial*. Also the number of non-reentrant initial releases in any execution is bounded by the number of locks.

Using this cut-off, a lock-sensitive predecessor set computation for DPNs can be implemented by guessing a sequence of initial releases and final acquisitions, and using lock-insensitive predecessor set computation to check whether there is a feasible execution with that sequence of initial releases and final acquisitions. However, guessing has to be implemented by iterating over all possible sequences, resulting in a super-exponential algorithm. We have not published these results, as they were superseded by subsequent research [79].

In [79], we extend the acquisition history method to DPNs with well-nested, non-reentrant locks (*Lock-DPNs*). We show how to compute predecessor sets in time polynomial in the program size and exponential in the number of locks.

The executions of the threads in a parallel pushdown system are completely independent. Thus, the execution of each thread can be seen as a linear sequence of steps. The acquisition history method computes a summary of the executions of each thread, and then combines these summaries to check whether there are compatible executions. However, when threads are dynamically created, they are not independent any more, as a thread does not start before it has been created. A natural generalization is to describe executions as trees (*execution-trees*), such that a step that creates a thread has two successors: The execution of the created thread, and the remainder of the execution of the creating thread. By generalizing the acquisition history method from linear executions to execution-trees, we obtain a tree automaton that characterizes all feasible execution-trees. Then, we construct the cross-product of the Lock-DPN and this tree automaton, resulting in a DPN without locks, whose executions correspond to feasible executions of the original Lock-DPN. The cross-product DPN is then analyzed using the known predecessor set algorithm of Bouajjani et al. [16].

Another generalization that we applied increased the number of threads that can be handled by acquisition histories. For the pairwise reachability properties of [57], it was sufficient to restrict to systems with two threads. A similar restriction was achieved in [78] by pruning the executions. However, when computing predecessor sets of arbitrary configurations, we have to consider more than two threads. Fortunately, the generalization of acquisition histories to more than two threads is quite straightforward.

Recall that DPNs and PA-processes are incomparable [16], and CDPNs are a generalization of both models. A slightly less expressive generalization of both models are DPNs with joins, where a thread may execute a join-statement to wait until all threads that it has spawned have terminated. In parallel to the preparation of this thesis, we have explored analysis of DPNs with joins and well-nested, non-reentrant locks (Join-Lock-DPNs). The acquisition history techniques can be generalized to this scenario, and as a first result, we have shown that reachability of regular sets of configurations is decidable in exponential time [44].

1.2 Contributions

In this thesis, we develop an algorithm for predecessor set computation for DPNs with reentrant monitors (Monitor-DPNs). Our methods are largely based on the ideas that we developed in [79] to compute predecessor sets of Lock-DPNs: We also use execution-trees, characterize execution-trees of feasible executions by *DPN-Acceptors* (a generalization of the tree automata used in [79]), and perform a cross-product construction of a Monitor-DPN and a DPN-Acceptor, thus reducing lock-sensitive predecessor set computation to lock-insensitive predecessor set computation, which is then performed by the algorithm of Bouajjani et al. [16]. The main technical differences are, (1) we analyze Monitor-DPNs rather than Lock-DPNs, (2) we use DPN-Acceptors rather than tree automata, and, (3) we use a more modular approach. Monitor-DPNs and Lock-DPNs are incomparable models, as Monitor-DPNs have reentrance and Lock-DPNs need not adhere to a monitor-discipline. DPN-Acceptors generalize tree automata by limited use of a stack, thus allowing to model reentrant monitors. Our approach to analysis of Monitor-DPNs consists of two phases. In the first phase, we reorder the steps of an execution—like in [78]—such that sequences between matching acquisition- and release-operations are scheduled atomically. Additionally, we eliminate reentrance in this phase. In the second phase, the acquisition history method is applied to the reordered executions. Compared to the direct approach taken in [79], our approach is more modular and results in simpler proofs.

Moreover, we show that many analysis problems for Lock-DPNs and Monitor-DPNs are NP-complete, where the high complexity is induced by the number of locks, not by the program size. This result also matches the runtime of our predecessor set algorithm, which is exponential only in the number of locks. When regarding slightly more expressive models, e.g. Join-Lock-DPNs, the analysis problems become PSPACE-hard.

Summarizing, the contributions of this thesis are the following:

- We characterize executions of DPNs by execution-trees, and show how to constrain predecessor set computation w.r.t. sets of execution-trees via a cross-product construction. While computing predecessor sets that are constrained with a regular set of interleaved executions is not effective, our results imply a polynomial algorithm for computing predecessor sets constrained with a regular set of execution-trees, or even with a set of execution-trees accepted by a DPN-Acceptor.

  The idea of execution-trees has been around in our research group for a while (cf. [111, 120]). The idea of the cross-product construction with a tree automaton has been developed in [79], and is generalized to DPN-Acceptors in this thesis.

- We generalize the acquisition histories of Kahlon et al. [55, 57] to execution-trees, thus supporting lock-sensitive analysis of DPNs. We mainly worked out this generalization in [79]. The main difference is that we analyze DPNs with reentrant monitors, while [79] analyzes DPNs with well-nested, non-reentrant locks. Also the technical approach in this thesis is different from [79]. While the formulation in [79] defines acquisition structures over execution-trees, making the correctness proof quite involved, this thesis presents a modular approach that results in simpler proofs.

- We use generalized acquisition histories and the cross-product construction to reduce lock-sensitive predecessor set computation to lock-*insensitive* predecessor set computation. This yields an algorithm for lock-sensitive predecessor set computation on Monitor-DPNs that can be applied to check various interesting properties, like absence of race-conditions and EF-logic.

- Finally, we show that various analysis problems for Monitor-DPNs are NP-complete, among them detection of race-conditions and model-checking negation-free EF-logic with a fixed number of operators. Slightly more general problems, like reachability analysis on Join-Lock-DPNs, are PSPACE-hard. Our NP-hardness results also apply to related problems on systems with locks. For example, the pairwise reachability problem on parallel pushdown systems (cf. [57]), as well as the model-checking problems studied by Kahlon and Gupta [55, 56], are NP-hard.

1.3 Outline

The remainder of this thesis is organized as follows: In Chapter 2, we present basic concepts that are frequently used in this thesis. In Chapter 3, we formally define Monitor-DPNs and their semantics. We define an *interleaving semantics* as a reference point, and a *tree-semantics* on execution-trees, and show that the two are equivalent. In Chapter 4, we generalize the concept of acquisition histories to execution-trees. As an intermediate model, we introduce *acquire/release-trees*, where matching acquisition- and release-nodes are summarized into single *use-nodes*. By reordering the steps of an execution such that steps between matching acquisitions and releases are scheduled atomically, we show that schedulability of a lock-execution-tree is equivalent to schedulability of the corresponding acquire/release-tree. Then, we use acquisition histories to show that the set of schedulable acquire/release-trees is regular. In Chapter 5, we show how to combine a Monitor-DPN with a regular constraint on its acquire/release-trees. This is done in two phases: In the first phase, we translate automata on acquire/release-trees to DPN-Acceptors on execution-trees. In the second phase, we do a cross-product construction between the Monitor-DPN and the DPN-Acceptor. In the first part of Chapter 6, we use the results of the previous chapters to reduce lock-sensitive predecessor set computation to lock-insensitive predecessor set computation. As lock-insensitive predecessor set computation is effective [16], we obtain an algorithm for lock-sensitive predecessor set computation, which is the main result of this thesis. In the second part of Chapter 6, we sketch some applications of this algorithm. In Chapter 7, we describe optimizations that simplify our algorithm for typical analysis scenarios, and present a simple example that illustrates how our analysis finds the data-race in the Java program from Example 1.1. Finally, we use the example to reveal problems that an implementation of our algorithm has to solve, and propose solutions. In Chapter 8, we briefly discuss how the methods of this thesis can be transferred to the analysis of DPNs with well-nested, non-reentrant locks, and discuss why our techniques do not apply to well-nested, reentrant locks. Moreover, we present a polynomial time algorithm that checks whether a given DPN uses locks only in a well-nested, non-reentrant fashion. In Chapter 9, we show that our analysis is NP-complete, and establish lower complexity bounds for various related analysis problems. Finally, Chapter 10 contains the conclusion of this thesis and indicates topics of current and future research.

2 Preliminaries

In this chapter, we briefly introduce some basic concepts and notations that we use in this thesis.

This chapter is organized as follows: In Section 2.1, we agree on conventions for basic notations of sets, sequences, etc. In Section 2.2, we describe the concepts of word and tree automata. In Section 2.3, we introduce concepts required to estimate the complexity of our algorithms.

2.1 Notations

The set $\mathbb{B} := \{\top, \bot\}$ is the set of truth values. The set of natural numbers including zero is denoted by $\mathbb{N}$. We set $\mathbb{N}_+ := \mathbb{N} \setminus \{0\}$. The symbol $\mathbb{R}$ denotes the real numbers. We set $\mathbb{R}_+ := \{r \in \mathbb{R} \mid r > 0\}$. We use the standard operations $\cup, \cap, \setminus, \in$ for sets. For a predicate P and a set S, the set $\{x \in S \mid P(x)\}$ denotes the set of all elements of S that satisfy P. We write $A \mathbin{\dot\cup} B$ for disjoint union of the sets A and B, i.e., $A \mathbin{\dot\cup} B = A \cup B$, and, additionally, we assume $A \cap B = \emptyset$. We use sets and their characteristic functions synonymously. For example, we write $\mathsf{valid}(c)$ or $c \in \mathsf{valid}$ to denote that the element c is contained in the set valid.

The set of finite length lists of elements from a set X is denoted by X^*. The empty list is denoted by ε. A list with distinct elements is denoted by $[x_1, \dots, x_n]$. List concatenation is written by the empty operator, i.e., $l_1 l_2$ is the concatenation of the lists l_1 and l_2. When clear from the context, we also write $x_1 \dots x_n$ instead of $[x_1, \dots, x_n]$, and mix lists and single elements to form a new list, i.e., $l x_1 x_2 l'$ instead of $l[x_1, x_2]l'$. With $|l|$, we denote the length of the list l. When unambiguous, we use lists in place of the set of their elements. For example, we write $x \in l$ to denote that l contains an element x, and $l_1 \cap l_2$ to denote the set of elements contained in both l_1 and l_2. To explicitly denote the set of elements of a list l, we write $\mathsf{set}(l)$. Variables of a list type are sometimes written with a bar, i.e., $\bar{x} \in X^*$, to distinguish them from variables $x \in X$ of the element type. For any set A, the predicate $\mathsf{disjoint} \subseteq A^*$ holds for exactly the lists whose elements are pairwise disjoint:

$$\mathsf{disjoint}(w) \iff \forall w_1, w_2, w_3 \in A^*, x, y \in A.\ w = w_1 x w_2 y w_3 \implies x \neq y.$$

Let $L_1 \subseteq X_1^*$, and $L_2 \subseteq X_2^*$ be sets of lists. Then $L_1 L_2$ is the set of lists $l_1 l_2$ with $l_1 \in L_1$ and $l_2 \in L_2$. The set of n-element lists of X is denoted by X^n. If types are clear from the context, X^1 may be written as just X. This allows a regular expression like notation for sets of lists For example, let $A = \{a, \dots, z\}$ be the Latin alphabet, then $\{a\}A^*(\{y\} \cup \{z\}) \subseteq A^*$ is the set of all words that start with the letter a and end with the letter y or z.

Given a relation $R \subseteq A \times B$, we write $aRb := (a, b) \in R$. We define the inverse $R^{-1} \subseteq B \times A$ of the relation R by

$$R^{-1} \subseteq B \times A := \{(b, a) \mid (a, b) \in R\}.$$

The image of a set S under R is denoted by $R(S)$, or, alternatively, by SR, and defined as

$$R(S) := SR := \{b \mid \exists a \in S.\ (a, b) \in R\}.$$

Given two relations $R \subseteq A \times B$ and $S \subseteq B \times C$, we define the composition of R and S by

$$(a, c) \in R \circ S \text{ iff } \exists b.\ (a, b) \in R \wedge (b, c) \in S.$$

This definition is also used if relations are given in arrow-notation, i.e., $\rightarrow_1 \circ \rightarrow_2$, and also for ternary relations that represent labeled transition systems:

$$c \xrightarrow{a}_1 \circ \xrightarrow{b}_2 c' \text{ iff } \exists \tilde{c}.\ c \xrightarrow{a}_1 \tilde{c} \xrightarrow{b}_2 c'.$$

For relations of the form $R \subseteq A \times A$, or $R \subseteq A \times \Sigma \times A$, R^* denotes the reflexive, transitive closure, and R^+ denotes the transitive closure.

2.2 Automata

In this section, we define the well-known concepts of word and tree automata.

2.2.1 Word Automata

Word automata are a well-known concept to characterize regular sets of sequences of symbols. A good introduction to word automata is [52]. We briefly sketch the definition of word automata and some basic results here. A *word automaton* $A = (\Sigma, Q, I, F, \delta)$ consists of a finite alphabet Σ, a finite set of states Q, a set of initial states $I \subseteq Q$, a set of final states $F \subseteq Q$, and a set of transitions $\delta \subseteq Q \times \Sigma \times Q$. If the alphabet is clear from the context, we omit it and write $A = (Q, I, F, \delta)$. We write $q \xrightarrow{a}_\delta q'$ instead of $(q, a, q') \in \delta$, and write $\rightarrow_\delta^*$ for the reflexive, transitive closure of $\rightarrow_\delta$.

A word $w \in \Sigma^*$ is *recognized in state* q by A, iff there is a final state $q' \in F$, such that $(q, w, q') \in \delta^*$. We write $A(q)$ for the set of words recognized in state q. A word w is *recognized* by A, iff it is recognized in an initial state. The language $\mathsf{L}(A)$ of an automaton A is the set of all recognized words:

$$\mathsf{L}(A) = \{w \in \Sigma^* \mid \exists q \in I, q' \in F.\ (q, w, q') \in \delta^*\}.$$

A set that is the language of some automaton is called *regular*. In order to store the automaton A, we need polynomial space in the size of the alphabet and the number of states:

$$|A| = \mathsf{poly}(|\Sigma||Q|).$$

See Section 2.3 for details.

The class of regular sets is closed under union, intersection, and complement. Given automata A and B over the alphabet Σ, automata $A \cup B$, $A \cap B$, and $\Sigma^* \setminus A$ can be constructed such that

$$\mathsf{L}(A \cup B) = \mathsf{L}(A) \cup \mathsf{L}(B) \quad \mathsf{L}(A \cap B) = \mathsf{L}(A) \cap \mathsf{L}(B) \quad \mathsf{L}(\Sigma^* \setminus A) = \Sigma^* \setminus \mathsf{L}(A).$$

Union and intersection can be computed in polynomial time, the complement can be computed in exponential time. Moreover, emptiness of the language of a given automaton can be decided in polynomial time. The sizes of the automata can be estimated by

$$|A \cup B| = O(|A| + |B|) \qquad |A \cap B| = O(|A||B|) \qquad |\Sigma^* \setminus A| = 2^{O(A)}.$$

A function $h : \Sigma \to \Gamma^*$ is called a homomorphism. We extend h to words and set of words by

$$h([a_1, \dots, a_n]) = h(a_1) \dots h(a_n) \qquad\qquad h(W) = \{h(w) \mid w \in W\}.$$

The inverse $h^{-1} : 2^{\Gamma^*} \to 2^{\Sigma^*}$ of h is defined by

$$h^{-1}(W) = \{w \in \Sigma^* \mid h(w) \in W\}.$$

The class of regular sets is closed under application of homomorphism and inverse homomorphism, and given automata A over Σ and B over Γ, and a homomorphism $h : \Sigma \to \Gamma^*$, automata $h(A)$ and $h^{-1}(B)$ can be constructed in polynomial time such that

$$\mathsf{L}(h(A)) = h(\mathsf{L}(A)) \qquad\qquad \mathsf{L}(h^{-1}(B)) = h^{-1}(\mathsf{L}(B)).$$

2.2.2 Tree Automata

Tree automata generalize the concept of word automata to trees. A good introduction to tree automata is [29]. We briefly sketch the definition of tree automata and some basic results here.

Given a finite, ranked alphabet $C = C_0 \dot\cup \dots \dot\cup C_n$, such that $C_0 \neq \emptyset$, the set $\mathbb{T}_C$ of *ranked trees* over C is defined as the least solution of the following constraints:

$$c(t_1, \dots, t_m) \in \mathbb{T}_C \qquad \text{for } c \in C_m,\ 0 \leq m \leq n, \text{ and } t_1, \dots, t_m \in \mathbb{T}_C$$

In order to simplify the definitions below, we also define $C_{n+1} = C_{n+2} = \dots := \emptyset$.

A tree automaton $A = (C, Q, F, \delta)$ consists of a ranked alphabet $C = C_0 \dot\cup \ldots \dot\cup C_n$, a finite set of states Q, a set $F \subseteq Q$ of final states, and a finite set $\delta \subseteq C \times Q^* \times Q$ of rules that are consistent w.r.t. the ranks of the constructors, i.e., $(c, r, q) \in \delta \implies c \in C_{|r|}$. If the alphabet is clear from the context, we omit it and write $A = (Q, F, \delta)$. We write a rule $(c, q_1 \ldots q_m, q) \in \delta$ as $c(q_1, \ldots, q_m) \to_\delta q$.

We define the relation $\to_\delta^* \subseteq \mathbb{T}_C \times Q$ as the least solution of the following constraints:

$$c(t_1, \ldots, t_m) \to_\delta^* q \qquad \text{if } t_1 \to_\delta^* q_1 \wedge \ldots \wedge t_m \to_\delta^* q_m \wedge c(q_1, \ldots, q_m) \to_\delta q$$

If we have $t \to_\delta^* q$, we say that the tree t is *accepted in state* q. A tree t is *accepted* by the tree automaton, iff it is accepted in a final state. The language $\mathsf{L}(A)$ of a tree automaton A is the set of all recognized trees:

$$\mathsf{L}(A) = \{t \in \mathbb{T}_C \mid \exists q \in F.\ t \to_\delta^* q\}$$

A set of trees is called *regular*, iff it is the language of some tree automaton.

An equivalent way of defining acceptance by a tree automaton is to include the states of the tree automaton into the alphabet with rank 0, and interpret the rules of the tree automaton as subtree-rewrite-rules, i.e., a rule $c(q_1, \ldots, q_m) \to_\delta q$ means that a subtree $c(q_1, \ldots, q_m)$ may be rewritten to q. A tree is accepted in state q iff it can be rewritten to q. Using this characterization, we write $t \to_\delta^* q$ also for trees t that contain states of the tree automaton. For example, we may write $f(g(q_1), q_2) \to_\delta^* q_3$.

In order to store a tree automaton $A = (C, Q, F, \delta)$, we need space polynomial in the size of the alphabet and the number of states:

$$|A| = \mathsf{poly}(|C||Q|).$$

See Section 2.3 for details.

Like for regular sets of words, the class of regular sets of trees is closed under union, intersection, and complement. Given tree automata A and B over an alphabet C, tree automata $A \cup B$, $A \cap B$, and $\mathbb{T}_C \setminus A$ can be constructed such that

$$\mathsf{L}(A \cup B) = \mathsf{L}(A) \cup \mathsf{L}(B) \quad \mathsf{L}(A \cap B) = \mathsf{L}(A) \cap \mathsf{L}(B) \quad \mathsf{L}(\mathbb{T}_C \setminus A) = \mathbb{T}_C \setminus \mathsf{L}(A).$$

Union and intersection can be computed in polynomial time, and the complement can be computed in exponential time. Moreover, it can be decided in polynomial time whether the language of a tree automaton is empty.

2.3 Complexity

In this section we define the preliminaries required for the analysis of the complexity of the algorithms that will be developed in this thesis. In Subsection 2.3.1, we describe the big-O notation, in Subsection 2.3.2 we define how we measure the size of the input data of our algorithms, and in Subsection 2.3.3 we briefly introduce the basic concepts of computational complexity.

2.3.1 Big-O Notation

The big-O notation is a well-known tool to specify the space and time complexity of algorithms. An introduction can be found in, for example, Wegener [119]. We introduce the basic definitions here:

Given a monotonous[1] function $g : \mathbb{N}_+^n \to \mathbb{R}_+$ for some $n \in \mathbb{N}_+$, we define the class of functions $O(g)$ by

$$f \in O(g) \;:\Longleftrightarrow\; f \text{ monotonous and } \exists c \in \mathbb{R}_+.\ \forall \vec{x} \in \mathbb{N}_+^n.\ \frac{f(\vec{x})}{g(\vec{x})} \leq c.$$

We extend this notion to classes of monotonous functions G by

$$f \in O(G) \;:\Longleftrightarrow\; \exists g \in G.\ f \in O(g).$$

As it is common in literature, we abuse the notion of equals in a non-symmetric way, and write $f = O(g)$ instead of $f \in O(g)$, and $O(f) = O(g)$ instead of $O(f) \subseteq O(g)$. If we mean real equality we write $O(f) \equiv O(g)$.

We have the following properties [119]:

$$\begin{aligned} cf &\in O(f) && \text{for } c \in \mathbb{R}_+ \\ cO(f) &\equiv O(f) && \text{for } c \in \mathbb{R}_+ \\ O(f_1) + \ldots + O(f_k) &\equiv O(f_1 + \ldots + f_k) \equiv O(\max\{f_1, \ldots, f_k\}) && \\ O(f)O(g) &\equiv O(fg) && \end{aligned}$$

Other useful laws are

$$\begin{aligned} O(O(f)) &\equiv O(f) \\ fO(g) &\equiv O(fg) \end{aligned}$$

For example, when regarding a polynomial with positive coefficients $c_m, \ldots, c_0 \in \mathbb{R}_+$, its order is only determined by the largest exponent:

$$O(c_n x^n + \ldots + c_1 x + c_0) \equiv O(x^n).$$

If we are not interested in the largest exponent of a polynomial, we also write $\mathsf{poly}(g)$ for the class of polynomials in g:

$$f \in \mathsf{poly}(g) \;:\Longleftrightarrow\; \exists k.\ f \in O(g^k)$$

[1]We use a pointwise ordering on $\mathbb{N}_+^n$, i.e., $(x_1, \ldots x_n) \leq (y_1, \ldots, y_n)$ iff $x_1 \leq y_1 \wedge \ldots \wedge x_n \leq y_n$.

and extend this notion to classes of functions:

$$f \in \mathsf{poly}(G) \;:\Longleftrightarrow\; \exists g \in G.\; f \in \mathsf{poly}(g).$$

For poly, we have similar laws as for O:

$$\begin{aligned}
cf^k &\in \mathsf{poly}(f) && \text{for } c \in \mathbb{R}_+ \text{ and } k \in \mathbb{N} \\
c\mathsf{poly}(f) &\equiv \mathsf{poly}(f) && \text{for } c \in \mathbb{R}_+ \\
\mathsf{poly}(f_1) + \ldots + \mathsf{poly}(f_k) &\equiv \mathsf{poly}(f_1 + \ldots + f_k) \equiv \mathsf{poly}(\max\{f_1, \ldots, f_k\}) \\
\mathsf{poly}(f)\mathsf{poly}(g) &\equiv \mathsf{poly}(fg) \\
\mathsf{poly}(\mathsf{poly}(f)) &\equiv \mathsf{poly}(f) \\
f\mathsf{poly}(g) &\subseteq \mathsf{poly}(fg) \\
\mathsf{poly}(2^{O(f)}) &\equiv 2^{O(f)}
\end{aligned}$$

Between the classes of functions described by O and poly, we have the following hierarchy:

$$O(f^0) \subset O(f^1) \subset \ldots \subset \mathsf{poly}(f) \subset 2^{O(f)}$$

Typical complexities encountered in this thesis are $\mathsf{poly}(n)$, which we call *polynomial in n*; $2^{\mathsf{poly}(n)}$, which we call *exponential in n*; and $\mathsf{poly}(n)2^{\mathsf{poly}(m)}$, which we call *polynomial in n and exponential in m*.

2.3.2 Size of Input Data

We typically specify the complexity of an algorithm in terms of the *size* of its input. The size of the input is the length of bits required to encode the input of the algorithm.

The input of our algorithms are usually automata-like structures, consisting of a finite alphabet, finite sets of symbols for states and stack, and a finite set of rules. The rules are a relation on boundedly many elements from the alphabet and symbol sets. For example, the rules of an automaton relate two states with one letter from the alphabet. A rule of a tree automaton may relate more than two states, depending on the arity of the function symbol.

Regard an automaton $A = (\Sigma, Q, I, F, \delta)$. We may drop letters and states that do not occur in rules, without changing the language of the automaton[2]. Hence, we may safely assume that Σ and Q do not contain such letters or states. Thus, to store A, it is sufficient to store the sets I, F, and δ, as Σ and Q can be reconstructed from these sets. With $|A|$, we denote the *size of A*, i.e., the number of bits required to store A.

In order to store finite sets, there are two common options: Explicit enumeration of all the elements in the set (e.g. by some tree or hashset data structure), or

[2]Actually, the precise alphabet is only important when complementing the automaton.

storing the set as a vector of bits representing its characteristic function. While the former method is usually better for sparse sets, containing few elements compared with the maximum possible elements, the latter method is better for dense sets, containing a number of elements in the order of the maximum possible elements. We choose the second possibility here, as it has a better worst-case size.

Hence, in order to store the sets of initial and final states, we need $2|Q|$ bits, and in order to store the set of rules, we need $|Q|^2|\Sigma|$ bits. Together, we need $2|Q| + |Q|^2|\Sigma|$ bits. Using big-O notation, we have

$$|A| = 2|Q| + |Q|^2|\Sigma| = O(|Q|^2|\Sigma|) = \mathsf{poly}(|Q||\Sigma|),$$

i.e., to store an automaton we need space quadratic in the number of states and linear in the size of the alphabet, or, estimated more roughly, space polynomial in the number of states and in the size of the alphabet.

A similar estimation works for tree automata $A = (C, Q, F, \delta)$. Here, the number of possible rules, and thus the number of bits required to store a set of rules, is

$$\sum_{i \in \mathbb{N}} |C_i||Q|^{i+1} = \sum_{i \leq \max\{i \ | \ |C_i| \neq 0\}} |C_i||Q|^{i+1} = |C|\mathsf{poly}(|Q|) = \mathsf{poly}(|C||Q|),$$

where we assume that the maximum rank is fixed. Thus, we have

$$|A| = O(|Q|) + |C|\mathsf{poly}(Q) = O(|C|\mathsf{poly}(Q)) = \mathsf{poly}(|C||Q|),$$

i.e., we need space polynomial in the number of states and linear in the size of the alphabet, or, more roughly, polynomial in the number of states and the size of the alphabet.

Example 2.1. *Given a finite set $\mathcal{X}$ we construct an automaton*

$$A = (\{x^b \mid x \in \mathcal{X} \wedge b \in \mathbb{B}\}, \{q\} \times 2^{\mathcal{X}}, \{q\} \times 2^{\mathcal{X}}, \{(q, \emptyset)\}, \delta)$$

with the following rules:

$$(q, X \cup \{x\}) \xrightarrow{x^\perp}_\delta (q, X) \qquad\qquad (q, X) \xrightarrow{x^\top}_\delta (q, X)$$

for all $X \subseteq \mathcal{X}$ and $x \in \mathcal{X}$.

Intuitively, this automaton computes the set of elements annotated with $\perp$, i.e., a sequence w is accepted in state (q, X), where X is the set of elements annotated with bottom occurring in w. Formally:

$$w \in A(q, X) \iff X = \{x \mid x^\perp \in \mathsf{set}(w)\}.$$

Automata that compute some sets based on the accepted word are frequently used in this thesis.

The size of the alphabet is $2|\mathcal{X}|$, the number of states is $2^{|\mathcal{X}|}$. Hence, the size of the automaton can be estimated by

$$|A| = O((2^{|\mathcal{X}|})^2 \cdot 2|\mathcal{X}|) = O(|\mathcal{X}|)2^{O(|\mathcal{X}|)} = 2^{O(|\mathcal{X}|)}.$$

In order to explicitly construct this automaton, one needs to instantiate the rule-templates for all possible sets $X \subseteq \mathcal{X}$ and elements $x \in X$. Hence, construction of the automaton requires time $2^{O(|\mathcal{X}|)}$. However, when implementing our methods, explicit construction of such automata should be avoided, and more compact symbolic representations should be used instead.

2.3.3 Computational Complexity

In this subsection, we briefly introduce the concepts of computational complexity that are used in this thesis. For a complete introduction, we refer to a textbook on computational complexity, e.g. [96, 119].

2.3.3.1 Polynomial Reduction and Completeness

The notion of NP-completeness and (polynomial) reduction was introduced in the landmark paper of Cook [30]. We briefly recall the basics here.

A *(decision) problem* consists of a set of inputs L. A *(decision) algorithm* decides, given an input i, whether $i \in L$. Note that we use the term *decision problem* here, while it is also common to use the term *language*. Moreover, we assume that algorithms are given by one-tape Turing machines.

A decision problem L is in P, if there is a deterministic algorithm that decides $i \in L$ in time $\mathsf{poly}(|i|)$. A decision problem L is in NP, if there is a nondeterministic algorithm that decides $i \in L$ in time $\mathsf{poly}(|i|)$. The problem is in PSPACE, if there is a deterministic algorithm that decides $i \in L$ in space $\mathsf{poly}(|i|)$. The problem is in NPSPACE, if there is a nondeterministic algorithm that decides $i \in L$ in space $\mathsf{poly}(|i|)$. A well-known result of Savitch [108] implies that PSPACE=NPSPACE. Finally, the problem L is in EXPTIME, if there is a deterministic algorithm that decides $i \in L$ in time $2^{\mathsf{poly}(|i|)}$.

A *polynomial reduction* from a problem L to another problem L' is a polynomial time deterministic algorithm that computes a function f, such that $i \in L$ if and only if $f(i) \in L'$. We write $L \leq_P L'$, if there is a polynomial reduction from L to L'. Intuitively, $L \leq_P L'$ means that the problem L is easier to solve than the problem L'.

A problem L is called *NP-hard* (*PSPACE-hard*), if it is harder than any problem L' in NP (PSPACE), i.e., if for any such problem L', we have $L' \leq_P L$. A problem is called *NP-easy* (*PSPACE-easy*), if it is in NP (PSPACE). A problem is called *NP-complete* (*PSPACE-complete*), if it is both, NP-hard and NP-easy (PSPACE-hard and PSPACE-easy).

Obviously, we have P $\subseteq$ NP $\subseteq$ PSPACE $\subseteq$ EXPTIME. It is conjectured that these inclusions are strict and, in particular, that there exist no polynomial time deterministic algorithms for NP-hard problems. Moreover, although the best known deterministic algorithms to solve NP-complete and PSPACE-complete problems require exponential time, PSPACE-hard problems are generally considered more difficult than problems in NP.

2.3.3.2 Standard Problems

The standard method to show that a problem is NP-easy (PSPACE-easy) is to specify a nondeterministic polynomial time (space) algorithm that solves the problem. Alternatively, one can reduce the problem to a known problem in NP (PSPACE). The standard method to show that a problem is NP-hard (PSPACE-hard) is to reduce a known NP-hard (PSPACE-hard) problem to this problem. Thus, it is useful to have a collection of NP-hard and PSPACE-hard problems, from which a suitable one can be selected. The first problem that was shown to be NP-hard is the Boolean satisfiability problem (SAT), i.e., checking whether there is a valuation that satisfies a given Boolean formula [30]. Based on this problem, many other problems can be shown to be NP-hard. A small collection of such problems is presented in [30], and was extended by Karp [58]. We only need the 3SAT-problem in this thesis.

3-Satisfiability The Boolean 3-satisfiability problem (3SAT for short) is one of the first problems shown to be NP-complete [30, 58]. The 3SAT problem is defined as follows: Given a set of n variables $V = \{v_1, \ldots, v_n\}$ ranging over Boolean values, and a set of m clauses $C = \{(l_{11}, l_{12}, l_{13}), \ldots, (l_{m1}, l_{m2}, l_{m3})\}$ over literals $l_{ij} \in V \mathbin{\dot\cup} \{\neg v \mid v \in V\}$, the problem is to decide whether there is a valuation $\sigma : V \to \{\top, \bot\}$ of the variables such that

$$\sigma \models \bigwedge_{1\leq i\leq m} \bigvee_{1\leq j\leq 3} l_{ij}.$$

If there is such a valuation, the 3SAT-instance (V, C) is said to be *satisfiable*, otherwise it is said to be *unsatisfiable*.

Quantified Boolean Formula A *quantified Boolean formula* (QBF) over n variables $V = \{v_1 \ldots v_n\}$ and m clauses $C = \{(l_{11}, l_{12}, l_{13}), \ldots, (l_{m1}, l_{m2}, l_{m3})\}$, where $l_{ij} \in V \mathbin{\dot\cup} \{\neg v \mid v \in V\}$, is a formula of the form

$$\exists v_1 \forall v_2 \ldots Q_n v_n. \bigwedge_{1\leq i\leq m} \bigvee_{1\leq j\leq 3} l_{ij},$$

such that $Q_i = \forall$, if i is even, and $Q_i = \exists$, if i is odd. We assume w.l.o.g. that n is even, i.e., $Q_n = \forall$. Deciding whether a given QBF is true is a well-known PSPACE-complete problem [96].

3 Models

In this chapter, we introduce the program model used in this thesis. *Dynamic pushdown networks with monitors* (Monitor-DPNs) are an extension of pushdown processes by dynamic thread creation and mutual exclusion via monitors. In Section 3.1, we motivate Monitor-DPNs from both, a practical and a theoretical point of view. In Section 3.2, we formally introduce *dynamic pushdown networks* (DPNs), a program model that supports recursive procedures and dynamic thread creation. In Section 3.3, we introduce the concepts of locks and the common *monitor* lock-usage discipline, and extend DPNs by monitors. For both, DPNs and Monitor-DPNs, we define an interleaving semantics and a *tree-semantics*, and show that both semantics are equivalent. Finally, in Section 3.4, we briefly summarize the results of this chapter and give an overview of related work.

3.1 Motivation

We motivate Monitor-DPNs from both, a practical and a theoretical point of view. From the practical point of view, one seeks for abstract models of parallel programs that have decidable properties and can express as much concepts of parallel programming as possible. Modern parallel programming languages—like Java [47] or C/C++ [112] with multi-threading libraries like pthreads [21]—support many concepts that pose challenges to automatic analysis. Among others, these are

- procedures,
- dynamically allocated memory and pointers to data and code (e.g. virtual methods),
- dynamic thread creation,
- shared memory between threads, and
- synchronization between threads (e.g. locks, join, wait/notify, message-passing).

Out of these concepts, Monitor-DPNs fully support procedures, dynamic thread creation, and synchronization between threads via reentrant monitors. However, they have no direct support for dynamically allocated memory and pointers, nor for shared memory or other synchronization primitives like join, wait/notify, or message-passing.

The choice of supported concepts can be justified as follows: Support of dynamically allocated memory and pointers immediately leads to Turing-powerful models, even without procedures and concurrency. For example, one can use linked lists as counters. A model with decidable properties that supports procedures and concurrency cannot support too powerful synchronization mechanisms like shared memory, wait/notify, or message-passing, as shown by Ramalingam [104]. Also non-well-nested locking leads to undecidability of simultaneous reachability properties, as was shown by Kahlon et al. [57]. (Recently, decidability for some weaker criterion than well-nestedness, called *bounded lock-chains*, was shown [54].) For locks, we have the choice of supporting well-nested, non-reentrant locks, or reentrant monitors[1]. We choose reentrant monitors, as they are used by the widespread Java programming language [47]. However, our methods also apply for well-nested, non-reentrant locks with minor changes (cf. Chapter 8 and [79]). Moreover, locks are typically used to avoid race-conditions. As verifying absence of race-conditions is an important application of our methods, it is essential not to abstract from locks.

From the theoretical point of view, one seeks for generalizations of existing models that still have decidable properties, such that further generalization results in undecidability or increased complexity. Monitor-DPNs are a generalization of dynamic pushdown networks (DPNs) by synchronization via monitors. DPNs have been introduced by Bouajjani et al. [16], and are, themselves, a generalization of the well-known pushdown systems (PDS) by dynamic thread creation. Predecessor set computation for PDSs and DPNs can be done in polynomial time. In this thesis, we provide a predecessor set computation for Monitor-DPNs that runs in polynomial time in the size of the DPN and in exponential time in the number of locks. To justify the exponential runtime, we show that the corresponding decision problem, i.e., lock-sensitive reachability between regular sets of configurations, is NP-complete (cf. Chapter 9). We also show that further extending Monitor-DPNs with join-synchronization makes this problem PSPACE-hard. As already mentioned, generalization to even more powerful synchronization mechanisms, like shared memory or rendezvous-communication, makes most problems undecidable [104].

3.2 Dynamic Pushdown Networks

Before we define Monitor-DPNs, we first introduce DPNs without locks. *Dynamic pushdown networks* (DPNs) are a model of programs with procedures and dynamic thread creation, which has been developed by Bouajjani et al. [16]. They

[1]Unfortunately, we have no results for well-nested, reentrant locks, cf. Section 8.1.

generalize pushdown systems by the ability to spawn new threads as side-effects of transitions. We define DPNs and their semantics along the lines of [16]:

Definition 3.1 (Dynamic Pushdown Network)**.** *A* dynamic pushdown network *(DPN) is a tuple $M = (P, \Gamma, \mathsf{Act}, \Delta)$, where P is a finite set of* control-symbols*, Γ is a finite set of* stack-symbols*, Act is a finite set of* actions*, and Δ is a finite set of* rules *of the following types:*

$$p\gamma \overset{a}{\hookrightarrow} p'w \qquad \text{for } p, p' \in P,\ a \in \mathsf{Act},\ \gamma \in \Gamma, \text{ and } w \in \Gamma^* \qquad \text{(local)}$$

$$p\gamma \overset{a}{\hookrightarrow} p_s w_s \sharp p'w \qquad \text{for } p, p_s, p' \in P,\ a \in \mathsf{Act},\ \gamma \in \Gamma, \text{ and } w_s, w \in \Gamma^* \qquad \text{(spawn)}$$

Intuitively, a (local)-rule describes a classic pushdown transition. A (spawn)-rule additionally creates a new thread as a side effect. The created thread starts with the control-state p_s and the stack w_s

Usually, we use variable p for control-symbols, variable γ for stack-symbols, variable a for actions, and variables w and r for stacks.

In order to store a DPN $M = (P, \Gamma, \mathsf{Act}, \Delta)$, we need space

$$|M| = \mathsf{poly}(|P||\Gamma||\mathsf{Act}|).$$

The argumentation is the same as for automata and tree automata (cf. Section 2.3).

3.2.1 Interleaving Semantics

The interleaving semantics describes an execution as a sequence of steps, where each step is made by one thread. Formally, it is defined as a labeled transition system over configurations.

Definition 3.2 (Configuration)**.** *A* configuration *of a DPN $M = (P, \Gamma, \mathsf{Act}, \Delta)$ is a list of* thread-configurations*, where each thread-configuration contains the thread's control-state and stack. The set of configurations over P and Γ is*

$$\mathsf{Conf}_{P,\Gamma} := (P\Gamma^*)^*$$

If P and Γ are clear from the context, we also write Conf instead of $\mathsf{Conf}_{P,\Gamma}$. Usually, we use φ as a variable for thread-configurations, and c for configurations.

Definition 3.3 (Interleaving Semantics of DPNs)**.** *Let $M = (P, \Gamma, \mathsf{Act}, \Delta)$ be a DPN. Then, the relation $\rightarrow_M \subseteq \mathsf{Conf} \times \mathsf{Act} \times \mathsf{Conf}$ is called* interleaving semantics *of M. It is defined as the least relation that satisfies the following constraints:*

$$c_1(p\gamma r)c_2 \xrightarrow{a}_M c_1(p'wr)c_2 \qquad \text{if } p\gamma \overset{a}{\hookrightarrow} p'w \in \Delta \qquad \text{(local)}$$

$$c_1(p\gamma r)c_2 \xrightarrow{a}_M c_1(p_s w_s)(p'wr)c_2 \qquad \text{if } p\gamma \overset{a}{\hookrightarrow} p_s w_s \sharp p'w \in \Delta \qquad \text{(spawn)}$$

With $\rightarrow_M^*$, we denote the reflexive, transitive closure of $\rightarrow_M$, i.e., for $\bar{a} = a_1 \dots a_n$, we have $c \xrightarrow{\bar{a}}_M^* c'$ iff there are configurations $c_0, \dots, c_n$ such that

$$c = c_0 \xrightarrow{a_1}_M c_1 \xrightarrow{a_2}_M \cdots \xrightarrow{a_n}_M c_n = c'.$$

If the DPN M is clear from the context, we write $\rightarrow$ instead of $\rightarrow_M$, and $\rightarrow^*$ instead of $\rightarrow_M^*$.

The (local)-constraint matches the usual definition of the semantics of a pushdown system. A step induced by the (spawn)-constraint additionally spawns a new thread and inserts its control-state and stack to the left of the spawning thread.

It is sometimes convenient to view configurations as plain sequences of elements from $P \cup \Gamma$ that start with an element from P. For this, we assume $P \cap \Gamma = \emptyset$. A thread-configuration starts with a control-symbol, followed by the stack-elements from top to bottom, and a configuration is the concatenation of its thread-configurations. For example, the configuration $[(p_1, [\gamma_1, \gamma_2]), (p_2, [\gamma_3, \gamma_4])] \in \mathsf{Conf}$ becomes the sequence $[p_1, \gamma_1, \gamma_2, p_2, \gamma_3, \gamma_4]$. Note that we used unambiguous list notation for this example, in order to emphasize the difference between a configuration and a sequence of elements from $P \cup \Gamma$. From now on, we identify configurations and sequences of elements from $P \cup \Gamma$ that start with an element from P. A set of configurations is called *regular*, iff it is the language of some automaton.

3.2.2 Tree-Semantics

Up to now, we have regarded an interleaving semantics of DPNs, i.e., an execution is a totally ordered sequence of actions. In an interleaving semantics, we have two types of nondeterministic choice: First, many rules may apply to a specific thread to derive its next step, and, second, there may be many threads ready to make a next step. The interleaving semantics makes no distinction between these two types of nondeterminism. The idea of the tree-semantics is to separate these two types of nondeterminism. In a first step, the rules that are applied to each thread are chosen, and, in a second step, the order in which concurrent steps are executed is chosen. The result of the first step is a partially ordered multiset (pomset) of actions, the result of the second step is a topological ordering of this set. As the only rules that affect more than one thread are spawn-rules, the partial ordering on the actions always has a tree shape.

Example 3.4. *Consider a DPN with the rules*

$$p\gamma \overset{a_1}{\hookrightarrow} p_1\gamma \sharp p_1\gamma'$$
$$p_1\gamma \overset{a_2}{\hookrightarrow} p_2\gamma \quad p_1\gamma' \overset{a_4}{\hookrightarrow} p_2\gamma'$$
$$p_2\gamma \overset{a_3}{\hookrightarrow} p_3\gamma \quad p_2\gamma' \overset{a_5}{\hookrightarrow} p_3\gamma'$$

It has the (maximal) executions $p\gamma \xrightarrow{\bar{a}} p_3\gamma p_3\gamma'$ *for* $\bar{a} \in S \subseteq \mathsf{Act}^*$ *with*

$$S = \{a_1a_2a_3a_4a_5, a_1a_2a_4a_3a_5, a_1a_2a_4a_5a_3, a_1a_4a_2a_5a_3, a_1a_4a_2a_3a_5, a_1a_4a_5a_2a_3\}.$$

Note that all 6 possible choices of $\bar{a}$ *result from exactly the same steps executed on the same threads in the same per-thread order. They only differ in the order in which steps of parallel threads have been executed. The dependency of the chosen steps yields the following partial ordering of the steps' actions:*

$$\{a_1 < a_2, a_1 < a_4, a_2 < a_3, a_4 < a_5\}$$

This ordering is visualized by the following tree:

$$\begin{tikzcd} & a_1 \arrow[dl] \arrow[dr] & \\ a_2 \arrow[d] & & a_4 \arrow[d] \\ a_3 & & a_5 \end{tikzcd}$$

Its set of topological orderings is exactly the set S*.*

Execution-trees represent such partial orderings of actions as ranked trees.

Definition 3.5 (Execution-Trees)**.** *The set* $\mathsf{T}_{\mathsf{Act}}$ *of* execution-trees *over actions* Act *is defined by the following grammar:*

$$\mathsf{T}_{\mathsf{Act}} ::= \tau \mid a(\mathsf{T}_{\mathsf{Act}}) \mid a(\mathsf{T}_{\mathsf{Act}}, \mathsf{T}_{\mathsf{Act}}), \textit{ for } a \in \mathsf{Act}$$

Executions that start at a configuration with more than one thread are described by a list of execution-trees, containing one tree per thread. Those lists are called execution-hedges*. We define the set* $\mathsf{H}_{\mathsf{Act}}$ *of execution-hedges by*

$$\mathsf{H}_{\mathsf{Act}} = (\mathsf{T}_{\mathsf{Act}})^*.$$

When clear from the context, we omit the index and write T *and* H *instead of* $\mathsf{T}_{\mathsf{Act}}$ *and* $\mathsf{H}_{\mathsf{Act}}$*. Usually, we use* t *as a variable for execution-trees, and* h *for execution-hedges.*

Next, we define a semantics of DPNs that defines executions between configurations as execution-trees (viz. execution-hedges).

Definition 3.6 (Tree-Semantics)**.** *Let* $M = (P, \Gamma, \mathsf{Act}, \Delta)$ *be a DPN. The tree-semantics* $\Rightarrow_M \subseteq P\Gamma^* \times \mathsf{T} \times \mathsf{Conf}$ *is defined as the least relation that satisfies the following constraints:*

$$p\gamma r \overset{\tau}{\Rightarrow}_M p\gamma r \qquad \text{(leaf)}$$

$$p\gamma r \overset{a(t)}{\Longrightarrow}_M c' \quad \textit{if } p\gamma \overset{a}{\hookrightarrow} p'w \in \Delta \textit{ and } p'wr \overset{t}{\Rightarrow}_M c' \qquad \text{(local)}$$

$$p\gamma r \overset{a(t_s,t)}{\Longrightarrow}_M c_sc' \quad \textit{if } p\gamma \overset{a}{\hookrightarrow} p_sw_s\sharp p'w \in \Delta,\ p_sw_s \overset{t_s}{\Rightarrow}_M c_s \textit{ and } p'wr \overset{t}{\Rightarrow}_M c' \qquad \text{(spawn)}$$

We lift $\Rightarrow_M$ to execution-hedges in the natural way, i.e.

$$\varphi_1 \dots \varphi_n \xRightarrow{t_1 \dots t_n}_M c_1 \dots c_n \text{ if } \varphi_1 \overset{t_1}{\Rightarrow}_M c_1 \wedge \dots \wedge \varphi_n \overset{t_n}{\Rightarrow}_M c_n$$

If the DPN M is clear from the context, we write $\Rightarrow$ instead of $\Rightarrow_M$.

The topological orderings of the labels of an execution-hedge are called *schedules* of the hedge.

Definition 3.7 (Schedules)**.** *Let* Act *be a set of actions. We define the* scheduling relation $\rightsquigarrow \subseteq \mathsf{H} \times \mathsf{Act} \times \mathsf{H}$ *as the least relation that satisfies the following constraints:*

$$h_1[a(t)]h_2 \overset{a}{\rightsquigarrow} h_1[t]h_2 \qquad \text{(local)}$$
$$h_1[a(t_1,t_2)]h_2 \overset{a}{\rightsquigarrow} h_1[t_1,t_2]h_2 \qquad \text{(spawn)}$$

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

The set $\mathsf{sched}(h)$ *of* schedules *of an execution-hedge h is the set of maximal runs of $\rightsquigarrow$ from this hedge:*

$$\mathsf{sched}(h) = \{\bar{a} \mid \exists h' \in \{\tau\}^*.\ h \overset{\bar{a}}{\rightsquigarrow}^* h'\}.$$

We now prove the coincidence between the interleaving semantics and the tree-semantics that was already suggested by Example 3.4.

Theorem 3.8 (Equality of Interleaving Semantics and Tree-Semantics)**.** *Let $M = (P, \Gamma, \mathsf{Act}, \Delta)$ be a DPN. The interleaving semantics has a run $c \overset{\bar{a}}{\rightarrow}^* c'$ if and only if there exists an execution-hedge $h \in \mathsf{H}$ with $c \overset{h}{\Rightarrow} c'$ and $\bar{a} \in \mathsf{sched}(h)$.*

Proof. For the $\Longrightarrow$-direction, we show that the scheduler can follow every step of the interleaving semantics, i.e.

$$c \overset{a}{\rightarrow} \hat{c} \overset{\hat{h}}{\Rightarrow} c' \implies \exists h.\ c \overset{h}{\Rightarrow} c' \wedge h \overset{a}{\rightsquigarrow} \hat{h} \qquad (*)$$

This is done by a case distinction on the constraint that was used to derive the $c \overset{a}{\rightarrow} \hat{c}$ step. If the (local)-constraint with $p\gamma \overset{a}{\hookrightarrow} p'w \in \Delta$ was used, we have $c = c_1[p\gamma r]c_2$ and $\hat{c} = c_1[p'wr]c_2$. Following the partitioning of $\hat{c}$, we can write $\hat{h}$ as $\hat{h} = h_1[\hat{t}]h_2$ with $|h_1| = |c_1|$ and $|h_2| = |c_2|$. Accordingly, we can write c' as $c' = c'_1 c'_t c'_2$ with $c_1 \overset{h_1}{\Rightarrow} c'_1$, $p'wr \overset{\hat{t}}{\Rightarrow} c'_t$, and $c_2 \overset{h_2}{\Rightarrow} c'_2$. By the (local)-constraint of $\Rightarrow$, we get $p\gamma r \xRightarrow{a(\hat{t})} c'_t$, and thus $c \xRightarrow{h_1[a(\hat{t})]h_2} c'$. Moreover, by the (local)-constraint of the scheduler, we have $h_1[a(\hat{t})]h_2 \overset{a}{\rightsquigarrow} \hat{h}$. If the $c \overset{a}{\rightarrow} c'$ step was derived by the (spawn)-constraint, the argumentation is analogous.

The $\Longrightarrow$-direction is now proved by induction over the reflexive, transitive closure in $c \overset{\bar{a}}{\rightarrow}^* c'$. If we have $c = c'$ and $\bar{a} = \varepsilon$, we set $h = \tau^{|c|}$, i.e., the execution-hedge that contains $|c|$ leafs. We observe that $c \overset{h}{\Rightarrow} c$ and $h \overset{\varepsilon}{\rightsquigarrow}^* h$. Due to $h \in \{\tau\}^*$, we get $\varepsilon \in \mathsf{sched}(h)$.

Now, assume we have $c \xrightarrow{a} \hat{c} \xrightarrow{\bar{a}}^* c'$. By induction hypothesis, we obtain $\hat{h}$ with $\hat{c} \overset{\hat{h}}{\Rightarrow} c'$, and $\bar{a} \in \mathsf{sched}(\hat{h})$. By $(*)$ we obtain h with $c \overset{h}{\Rightarrow} c'$ and $h \overset{a}{\rightsquigarrow} \hat{h}$. Thus we also get $a\bar{a} \in \mathsf{sched}(h)$.

For the $\Longleftarrow$-direction, we show that the interleaving semantics can follow every step of the scheduler, i.e.

$$c \overset{h}{\Rightarrow} c' \wedge h \overset{a}{\rightsquigarrow} \hat{h} \implies \exists \hat{c}.\ c \xrightarrow{a} \hat{c} \wedge \hat{c} \overset{\hat{h}}{\Rightarrow} c'. \qquad (\dagger)$$

This is, similar to the above proof of $(*)$, shown by case distinction over the constraint used to derive $h \overset{a}{\rightsquigarrow} h'$.

The $\Longleftarrow$-direction now follows from induction over $\bar{a}$. In the case $\bar{a} = \varepsilon$, we have $h \in \{\tau\}^*$, and hence $c = c'$. Thus we have $c \xrightarrow{\varepsilon}^* c'$. In the case $\bar{a} = a\bar{a}'$, by unfolding the definition of sched, we obtain $\hat{h} \in \mathsf{H}$ and $h_f \in \{\tau\}^*$ such that $h \overset{a}{\rightsquigarrow} \hat{h}$ and $\bar{a}' \in \mathsf{sched}(\hat{h})$. By $(\dagger)$, we obtain $\hat{c}$ such that $c \xrightarrow{a} \hat{c}$ and $\hat{c} \overset{\hat{h}}{\Rightarrow} c'$. By induction hypothesis, we get $\hat{c} \xrightarrow{\bar{a}'}^* c'$, and thus $c \xrightarrow{a\bar{a}'}^* c'$. □

Obviously, every execution-hedge has at least one schedule. Thus we get:

Corollary 3.9. *Let $M = (P, \Gamma, \mathsf{Act}, \Delta)$ be a DPN and $c, c' \in \mathsf{Conf}$ be configurations. There is a run between c and c', if and only if there is an execution-hedge from c to c'. Formally:*

$$\exists \bar{a} \in \mathsf{Act}^*.\ c \xrightarrow{\bar{a}}^* c' \iff \exists h \in \mathsf{H}.\ c \overset{h}{\Rightarrow} c'.$$

□

3.3 Locks and Monitors

A disadvantage of DPNs is that there is no communication between threads. Once a thread has been spawned, it runs completely independent of the other threads of the system. One possibility for adding communication between threads are locks. Locks allow to synchronize the access on resources between parallel threads. A thread can acquire and release locks, and each lock may be acquired by at most one thread at the same time. If a thread wants to acquire a lock that is currently acquired by another thread, it has to wait until the lock is released.

Locks may be reentrant, i.e., the same thread may re-acquire a lock that it already holds. Intuitively, acquiring a lock increments a counter on that lock and releasing decrements the counter. The thread owns the lock as long as the counter is positive.

A common lock-usage discipline is to acquire locks only on procedure calls, and release them on the matching return. This corresponds to, e.g. `synchronized`-methods in the Java programming language [47]. Locks used in this way are called *monitors*. We model DPNs with monitors by binding locks to stack-symbols. Moreover, we label rules such that their effect on the locks becomes visible:

Definition 3.10 (Lock-Actions)**.** *Let* Act *be a finite set of actions, and* $\mathcal{X}$ *be a finite set of locks. Then,*

$$\mathsf{Act}_{\mathcal{X}} := \{\square_a \mid a \in \mathsf{Act}\} \cup \{\langle_x, \rangle_x \mid x \in \mathcal{X}\}$$

denotes the set of lock-actions over actions Act *and locks* $\mathcal{X}$*. Actions of the form* $\square_a$ *are called* base-actions*, actions of the form* $\rangle_x$ *are called* release-actions*, and actions of the form* $\langle_x$ *are called* acquisition-actions*. Usually, we use* x *as a variable for locks,* X *for sets of locks, and* o *for lock-actions.*

In the context of lock-actions, also the actions from Act *are called* base-actions*.*

Definition 3.11 (Monitor-DPN)**.** *A* Monitor-DPN M *is a tuple*

$$M = (P, \Gamma, \Gamma_\perp, \mathsf{Act}, \mathcal{X}, \Delta, \mathsf{locks}),$$

where Act *is a finite set of* base-actions*,* $\mathcal{X}$ *is a finite set of* locks*,* $(P, \Gamma, \mathsf{Act}_{\mathcal{X}}, \Delta)$ *is a DPN,* $\mathsf{locks} : \Gamma \to 2^{\mathcal{X}}$*, with* $\forall \gamma \in \Gamma.\ |\mathsf{locks}(\gamma)| \leq 1$ *is a mapping from stack-symbols to sets of locks, where no stack-symbol is assigned more than one lock, and* $\Gamma_\perp$ *with* $\emptyset \subset \Gamma_\perp \subseteq \Gamma$*, such that* $\gamma \in \Gamma_\perp \implies \mathsf{locks}(\gamma) = \emptyset$ *is the set of* bottom stack-symbols*. Moreover, the rules in* Δ *are of the following types:*

Base-rules:	$p\gamma \overset{\square_a}{\hookrightarrow} p'\gamma'$	*with* $\gamma \sim \gamma'$
Push-rules:	$p\gamma \overset{\square_a}{\hookrightarrow} p'\gamma_1\gamma_2$	*with* $\gamma \sim \gamma_2$, $\mathsf{locks}(\gamma_1) = \emptyset$*, and* $\gamma_1 \notin \Gamma_\perp$
Pop-rules:	$p\gamma \overset{\square_a}{\hookrightarrow} p'$	*with* $\mathsf{locks}(\gamma) = \emptyset$ *and* $\gamma \notin \Gamma_\perp$
Spawn-rules:	$p\gamma \overset{\square_a}{\hookrightarrow} p_s\gamma_s \sharp p'\gamma'$	*with* $\gamma \sim \gamma'$ *and* $\gamma_s \in \Gamma_\perp$
Acquire-rules:	$p\gamma \overset{\langle_x}{\hookrightarrow} p'\gamma_1\gamma_2$	*with* $\gamma \sim \gamma_2$ *and* $\mathsf{locks}(\gamma_1) = \{x\}$
Release-rules:	$p\gamma \overset{\rangle_x}{\hookrightarrow} p'$	*with* $\mathsf{locks}(\gamma) = \{x\}$

where $p, p', p_s \in P$*,* $\gamma, \gamma', \gamma_s, \gamma_1, \gamma_2 \in \Gamma$*,* $a \in \mathsf{Act}$*,* $x \in \mathcal{X}$*, and* $\gamma \sim \gamma'$ *means that* γ *and* γ' *hold the same locks and are both in* $\Gamma_\perp$ *or both not in* $\Gamma_\perp$*, i.e.*

$$\gamma \sim \gamma' \ :\Longleftrightarrow\ \mathsf{locks}(\gamma) = \mathsf{locks}(\gamma') \wedge (\gamma \in \Gamma_\perp \iff \gamma' \in \Gamma_\perp).$$

We lift the locks *function to stacks, and configurations in the following way:*

$$\begin{aligned} \mathsf{locks}(\gamma_1 \dots \gamma_n) &= \mathsf{locks}(\gamma_1) \cup \dots \cup \mathsf{locks}(\gamma_n) && \text{for } \gamma_1, \dots, \gamma_n \in \Gamma \\ \mathsf{locks}(p_1 w_1 \dots p_n w_n) &= \mathsf{locks}(w_1) \cup \dots \cup \mathsf{locks}(w_n) && \text{for } p_1 w_1 \dots p_n w_n \in \mathsf{Conf} \end{aligned}$$

A stack of a Monitor-DPN is called valid if it is not empty, its bottommost symbol is from $\Gamma_\perp$, and all other symbols are from $\Gamma \setminus \Gamma_\perp$. We define the predicate $\mathsf{valid} \subseteq \Gamma^*$ *by:*

$$\mathsf{valid}(w) \iff w \in (\Gamma \setminus \Gamma_\perp)^*\Gamma_\perp.$$

We lift the valid*-predicate to configurations in the natural way:*

$$\mathsf{valid}(p_1 w_1 \dots p_n w_n) \iff \mathsf{valid}(w_1) \wedge \dots \wedge \mathsf{valid}(w_n).$$

Intuitively, base-rules make a transition without pushing symbols on the stack, changing the locks, or creating a thread. Threads are created by spawn-rules. Push- and pop-rules push and pop symbols from the stack, without changing the lockstack, while acquisition- and release-rules push and pop symbols from the stack that are bound to locks, thus changing the lockstack.

Moreover, there is a special class of stack-symbols $\Gamma_\perp$ that occur as the bottommost symbol of each stack and cannot be pushed or popped. Hence, all valid stacks are non-empty, which simplifies the constructions in Chapter 6. As Monitor-DPNs are a special case of DPNs, we can apply the lock-insensitive interleaving and tree-semantics (cf. Definitions 3.3 and 3.6) to them. Validity of configurations is preserved by transitions of the lock-insensitive semantics, i.e.

$$\mathsf{valid}(c) \wedge c \xrightarrow{\bar{o}}{}^{*} c' \implies \mathsf{valid}(c')$$

$$\mathsf{valid}(c) \wedge c \overset{h}{\Rightarrow} c' \implies \mathsf{valid}(c')$$

Proof. By inspecting the rules of a Monitor-DPN, we observe that symbols from $\Gamma_\perp$ cannot be pushed or popped, nor can a rule change the membership in $\Gamma_\perp$ of the topmost stack-symbol. Hence, stacks of existing threads remain valid. The bottommost stack-symbols of spawned threads are from $\Gamma_\perp$, hence stacks of spawned threads are initially valid. Thus, a single step preserves validity, i.e.

$$\mathsf{valid}(c) \wedge c \xrightarrow{o} c' \implies \mathsf{valid}(c').$$

From this, the first proposition is shown by straightforward induction. The second proposition follows from the first one using Corollary 3.9. □

Moreover, restricting the rules of a DPN to base-, push-, pop-, and spawn-rules does not limit the modeling power of DPNs w.r.t. reachability properties, as rules pushing arbitrary many stack-symbols in one step can be simulated by a sequence of push-rules, using intermediate states. I.e., for each rule $p\gamma \overset{a}{\hookrightarrow} p'\gamma_1 \dots \gamma_n$ with $n > 2$, we introduce new control-states $p_1 \dots p_{n-1}$ and the rules

$$\begin{aligned}
&p\gamma \overset{a}{\hookrightarrow} p_1\gamma_{n-1}\gamma_n\\
&p_1\gamma_{n-1} \overset{a}{\hookrightarrow} p_2\gamma_{n-2}\gamma_{n-1}\\
&\dots\\
&p_{n-1}\gamma_2 \overset{a}{\hookrightarrow} p'\gamma_1\gamma_2
\end{aligned}$$

Obviously, each execution of the original DPN corresponds to an execution of the new DPN, with some additional steps performing a-actions. Vice versa, each execution of the new DPN that reaches a configuration with none of the new control-states corresponds to an execution of the original DPN, when removing the additional a-actions.

For the remainder of this section, we fix a Monitor-DPN

$$M = (P, \Gamma, \Gamma_\perp, \mathsf{Act}, \mathcal{X}, \Delta, \mathsf{locks}).$$

In order to store a Monitor-DPN, we may assume that every lock actually occurs in a rule. Thus, we have $|\mathcal{X}| = O(|\Gamma|)$, and hence

$$|M| = \mathsf{poly}(|P||\Gamma||\mathsf{Act}||\mathcal{X}|) = \mathsf{poly}(|P||\Gamma||\mathsf{Act}|).$$

Each stack of a Monitor-DPN encodes a stack of locks.

Definition 3.12 (Lockstacks)**.** *The function $\mathsf{ls}_M : \Gamma \to \mathcal{X}^*$ maps a stack-symbol to the list of locks that corresponds to this stack-symbol. This is either a singleton list or the empty list. We define:*

$$\mathsf{ls}_M(\gamma) = \begin{cases} x & \textit{if } \mathsf{locks}(\gamma) = \{x\} \\ \varepsilon & \textit{else} \end{cases}$$

When clear from the context, we omit the DPN M and write ls instead of ls_M. We lift ls to stacks, thread-configurations, and configurations:

$$\mathsf{ls} : \Gamma^* \to \mathcal{X}^* \qquad \mathsf{ls} : P\Gamma^* \to \mathcal{X}^* \qquad \mathsf{ls} : \mathsf{Conf} \to (\mathcal{X}^*)^*$$

where

$$\begin{aligned} \mathsf{ls}(\gamma_1 \dots \gamma_n) &= \mathsf{ls}(\gamma_1) \dots \mathsf{ls}(\gamma_n) && \textit{for } (\gamma_1 \dots \gamma_n) \in \Gamma^* \\ \mathsf{ls}(pw) &= \mathsf{ls}(w) && \textit{for } pw \in P\Gamma^* \\ \mathsf{ls}(\varphi_1 \dots \varphi_n) &= [\mathsf{ls}(\varphi_1), \dots, \mathsf{ls}(\varphi_n)] && \textit{for } (\varphi_1 \dots \varphi_n) \in \mathsf{Conf} \end{aligned}$$

We use the variable name μ for lockstacks.

3.3.1 Lock-Sensitive Interleaving Semantics

A lock-sensitive semantics has to ensure that no lock is owned by two threads at the same time. For this, we define the set of *consistent* configurations, which are those configurations where no lock is on the lockstack of more than one thread:

Definition 3.13 (Consistent Configurations)**.** *A configuration $c \in \mathsf{Conf}$ is called* consistent*, if and only if no lock is on the lockstack of more than one thread. We define the set of consistent configurations by*

$$\mathsf{Conf}^{\mathsf{ls}} := \{p_1 w_1 \dots p_n w_n \mid n \in \mathbb{N} \wedge \forall 1 \le i < j \le n.\ \mathsf{locks}(w_i) \cap \mathsf{locks}(w_j) = \emptyset\}.$$

The lock-sensitive interleaving semantics is then defined as the restriction of the lock-insensitive semantics to consistent configurations:

Definition 3.14 (Lock-Sensitive Interleaving Semantics)**.** *The lock-sensitive interleaving semantics*

$$\rightarrow_{\mathsf{ls},M} \subseteq \mathsf{Conf}^{\mathsf{ls}} \times \mathsf{Act}_{\mathcal{X}} \times \mathsf{Conf}^{\mathsf{ls}}$$

is defined as the restriction of the lock-insensitive interleaving semantics to consistent configurations:

$$c \xrightarrow{o}_{\mathsf{ls},M} c' \;:\Longleftrightarrow\; c, c' \in \mathsf{Conf}^{\mathsf{ls}} \wedge c \xrightarrow{o}_{M} c'.$$

If the Monitor-DPN M is clear from the context, we write $\rightarrow_{\mathsf{ls}}$ instead of $\rightarrow_{\mathsf{ls},M}$.

Lemma 3.15. *The lock-sensitive interleaving semantics can be equivalently characterized as the least solution of the following constraints:*

$$\begin{aligned}
&c_1(p\gamma r)c_2 \xrightarrow{\Box_a}_{\mathsf{ls},M} c_1(p'wr)c_2 && \text{if } c_1(p\gamma r)c_2 \in \mathsf{Conf}^{\mathsf{ls}} \wedge p\gamma \overset{\Box_a}{\hookrightarrow} p'w \in \Delta && \text{(local)}\\
&c_1(p\gamma r)c_2 \xrightarrow{\Box_a}_{\mathsf{ls},M} c_1(p_s\gamma_s)(p'\gamma' r)c_2 && \text{if } c_1(p\gamma r)c_2 \in \mathsf{Conf}^{\mathsf{ls}} \wedge p\gamma \overset{\Box_a}{\hookrightarrow} p_s\gamma_s\sharp p'\gamma' \in \Delta && \text{(spawn)}\\
&c_1(p\gamma r)c_2 \xrightarrow{\langle_x}_{\mathsf{ls},M} c_1(p'\gamma_1\gamma_2 r)c_2 && \text{if } c_1(p\gamma r)c_2 \in \mathsf{Conf}^{\mathsf{ls}} \wedge p\gamma \overset{\langle_x}{\hookrightarrow} p'\gamma_1\gamma_2 \in \Delta \wedge x \notin \mathsf{locks}(c_1c_2) && \text{(acquire)}\\
&c_1(p\gamma r)c_2 \xrightarrow{\rangle_x}_{\mathsf{ls},M} c_1(p'r)c_2 && \text{if } c_1(p\gamma r)c_2 \in \mathsf{Conf}^{\mathsf{ls}} \wedge p\gamma \overset{\rangle_x}{\hookrightarrow} p' \in \Delta && \text{(release)}
\end{aligned}$$

Proof. For the $\Longrightarrow$-direction, we assume that we have a step $c \xrightarrow{o}_{\mathsf{ls}} c'$. Unfolding the definition of $\rightarrow_{\mathsf{ls}}$, we get $c, c' \in \mathsf{Conf}^{\mathsf{ls}}$ and $c \xrightarrow{o} c'$. This step is either a local or a spawn-step. In case of a local step, we get $c = c_1p\gamma rc_2$, $c' = c_1p'wrc_2$, and $p\gamma \overset{o}{\hookrightarrow} p'w \in \Delta$. We make a case distinction over o. In the case $o = \Box_a$, we get the proposition due to the (local)-constraint. In the case $o = \langle_x$, we have $w = \gamma_1\gamma_2$ and $\mathsf{locks}(\gamma_1) = \{x\}$, due to the constraints for rules of Monitor-DPNs. As c' is consistent, we have $x \notin \mathsf{locks}(c_1c_2)$, and we get the proposition due to the (acquire)-constraint. In the case $o = \rangle_x$, we have $w = \varepsilon$, and get the proposition due to the (release)-constraint. In case of a spawn-step, we have $c = c_1p\gamma rc_2$, $c' = c_1p_sw_sp'wrc_2$, and $p\gamma \overset{o}{\hookrightarrow} p_sw_s\sharp p'w \in \Delta$. Due to the constraints for rules of Monitor-DPNs, we have $w_s = \gamma_s$ and $w = \gamma'$ for some $\gamma_s, \gamma' \in \Gamma$, and the proposition follows by the (spawn)-constraint.

For the $\Longleftarrow$-direction, assume that $c \xrightarrow{o}_{\mathsf{ls}} c'$ was derived due to the above constraints. We have to show that $c, c' \in \mathsf{Conf}^{\mathsf{ls}}$ and $c \xrightarrow{o} c'$.

We observe that the above constraints are a specialization of the constraints defining $\rightarrow$. Thus, we have $c \xrightarrow{o} c'$. Moreover, all constraints require $c \in \mathsf{Conf}^{\mathsf{ls}}$. It remains to show $c' \in \mathsf{Conf}^{\mathsf{ls}}$.

We distinguish due to which constraint the step $c \xrightarrow{o}_{\mathsf{ls}} c'$ was derived. In case of the (local)- and (spawn)-constraints, we observe that, due to the constraints on the rules of a Monitor-DPN, the lockstack of the local thread is not changed, and the spawned thread's lockstack is initially empty. Hence, with $c \in \mathsf{Conf}^{\mathsf{ls}}$, we also get $c' \in \mathsf{Conf}^{\mathsf{ls}}$.

In case of the (acquire)-constraint, we have $\mathsf{locks}(\gamma_1) = \{x\}$ and $\mathsf{locks}(\gamma_2) = \mathsf{locks}(\gamma)$. As we have $c \in \mathsf{Conf}^{\mathsf{ls}}$, we have $c' \in \mathsf{Conf}^{\mathsf{ls}}$ if the newly acquired lock x does not break consistency. This is guaranteed by the side condition $x \notin \mathsf{locks}(c_1 c_2)$.

In case of the (release)-constraint, we pop a lock from the thread's lockstack. Thus, with $c \in \mathsf{Conf}^{\mathsf{ls}}$, we also get $c' \in \mathsf{Conf}^{\mathsf{ls}}$. □

3.3.2 Well-Nestedness

The binding of locks to the stack in Monitor-DPNs ensures that locks are always used in a well-nested fashion, i.e., if a lock is released, it is the last lock that was acquired and not yet released.

We want to define a lock-sensitive scheduler for execution-hedges. As an execution-hedge does not contain information about the configuration from that it was generated, there is also no information about the lockstacks. In order to define the notion of lock-sensitive scheduling, we explicitly add the lockstacks to the execution-hedge. Here, it makes sense to assume that the execution-hedge is well-nested w.r.t. those explicit lockstacks. Hence, we have to formalize the notion of a well-nested execution-hedge.

In order to make the presentation of execution-trees more readable, we introduce the convention to write a spawn-node annotated with a base-action as $\triangleright_a(t_1, t_2)$, instead of $\square_a(t_1, t_2)$.

We define the relation $\rightharpoonup$ that describes how an execution-tree modifies a given lockstack. Later in the thesis, we only need the resulting lockstack of the root thread. Hence, we omit the resulting lockstacks of spawned threads. However, we still assume that spawned threads are well-nested w.r.t. the empty lockstack.

Definition 3.16 (Lock-Transition Relation)**.** *Let* Act *be a set of base-actions and* $\mathcal{X}$ *be a set of locks. The* lock-transition relation $\rightharpoonup_{\mathsf{Act},\mathcal{X}} \subseteq \mathcal{X}^* \times \mathsf{T}_{\mathsf{Act}_\mathcal{X}} \times \mathcal{X}^*$, *is defined as the least relation that satisfies the following constraints:*

$$\begin{array}{lll}
\mu \overset{\tau}{\rightharpoonup}_{\mathsf{Act},\mathcal{X}} \mu & \textit{if } \mu \in \mathcal{X}^* & \text{(leaf)} \\
\mu \overset{\square_a(t)}{\rightharpoonup}_{\mathsf{Act},\mathcal{X}} \mu' & \textit{if } \mu \overset{t}{\rightharpoonup}_{\mathsf{Act},\mathcal{X}} \mu' & \text{(base)} \\
\mu \overset{\triangleright_a(t_s,t)}{\rightharpoonup}_{\mathsf{Act},\mathcal{X}} \mu' & \textit{if } \exists \mu_s .\ \varepsilon \overset{t_s}{\rightharpoonup}_{\mathsf{Act},\mathcal{X}} \mu_s \textit{ and } \mu \overset{t}{\rightharpoonup}_{\mathsf{Act},\mathcal{X}} \mu' & \text{(spawn)} \\
\mu \overset{\langle_x(t)}{\rightharpoonup}_{\mathsf{Act},\mathcal{X}} \mu' & \textit{if } x\mu \overset{t}{\rightharpoonup}_{\mathsf{Act},\mathcal{X}} \mu' & \text{(acquire)} \\
x\mu \overset{\rangle_x(t)}{\rightharpoonup}_{\mathsf{Act},\mathcal{X}} \mu' & \textit{if } \mu \overset{t}{\rightharpoonup}_{\mathsf{Act},\mathcal{X}} \mu' & \text{(release)}
\end{array}$$

If Act *and* $\mathcal{X}$ *are clear from the context, we write* $\rightharpoonup$ *instead of* $\rightharpoonup_{\mathsf{Act},\mathcal{X}}$.

Based on the lock-transition relation, we define well-nestedness and some related notions on execution-trees.

Definition 3.17. *Let* $t \in \mathsf{T}_{\mathsf{Act}_\mathcal{X}}$ *be an execution-tree.*

1. t *is called* well-nested w.r.t. a lockstack $\mu \in \mathcal{X}^*$*, iff there is a lockstack* μ' *such that* $\mu \overset{t}{\rightharpoonup} \mu'$*.*
2. t *is called* well-nested*, iff it is well-nested w.r.t. some lockstack. We define* T^{wn} *to be the set of well-nested execution-trees:*
$$t \in \mathsf{T}^{\mathsf{wn}} \;:\Longleftrightarrow\; \exists \mu, \mu'.\ \mu \overset{t}{\rightharpoonup} \mu'.$$
3. t *is called* non-releasing*, iff it is well-nested w.r.t. the empty lockstack. We define* T^{nr} *to be the set of non-releasing execution-trees:*
$$t \in \mathsf{T}^{\mathsf{nr}} \;:\Longleftrightarrow\; \exists \mu.\ \varepsilon \overset{t}{\rightharpoonup} \mu.$$
4. t *is called a* same-level tree*, iff* $\varepsilon \overset{t}{\rightharpoonup} \varepsilon$*. We define* T^{sl} *to be the set of same-level trees:*
$$t \in \mathsf{T}^{\mathsf{sl}} \;:\Longleftrightarrow\; \varepsilon \overset{t}{\rightharpoonup} \varepsilon.$$

The $\rightharpoonup$-relation is a transition relation on the *local branch* of a tree, i.e., the nodes of the tree corresponding to steps of the thread at the root of the tree. In order to describe its properties, it is useful to define concatenation of execution-trees at the root thread.

Definition 3.18 (Tree Concatenation)**.** *The operator* $\cdot;\cdot : \mathsf{T} \to \mathsf{T} \to \mathsf{T}$ *is inductively defined as follows:*

$$\begin{aligned} \tau; t' &= t' \\ a(t); t' &= a(t; t') \\ a(t_s, t); t' &= a(t_s, t; t') \end{aligned}$$

Example 3.19. *Intuitively, tree concatenation can be seen as a generalization of list concatenation: If an execution-tree is understood as a list of operations of the root thread, where spawn-operations have an additional execution-tree as argument, tree concatenation is concatenation of such lists. For example, consider the execution-trees* $t_1 = a_1(a_2(t_a, a_3(t_b, \tau)))$ *and* $t_2 = a_4(a_5(t_c, \tau))$*. Their concatenation* $t_1; t_2$ *can be visualized as follows:*

$$\begin{tikzcd}
a_1 \arrow[r, no head] & a_2 \arrow[r, no head] \arrow[d, no head] & a_3 \arrow[r, no head] \arrow[d, no head] & \tau & ; & a_4 \arrow[r, no head] & a_5 \arrow[r, no head] \arrow[d, no head] & \tau & = & a_1 \arrow[r, no head] & a_2 \arrow[r, no head] \arrow[d, no head] & a_3 \arrow[r, no head] \arrow[d, no head] & a_4 \arrow[r, no head] & a_5 \arrow[r, no head] \arrow[d, no head] & \tau \\
 & t_a & t_b & & & & t_c & & & & t_a & t_b & & t_c &
\end{tikzcd}$$

Tree concatenation has similar properties as list concatenation. Some important ones are formalized in the next lemma.

Lemma 3.20 (Properties of Tree Concatenation)**.** *The following properties hold for all execution-trees $t, t_1, t_2, t_3 \in \mathsf{T}$:*

1. *Tree concatenation is associative, i.e., $(t_1; t_2); t_3 = t_1; (t_2; t_3)$*
2. *The empty tree is a left- and right-neutral, i.e., $t; \tau = \tau; t = t$*
3. *The empty tree can only be produced as the concatenation of empty trees, i.e., $t_1; t_2 = \tau$ if and only if $t_1 = \tau$ and $t_2 = \tau$.*

Proof. By straightforward induction. □

Associativity of tree concatenation justifies to omit parenthesis, i.e., we write $t_1; t_2; t_3$ instead of $t_1; (t_2; t_3)$ or $(t_1; t_2); t_3$. Moreover, we omit the ;-operator if there is no ambiguity with list concatenation. In chains of ;-operations, we write just the label o for a tree $o(\tau)$. For example, we write $\langle_x t_1 \rhd_a(t_s) \rangle_x t_2$ instead of $\langle_x(\tau); t_1; \rhd_a(t_s, \tau); \rangle_x(\tau); t_2$. This notation represents an execution-tree as a sequence of steps of the root thread.

We now describe some basic properties of the lock-transition relation.

Lemma 3.21 (Properties of the Lock-Transition Relation)**.** *Let* Act *be a set of actions and $\mathcal{X}$ be a set of locks. Moreover, let $t, t' \in \mathsf{T}_{\mathsf{Act}_\mathcal{X}}$ be execution-trees and $\mu, \mu', \mu'' \in \mathcal{X}^*$ be lockstacks, and $x \in \mathcal{X}$ be a lock. Then, the following holds:*

1. *Chaining $\rightharpoondown$-transitions commutes with tree concatenation, i.e., $\exists \mu'.\ \mu \overset{t}{\rightharpoondown} \mu' \overset{t'}{\rightharpoondown} \mu''$ iff $\mu \overset{t;t'}{\rightharpoondown} \mu''$.*
2. *Transitions of $\rightharpoondown$ remain valid when elements are added at the bottom of the lockstack, i.e., $\mu \overset{t}{\rightharpoondown} \mu'$ implies $\mu\mu'' \overset{t}{\rightharpoondown} \mu'\mu''$.*
3. *The resulting lockstack is determined by the initial lockstack and the tree, i.e., $\mu \overset{t}{\rightharpoondown} \mu' \wedge \mu \overset{t}{\rightharpoondown} \mu''$ implies $\mu' = \mu''$.*
4. *Same-level trees can be prepended to runs of $\rightharpoondown$, i.e., for $\varepsilon \overset{t}{\rightharpoondown} \varepsilon$, we have $\mu \overset{t;t'}{\rightharpoondown} \mu'$ if and only if $\mu \overset{t'}{\rightharpoondown} \mu'$.*
5. *The topmost lock on the lockstack is either popped at some point, or not used at all, i.e., $x\mu \overset{t}{\rightharpoondown} \mu'$ implies one of the following disjoint cases:*
   a) *Either we obtain t_1, t_2 such that $t = t_1 \rangle_x t_2$ with $\varepsilon \overset{t_1}{\rightharpoondown} \varepsilon$, or*
   b) *we obtain μ_1 such that $\mu' = \mu_1 x \mu$ and $\varepsilon \overset{t}{\rightharpoondown} \mu_1$.*

   Additionally, case a) implies $\mu \overset{t_2}{\rightharpoondown} \mu'$.

Proof. Propositions 1-3 are shown by straightforward induction. Proposition 4 is a straightforward corollary of Propositions 1-3.

Disjointness of the cases of Proposition 5 immediately follows from Propositions 1-4. Completeness of the cases is shown by induction on t, followed by a case distinction over the definition of $\rightharpoonup$. The proof for the (leaf)-, (base)-, (spawn), and (release)-constraints is straightforward. Here, we demonstrate the case that $x\mu \overset{t}{\rightharpoonup} \mu'$ was derived due to the (acquire)-constraint, i.e., we have a lock $y \in \mathcal{X}$, can write t as $t = \langle_y t'$, and have $yx\mu \overset{t'}{\rightharpoonup} \mu'$. By applying the induction hypothesis, we get two cases:

1) We can write t' as $t' = t_1\rangle_y t_2$ with $\varepsilon \overset{t_1}{\rightharpoonup} \varepsilon$ and $x\mu \overset{t_2}{\rightharpoonup} \mu'$. In this case, we apply the induction hypothesis again, and get the cases:
   1.1) We can write t_2 as $t_2 = t_{21}\rangle_x t_{22}$ with $\varepsilon \overset{t_{21}}{\rightharpoonup} \varepsilon$ and $\mu \overset{t_{22}}{\rightharpoonup} \mu'$. Hence, we get $\varepsilon \overset{\langle_y t_1\rangle_y t_{21}}{\rightharpoonup} \varepsilon$, and thus we get case a) of the proposition.
   1.2) We obtain μ_1 with $\mu' = \mu_1 x\mu$ and $\varepsilon \overset{t_2}{\rightharpoonup} \mu_1$. Hence we have $\varepsilon \overset{\langle_y t_1\rangle_y t_2}{\rightharpoonup} \mu_1$, and thus we get case b) of the proposition.
2) We obtain a μ_1 with $\mu' = \mu_1 yx\mu$ and $\varepsilon \overset{t'}{\rightharpoonup} \mu_1$. Hence, we have $\varepsilon \overset{\langle_y t'}{\rightharpoonup} \mu_1 y$, and thus we get case b) of the proposition.

□

Moreover, the lock-transition relation simulates the tree-semantics w.r.t. the lockstack of the local thread:

Lemma 3.22. *For all configurations $c' \in \mathsf{Conf}$, thread-configurations $\varphi, \varphi' \in P\Gamma^*$, and execution-trees $t \in \mathsf{T}$, the following holds:*

$$\varphi \overset{t}{\Rightarrow} c'\varphi' \implies \mathsf{ls}(\varphi) \overset{t}{\rightharpoonup} \mathsf{ls}(\varphi').$$

Proof. By straightforward induction over $\overset{t}{\Rightarrow}$, exploiting the restrictions of the rules of a Monitor-DPN. □

3.3.3 Lock-Sensitive Scheduler

As sketched in the previous section, the lock-sensitive scheduler has to explicitly keep track of the lockstacks of each thread. For this purpose, we define *lock-execution-trees* and *lock-execution-hedges* as execution-trees and -hedges paired with lockstacks, such that the trees are well-nested w.r.t. their corresponding lockstacks, and the lockstacks are consistent, i.e., no lock is on more than one lockstack at the same time:

Definition 3.23 (Lock-Execution-Hedge). *Let* Act *be a set of actions and* $\mathcal{X}$ *be a set of locks. We define the set* $\mathsf{T}^{\mathsf{ls}} \subseteq \mathsf{T}_{\mathsf{Act}_\mathcal{X}} \times \mathcal{X}^*$ *of* lock-execution-trees *and the set* $\mathsf{H}^{\mathsf{ls}} \subseteq (\mathsf{T}_{\mathsf{Act}_\mathcal{X}} \times \mathcal{X}^*)^*$ *of* lock-execution-hedges *by:*

$$\begin{aligned}\mathsf{T}^{\mathsf{ls}} &:= \{(t, \mu) \mid \exists \mu'.\ \mu \xrightarrow{t} \mu'\}\\ \mathsf{H}^{\mathsf{ls}} &:= \{(t_1, \mu_1) \dots (t_n, \mu_n) \in (\mathsf{T}^{\mathsf{ls}})^* \mid \forall 1 \leq i < j \leq n.\ \mu_i \cap \mu_j = \emptyset\}\end{aligned}$$

If Act *and* $\mathcal{X}$ *are not clear from the context, we make them explicit by writing* $\mathsf{T}^{\mathsf{ls}}_{\mathsf{Act},\mathcal{X}}$ *and* $\mathsf{H}^{\mathsf{ls}}_{\mathsf{Act},\mathcal{X}}$.

The operations $|_1 : \mathsf{H}^{\mathsf{ls}} \to \mathsf{H}$ *and* $|_2 : \mathsf{H}^{\mathsf{ls}} \to (\mathcal{X}^*)^*$ *project a lock-execution-hedge to the execution-hedge and the list of lockstacks:*

$$\begin{aligned}((t_1, \mu_1) \dots (t_n, \mu_n))|_1 &:= t_1 \dots t_n\\ ((t_1, \mu_1) \dots (t_n, \mu_n))|_2 &:= \mu_1 \dots \mu_n\end{aligned}$$

The operation $\times : \mathsf{H} \times (\mathcal{X}^*)^* \to \mathsf{H}^{\mathsf{ls}}$ *pairs an execution-hedge with a list of lockstacks. It is only defined if the execution-hedge and the list of lockstacks have the same length, and the result is a lock-execution-hedge, i.e., if the list of lockstacks is consistent and the execution-trees are well-nested w.r.t. their corresponding lockstacks:*

$$(t_1, \dots, t_n) \times (\mu_1, \dots, \mu_m) := \begin{cases} (t_1, \mu_1) \dots (t_n, \mu_n) & \text{if } m = n \\ & \quad \text{and } (t_1, \mu_1) \dots (t_n, \mu_n) \in \mathsf{H}^{\mathsf{ls}} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Note that Lemma 3.22 implies that, for a Monitor-DPN with a configuration $c \in \mathsf{Conf}^{\mathsf{ls}}$ and an execution-hedge $c \overset{h}{\Rightarrow} c'$, the function $h \times \mathsf{ls}(c)$ is always defined.

The lock-sensitive scheduler is now defined as a labeled transition system over lock-execution-hedges:

Definition 3.24 (Lock-Sensitive Scheduler). *Let* Act *be a set of actions and* $\mathcal{X}$ *be a set of locks. We define the* lock-sensitive scheduling relation $\rightsquigarrow_{\mathsf{ls}} \subseteq \mathsf{H}^{\mathsf{ls}} \times \mathsf{Act}_\mathcal{X} \times \mathsf{H}^{\mathsf{ls}}$ *as the least relation that satisfies the following constraints:*

$$\begin{aligned}
&h_1[(\Box_a t, \mu)]h_2 \overset{\Box_a}{\rightsquigarrow} h_1[(t, \mu)]h_2 && && \text{(base)}\\
&h_1[(\langle_x t, \mu)]h_2 \overset{\langle_x}{\rightsquigarrow} h_1[(t, x\mu)]h_2 && \text{if } x \notin (h_1 h_2)|_2 && \text{(acquire)}\\
&h_1[(\rangle_x t, x\mu)]h_2 \overset{\rangle_x}{\rightsquigarrow} h_1[(t, \mu)]h_2 && && \text{(release)}\\
&h_1[(\rhd_a(t_1)t_2, \mu)]h_2 \overset{\Box_a}{\rightsquigarrow} h_1[(t_1, \varepsilon), (t_2, \mu)]h_2 && && \text{(spawn)}
\end{aligned}$$

The reflexive, transitive closure of $\rightsquigarrow_{\mathsf{ls}}$ *is denoted by* $\rightsquigarrow^*_{\mathsf{ls}}$.

The set of lock-sensitive schedules *of a lock-execution-hedge is the set of runs of* $\rightsquigarrow_{\mathsf{ls}}$ *that consume the entire hedge: We define* $\mathsf{sched}_{\mathsf{ls}} : \mathsf{H}^{\mathsf{ls}} \to 2^{(\mathsf{Act}_\mathcal{X})^*}$ *by*

$$\mathsf{sched}_{\mathsf{ls}}(h) = \{\bar{o} \mid \exists h'.\ h'|_1 \in \{\tau\}^* \wedge h \overset{\bar{o}}{\rightsquigarrow}{}^*_{\mathsf{ls}} h'\}.$$

The (base)- and (spawn)-constraints work as their lock-insensitive counterparts, and do not modify the lockstack. The side condition of the (acquire)-constraint ensures that a lock-acquisition is only executed if the lock is not acquired by some other thread. The (release)-constraint releases the topmost lock of the lockstack, if it matches the lock from the node's label (which it always does for well-nested execution-trees). Note that we defined the scheduler only on well-nested lock-execution-hedges where no two processes hold the same lock (H^{ls}). The side condition of the (acquire)-constraint makes this restriction explicit.

We show that the lock-sensitive scheduler matches the lock-sensitive interleaving semantics:

Theorem 3.25 (Equality of Lock-Sensitive Interleaving and Tree-Semantics)**.** *Let $c \in \mathsf{Conf}^{\mathsf{ls}}$ be a consistent configuration and $c' \in \mathsf{Conf}$ be an arbitrary configuration. Then, any run of the lock-sensitive interleaving semantics between c and c' corresponds to a lock-sensitive schedule of an execution-hedge between c and c'. Formally:*

$$c \xrightarrow{\bar{o}}{}^{*}_{\mathsf{ls}} c' \iff \exists h \in \mathsf{H}.\ c \overset{h}{\Rightarrow} c' \wedge \bar{o} \in \mathsf{sched}_{\mathsf{ls}}(h \times \mathsf{ls}(c)).$$

Proof. Regard the proof of Theorem 3.8. Analogously, we prove:

$$c \xrightarrow{o}_{\mathsf{ls}} \hat{c} \overset{\hat{h}}{\Rightarrow} c' \implies \exists h.\ c \overset{h}{\Rightarrow} c' \wedge h \times \mathsf{ls}(c) \overset{a}{\rightsquigarrow}_{\mathsf{ls}} \hat{h} \times \mathsf{ls}(\hat{c}) \qquad (*)$$

$$c \overset{h}{\Rightarrow} c' \wedge h \times \mathsf{ls}(c) \overset{o}{\rightsquigarrow}_{\mathsf{ls}} \hat{h} \times \hat{\mu} \implies \exists \hat{c} \in \mathsf{Conf}^{\mathsf{ls}}.\ c \xrightarrow{o}_{\mathsf{ls}} \hat{c} \overset{\hat{h}}{\Rightarrow} c' \wedge \hat{\mu} = \mathsf{ls}(\hat{c}) \qquad (\dagger)$$

The proof of $(*)$ is analogous to that of Theorem 3.8: We split the configuration and the execution-hedge into the thread that executes the step, and the surrounding threads. The hedge h is obtained by prepending the executed step to the corresponding tree of $\hat{h}$. The step $h \times \mathsf{ls}(c) \overset{a}{\rightsquigarrow}_{\mathsf{ls}} \hat{h} \times \mathsf{ls}(\hat{c})$ is obtained by using the corresponding constraint of the lock-sensitive scheduler. The side condition for an (acquire)-constraint follows from the corresponding side condition of the (acquire)-constraint of the lock-sensitive interleaving semantics.

Also the proof of $(\dagger)$ is analogous to that of Theorem 3.8: We split h and c into the thread that was scheduled and the other threads. The node of h that was scheduled corresponds to a possible step of $\xrightarrow{o}_{\mathsf{ls}}$. As the node can be scheduled lock-sensitively, the corresponding step can also be executed lock-sensitively, yielding a configuration $\hat{c} \in \mathsf{Conf}^{\mathsf{ls}}$ with $c \xrightarrow{o}_{\mathsf{ls}} \hat{c}$. As the lock-sensitive scheduler and the interleaving semantics modify the lockstacks in the same way, we have $\hat{\mu} = \mathsf{ls}(\hat{c})$.

From $(*)$, the $\implies$-direction is shown by induction over $\bar{o}$, analogously to the proof of Theorem 3.8: In the case $\bar{o} = \varepsilon$, we have $c = c'$. We choose $h = \tau^{|c|}$. Obviously, we have $c \overset{\tau^{|c|}}{\Longrightarrow} c$ and $\varepsilon \in \mathsf{sched}_{\mathsf{ls}}(\tau^{|c|} \times \mathsf{ls}(c))$.

In the case $\bar{o} = o\bar{o}'$, we obtain $\hat{c}$ with $c \xrightarrow{o}_{\mathsf{ls}} \hat{c} \xrightarrow{\bar{o}'}{}^{*}_{\mathsf{ls}} c'$. By induction hypothesis, we obtain $\hat{h}$ such that $\hat{c} \overset{\hat{h}}{\Rightarrow} c'$ and $\bar{o}' \in \mathsf{sched}_{\mathsf{ls}}(\hat{h} \times \mathsf{ls}(\hat{c}))$. By $(*)$, we obtain h with $c \overset{h}{\Rightarrow} c'$ and $h \times \mathsf{ls}(c) \overset{o}{\rightsquigarrow}_{\mathsf{ls}} \hat{h} \times \mathsf{ls}(\hat{c})$. Together, we get $o\bar{o} \in \mathsf{sched}_{\mathsf{ls}}(h \times \mathsf{ls}(c))$.

Similar, the $\Longleftarrow$-direction is shown from (†) by induction over $\bar{o}$: In the case $\bar{o} = \varepsilon$, we have $h \in \{\tau\}^*$, hence $c = c'$, and thus $c \xrightarrow{\varepsilon}^*_{\mathsf{ls}} c'$.

In the case $\bar{o} = o\bar{o}'$, we obtain $\hat{h}$ and $\hat{\bar{\mu}}$, such that $\hat{h}\times\hat{\bar{\mu}}$ is defined and we have $h\times\mathsf{ls}(c) \overset{o}{\rightsquigarrow}_{\mathsf{ls}} \hat{h}\times\hat{\bar{\mu}}$ and $\bar{o}' \in \mathsf{sched}_{\mathsf{ls}}(\hat{h}\times\hat{\bar{\mu}})$. From (†), we obtain $\hat{c} \in \mathsf{Conf}^{\mathsf{ls}}$ with $\hat{c} \overset{\hat{h}}{\Rightarrow} c'$, $c \xrightarrow{o}_{\mathsf{ls}} \hat{c}$, and $\hat{\bar{\mu}} = \mathsf{ls}(\hat{c})$. The induction hypothesis yields $\hat{c} \xrightarrow{\bar{o}'}^*_{\mathsf{ls}} c'$, and together we get $c \xrightarrow{o\bar{o}'}^*_{\mathsf{ls}} c'$. □

Similar to Corollary 3.9, we have:

Corollary 3.26. *Let $c \in \mathsf{Conf}^{\mathsf{ls}}$ be a consistent configuration and $c' \in \mathsf{Conf}$ be an arbitrary configuration. Then, there is a run between c and c', if and only if there is a schedulable lock-execution-hedge between c and c'. Formally:*

$$\exists \bar{o} \in \mathsf{Act}^*_{\mathcal{X}}.\ c \xrightarrow{\bar{o}}_{\mathsf{ls}} c' \iff \exists h \in \mathsf{H}.\ c \overset{h}{\Rightarrow} c' \wedge \mathsf{sched}_{\mathsf{ls}}(h\times\mathsf{ls}(c)) \neq \emptyset.$$

□

3.4 Summary and Related Work

In this chapter, we have introduced DPNs and Monitor-DPNs. DPNs are an extension of pushdown systems by dynamic thread creation, and Monitor-DPNs extend DPNs by synchronization between threads via monitors. For both models, we have defined an interleaving semantics and a tree-semantics. While an execution of the interleaving semantics is a totally ordered sequence of steps, an execution of the tree-semantics is a tree, that does not order unrelated steps of different threads. In order to obtain a sequential execution from an execution-tree, it is processed by a scheduler. In Theorems 3.8 and 3.25, we have shown the equivalence of the interleaving semantics and the tree-semantics—i.e., every execution of the interleaving semantics corresponds to a schedule of an execution of the tree-semantics.

In the remainder of this section, we discuss related work. We first discuss extensions of pushdown systems by concurrency, then we discuss concurrent pushdown systems with locks and with more powerful communication mechanisms. Finally, we discuss the relation of our tree-semantics to true-concurrency semantics, and to semantics that model configurations as multisets of threads rather than lists of threads.

Pushdown Systems with Concurrency A well-known model that extends pushdown systems by concurrency are PA-processes [7, 8], a process-algebraic model of parallelism, where a configuration is represented as a term containing sequential- and parallel-composition operators.

PA-processes have been integrated into a nice hierarchy of process models, called the *process rewrite system hierarchy* (PRS-hierarchy), by Mayr [84]. We briefly present Mayr's definition of PA-processes (called (1,G)-PRS in the PRS-hierarchy): PA-processes can be expressed as a system of transition rules on process terms. In Mayr's setting, there are sets Act of actions and sets Var of *process variables*. A process term has the form:

$$P ::= \varepsilon \mid \mathsf{Var} \mid P.P \mid P \parallel P$$

where ε is the empty term, Var is a process variable, $t_1.t_2$ is interpreted as sequential composition of t_1 and t_2, and $t_1 \parallel t_2$ is interpreted as parallel composition of t_1 and t_2. Mayr always regards equivalence classes of process terms modulo associativity and commutativity of parallel composition, associativity of sequential composition, and neutrality of the empty process w.r.t. sequential and parallel composition (i.e., $\varepsilon.t = t.\varepsilon = t$ and $t \parallel \varepsilon = t$). A PA-process is given by a set of rules Δ that are of the form $x \xrightarrow{a} t$, where $x \in \mathsf{Var}$ is a process variable, $a \in \mathsf{Act}$ is an action and $t \in P$ is an arbitrary process term. The rules of a PA-process induce a labeled transition system on process terms as follows:

$$\frac{(x \xrightarrow{a} t) \in \Delta}{x \xrightarrow{a} t} \qquad \frac{t_1 \xrightarrow{a} t_1'}{t_1 \parallel t_2 \xrightarrow{a} t_1' \parallel t_2} \qquad \frac{t_2 \xrightarrow{a} t_2'}{t_1 \parallel t_2 \xrightarrow{a} t_1 \parallel t_2'} \qquad \frac{t_1 \xrightarrow{a} t_1'}{t_1.t_2 \xrightarrow{a} t_1'.t_2}$$

When compared with DPNs, process variables match the stack-symbols and sequential composition matches the stack. As there is no notion of control-state, any state of the process must be encoded into the stack-symbols. This implies the first restriction compared to DPNs: While in a DPN, a rule of the form $p\gamma \overset{a}{\hookrightarrow} p'$ can return a result to the caller, there is no such possibility in PA-processes, i.e., in a process term of the form $t_1.t_2$, the outcome of t_1 cannot affect the result of t_2. In the PRS-hierarchy, this can be achieved by PAD-processes ((S,G)-PRS) that have rules of the form $x.y \xrightarrow{a} t'$.

Parallelism in PA-processes and DPNs is, however, handled completely different. While concurrency in PA-systems is bound to the callstack, in DPNs, concurrency adds a new dimension of infinity to the state-space. For example, after executing a spawn-rule in a DPN, say $p\gamma \overset{a}{\hookrightarrow} p\gamma_s \sharp p\gamma'$ that induces the transition $p\gamma\gamma_c \xrightarrow{a} p\gamma_s p\gamma'\gamma_c$, both threads continue there executions independently, each on its own stack. In particular, the spawning thread may pop stack-symbols, while the spawned thread is still running. For example, we may have the transition $p\gamma_s p\gamma'\gamma_c \rightarrow p\gamma_s p\gamma_c$, where the spawning thread pops its topmost stack-symbol, while the spawned thread is still running. However, in PA-processes, after execution of a rule of the form $x \xrightarrow{a} t_1 \parallel t_2$, inducing, for example, the transition $x.x_c \xrightarrow{a} (t_1 \parallel t_2).x_c$, we have to wait until both processes t_1 and t_2 have terminated before we can return to x_c.

PA-systems (and also PAD-systems) are not an adequate model for dynamic thread creation. This inadequacy is postulated by Bouajjani et al. [16], and supported by showing that there is a DPN whose trace language does not match the trace language of any PA-process. On the other hand, DPNs cannot model the synchronization on termination of two parallel threads.

Unifying both models leads to Constrained DPNs (CDPNs) [16] that are a generalization of both DPNs and PAD-processes. A CDPN is a DPN, where a rule may be constrained by a *stable* regular condition on the control-states of the threads spawned by the thread that executes the rule. A stable regular condition is a regular expression over stable sets of control-states, where a stable set is a set of control-states that is closed under transitions, i.e., once the control-state of a thread is in a stable set, it's control-state will remain in that stable set forever. In particular, stable constraints allow a thread to wait until a spawned thread has terminated.

In order to model the relationship of a thread to the threads that it has spawned, configurations of CDPNs are represented as trees rather than lists. In such a configuration tree, each node stores the configuration of a single thread, and the successors of a node hold the configurations of the threads spawned by the thread of this node. As shown by Bouajjani et al. [16], predecessor sets of regular sets of configuration trees are again regular and can be computed. Hence, reachability properties of CDPNs are decidable.

A common model when considering analyses with locks are parallel pushdown systems (PPDS). A PPDS consists of a fixed number of threads, where each thread is described by a pushdown system. PPDS are less general than both PA and DPN. In the first paper on precise analysis of concurrent programs with procedures and locks, Kahlon et al. [57] use parallel pushdown systems, and so does most of the work based on this paper [39, 40, 54–56, 61, 63].

Regarding dynamic thread creation, we consider interprocedural flowgraphs with monitors in [78], which are equivalent to Monitor-DPNs with a single control-state. In [79], we consider DPNs with well-nested, non-reentrant locks and in [44], we consider DPNs with well-nested, non-reentrant locks and join-operations. In this thesis, we consider Monitor-DPNs. We do not handle join-operations, but it should be possible to combine the techniques in this thesis and the techniques that we developed in [44], in order to analyze Monitor-DPNs with join-operations. However, lock-sensitive reachability analysis is PSPACE-hard when considering join-operations, while it is NP-complete without join-operations (cf. Chapter 9).

Locks and Monitors All precise analyses for concurrent programs with procedures and locks that the author is aware of either restrict to non-reentrant locking where locks are well-nested [39, 40, 44, 55–57, 61, 63, 79] or have bounded lock-chains [54], or they consider reentrant monitors [60, 78]. We discuss the problem of analyzing models with reentrant locks that do not adhere to a monitor-discipline in Section 8.1. Analysis of pushdown systems with locks that are not well-nested is undecidable in general [57]. Recently, it has been shown that a generalization of well-nestedness, so called *bounded lock-chains*, are sufficient to decide reachability [54].

Regarding well-nested, non-reentrant locks and reentrant monitors, neither is more general than the other. As an example, consider the following pushdown process:

$$p \rightarrow \mathsf{acquire}\ x; q$$
$$q \rightarrow s_1; q; s_2 \qquad\qquad q \rightarrow \mathsf{release\ x}$$

Where s_1 and s_2 are some arbitrary statements. Clearly, executions starting at p use locks in a well-nested, non-reentrant fashion. However, there is no pushdown process using monitors that has the same set of executions. Vice versa, reentrant acquisition of monitors obviously cannot be modeled by non-reentrant locks. However, DPNs with reentrant monitors can be converted to DPNs with non-reentrant monitors, at the cost of an exponential blowup in the number of monitors. For PPDS, such a conversion is presented in [60]. For Monitor-DPNs, the idea of the conversion is to encode the set of currently acquired locks into the control-state of the DPN, and flag each symbol on the stack whether the lock associated to it is reentrant or not. These flags are required to distinguish reentrant from non-reentrant release-operations. In this thesis, a similar conversion is performed in Section 4.1.

More Powerful Communication In PA-processes, communication is limited to allow parallel processes to synchronize on termination. In DPNs, apart from dynamic creation of threads, there is no possibility for threads to communicate. CDPNs allow a restricted form of communication, as a thread may observe the state of its children, but only with stable constraints. Locks and monitors add some limited form of communication, by ensuring mutual exclusion.

In contrast, real programming languages allow for more powerful communication mechanisms. A common concept is to use shared memory that may be read and written by any thread. Other well-known concepts include rendezvous-communication, where two threads synchronize on a specific statement that both threads must execute simultaneously, or message-passing, where data is passed between threads via messages. However, precise reachability analysis of systems with (at least) two pushdown threads and any of the above communication concepts (shared memory, rendezvous, message-passing) is undecidable [104]. The basic idea is to reduce the emptiness problem of the intersection of two context-free languages to reachability queries of the pushdown systems, using communication to ensure that the executions of both pushdown systems produce the same terminals.

This problem can be attacked by over- or under-approximation. The simplest method is to ignore the effects of the additional synchronization, leading to an over-approximation. Properties like absence of data-races are often ensured by using locks, such that this approximation is often precise enough to prove those properties. Another over-approximation, which is based on over-approximating runs of *communicating pushdown systems* (CPDS), has been proposed by Bouajjani et al. [14].

A natural way to design under-approximations is bounded model-checking [9, 10], where the state-space of the system is only explored up to a certain depth. For example, the KISS [102] tool regards only executions up to two context switches. This is extended by [101], where parallel pushdown systems[2] are analyzed for executions up to k context switches. In [15], this is generalized to DPNs with shared global state. Moreover, while the bound in [101] is based on context switches, the bound in [15] is based on how often information is passed between threads via shared global state, while there may be unboundedly many context switches and threads. The idea of [101] has also been adapted to concurrent weighted pushdown systems [68], allowing for certain infinite-state abstractions of program data. In Section 6.4, we discuss how to use the methods of our thesis to increase the precision of bounded model-checking for Monitor-DPNs.

True-Concurrency Semantics The ideas behind our tree-semantics are based on true-concurrency semantics. True-concurrency semantics has first been studied for Petri-nets [98, 99], and also inspired the theory of traces [87]. As illustrated in Example 3.4, an execution-tree induces a partial ordering on the multiset of actions. Such *partially ordered multisets*, also called *pomsets*, have been introduced as *partial strings* by Grabowski [48], and applied to Petri-nets. The term „pomset" was first used by Pratt [100].

Apart from encoding a pomset, an execution-tree specifies an order on the successors of an action, i.e., a spawn-node has a left successor, which describes the spawned thread, and a right successor, which describes the continuation of the spawning thread. This additional information is essential, as it allows to track the execution of a thread throughout the execution-tree. Examples are tree concatenation, which continues the execution of the root thread (cf. Example 3.19), and the lock-sensitive scheduler (cf. Section 3.3.3), which assigns the empty lockstack to the spawned thread and the original lockstack to the spawning thread when executing a spawn-node.

Ordered Configurations As mentioned above, the successors of a node in an execution-tree are ordered, and also the configurations of a DPN are an ordered list of thread-configurations. When regarding CDPNs [16] or Join-Lock-DPNs [44], configurations are modeled as trees, where each node represents a thread-configuration, and the successors of a node are the spawned threads of this node. Also in these models, the successors of a node are ordered. However, in typical concurrent programs, there is no notion of ordering on threads. Indeed, in [78] we

[2]They handle creation of a constant number of threads: The pushdown system that describes the thread is started by the spawn-statement using communication.

represent configurations as unordered multisets of thread-configurations. However, in order to apply the automata-theoretic techniques developed for DPNs, we have to specify configurations as lists.

Lugiez [81] studies DPNs with configurations that are unordered, unranked trees. Sets of such trees are described by *Presburger Tree Automata* [123], and their extension, *Presburger Weighted Tree Automata* [81]. While for the ordered setting, the set of forward reachable configurations of a CDPN is not regular [16], the main result of [81] is that, in the unordered setting, the set of forward reachable configurations is accepted by a Presburger Weighted Tree Automaton. Combination of these methods with our methods for lock-sensitive analysis is left to future research.

4 Lock-Sensitive Schedulability

In the last chapter, we have defined Monitor-DPNs with an interleaving semantics and a tree-semantics. Corollary 3.26 states that there is a lock-sensitive execution between two configurations, if and only if there is a schedulable lock-execution-hedge between those configurations. In this chapter, we show how to characterize schedulable lock-execution-hedges, using a method based on *acquisition histories* [57]. In the next chapter, we then show how to combine this characterization with the Monitor-DPN to be analyzed.

The characterization of schedulable lock-execution-hedges is simpler to describe and prove correct if only schedules are regarded that execute the steps between matching acquisition- and release-operations atomically, i.e., no steps of other threads are executed in between. Schedules with this property are called *disciplined schedules*. An arbitrary schedule of a lock-execution-hedge can always be reordered to a disciplined schedule. Thus, it is sufficient to regard disciplined schedules in order to describe schedulable lock-execution-hedges. Moreover, we only need to consider non-reentrant acquisition- and release-steps, as reentrant ones are always executable and have no effect on the set of allocated locks.

Lock-execution-hedges are an adequate model for arbitrary schedules, as the scheduler processes one node per step. Similarly, we define *lock-acquire/release-hedges* (lock-a/r-hedges) as the adequate model for disciplined schedules. In lock-a/r-hedges, the nodes between (outermost) matched acquisition and release pairs are represented by single nodes. Moreover, they contain no reentrant acquisitions and releases.

This chapter is organized as follows: In Section 4.1, we define lock-a/r-hedges and show how to transform lock-execution-hedges to lock-a/r-hedges. In Section 4.2, we show that schedulability of lock-a/r-hedges and lock-execution-hedges matches, exploiting that schedules can be reordered to disciplined schedules. In Section 4.3, we use *acquisition structures*, a generalization of acquisition histories [57], to characterize the set of schedulable lock-a/r-hedges as a tree automaton. Finally, we briefly summarize the results of this chapter and discuss related work in Section 4.4.

4.1 Acquire/Release-Hedges

In this section, we transform a lock-execution-hedge by collapsing nodes between matched acquisition- and release-operations into single nodes, and renaming reentrant lock-operations to operations that have no effect. The resulting structure is called *lock-acquire/release-hedge.*

For this section, let $M = (P, \Gamma, \Gamma_\bot, \mathsf{Act}, \mathcal{X}, \Delta, \mathsf{locks})$ be a fixed Monitor-DPN. We first define lock-acquire/release-hedges:

Definition 4.1 (Lock-Acquire/Release-Hedges)**.** *We define the following signature* $\mathsf{HAR}^{\mathsf{ls}}$ *for* lock-acquire/release-hedges *(lock-a/r-hedges for short):*

$$\begin{aligned} \mathsf{HAR}^{\mathsf{ls}} &::= \varepsilon_{\mathsf{h}} \mid (\mathsf{TAR}, X)\#_{\mathsf{h}}\mathsf{HAR}^{\mathsf{ls}} && \textit{for } X \subseteq \mathcal{X} \\ \mathsf{TAR} &::= \tau \mid \langle\rangle_X(\mathsf{SAR}, \mathsf{TAR}) \mid \rangle_x(\mathsf{TAR}) \mid \langle_x(\mathsf{TAR}) && \textit{for } x \in \mathcal{X},\ X \subseteq \mathcal{X} \\ \mathsf{SAR} &::= \varepsilon_{\mathsf{s}} \mid \mathsf{TAR}\#_{\mathsf{s}}\mathsf{SAR} \end{aligned}$$

The elements of TAR *are called* acquire/release-trees *(a/r-trees for short). Elements of* $\mathsf{HAR}^{\mathsf{ls}}$ *and* SAR *are also written as lists, omitting the* ε*- and* $\#$*-symbols, and we use a similar notation as for execution-trees to avoid parentheses: We write* $\langle_x t$*,* $\rangle_x t$*, and* $\langle\rangle_X(s)t$ *instead of* $\langle_x(t)$*,* $\rangle_x(t)$*, and* $\langle\rangle_X(s, t)$*.*

Moreover, we assume that the sets of locks in the second components of the elements of a lock-a/r-hedge are pairwise disjoint, that the locks in the trees are non-reentrant, and that there are no matched acquire/release-pairs. Formally, we define the set of well-formed lock-a/r-hedges by $\mathsf{WF} := \mathsf{WF}_{\mathsf{h}}^{\emptyset}$*. For all* $X \subseteq \mathcal{X}$*, the auxiliary sets* $\mathsf{WF}_{\mathsf{h}}^X \subseteq \mathsf{HAR}^{\mathsf{ls}}$*,* $\mathsf{WF}_{\mathsf{t}}^X \subseteq \mathsf{TAR}$*, and* $\mathsf{WF}_{\mathsf{s}} \subseteq \mathsf{SAR}$ *are defined as the least solution of the following constraints:*

$$\begin{aligned} \varepsilon_{\mathsf{h}} &\in \mathsf{WF}_{\mathsf{h}}^X && \textit{if } X \subseteq \mathcal{X} && \text{(h-empty)} \\ (t, Y)\#_{\mathsf{h}}h &\in \mathsf{WF}_{\mathsf{h}}^X && \textit{if } t \in \mathsf{WF}_{\mathsf{t}}^Y \wedge X \cap Y = \emptyset \wedge h \in \mathsf{WF}_{\mathsf{h}}^{X \cup Y} && \text{(h-cons)} \end{aligned}$$

$$\begin{aligned} \tau &\in \mathsf{WF}_{\mathsf{t}}^X && \textit{if } X \subseteq \mathcal{X} && \text{(leaf)} \\ \rangle_x t &\in \mathsf{WF}_{\mathsf{t}}^{\{x\} \cup X} && \textit{if } x \notin X \wedge t \in \mathsf{WF}_{\mathsf{t}}^X && \text{(release)} \\ \langle\rangle_Y(s)t &\in \mathsf{WF}_{\mathsf{t}}^X && \textit{if } X \cap Y = \emptyset \wedge s \in \mathsf{WF}_{\mathsf{s}} \wedge t \in \mathsf{WF}_{\mathsf{t}}^X && \text{(use)} \\ \langle_x t &\in \mathsf{WF}_{\mathsf{t}}^X && \textit{if } x \notin X \wedge t \in \mathsf{WF}_{\mathsf{t}}^{\emptyset} \cap \mathsf{WF}_{\mathsf{t}}^{\{x\} \cup X} && \text{(acquire)} \end{aligned}$$

$$\begin{aligned} \varepsilon_{\mathsf{s}} &\in \mathsf{WF}_{\mathsf{s}} && && \text{(s-empty)} \\ t\#_{\mathsf{s}}s &\in \mathsf{WF}_{\mathsf{s}} && \textit{if } t \in \mathsf{WF}_{\mathsf{t}}^{\emptyset} \wedge s \in \mathsf{WF}_{\mathsf{s}} && \text{(s-cons)} \end{aligned}$$

For the remainder of this thesis, we restrict the set $\mathsf{HAR}^{\mathsf{ls}}$ *to well-formed lock-a/r-hedges.*

As for lock-execution-hedges, we use the operators $|_1$ *and* $|_2$ *to project a lock-a/r-hedge to its a/r-hedge and its lockstacks, respectively.*

Note that we explicitly defined the constructors ε and $\#$ for hedges and lists of spawned threads, such that tree automata can be defined over those structures.

Intuitively, an element (t, X) of a lock-a/r-hedge consists of an a/r-tree t describing the execution of a thread, and a set $X \subseteq \mathcal{X}$ of locks that the thread holds initially. In an a/r-tree, a $\langle\rangle_X(s,t)$-node summarizes a sequence of nodes between a matched acquisition- and release-node. The set $X \subseteq \mathcal{X}$ is the set of *used* locks, i.e., the set of locks acquired (and released) in the summarized nodes. The list $s \in \mathsf{SAR}$ is the list of spawned threads, and $t \in \mathsf{TAR}$ is the remainder of the root thread. Nodes of the form $\langle_x t$ and $\rangle_x t$ represent unmatched acquisitions and releases.

Intuitively, in the WF_h^X-predicate, the set X models the locks that are initially held by other elements of the hedge. The (h-cons)-constraint ensures that the sets of initially held locks are pairwise disjoint, and that the lock-a/r-trees are well-formed w.r.t. their set of initially held locks.

In the WF_t^X-predicate, the set X models the locks that are currently held by the thread. The (release)-constraint ensures that the released lock is really held by the thread. The (use)-constraint ensures that the set of used locks is non-reentrant and that the list of spawned threads is well-formed. The (acquire)-constraint ensures that the acquisition is non-reentrant ($x \notin X$), and that the remainder of the tree does not free any locks, i.e., that the acquisition is really unmatched ($t \in \mathsf{WF}_\mathsf{t}^\emptyset$). Finally, the WF_s-predicate ensures that all a/r-trees in the list are well-formed w.r.t. the empty set of initially held locks.

Note that the WF_t-predicate is defined similar to the lock-transition relation on execution-trees (cf. Definition 3.16). Because we have no reentrance, it is sufficient to use sets of locks, instead of the lockstacks used for the lock-transition relation. Additionally, the WF_t-predicate makes explicit only the start lockset, while the lock-transition relation is defined as a relation between two lockstacks.

The well-formedness of lock-a/r-hedges is similar to the consistency constraint for lock-execution-hedges (cf. Definition 3.23). Additionally, it ensures that all matching acquisition- and release-nodes are collapsed into use-nodes, and that reentrance is completely eliminated, i.e., that there are no reentrant acquisition- and release-nodes and that use-nodes are not annotated with reentrant locks.

4.1.1 Mapping Execution-Hedges to A/R-Hedges

In order to map lock-execution-hedges to lock-a/r-hedges, we have to identify matching acquisitions and releases in the lock-execution-hedge. This is achieved by the lock-transition relation (cf. Definition 3.16):

Lemma 4.2 (Case Distinction for Lock-Execution-Trees)**.** *The following case distinction for lock-execution-trees $(t,\mu) \in \mathsf{T}^{\mathsf{ls}}$ is exhaustive and unambiguous:*

$$\begin{array}{lll}
t = \tau & & \text{(leaf)} \\
t = \Box_a t' \wedge (t',\mu) \in \mathsf{T}^{\mathsf{ls}} & \textit{for some } a, t' & \text{(base)} \\
t = \rhd_a(t_s)t' \wedge (t_s,\varepsilon) \in \mathsf{T}^{\mathsf{ls}} \wedge (t',\mu) \in \mathsf{T}^{\mathsf{ls}} & \textit{for some } a, t_s, t' & \text{(spawn)} \\
t = \langle_x t_s \rangle_x t' \wedge t_s \in \mathsf{T}^{\mathsf{sl}} \wedge (t',\mu) \in \mathsf{T}^{\mathsf{ls}} & \textit{for some } x, t_s, t' & \text{(use)} \\
t = \langle_x t' \wedge t' \in \mathsf{T}^{\mathsf{nr}} & \textit{for some } x, t' & \text{(acquire)} \\
t = \rangle_x t' \wedge \mu = x\mu' \wedge (t',\mu') \in \mathsf{T}^{\mathsf{ls}} & \textit{for some } x, \mu', t' & \text{(release)}
\end{array}$$

Proof. We show that (t,μ) matches exactly one of the cases. From the definition of T^{ls} we obtain a μ' such that $\mu \overset{t}{\rightharpoonup} \mu'$. If $t = \tau$, it is matched exactly by the (leaf)-case. Otherwise, by definition of $\rightharpoonup$, the subtrees of t are also well-nested. Hence, if the root-node of t is a base- or spawn-node, it is matched exactly by the (base)- and (spawn)-case, respectively. If the root-node of t is a release-node, by definition of $\rightharpoonup$, the topmost lock on the lockstack matches the released lock. Thus, this case is matched exactly by the (release)-case. If the root-node of t is an acquisition, i.e., $t = \langle_x t'$, we have $\mu \overset{\langle_x}{\rightharpoonup} x\mu \overset{t'}{\rightharpoonup} \mu'$. We apply Lemma 3.21 (Proposition 5), and the two resulting disjoint cases a) and b) match the (use)- and (acquire)-case. □

Intuitively, Lemma 4.2 distinguishes whether an acquisition-node is matched (use) or unmatched (acquire). In the matched case, the tree between the acquisition and the matching release is a same-level tree, and in the unmatched case, it is a non-releasing tree.

Now, we define the mapping from lock-execution-hedges to lock-a/r-hedges:

Definition 4.3. *The functions* $\mathsf{ar} := \mathsf{ar_h}$, $\mathsf{ar_h} : \mathsf{H}^{\mathsf{ls}} \to \mathsf{HAR}^{\mathsf{ls}}$, $\mathsf{ar_t} : \mathsf{T}^{\mathsf{ls}} \to \mathsf{TAR}$, *and* $\mathsf{ar_s} : \mathsf{T}^{\mathsf{wn}} \times 2^{\mathcal{X}} \to 2^{\mathcal{X}} \times \mathsf{TAR}^*$ *are inductively defined over the structure of* *lock-execution-hedges, using the case distinction of Lemma 4.2.*

$$\begin{aligned}
\mathsf{ar_h}(\varepsilon) &= \varepsilon\\
\mathsf{ar_h}((t,\mu)h) &= (\mathsf{ar_t}(t,\mu),\mathsf{set}(\mu))\mathsf{ar_h}(h)
\end{aligned}$$

$$\begin{aligned}
\mathsf{ar_t}(\tau,\mu) &= \tau\\
\mathsf{ar_t}(\Box_a t,\mu) &= \langle\rangle_\emptyset(\varepsilon)\mathsf{ar_t}(t,\mu)\\
\mathsf{ar_t}(\rhd_a(t_1)t_2,\mu) &= \langle\rangle_\emptyset([\mathsf{ar_t}(t_1,\varepsilon)])\mathsf{ar_t}(t_2,\mu)\\
\mathsf{ar_t}(\langle_x t,\mu) &= \begin{cases}\langle\rangle_\emptyset(\varepsilon)\mathsf{ar_t}(t,x\mu) & \textit{if } x\in\mathsf{set}(\mu)\\ \langle_x\mathsf{ar_t}(t,x\mu) & \textit{if } x\notin\mathsf{set}(\mu)\end{cases} \qquad \textit{if } t\in\mathsf{T^{nr}}\\
\mathsf{ar_t}(\rangle_x t,x\mu) &= \begin{cases}\langle\rangle_\emptyset(\varepsilon)\mathsf{ar_t}(t,\mu) & \textit{if } x\in\mathsf{set}(\mu)\\ \rangle_x\mathsf{ar_t}(t,\mu) & \textit{if } x\notin\mathsf{set}(\mu)\end{cases}\\
\mathsf{ar_t}(\langle_x t_1\rangle_x t_2,\mu) &= \langle\rangle_{(\{x\}\setminus\mathsf{set}(\mu))\cup u}(s)\mathsf{ar_t}(t_2,\mu) \ \textit{if } t_1\in\mathsf{T^{sl}}\wedge\mathsf{ar_s}(t_1,\mathsf{set}(\mu))=(u,s)
\end{aligned}$$

$$\begin{aligned}
\mathsf{ar_s}(\tau,X) &= (\emptyset,\varepsilon)\\
\mathsf{ar_s}(\Box_a t,X) &= \mathsf{ar_s}(t,X)\\
\mathsf{ar_s}(\rhd_a(t_1)t_2,X) &= (u,\mathsf{ar_t}(t_1,\varepsilon)s) && \textit{if } \mathsf{ar_s}(t_2,X)=(u,s)\\
\mathsf{ar_s}(\langle_x t,X) &= (\{x\}\setminus X\cup u,s) && \textit{if } \mathsf{ar_s}(t,X)=(u,s)\\
\mathsf{ar_s}(\rangle_x t,X) &= \mathsf{ar_s}(t,X)
\end{aligned}$$

The definition of $\mathsf{ar_t}$ follows the case distinction of Lemma 4.2. Moreover, it keeps track of the current lockstack, to eliminate reentrance. Reentrant locks in use-nodes are simply not included into the set of used locks. Reentrant acquisition- or release-nodes are replaced by a dummy use-node $\langle\rangle_\emptyset(\varepsilon)$ that spawns no threads and uses no locks. Also base-steps are translated to such dummy nodes. Note that we intentionally defined ar such that reentrance elimination preserves the structure of the tree, and replaced reentrant nodes as well as base-nodes by dummy nodes, instead of omitting them in the translated tree. This simplifies the simulation proof between the scheduler on lock-execution-hedges and the one on lock-a/r-hedges (cf. proof of Lemma 4.11).

Well-Definedness of Definition 4.3. We have to show that the result of the ar-function is well-formed.

Let $\tilde{h}=\mathsf{ar}(h)$ for a lock-execution-hedge $h\in\mathsf{H^{ls}}$. We show $\tilde{h}\in\mathsf{WF_h^\emptyset}$. Disjointness of the sets of initially held locks in $\tilde{h}$ follows from disjointness of the lockstacks in h. It remains to show that $\mathsf{ar_t}(t,\mu)\in\mathsf{WF_t^{set(\mu)}}$ for $(t,\mu)\in\mathsf{T^{ls}}$. This is done by induction on t, according to the case distinction of Lemma 4.2.

Generalizing the goal yields:

$$\begin{aligned}(t,\mu) \in \mathsf{T}^{\mathsf{ls}} &\implies \mathsf{ar_t}(t,\mu) \in \mathsf{WF}_{\mathsf{t}}^{\mathsf{set}(\mu)}\\ \mu \xrightarrow{t_s}\!\!\!\!\triangle\ \varepsilon &\implies \mathsf{ar_s}(t_s, X) \in 2^{\mathcal{X}\setminus X} \times \mathsf{WF_s}\end{aligned}$$

We demonstrate the case $t = \langle_x t_1$ for $x \notin \mu$ and $t_1 \in \mathsf{T}^{\mathsf{nr}}$ and the case $t = \langle_x t_1 \rangle_x t_2$ for $t_1 \in \mathsf{T}^{\mathsf{sl}}$. The other cases are straightforward or analogous.

In the case $t = \langle_x t_1$ for $x \notin \mu$ and $t_1 \in \mathsf{T}^{\mathsf{nr}}$, we have $\mathsf{ar_t}(t,\mu) = \langle_x \mathsf{ar_t}(t_1, x\mu)$. By induction hypothesis, we have $\mathsf{ar_t}(t_1, x\mu) \in \mathsf{WF}_{\mathsf{t}}^{\mathsf{set}(x\mu)}$. As $t_1 \in \mathsf{T}^{\mathsf{nr}}$, it contains no unmatched release-nodes, and so does $\mathsf{ar_t}(t_1, x\mu)$, hence we also have $\mathsf{ar_t}(t_1, x\mu) \in \mathsf{WF}_{\mathsf{t}}^{\emptyset}$, and the proposition follows with the (acquire)-constraint of $\mathsf{WF_t}$.

In the case $t = \langle_x t_1 \rangle_x t_2$ for $t_1 \in \mathsf{T}^{\mathsf{sl}}$, we have $\mathsf{ar_t}(t,\mu) = \langle\rangle_{(\{x\}\setminus \mathsf{set}(\mu))\cup u}(s)\mathsf{ar_t}(t_2,\mu)$ for $(u,s) := \mathsf{ar_s}(t_1, \mathsf{set}(\mu))$. By induction hypothesis, we have $u \cap \mathsf{set}(\mu) = \emptyset$, $s \in \mathsf{WF_s}$ and $\mathsf{ar_t}(t_2,\mu) \in \mathsf{WF}_{\mathsf{t}}^{\mathsf{set}(\mu)}$. The proposition follows with the (use)-constraint of $\mathsf{WF_s}$. □

4.2 Schedules of Acquire/Release-Hedges

In the last section, we defined lock-a/r-hedges and showed how to transform lock-execution-hedges to lock-a/r-hedges. In this section, we define a scheduler on lock-a/r-hedges and show that a lock-execution-hedge is schedulable if and only if its lock-a/r-hedge is schedulable. While the *if*-direction of this theorem is rather straightforward, the *only if*-direction requires reordering of the steps of a schedule such that no steps of other threads are scheduled between matched acquisition- and release-nodes of a thread.

The scheduler on lock-a/r-hedges is defined similar to that on lock-execution-hedges: In each step, it selects a schedulable root-node of the hedge, and replaces it by its successors. Additionally, we force the scheduler not to schedule release-nodes after acquisition-nodes.

Definition 4.4 (Scheduler on Lock-A/R-Hedges)**.** *We define the relations*

$$\rightsquigarrow_{\mathsf{u}}, \rightsquigarrow_{\mathsf{a}}, \rightsquigarrow_{\mathsf{r}} \subseteq \mathsf{HAR}^{\mathsf{ls}} \times \mathsf{HAR}^{\mathsf{ls}}$$

as the least relations that satisfy the following constraints:

$$\begin{aligned}
&h_1(\langle\rangle_Y(s,t),X)h_2 \rightsquigarrow_{\mathsf{u}} h_1\mathsf{lift}(s)(t,X)h_2 && \textit{if } Y \cap \bigcup(h_1h_2)|_2 = \emptyset && \text{(use)}\\
&h_1(\langle_x(t),X)h_2 \rightsquigarrow_{\mathsf{a}} h_1(t,\{x\}\cup X)h_2 && \textit{if } x \notin \bigcup(h_1h_2)|_2 && \text{(acquire)}\\
&h_1(\rangle_x(t),\{x\}\cup X)h_2 \rightsquigarrow_{\mathsf{r}} h_1(t,X)h_2 && \textit{if } x \notin X && \text{(release)}
\end{aligned}$$

Where $\mathsf{lift}(t_1 \dots t_n) := (t_1,\emptyset)\dots(t_n,\emptyset)$*. Moreover, we define* $\rightsquigarrow_{\mathsf{ru}} := \rightsquigarrow_{\mathsf{r}} \cup \rightsquigarrow_{\mathsf{u}}$*,* $\rightsquigarrow_{\mathsf{au}} := \rightsquigarrow_{\mathsf{a}} \cup \rightsquigarrow_{\mathsf{u}}$*, and* $\rightsquigarrow := \rightsquigarrow_{\mathsf{r}} \cup \rightsquigarrow_{\mathsf{u}} \cup \rightsquigarrow_{\mathsf{a}}$*.*

A schedule of a lock-a/r-hedge completely schedules the hedge by first applying a sequence of release- and use-steps ($\rightsquigarrow^*_{\mathsf{ru}}$), and then a sequence of acquisition- and use-steps ($\rightsquigarrow^*_{\mathsf{au}}$). Analogously to the set $\mathsf{sched}_{\mathsf{ls}} : \mathsf{H}^{\mathsf{ls}} \to 2^{(\mathsf{Act}_{\mathcal{X}})^*}$*, we define the predicate* $\mathsf{sched}_{\mathsf{ar}} : \mathsf{HAR}^{\mathsf{ls}} \to \mathbb{B}$ *by*

$$\mathsf{sched}_{\mathsf{ar}}(h) \;:\Longleftrightarrow\; \exists h'.\ h'|_1 \in \{\tau\}^* \wedge h \rightsquigarrow^*_{\mathsf{ru}} \circ \rightsquigarrow^*_{\mathsf{au}} h'.$$

The following theorem connects the scheduler on lock-a/r-hedges with the scheduler on lock-execution-hedges.

Theorem 4.5. *A lock-execution-hedge has a schedule if and only if its corresponding lock-a/r-hedge has a schedule. Formally, for any lock-execution-hedge* $h \in \mathsf{H}^{\mathsf{ls}}$*, we have:*

$$\mathsf{sched}_{\mathsf{ls}}(h) \neq \emptyset \iff \mathsf{sched}_{\mathsf{ar}}(\mathsf{ar}(h))$$

In the remainder of this section, we prove this theorem. We proceed as follows: First, we define a *disciplined scheduler* on lock-execution-hedges, that schedules sequences of nodes between matched acquisitions and releases atomically. We show that there exists a corresponding disciplined schedule for any schedule. In this step, the main work of the proof is done. Next, we show a stepwise simulation between the disciplined scheduler on lock-execution-hedges and the scheduler on lock-a/r-hedges. In this step, we essentially prove the reentrance elimination correct.

4.2.1 A Theory of Movers

For reordering of schedules to disciplined schedules, we adopt the concept of *movers* from Lipton [80]. Section 4.4 contains a detailed comparison of the theory developed here and [80].

In this subsection, we first describe a theory of movers for a rather general setting, and then instantiate it to lock-sensitive scheduling. We assume that a configuration is described by a list of process-configurations from $\mathcal{P}$. Each process may own locks from $\mathcal{X}$, described by a function $\mathsf{locks} : \mathcal{P} \to 2^{\mathcal{X}}$. The locks held by the processes in a configuration are assumed to be disjoint. The key observation is that a step of a process in a configuration only depends on the state of the process itself and the locks held by the other processes. We describe transitions by a *step-relation* $r \subseteq \mathcal{P} \times \mathcal{P}^* \times \mathcal{P} \times 2^{\mathcal{X}}$. Instead of $(p, h, p', X) \in r$, we write $p \underset{r}{\rightarrow} h, p'[X]$, meaning that, within a *context* where the other processes hold the locks from X, process p makes a step to p', spawning the processes in h. We lift the step-relation to configurations:

$$h_1 p h_2 \underset{r}{\rightsquigarrow} h_1 h_s p' h_2 \text{ iff } p \underset{r}{\rightarrow} h_s, p'[\mathsf{locks}(h_1 h_2)],$$

where $\mathsf{locks}(p_1 \ldots p_n) := \mathsf{locks}(p_1) \cup \ldots \cup \mathsf{locks}(p_n)$.

Moreover, we assume that step-relations preserve disjointness of locks, are antimonotone w.r.t. the context, and freshly spawned processes own no locks:

Definition 4.6 (Valid Step-Relations). *A step-relation r is called* valid*, iff*

$$\begin{aligned} \mathsf{locks}(p) \cap X = \emptyset \wedge p \xrightarrow[r]{} h, p'[X] &\implies \mathsf{locks}(p') \cap X = \emptyset \\ X \subseteq X' \wedge p \xrightarrow[r]{} h, p'[X'] &\implies p \xrightarrow[r]{} h, p'[X] \\ p \xrightarrow[r]{} h, p'[X'] &\implies \mathsf{locks}(h) = \emptyset \end{aligned}$$

Steps that release locks or do not alter the set of locks owned by the process are called decreasing steps. Steps that acquire locks or do not alter the set of locks are called increasing steps. Formally:

$$\begin{array}{lll} r \text{ decreasing} & \text{iff } p \xrightarrow[r]{} h, p'[X] \implies \mathsf{locks}(p) \supseteq \mathsf{locks}(p') \\ r \text{ increasing} & \text{iff } p \xrightarrow[r]{} h, p'[X] \implies \mathsf{locks}(p) \subseteq \mathsf{locks}(p') \end{array}$$

Two consecutive steps may either be unrelated, on the same process, or the second step may be on a process spawned by the first step. The last two cases are expressed by the combinators $\bullet$ and $\rhd$, that combine two step-relations into a new step-relation:

$$\begin{array}{lll} p \xrightarrow[r \bullet s]{} h_1 h_2, p'[X] & \text{if } p \xrightarrow[r]{} h_1, \tilde{p}[X] \wedge \tilde{p} \xrightarrow[s]{} h_2, p'[X] & \text{(seq)} \\ p \xrightarrow[r \rhd s]{} h_1 h_2 p'_s h_3, p'[X] & \text{if } p \xrightarrow[r]{} h_1 p_s h_3, p'[X] \wedge p_s \xrightarrow[s]{} h_2, p'_s[X \cup \mathsf{locks}(p')] & \text{(spawn)} \end{array}$$

Recall that we assume that spawned processes do not acquire locks, thus processes spawned by the first step are not considered for the context of the second step.

Moreover, we define the identity step-relation id by

$$p \xrightarrow[\mathsf{id}]{} h, p' \text{ iff } p' = p \wedge h = \varepsilon.$$

It is straightforward to show the following properties:

Lemma 4.7.

1. *If r and s are valid, so are $r \bullet s$ and $r \rhd s$. Moreover, id is valid.*
2. *$\bullet$ is associative, and id is a left- and right-neutral:*

$$(r \bullet s) \bullet t = r \bullet (s \bullet t) \text{ and } r \bullet \mathsf{id} = \mathsf{id} \bullet r = r.$$

3. *Sequential composition and spawn-composition imply composition of the lifted relations:*

$$\underset{r \bullet s}{\rightsquigarrow} \subseteq \underset{r}{\rightsquigarrow} \circ \underset{s}{\rightsquigarrow} \text{ and } \underset{r \rhd s}{\rightsquigarrow} \subseteq \underset{r}{\rightsquigarrow} \circ \underset{s}{\rightsquigarrow}.$$

4. *Union distributes over lifting: $\underset{r \cup s}{\rightsquigarrow} = \underset{r}{\rightsquigarrow} \cup \underset{s}{\rightsquigarrow}$.*

□

The main-statement of this section is the following:

Lemma 4.8 (Mover-Lemma)**.** *If there is an increasing step, followed by a decreasing step on an unrelated process, these steps can be swapped.*

Formally, this is expressed as a case distinction: Given configurations $h, h' \in \mathcal{P}^*$, an increasing step-relation r, and a decreasing step-relation s, we have:

$$h \underset{r}{\rightsquigarrow} \circ \underset{s}{\rightsquigarrow} h' \implies (h \underset{r\bullet s}{\rightsquigarrow} h' \quad \vee \quad h \underset{r\triangleright s}{\rightsquigarrow} h' \quad \vee \quad h \underset{s}{\rightsquigarrow} \circ \underset{r}{\rightsquigarrow} h')$$

Note that the above cases are not necessarily disjoint. For example, we have $h \underset{\mathsf{id}}{\rightsquigarrow} h \underset{\mathsf{id}}{\rightsquigarrow} h$ and also $h \underset{\mathsf{id}\bullet\mathsf{id}}{\rightsquigarrow} h$ for the identity step-relation id.

Proof. The proof works by analyzing the relation of the first and second step. If the first and second step are in the same process, we get $h \underset{r\bullet s}{\rightsquigarrow} h'$. If the second step is in a process that was spawned by the first step, we get $h \underset{r\triangleright s}{\rightsquigarrow} h'$. Finally, if the first and second step are in independent processes, they can be swapped: The first step is increasing, hence, before the first step, the configuration has fewer locks. As the step-relation is antimonotone, the second step can be performed on that configuration. As the second step is decreasing, the resulting configuration has fewer locks than the configuration the first step was originally executed on. Thus, the first step can also be executed on that configuration.

Formally, we have to do rather extensive splitting of configurations: We assume $h \underset{r}{\rightsquigarrow} \tilde{h} \underset{s}{\rightsquigarrow} h'$. By definition of the lifting of $\rightarrow$ to configurations, we can write $h = h_1 p h_2$ and $\tilde{h} = h_1 h_s \tilde{p} h_2$, such that $p \underset{r}{\rightarrow} h_s, \tilde{p}[\mathsf{locks}(h_1 h_2)]$. We now distinguish on which part of the configuration the second step is performed. If it is performed on $\tilde{p}$, we get h'_s and p' such that $h' = h_1 h_s h'_s p' h_2$ and $\tilde{p} \underset{s}{\rightarrow} h'_s p'[\mathsf{locks}(h_1 h_2)]$, and thus $p \underset{r\bullet s}{\rightrightarrows} h_s h'_s, p'[\mathsf{locks}(h_1 h_2)]$. This yields $h \underset{r\bullet s}{\rightsquigarrow} h'$.

If the second step is performed on a process from h_s, we obtain h_a, h_b, h'_s, and p'_s such that $h_s = h_a p_s h_b$, $h' = h_1 h_a h'_s p'_s h_b \tilde{p} h_2$, and $p_s \underset{r}{\rightarrow} h'_s, p'_s[\mathsf{locks}(h_1 \tilde{p} h_2)]$. Thus, we have $p \underset{r\triangleright s}{\rightrightarrows} h_a h'_s p'_s h_b, \tilde{p}[\mathsf{locks}(h_1 h_2)]$, which implies $h \underset{r\triangleright s}{\rightsquigarrow} h'$.

If the second step is performed on h_1, we obtain h_a, h'_s, h_b, p_2, and p'_2 such that $h_1 = h_a p_2 h_b$, $h' = h_a h'_s p'_2 h_b h_s \tilde{p} h_2$, and $p_2 \underset{s}{\rightsquigarrow} h'_s, p'_2[\mathsf{locks}(h_a h_b \tilde{p} h_2)]$. As r is increasing, we have $\mathsf{locks}(p) \subseteq \mathsf{locks}(\tilde{p})$. Due to antimonotonicity of the step-relation, we get $p_2 \underset{s}{\rightarrow} h'_s, p'_2[\mathsf{locks}(h_a h_b p h_2)]$, and thus $h \underset{s}{\rightsquigarrow} h_a h'_s p'_2 h_b p h_2$. As s is decreasing, we have $\mathsf{locks}(p_2) \supseteq \mathsf{locks}(p'_2)$. Moreover, freshly spawned processes hold no locks, thus we have $\mathsf{locks}(h'_s) = \emptyset$. Due to antimonotonicity of the step-relation, we get $p \underset{r}{\rightarrow} h_s, \tilde{p}[\mathsf{locks}(h_a h'_s p'_2 h_b h_2)]$, and thus $h_a h'_s p'_2 h_b p h_2 \underset{r}{\rightsquigarrow} h'$. Together, we get $h \underset{s}{\rightsquigarrow} \circ \underset{r}{\rightsquigarrow} h'$.

The case that the second step is performed on h_2 is shown analogously. □

Moreover, we denote by r^* the reflexive, transitive closure of r w.r.t. $\bullet$, i.e., r^* is the least relation that satisfies the following constraints:

$$r^* \supseteq \mathsf{id} \text{ and } r^* \supseteq r \bullet r^*$$

Instantiation of the above theory to scheduling of lock-execution-hedges is straightforward: A process-state is a lock-execution-tree, i.e., $\mathcal{P} := \mathsf{T}^{\mathsf{ls}}$. The

locks-function maps a lock-execution-tree to the set of locks of its lockstack, i.e. $\mathsf{locks}(t,\mu) = \mathsf{set}(\mu)$. Then, configurations correspond to lock-execution-hedges, and we have $\mathsf{locks}(h) = \{x \mid \exists \mu \in h|_2.\ x \in \mu\}$.

4.2.2 Disciplined Schedules of Execution-Hedges

We now define the step-relations of the disciplined scheduler on lock-execution-hedges. The disciplined scheduler on lock-execution-hedges schedules nodes between matched acquisitions and releases in one step.

Definition 4.9 (Disciplined Scheduler on Lock-Execution-Hedges)**.** *We first define the auxiliary function* $\mathsf{used} : \mathsf{T}^{\mathsf{wn}} \to 2^{\mathcal{X}}$ *that maps a well-nested tree to the set of locks used in the local branch of this tree. The function* used *is defined inductively over the structure of an execution-tree:*

$$\begin{aligned} \mathsf{used}(\tau) &= \emptyset & \mathsf{used}(\square_a t) &= \mathsf{used}(t)\\ \mathsf{used}(\rhd_a(t_1)t_2) &= \mathsf{used}(t_2) & \mathsf{used}(\langle_x t) &= \{x\} \cup \mathsf{used}(t)\\ \mathsf{used}(\rangle_x t) &= \mathsf{used}(t) \end{aligned}$$

As a second auxiliary function, we define the function $\mathsf{spawned} : \mathsf{T}^{\mathsf{wn}} \to \mathsf{H}^{\mathsf{ls}}$ *that collects the spawned trees of the local branch of a well-nested tree, and pairs them with empty lockstacks:*

$$\begin{aligned} \mathsf{spawned}(\tau) &= \varepsilon & \mathsf{spawned}(\square_a t) &= \mathsf{spawned}(t)\\ \mathsf{spawned}(\rhd_a(t_1)t_2) &= (t_1,\varepsilon)\mathsf{spawned}(t_2) & \mathsf{spawned}(\langle_x t) &= \mathsf{spawned}(t)\\ \mathsf{spawned}(\rangle_x t) &= \mathsf{spawned}(t) \end{aligned}$$

Then, we define the relations $\xrightarrow[\mathsf{u}]{}$, $\xrightarrow[\mathsf{a}]{}$, *and* $\xrightarrow[\mathsf{r}]{}$ *as the least relations that satisfy the following constraints:*

$$\begin{aligned}
&(\square_a t,\mu) \xrightarrow[\mathsf{u}]{} \varepsilon,(t,\mu)[X] && && \text{(base)}\\
&(\rhd_a(t_s,t),\mu) \xrightarrow[\mathsf{u}]{} (t_s,[]),(t,\mu)[X] && && \text{(spawn)}\\
&(\langle_x t_s \rangle_x t,\mu) \xrightarrow[\mathsf{u}]{} \mathsf{spawned}(t_s),(t,\mu)[X] && && \text{(use)}\\
&\quad \textit{if } t_s \in \mathsf{T}^{\mathsf{sl}} \wedge (\{x\} \cup \mathsf{used}(t_s)) \cap X = \emptyset\\
&(\langle_x t,\mu) \xrightarrow[\mathsf{a}]{} \varepsilon,(t,x\mu)[X] && \textit{if } x \notin X && \text{(acquire)}\\
&(\rangle_x t,x\mu) \xrightarrow[\mathsf{r}]{} \varepsilon,(t,\mu)[X] && && \text{(release)}
\end{aligned}$$

Moreover, we define:

$$\xrightarrow[\mathsf{ru}]{} := \xrightarrow[\mathsf{r}]{} \cup \xrightarrow[\mathsf{u}]{} \quad \textit{and} \quad \xrightarrow[\mathsf{au}]{} := \xrightarrow[\mathsf{a}]{} \cup \xrightarrow[\mathsf{u}]{}\ .$$

Obviously, these relations fulfill the assumptions for step-relations, i.e., they preserve disjointness of locks, are antimonotone, and spawned processes own no locks. Using the mover-lemma, we can reorder the steps of a schedule to a disciplined schedule:

Lemma 4.10. *Any lock-sensitive schedule has a corresponding disciplined schedule on lock-execution-hedges and, vice versa, any disciplined schedule has a corresponding lock-sensitive schedule, i.e., for any lock-execution-hedge $h \in \mathsf{H}^{\mathsf{ls}}$, we have*

$$\mathsf{sched}_{\mathsf{ls}}(h) \neq \emptyset \iff \exists h' \in \mathsf{H}^{\mathsf{ls}}.\ h'|_1 \in \{\tau\}^* \wedge h \underset{\mathsf{ru}}{\rightsquigarrow}^* \circ \underset{\mathsf{au}}{\rightsquigarrow}^* h'.$$

Proof. First, we generalize the statement by omitting the condition that the schedule must be complete:

$$(\exists \bar{o}.\ h \overset{\bar{o}}{\rightsquigarrow}{}^*_{\mathsf{ls}} h') \iff h \underset{\mathsf{ru}}{\rightsquigarrow}^* \circ \underset{\mathsf{au}}{\rightsquigarrow}^* h'.$$

For the $\Longleftarrow$-direction, we observe that steps of the disciplined scheduler correspond to single steps or sequences of steps of the original scheduler: For $\underset{\mathsf{a}}{\rightsquigarrow}$ and $\underset{\mathsf{r}}{\rightsquigarrow}$ steps, we have

$$\underset{\mathsf{a}}{\rightsquigarrow} = \bigcup_{x \in \mathcal{X}} \overset{\langle x}{\rightsquigarrow}_{\mathsf{ls}} \text{ and } \underset{\mathsf{r}}{\rightsquigarrow} = \bigcup_{x \in \mathcal{X}} \overset{\rangle x}{\rightsquigarrow}_{\mathsf{ls}}.$$

Moreover, for $\underset{\mathsf{u}}{\rightsquigarrow}$-steps, we show

$$\underset{\mathsf{u}}{\rightsquigarrow} \subseteq \bigcup_{\bar{o} \in (\mathsf{Act}_{\mathcal{X}})^*} \overset{\bar{o}}{\rightsquigarrow}{}^*_{\mathsf{ls}}.$$

If the $\underset{\mathsf{u}}{\rightsquigarrow}$-step was derived by the (base)- or (spawn)-rule, we can immediately derive the corresponding step from $\rightsquigarrow_{\mathsf{ls}}$. If it was derived by the (use)-rule, the proposition follows from the following statement, which is easily shown by induction on t_s:

$$\mu \overset{t_s}{\rightharpoonup} \mu' \wedge \mathsf{used}(t_s) \cap (h_1 h_2)|_2 = \emptyset \implies \exists \bar{o}.\ (t_s, \mu) \overset{\bar{o}}{\rightsquigarrow}{}^*_{\mathsf{ls}} \mathsf{spawned}(t_s)(\tau, \mu').$$

For the $\Longrightarrow$-direction, we show how steps that we append to a disciplined schedule can be moved left into this schedule to a valid position. Formally, we show:

$$h \underset{\mathsf{ru}}{\rightsquigarrow}^* \circ \underset{\mathsf{au}}{\rightsquigarrow}^* \tilde{h} \underset{\mathsf{u}^*\bullet\mathsf{r}}{\rightsquigarrow} h' \implies h \underset{\mathsf{ru}}{\rightsquigarrow}^* \circ \underset{\mathsf{au}}{\rightsquigarrow}^* h' \qquad (*)$$

Note that we show how to shift left a sequence of use-steps, followed by a release-step. This generalization is needed for the induction proof.

Moreover, for all base-actions $a \in \mathsf{Act}$ and locks $x \in \mathcal{X}$, we have:

$$\overset{\Box_a}{\rightsquigarrow}_{\mathsf{ls}} \subseteq \underset{\mathsf{au}}{\rightsquigarrow}, \text{ and } \overset{\langle x}{\rightsquigarrow}_{\mathsf{ls}} \subseteq \underset{\mathsf{au}}{\rightsquigarrow}, \text{ and } \overset{\rangle x}{\rightsquigarrow}_{\mathsf{ls}} \subseteq \underset{\mathsf{u}^*\bullet\mathsf{r}}{\rightsquigarrow}.$$

Now, it is straightforward to construct the disciplined schedule, by appending the steps of the lock-sensitive schedule one by one to the end of the already constructed disciplined schedule, using $(*)$ to integrate release-steps into the disciplined schedule.

It remains to prove $(*)$. This is done by induction on the number of $\underset{\mathsf{au}}{\rightsquigarrow}$-steps in $h \underset{\mathsf{ru}}{\rightsquigarrow}^* \circ \underset{\mathsf{au}}{\rightsquigarrow}^* \tilde{h}$. In the base-case, we have $h \underset{\mathsf{ru}}{\rightsquigarrow}^* \tilde{h} \underset{\mathsf{u}^*\bullet\mathsf{r}}{\rightsquigarrow} h'$. Due to Lemma 4.7, we get

$$\underset{\mathsf{u}^*\bullet\mathsf{r}}{\rightsquigarrow} \subseteq \underset{\mathsf{ru}}{\rightsquigarrow}^*,$$

and thus $h \underset{\mathsf{ru}}{\rightsquigarrow}^* h'$. If there is at least one $\underset{\mathsf{au}}{\rightsquigarrow}$-step, we pick the last one, and get

$$h \underset{\mathsf{ru}}{\rightsquigarrow}^* \circ \underset{\mathsf{au}}{\rightsquigarrow}^* \hat{h} \underset{\mathsf{au}}{\rightsquigarrow} \tilde{h} \underset{\mathsf{u}^*\bullet\mathsf{r}}{\rightsquigarrow} h'.$$

As au is increasing and $\mathsf{u}^* \bullet \mathsf{r}$ is decreasing, we can apply the mover-lemma (Lemma 4.8), and get the following cases:

$\hat{h} \underset{\mathsf{au}\bullet\mathsf{u}^*\bullet\mathsf{r}}{\rightsquigarrow} h'$ In this case, the au and $\mathsf{u}^* \bullet \mathsf{r}$ steps are executed on the same thread. We distinguish whether the first step is an acquisition- or a use-step. In case of a use-step, we prepend the use-step to the sequence of use-steps before the release-step, and apply the induction hypothesis. Formally, this corresponds to the inequation

$$\mathsf{u} \bullet \mathsf{u}^* \bullet \mathsf{r} \subseteq \mathsf{u}^* \bullet \mathsf{r},$$

that is easily shown. If the first step is an acquisition-step, it is matched with the release-step, obtaining a single use-step that can be executed as last step of the disciplined schedule. Formally, this corresponds to the inequation

$$\mathsf{a} \bullet \mathsf{u}^* \bullet \mathsf{r} \subseteq \mathsf{u}^*,$$

that is also easily shown. (Note that the acquired and released lock do match, as the intermediate use-steps do not modify the lockstack.)

$\hat{h} \underset{\mathsf{au}\rhd(\mathsf{u}^*\bullet\mathsf{r})}{\rightsquigarrow} h'$ In this case, a freshly spawned thread would execute some use-steps, and then release a lock. However, as freshly spawned threads have an empty lockstack, use-steps do not change the lockstack, and a lock cannot be released from an empty lockstack, this yields a contradiction. Formally, we easily show:

$$\mathsf{au} \rhd (\mathsf{u}^* \bullet \mathsf{r}) = \emptyset$$

$\hat{h} \underset{\mathsf{u}^*\bullet\mathsf{r}}{\rightsquigarrow} \circ \underset{\mathsf{au}}{\rightsquigarrow} h'$ In this case, the induction hypothesis yields $h \underset{\mathsf{ru}}{\rightsquigarrow}^* \circ \underset{\mathsf{au}}{\rightsquigarrow}^* \circ \underset{\mathsf{au}}{\rightsquigarrow} h'$, and thus we get the proposition.

□

Next, we connect disciplined schedules of lock-execution-hedges with schedules of lock-a/r-hedges.

Lemma 4.11. *A disciplined schedule of a lock-execution-hedge corresponds to a schedule of the lock-a/r-hedge. Formally, for any lock-execution-hedge* $h \in \mathsf{H}^{\mathsf{ls}}$*, we have:*

$$(\exists h' \in \mathsf{H}^{\mathsf{ls}}.\ h'|_1 \in \{\tau\}^* \wedge h \underset{\mathsf{ru}}{\rightsquigarrow}^* \circ \underset{\mathsf{au}}{\rightsquigarrow}^* h') \iff \mathsf{sched}_{\mathsf{ar}}(\mathsf{ar}(h)).$$

Proof. After unfolding the definition of $\mathsf{sched}_{\mathsf{ar}}$, we have to show

$$\begin{aligned}&(\exists h' \in \mathsf{H}^{\mathsf{ls}}.\ h'|_1 \in \{\tau\}^* \wedge h \underset{\mathsf{ru}}{\hookrightarrow}^* \circ \underset{\mathsf{au}}{\hookrightarrow}^* h')\\ \iff &(\exists h^a \in \mathsf{HAR}^{\mathsf{ls}}.\ h^a|_1 \in \{\tau\}^* \wedge \mathsf{ar}(h) \rightsquigarrow^*_{\mathsf{ru}} \circ \rightsquigarrow^*_{\mathsf{au}} h^a).\end{aligned}$$

It is straightforward to show $h'|_1 \in \{\tau\}^* \iff \mathsf{ar}(h')|_1 \in \{\tau\}^*$. Then, the statement is proved by the following bisimulation argument:

$$(\exists h'.\ h^a = \mathsf{ar}(h') \wedge h \underset{\mathsf{r}}{\hookrightarrow} h') \iff \mathsf{ar}(h) \rightsquigarrow_{\mathsf{r}} h^a \tag{1}$$
$$(\exists h'.\ h^a = \mathsf{ar}(h') \wedge h \underset{\mathsf{u}}{\hookrightarrow} h') \iff \mathsf{ar}(h) \rightsquigarrow_{\mathsf{u}} h^a \tag{2}$$
$$(\exists h'.\ h^a = \mathsf{ar}(h') \wedge h \underset{\mathsf{a}}{\hookrightarrow} h') \iff \mathsf{ar}(h) \rightsquigarrow_{\mathsf{a}} h^a \tag{3}$$

The proofs of the $\Longrightarrow$-directions are straightforward, by unfolding the definitions. Reentrance elimination of ar causes no problems here, as the $\langle\rangle_\emptyset$-nodes resulting from reentrance elimination are always schedulable.

For the $\Longleftarrow$-directions, we have to consider reentrance elimination. We demonstrate the proof for (2) here, the other proofs are similar: So assume we have $\mathsf{ar}(h) \rightsquigarrow_{\mathsf{u}} h^a$. Hence, we can write $\mathsf{ar}(h)$ as $\mathsf{ar}(h) = h_1^a(\langle\rangle_Y(s^a, t^a), X)h_2^a$, and have $h^a = h_1^a\mathsf{lift}(s^a)(t^a, X)h_2^a$ and $Y \cap \bigcup(h_1^a h_2^a)|_2 = \emptyset$. Accordingly, we can write h as $h = h_1(\tilde{t}, \mu)h_2$ with $\mathsf{ar}(h_1) = h_1^a$, $\mathsf{ar}(h_2) = h_2^a$, and

$$\mathsf{ar}(\tilde{t}, \mu) = (\langle\rangle_Y(s^a, t^a), X) \tag{$*$}$$

This implies $X = \mathsf{set}(\mu)$ and $\mathsf{locks}(h_1h_2) = \bigcup(h_1^a h_2^a)|_2$.

Now, we distinguish on the definition of ar (Definition 4.3), how $(*)$ was produced. The use-node may be the result of a translated base-node (i) or spawn-node (ii), a sequence of nodes between a matched acquire/release-pair (iii), a reentrant acquisition-node (iv), or a reentrant release-node (v). We demonstrate the proof for case (iii) here, the other cases are trivial or analogous.

So assume we have $\tilde{t} = \langle_x t_s\rangle_x t$ with $t_s \in \mathsf{T}^{\mathsf{sl}}$ and $Y = \{x\} \setminus X \cup u$ where $(u, s^a) = \mathsf{ar}_{\mathsf{s}}(t_s, X)$. By straightforward induction over t_s, we get $u = \mathsf{used}(t_s) \setminus (X)$ and $s^a = \mathsf{ar}(\mathsf{spawned}(t_s))$. Hence, we get

$$((\{x\} \cup \mathsf{used}(t_s)) \setminus \mathsf{set}(\mu)) \cap \mathsf{locks}(h_1h_2) = \emptyset.$$

As h is consistent, we have $\mathsf{set}(\mu) \cap \mathsf{locks}(h_1h_2) = \emptyset$, and thus $(\{x\} \cup \mathsf{used}(t_s)) \cap \mathsf{locks}(h_1h_2) = \emptyset$. We set $h' := h_1(\mathsf{spawned}(t_s))(t, \mu)h_2$, and apply the (use)-rule to obtain $h \underset{\mathsf{u}}{\hookrightarrow} h'$. Obviously, we have $\mathsf{ar}(h') = h^a$, and this implies the proposition. □

Now, we complete the proof of Theorem 4.5:

Proof of Theorem 4.5. This theorem follows directly from Lemmas 4.10 and 4.11:

$$\begin{aligned}&\mathsf{sched}_{\mathsf{ls}}(h) \neq \emptyset\\ \iff &\exists h' \in \mathsf{H}^{\mathsf{ls}}.\ h'|_1 \in \{\tau\}^* \wedge h \underset{\mathsf{ru}}{\hookrightarrow}^* \circ \underset{\mathsf{au}}{\hookrightarrow}^* h' && \text{(Lem. 4.10)}\\ \iff &\mathsf{sched}_{\mathsf{ar}}(\mathsf{ar}(h)) && \text{(Lem. 4.11)}\end{aligned}$$

□

4.3 Acquisition Structures

In the last sections we have introduced lock-a/r-hedges that are built from lock-execution-hedges by summarizing matched acquisition- and release-nodes into use-nodes. We have defined a scheduler on lock-a/r-hedges, and shown that a lock-execution-hedge has a schedule if and only if its lock-a/r-hedge is schedulable (Theorem 4.5). In this section, we characterize the set of schedulable lock-a/r-hedges by a tree automaton.

This section is structured as follows: In Subsection 4.3.1, we define the notion of a dependence-graph. The dependence-graph captures the ordering constraints between the nodes of a lock-a/r-hedge that a schedule must adhere to. We show that schedulability corresponds to acyclicity of the dependence-graph and some additional consistency properties, regarding initially held locks and duplicate acquisitions. While the additional consistency properties are obviously regular, showing that acyclicity of the (unbounded) dependence-graph is regular requires some argumentation. The main idea is to use so called *acquisition-graphs* and *release-graphs*, which we introduce in Subsection 4.3.2. Finally, in Subsection 4.3.3, we put together the results to construct a tree automaton that accepts exactly the schedulable lock-a/r-hedges.

4.3.1 Dependence-Graph

Given a lock-a/r-hedge, the following criteria must be satisfied by any schedule:

1. No lock may be acquired more than once.
2. A lock that is used or acquired, and not released must not be held initially by any thread.
3. The nodes must be scheduled in tree-order.
4. No release may be scheduled after an acquisition.
5. A release of a lock must be scheduled before any usage of that lock, and, symmetrically, an acquisition of a lock must be scheduled after any usage of that lock.

Due to well-formedness, all acquisitions are unmatched. Hence, if the first criterion is violated, the acquisition that is scheduled first will prevent the other acquisitions from ever being scheduled. If the second criterion is violated, there is a lock that is acquired throughout the whole execution. A usage or acquisition of this lock can never be scheduled. The third and fourth criteria are obvious by definition of the scheduler. Finally, if the fifth criterion is violated, there is a lock that is initially owned by some thread until it is released. A usage of this lock cannot be scheduled before this release. Symmetrically, a lock that is acquired will never be released again, hence no usage of such a lock can be scheduled after the acquisition.

Note that the first and second criteria are not symmetric w.r.t. acquisition- and release-nodes. The counterpart of the first criteria, (i.e., no lock is released more than once) follows from well-formedness of the hedge. The counterpart of the second criteria, (i.e., a lock that is used or released, and not acquired, is not held finally by any thread) also holds for any schedule.

Also note that the first and second criteria are independent of the order in that the nodes of the hedge are scheduled. To capture whether the third to fifth criteria can be satisfied, we define the notion of a dependence-graph, that describes these criteria as edges in a graph over the nodes of the hedge.

Definition 4.12 (Dependence-Graph)**.** *Let $h \in \mathsf{HAR}$ be an a/r-hedge. The hedge h induces a graph $\mathsf{g}(h) = (\mathsf{V}(h), \mathsf{E}(h))$ over vertices $\mathsf{V}(h)$ and edges $\mathsf{E}(h)$, such that each node $v \in \mathsf{V}(h)$ corresponds to exactly one non-leaf-node $n(v)$ in h, and $\mathsf{E}(h)$ contains an edge $(v, v') \in \mathsf{E}(h)$, if and only if v' corresponds to a direct successor node of v in h.*

From the graph $\mathsf{g}(h)$ of a hedge, we construct the dependence-graph $\mathsf{g}^{\mathsf{dep}}(h) = (\mathsf{V}(h), \mathsf{E}^{\mathsf{dep}}(h))$ with $\mathsf{E}^{\mathsf{dep}}(h) = \mathsf{E}(h) \cup \mathsf{E}^{\mathsf{add}}$, where $\mathsf{E}^{\mathsf{add}}$ contains the following edges:

$$\begin{aligned} \mathsf{E}^{\mathsf{add}} \; := \; & \{(v, v') \mid \exists x, y.\; n(v) = \rangle_x \wedge n(v') = \langle_y\} & (1)\\ & \cup \{(v, v') \mid \exists x, X.\; n(v) = \rangle_x \wedge n(v') = \langle\rangle_{\{x\}\cup X}\} & (2)\\ & \cup \{(v, v') \mid \exists x, X.\; n(v) = \langle\rangle_{\{x\}\cup X} \wedge n(v') = \langle_x\} & (3) \end{aligned}$$

We overload g,$\mathsf{g}^{\mathsf{dep}}$,$\mathsf{V}$,$\mathsf{E}$,$\mathsf{E}^{\mathsf{dep}}$, and $\mathsf{E}^{\mathsf{add}}$ to lock-a/r-hedges in the natural way, i.e., ignoring the lockstacks and just considering the trees.

Intuitively, in $\mathsf{E}^{\mathsf{add}}$, the edges due to (1) describe that releases must be scheduled before acquisitions. The edges due to (2) describe that a usage has to be scheduled after the release of a used lock and the edges due to (3) describe that an acquisition has to be scheduled after a usage of the acquired lock.

Well-Definedness. Up to naming of the nodes in V, the graph $\mathsf{g}(h)$ is unique for a given hedge h. To formally define $\mathsf{g}(h)$, we use access strings to uniquely identify the nodes in h. An access string $s \in (\mathbb{N})^+$ is a non-empty sequence of natural numbers, where each number indexes the next node on a path through the hedge. The nodes in $\mathsf{V}(h)$ are then pairs of access strings and nodes from h. The edges $\mathsf{E}(h)$ are inductively defined as follows:

$$\begin{aligned} \mathsf{E}(t_1 \ldots t_n) &= 1 \cdot \mathsf{E}(t_1) \cup \ldots \cup n \cdot \mathsf{E}(t_n)\\ \mathsf{E}(\tau) &= \emptyset\\ \mathsf{E}(\rangle_x t) &= \{((\varepsilon, \rangle_x), (1, \mathsf{root}(t)))\} \cup 1 \cdot \mathsf{E}(t)\\ \mathsf{E}(\langle_x t) &= \{((\varepsilon, \langle_x), (1, \mathsf{root}(t)))\} \cup 1 \cdot \mathsf{E}(t)\\ \mathsf{E}(\langle\rangle_X(t_1 \ldots t_n, t_{n+1})) &= \{((\varepsilon, \langle\rangle_X), (i, \mathsf{root}(t_i))) \mid 1 \le i \le n+1\} \cup \bigcup_{1 \le i \le n+1} i \cdot \mathsf{E}(t_i) \end{aligned}$$

where $i \cdot E := \{(([i]a, b), ([i]a', b')) \mid ((a, b), (a', b')) \in E\}$, and $\mathsf{root}(t)$ is the root-node of t. The nodes $\mathsf{V}(h)$ are the nodes that occur in $\mathsf{E}(h)$, i.e., $\mathsf{V}(h) := \{v \mid \exists v'.\ (v, v') \in \mathsf{E}(h) \vee (v', v) \in \mathsf{E}(h)\}$. The mapping $n(v)$ is defined as projection to the second component: $n(a, b) = b$. $\square$

In order to reduce the notational overhead for the following proofs, we introduce some notation for dependence-graphs: We omit the argument h from $\mathsf{E}(h)$ and $\mathsf{E}^{\mathsf{dep}}(h)$ if it is clear from the context. Moreover, we write $v \rightarrow_{\mathsf{E}} v'$ instead of $(v, v') \in \mathsf{E}$, and $v \rightarrow^*_{\mathsf{E}} v'$ instead of $(v, v') \in \mathsf{E}^*$, and use analogous notations for $\mathsf{E}^{\mathsf{add}}$, $\mathsf{E}^{\mathsf{dep}}$, and $\rightarrow^+$.

When referring to nodes of the dependence-graph, we are often only interested in the corresponding hedge-node. In such cases, we omit the access string, and write, for example, $\langle\rangle_X \rightarrow^*_{\mathsf{E}} \rangle_x$, instead of $\exists a, a'.\ ((a, \langle\rangle_X), (a', \rangle_x)) \in \mathsf{E}^*$. Moreover, we sometimes are only interested whether a use-node uses a particular lock x. If the type of x is clear from the context, we may write $\langle\rangle_x$ instead of $\langle\rangle_{\{x\} \cup X}$ for some X.

Using the dependence-graph, we formally capture the criteria specified above, and show that they are not only necessary, but also sufficient for the existence of a schedule:

Definition 4.13. *For an a/r-hedge $h \in \mathsf{HAR}$, we define the set of used, acquired, and released locks as follows:*

$$\begin{aligned} \mathsf{used}(h) &= \bigcup \{X \mid \langle\rangle_X \in \mathsf{V}(h)\} \\ \mathsf{acq}(h) &= \{x \mid \langle_x \in \mathsf{V}(h)\} \\ \mathsf{rel}(h) &= \{x \mid \rangle_x \in \mathsf{V}(h)\} \end{aligned}$$

We overload used*,* acq*, and* rel *to lock-a/r-hedges in the natural way.*

Lemma 4.14. *Given a lock-a/r-hedge $h \in \mathsf{HAR}^{\mathsf{ls}}$, it is schedulable (i.e., $\mathsf{sched}_{\mathsf{ar}}(h)$ holds), if and only if the following criteria are satisfied:*

$$\forall v, v' \in \mathsf{V}(h).\ n(v) = \langle_x \wedge n(v') = \langle_x \implies v = v' \tag{C1}$$

$$((\mathsf{used}(h) \cup \mathsf{acq}(h)) \setminus \mathsf{rel}(h)) \cap h|_2 = \emptyset \tag{C2}$$

$$\forall v.\ (v, v) \notin \mathsf{E}^{\mathsf{dep}}(h)^+ \tag{C3}$$

Intuitively, (C1) states that no lock may be acquired more than once, (C2) states that a lock that is used or acquired, and not released must not be held initially by any thread, and (C3) states that the dependence-graph must be acyclic.

Proof. For the $\Longrightarrow$-direction, assume $\mathsf{sched}_{\mathsf{ar}}(h)$ holds, i.e., we have an h' with $h'|_1 \in \{\tau\}^*$ and $h \rightsquigarrow^*_{\mathsf{ru}} \circ \rightsquigarrow^*_{\mathsf{au}} h'$. The proof is done by induction on $\rightsquigarrow^*_{\mathsf{ru}} \circ \rightsquigarrow^*_{\mathsf{au}}$. In the base-case, we have $h = h'$, hence $h|_1 \in \{\tau\}^*$, i.e., h only contains leaf-nodes and we have $\mathsf{acq}(h) = \mathsf{used}(h) = \mathsf{rel}(h) = \emptyset$, and $\mathsf{V}(h) = \emptyset$. Thus, (C1)–(C3) are trivially satisfied.

Now assume we have $h \rightsquigarrow_{\mathsf{ru}} \tilde{h} \rightsquigarrow_{\mathsf{ru}}^{*} \circ \rightsquigarrow_{\mathsf{au}}^{*} h'$ or $h \rightsquigarrow_{\mathsf{au}} \tilde{h} \rightsquigarrow_{\mathsf{au}}^{*} h'$, and $\tilde{h}$ satisfies (C1)–(C3). We have to show that h also satisfies (C1)–(C3). For this, let v be the node scheduled[1] in the first step, i.e., we have $\mathsf{V}(h) = \{v\} \dot\cup \mathsf{V}(\tilde{h})$.

On (C1): If v is a use- or release-node, the set of acquisition-nodes in h and $\tilde{h}$ is the same, and thus h satisfies (C1), because $\tilde{h}$ does. So assume v is an acquisition-node, i.e., $v = \langle_x$ for some lock x. Hence we have $h \rightsquigarrow_{\mathsf{a}} \tilde{h} \rightsquigarrow_{\mathsf{au}}^{*} h'$ and $x \in \tilde{h}|_2$. Then, $\tilde{h}$ contains no release-nodes, as such nodes cannot be scheduled by $\rightsquigarrow_{\mathsf{au}}$. Hence, we have $\mathsf{rel}(\tilde{h}) = \emptyset$. As $\tilde{h}$ satisfies (C2), we have $x \notin \mathsf{acq}(\tilde{h})$, and thus h does not violate (C1) involving v. Moreover, h does not violate (C1) involving any other nodes, as those nodes would also be contained in $\tilde{h}$. Hence, h satisfies (C1).

On (C2): If v is a use-node, i.e., $v = \langle\rangle_X$, we have $\mathsf{used}(h) = X \cup \mathsf{used}(\tilde{h})$, $\mathsf{acq}(h) = \mathsf{acq}(\tilde{h})$, $\mathsf{rel}(h) = \mathsf{rel}(\tilde{h})$, and $h|_2 = \tilde{h}|_2$. Because the first step was scheduled, and h is well-formed, we have $X \cap h|_2 = \emptyset$, and thus (C2) also holds for h.

If v is a release-node, we have $\mathsf{used}(h) = \mathsf{used}(\tilde{h})$, $\mathsf{acq}(h) = \mathsf{acq}(\tilde{h})$, and $\mathsf{rel}(h) = \{x\} \cup \mathsf{rel}(\tilde{h})$, where x is the released lock. Moreover, we have $h|_2 = \{x\} \cup \tilde{h}|_2$, and thus h satisfies (C2) because $\tilde{h}$ does.

If v is an acquisition-node, we have $\mathsf{used}(h) = \mathsf{used}(\tilde{h})$, $\mathsf{rel}(h) = \mathsf{rel}(\tilde{h})$, and $\mathsf{acq}(h) = \{x\} \cup \mathsf{acq}(\tilde{h})$ where x is the acquired lock. Because the acquisition was scheduled, and h is well-formed, we have $x \notin h|_2$, and thus h satisfies (C2) because $\tilde{h}$ does.

On (C3): We show that v has no incoming edges in $\mathsf{E}^{\mathsf{dep}}(h)$, and thus acyclicity of $\mathsf{g}^{\mathsf{dep}}(h)$ is implied by acyclicity of $\mathsf{g}^{\mathsf{dep}}(\tilde{h})$. Because v was scheduled, it is a root-node of h, and thus has no incoming edges in $\mathsf{E}(h)$. We now show that it also has no incoming edges in $\mathsf{E}^{\mathsf{add}}(h)$: An edge $(\tilde{v}, v) \in \mathsf{E}^{\mathsf{add}}(h)$ may be due to one of the following cases (cf. Definition 4.12):

(1) v is an acquisition-node, and $\tilde{v}$ is a release-node. However, if v is an acquisition-node, we are in the case $\tilde{h} \rightsquigarrow_{\mathsf{au}}^{*} h'$, Moreover, we have $\tilde{v} \in \tilde{h}$. However, the release-node $\tilde{v}$ can never be scheduled by $\rightsquigarrow_{\mathsf{au}}$, yielding a contradiction.

(2) v is a use-node, and $\tilde{v}$ is a release-node, i.e., we have $v = \langle\rangle_{\{x\}\cup X}$ and $\tilde{v} = \rangle_x$. As h is well-formed, the release-node $\tilde{v}$ has no matching acquisition, and thus releases a lock that is initially held by h, i.e., we have $x \in h|_2$. However, as the use-node v is schedulable and h is well-formed, we also have $(\{x\} \cup X) \cap h|_2 = \emptyset$, which yields a contradiction.

[1]Note that, in each step, the scheduler removes exactly one non-leaf root-node from the hedge.

(3) v is an acquisition-node, and $\tilde{v}$ is a use-node, i.e., $v = \langle_x$ and $\tilde{v} = \langle\rangle_{\{x\}\cup X}$. We are in the case $\tilde{h} \rightsquigarrow^*_{\mathsf{au}} h'$, and thus have $\mathsf{rel}(\tilde{h}) = \emptyset$. Moreover, we have $x \in \mathsf{used}(\tilde{h})$, and because we just scheduled an acquisition of lock x, we also have $x \in \tilde{h}|_2$. Thus, $\tilde{h}$ violates (C2), in contradiction to the assumption.

For the $\Longleftarrow$-direction, we construct a schedule for h by induction on h. If h contains only leaf-nodes, it is trivially schedulable. So assume h contains at least one non-leaf-node, and satisfies (C1)–(C3). As $\mathsf{g}^{\mathsf{dep}}(h)$ is acyclic and contains no leaf-nodes at all, there is a minimal (non-leaf) node v from $\mathsf{g}^{\mathsf{dep}}(h)$, i.e., $\forall \tilde{v}.\ (\tilde{v}, v) \notin \mathsf{E}^{\mathsf{dep}}(h)$. We show that v can be scheduled, i.e., we obtain $\tilde{h}$ with $h \rightsquigarrow_x \tilde{h}$ for $x \in \{\mathsf{r}, \mathsf{u}, \mathsf{a}\}$ and $\mathsf{V}(h) = \mathsf{V}(\tilde{h}) \mathbin{\dot{\cup}} \{v\}$. Moreover, we show that $\tilde{h}$ satisfies (C1)–(C3), such that we can apply the induction hypothesis to continue the schedule. Note that $\tilde{h}$ trivially satisfies (C1) and (C3), as it contains less nodes than h. We now distinguish over the node v:

- If v is a release-node, i.e., $v = \rangle_x$, it can be scheduled, i.e., $h \rightsquigarrow_{\mathsf{r}} \tilde{h}$. Moreover, as h is well-formed, we have $x \in h|_2$. As the locksets in $h|_2$ are disjoint, and the scheduling step $h \rightsquigarrow_{\mathsf{r}} \tilde{h}$ removes x from one of the locksets, we have $x \notin \tilde{h}|_2$. Hence, $\tilde{h}$ satisfies (C2). By induction hypothesis, we have $\mathsf{sched}_{\mathsf{ar}}(\tilde{h})$, and thus $\mathsf{sched}_{\mathsf{ar}}(h)$.

- If v is a use-node, i.e., $v = \langle\rangle_X$, we first show $X \cap h|_2 = \emptyset$, which implies that v can be scheduled, i.e., $h \rightsquigarrow_{\mathsf{u}} \tilde{h}$. So assume that there is a lock $x \in X \cap h|_2$. From $x \in X$, we get $x \in \mathsf{used}(h)$, and (C2) implies $x \in \mathsf{rel}(h)$. Hence there is a release-node $\rangle_x \in \mathsf{V}(h)$. However, this implies the edge $(\rangle_x, v) \in \mathsf{E}^{\mathsf{add}} \subseteq \mathsf{E}^{\mathsf{dep}}$, in contradiction to minimality of v.

  Moreover, we have $\mathsf{used}(\tilde{h}) \subseteq \mathsf{used}(h)$, $\mathsf{acq}(\tilde{h}) = \mathsf{acq}(h)$, $\mathsf{rel}(\tilde{h}) = \mathsf{rel}(h)$, and $\tilde{h}|_2 = h|_2$, and thus $\tilde{h}$ satisfies (C2) because h does. By induction hypothesis, we get $\mathsf{sched}_{\mathsf{ar}}(\tilde{h})$, and thus $\mathsf{sched}_{\mathsf{ar}}(h)$.

- Now let v be an acquisition-node, i.e., $v = \langle_x$. First note that h does not contain any release-nodes, as $\mathsf{E}^{\mathsf{add}}$ contains an edge from any release to any acquisition-node, and v is minimal. Hence we have $\mathsf{rel}(h) = \emptyset$. Moreover, we have $x \in \mathsf{acq}(h)$, and with (C2) we get $x \notin h|_2$. Hence v can be scheduled, i.e., $h \rightsquigarrow_{\mathsf{a}} \tilde{h}$. As h satisfies (C1), it contains no second acquisition of x, and we have $x \notin \mathsf{acq}(\tilde{h})$. Moreover, as any use-node $\langle\rangle_{\{x\}\cup X} \in \mathsf{V}(h)$ would imply an edge $(\langle\rangle_{\{x\}\cup X}, v)$, and v is minimal, we have $x \notin \mathsf{used}(h) = \mathsf{used}(\tilde{h})$. Hence $\tilde{h}$ satisfies (C2) because h does. By induction hypothesis, we get $\mathsf{sched}_{\mathsf{ar}}(\tilde{h})$, i.e., there is an h' with $h'|_1 \in \{\tau\}^*$ and $\tilde{h} \rightsquigarrow^*_{\mathsf{ru}} \circ \rightsquigarrow^*_{\mathsf{au}} h'$. Because $\tilde{h}$ contains no release-nodes, we even have $\tilde{h} \rightsquigarrow^*_{\mathsf{au}} h'$, and thus $h \rightsquigarrow^*_{\mathsf{au}} h'$. Together, we get $\mathsf{sched}_{\mathsf{ar}}(h)$.

□

4.3.2 Acquisition- and Release-Graphs

Criteria (C1) and (C2) can obviously be checked by a finite tree automaton that collects the acquisition-, release-, and use-sets in a bottom-up computation. However, (C3) involves building the dependence-graph, which has unbounded size. In this subsection, we show how to check acyclicity of the dependence-graph within finite state, i.e., without actually computing the dependence-graph. Instead, we compute a compressed representation of the dependence-graph, called the release- and acquisition-graph. We first observe that any cycle in the dependence-graph consists of either use- and release-nodes, or use- and acquisition-nodes, i.e., there is no cycle containing both, acquisition- and release-nodes. Moreover, each cycle contains at least one acquisition- or release-node. This motivates a classification of cycles whether they contain an acquisition- or a release-nodes, and to handle each class separately. Lets regard a cycle in the dependence-graph that contains at least one acquisition-node. It has the form

$$\langle_x \rightarrow^+_{\mathsf{E}^{\mathsf{dep}}} \langle_x$$

As acquisition-nodes have no outgoing edges in $\mathsf{E}^{\mathsf{add}}$, the first edge of the cycle stems from E. As E is acyclic, the cycle contains at least one edge from $\mathsf{E}^{\mathsf{add}}$, and as in E there is no path from an acquisition- to a release-node, the first edge from $\mathsf{E}^{\mathsf{add}}$ goes from a use-node to an acquisition-node. Thus we have

$$\langle_x \rightarrow^+_{\mathsf{E}} \langle\rangle_{x_1} \rightarrow_{\mathsf{E}^{\mathsf{add}}} \langle_{x_1} \rightarrow^* \langle_x.$$

By iterating this argumentation we get

$$\langle_x \rightarrow^+_{\mathsf{E}} \langle\rangle_{x_1} \rightarrow_{\mathsf{E}^{\mathsf{add}}} \langle_{x_1} \rightarrow^+_{\mathsf{E}} \dots \rightarrow^+_{\mathsf{E}} \langle\rangle_{x_n} \rightarrow_{\mathsf{E}^{\mathsf{add}}} \langle_{x_n}$$

with $x_n = x$. Note that we use the notation $\langle\rangle_x$ instead of $\exists X.\ \langle\rangle_{\{x\}\cup X}$ here.

The key idea of checking acyclicity in finite state is to represent the sequences of the form $\langle_{x_i} \rightarrow^+_{\mathsf{E}} \langle\rangle_{x_{i+1}} \rightarrow_{\mathsf{E}^{\mathsf{add}}} \langle_{x_{i+1}}$ in the dependence-graph by a single edge in the acquisition-graph, and identify the acquisition-nodes by the acquired locks. For cycles over release-nodes, we do the symmetric construction. This leads to the following definition of the acquisition- and release-graph:

Definition 4.15 (Acquisition- and Release-Graph)**.** *Let $h \in \mathsf{HAR}$ be an a/r-hedge. We define the* acquisition-graph $\mathsf{E}^{\mathsf{acq}}(h) \subseteq \mathcal{X} \times \mathcal{X}$ *and the* release-graph $\mathsf{E}^{\mathsf{rel}}(h) \subseteq \mathcal{X} \times \mathcal{X}$ *as the least solution of the following constraints:*

$$\begin{array}{lll} (x, x') \in \mathsf{E}^{\mathsf{acq}}(h) & \textit{if} & \langle_x \rightarrow^+_{\mathsf{E}} \langle\rangle_{x'} \\ (x, x') \in \mathsf{E}^{\mathsf{rel}}(h) & \textit{if} & \langle\rangle_x \rightarrow^+_{\mathsf{E}} \rangle_{x'} \end{array}$$

Again, we omit the argument h of $\mathsf{E}^{\mathsf{acq}}(h)$ and $\mathsf{E}^{\mathsf{rel}}(h)$ when clear from the context.

We now show that acquisition and release-graphs can be used to correctly detect cycles in the dependence-graph:

Lemma 4.16. *Let $h \in \mathsf{HAR}$ be an a/r-hedge. The dependence-graph $\mathsf{E}^{\mathsf{dep}}(h)$ contains a cycle, if and only if the acquisition-graph $\mathsf{E}^{\mathsf{acq}}(h)$ or the release-graph $\mathsf{E}^{\mathsf{rel}}(h)$ contain a cycle.*

Proof. First of all, we show that the dependence-graph does not contain a path from an acquisition- to a release-node, i.e.

$$\nexists x, x'.\ \langle_x \rightarrow^*_{\mathsf{E}^{\mathsf{dep}}} \rangle_{x'}. \qquad (*)$$

The proof is done by contradiction. Assume we have $\langle_x \rightarrow^*_{\mathsf{E}^{\mathsf{dep}}} \rangle_{x'}$. We proceed by induction on the length of the path. Clearly, the path cannot be empty. As h is well-formed, there are no paths from acquisition- to release-nodes only in E. So we pick the first edge in $\mathsf{E}^{\mathsf{add}}$ on the path, and get nodes v_1, v_2 with

$$\langle_x \rightarrow^*_{\mathsf{E}} v_1 \rightarrow_{\mathsf{E}^{\mathsf{add}}} v_2 \rightarrow^*_{\mathsf{E}^{\mathsf{dep}}} \rangle_{x'}.$$

We distinguish over type of the edge $v_1 \rightarrow_{\mathsf{E}^{\mathsf{add}}} v_2$ (cf. Definition 4.12). In case (1) and (3), v_2 is an acquisition-node, and the contradiction follows by induction hypothesis. In case (2), v_1 is a release-node, and the path $\langle_x \rightarrow^*_{\mathsf{E}} v_1$ contradicts well-formedness of h.

Now we prove the lemma: For the $\Longrightarrow$-direction, assume that $\mathsf{E}^{\mathsf{dep}} = \mathsf{E} \cup \mathsf{E}^{\mathsf{add}}$ contains a cycle. As E is acyclic (it is the graph of a hedge), the cycle contains at least one edge from $\mathsf{E}^{\mathsf{add}}$, i.e., it can be written as $v \rightarrow_{\mathsf{E}^{\mathsf{add}}} v' \rightarrow^*_{\mathsf{E}^{\mathsf{dep}}} v$ for some nodes $v, v' \in \mathsf{V}$. We distinguish over the type of the edge $v \rightarrow_{\mathsf{E}^{\mathsf{add}}} v'$ (cf. Definition 4.12):

(1) v is a release-node and v' is an acquisition-node. As we have $v' \rightarrow^*_{\mathsf{E}^{\mathsf{dep}}} v$, this yields a contradiction to $(*)$.

(2) v is a release-node and v' is a use-node. This case is shown analogously to the next case.

(3) v is a use-node and v' is an acquisition-node, and we have $v = \langle\rangle_x$ and $v' = \langle_x$ for some lock x. We show that any path starting with such an edge yields a corresponding path in the acquisition-graph, i.e.

$$\forall x, x''.\ \langle\rangle_x \rightarrow_{\mathsf{E}^{\mathsf{add}}} \langle_x \rightarrow^*_{\mathsf{E}^{\mathsf{dep}}} \langle\rangle_{x''} \implies x \rightarrow^+_{\mathsf{E}^{\mathsf{acq}}} x''. \qquad (\dagger)$$

Applied to our path $v \rightarrow_{\mathsf{E}^{\mathsf{add}}} v' \rightarrow^*_{\mathsf{E}^{\mathsf{dep}}} v$ this yields the cycle $x \rightarrow^+_{\mathsf{E}^{\mathsf{acq}}} x$.

The statement $(\dagger)$ is proved by induction on the length of the path $\langle_x \rightarrow^*_{\mathsf{E}^{\mathsf{dep}}} \langle\rangle_{x''}$. Clearly, the path is not empty. If it only contains edges from E, we have $x \rightarrow_{\mathsf{E}^{\mathsf{acq}}} x''$ by definition of $\mathsf{E}^{\mathsf{acq}}$. So lets assume there is at least one edge from $\mathsf{E}^{\mathsf{add}}$. If we pick the first one, we can write the path as

$$\langle\rangle_x \rightarrow_{\mathsf{E}^{\mathsf{add}}} \langle_x \rightarrow^*_{\mathsf{E}} v_1 \rightarrow_{\mathsf{E}^{\mathsf{add}}} v_2 \rightarrow^*_{\mathsf{E}^{\mathsf{dep}}} \langle\rangle_{x''}.$$

The edge $v_1 \rightarrow_{\mathsf{E}^{\mathsf{add}}} v_2$ must be of type (3), otherwise v_1 or v_2 would be a release-node, in contradiction to $(*)$. Hence we have

$$\rangle_x \rightarrow_{\mathsf{E}^{\mathsf{add}}} \langle_x \rightarrow^*_{\mathsf{E}} \rangle_y \rightarrow_{\mathsf{E}^{\mathsf{add}}} \langle_y \rightarrow^*_{\mathsf{E}^{\mathsf{dep}}} \rangle_{x''}.$$

By definition of $\mathsf{E}^{\mathsf{acq}}$, we have $x \rightarrow_{\mathsf{E}^{\mathsf{acq}}} y$. By induction hypothesis, we get $y \rightarrow^*_{\mathsf{E}^{\mathsf{acq}}} x''$, and thus $x \rightarrow^*_{\mathsf{E}^{\mathsf{acq}}} x''$.

For the $\Longleftarrow$-direction, we have to construct a cycle in the dependence-graph from a cycle in the acquisition or release-graph. We do the construction for acquisition-graphs here, the argumentation for release-graphs is analogous. So assume we have a cycle $x_1 \rightarrow_{\mathsf{E}^{\mathsf{acq}}} \dots \rightarrow_{\mathsf{E}^{\mathsf{acq}}} x_n$ with $x_n = x_1$. By definition of $\mathsf{E}^{\mathsf{acq}}$, the edges result from paths in E, i.e., we have

$$\langle_{x_1} \rightarrow^+_{\mathsf{E}} \rangle_{x_2} \quad \langle_{x_2} \rightarrow^+_{\mathsf{E}} \rangle_{x_3} \quad \cdots \quad \langle_{x_{n-1}} \rightarrow^+_{\mathsf{E}} \rangle_{x_n}.$$

Moreover, $\mathsf{E}^{\mathsf{add}}$ contains the edges $\rangle_{x_i} \rightarrow_{\mathsf{E}^{\mathsf{add}}} \langle_{x_i}$ that close the gaps in the path above, and we get

$$\langle_{x_1} \rightarrow^+_{\mathsf{E}} \rangle_{x_2} \rightarrow_{\mathsf{E}^{\mathsf{add}}} \langle_{x_2} \rightarrow^+_{\mathsf{E}} \rangle_{x_3} \rightarrow_{\mathsf{E}^{\mathsf{add}}} \cdots \rightarrow_{\mathsf{E}^{\mathsf{add}}} \langle_{x_{n-1}} \rightarrow^+_{\mathsf{E}} \rangle_{x_n} \rightarrow_{\mathsf{E}^{\mathsf{add}}} \langle_{x_n}.$$

As we defined $\mathsf{E}^{\mathsf{dep}} = \mathsf{E} \cup \mathsf{E}^{\mathsf{add}}$, and set $x_n = x_1$, we get the cycle $\langle_{x_1} \rightarrow^+_{\mathsf{E}^{\mathsf{dep}}} \langle_{x_1}$ in the dependence-graph.

Note that we used quite informal ...-notation in the proof of the $\Longleftarrow$-direction. Formally, the proof is done by induction on n, generalizing the statement to non-cyclic paths, i.e., removing the assumption $x_n = x_1$. $\square$

4.3.3 A Tree Automaton for Schedulable A/R-Hedges

In the last two subsections, we have defined criteria for schedulability of a lock-a/r-hedge, and have shown how to encode them into finite state. In order to decide schedulability of a lock-a/r-hedge, we need the set of finally acquired, initially released, used, and initially held locks, as well as the acquisition and release-graph. In this section, we show how to compute this information bottom-up over the lock-a/r-hedge. Thus, schedulability of a lock-a/r-hedge can be expressed as a deterministic bottom-up tree automaton.

The states of the tree automaton are called *acquisition structures* and *hedge acquisition structures*. An acquisition structure is a five-tuple $(r, g_r, u, a, g_a) \in \mathcal{X} \times \mathcal{X}^2 \times \mathcal{X} \times \mathcal{X} \times \mathcal{X}^2$, where r is the set of released locks, g_r is the release-graph, u is the set of used locks, a is the set of acquired locks, and g_a is the acquisition-graph. The set of acquisition structures is denoted by

$$\mathsf{AS} := 2^{\mathcal{X} \times \mathcal{X}^2 \times \mathcal{X} \times \mathcal{X} \times \mathcal{X}^2}.$$

When the tree automaton processes the list of lock-a/r-trees and sets of initially held locks in the lock-a/r-hedge, it has to additionally collect the initially held locks. At this phase, its states are *hedge acquisition structures* of the form (σ, X), where $\sigma \in \mathsf{AS}$ is the acquisition structure of the hedge processed so far, and $X \subseteq \mathcal{X}$ is a the set of initially held locks. The set of hedge acquisition structures is denoted by

$$\mathsf{AS}_{\mathsf{h}} := \mathsf{AS} \times 2^{\mathcal{X}}.$$

We introduce some operations on acquisition structures. First of all, we abbreviate the *empty acquisition structure* that consists of empty sets only by $\emptyset^5 \in \mathsf{AS}$. Analogously, the *empty hedge acquisition structure* is abbreviated by $\emptyset^6 \in \mathsf{AS}_{\mathsf{h}}$, i.e.

$$\emptyset^5 := (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset) \qquad\qquad \emptyset^6 := (\emptyset^5, \emptyset).$$

Next, we define an operation for parallel composition of two acquisition structures. This operator is used to combine the acquisition structures at use-nodes and the acquisition structures of the lock-a/r-trees in the hedge. The information in the hedge acquisition structure can be used to check (C2) and (C3) after the lock-a/r-hedge is completely processed. However, (C1), (i.e., the lock-a/r-hedge contains no two final acquisitions of the same lock) must be checked during the computation. For this purpose, we define the operations on acquisition structures as partial functions that are undefined iff Criterion (C1) is violated. We explicitly denote an undefined value with the symbol $\perp$. We define $\|: \mathsf{AS} \times \mathsf{AS} \to \mathsf{AS}$ by:

$$(r, g_r, u, a, g_a) \parallel (r', g_r', u', a', g_a') := \begin{cases} \perp & \text{if } a \cap a' \neq \emptyset \\ (r \cup r', g_r \cup g_r', u \cup u', a \cup a', g_a \cup g_a') & \text{else} \end{cases}$$

We overload parallel composition to hedge acquisition structures, and define

$$(\sigma, X) \parallel (\sigma', X') := (\sigma \parallel \sigma', X \cup X').$$

Obviously, the $\|$-operator is associative and commutative, and $\emptyset^5$ ($\emptyset^6$ respectively) is a neutral element.

Finally, we define the partial function $\mathsf{as} : \mathsf{HAR}^{\mathsf{ls}} \to \mathsf{AS}_{\mathsf{h}}$ inductively over the structure of lock-a/r-hedges. For this purpose, we set $\mathsf{as} := \mathsf{as}_{\mathsf{h}}$, and define the partial functions $\mathsf{as}_{\mathsf{h}} : \mathsf{HAR}^{\mathsf{ls}} \to \mathsf{AS}_{\mathsf{h}}$, $\mathsf{as}_{\mathsf{s}} := \mathsf{TAR}^* \to \mathsf{AS}$, and $\mathsf{as}_{\mathsf{t}} : \mathsf{TAR}^{\mathsf{ls}} \to \mathsf{AS}$ by:

$$\begin{aligned} \mathsf{as}_{\mathsf{h}}(\varepsilon_{\mathsf{h}}) &:= \mathsf{as}_{\mathsf{h}}^{\varepsilon} & \mathsf{as}_{\mathsf{h}}((t, X)\#_{\mathsf{h}}h) &:= \mathsf{as}_{\mathsf{h}}^{\#}(\mathsf{as}_{\mathsf{t}}(t), X, \mathsf{as}_{\mathsf{h}}(h)) \\ \mathsf{as}_{\mathsf{s}}(\varepsilon_{\mathsf{s}}) &:= \mathsf{as}_{\mathsf{s}}^{\varepsilon} & \mathsf{as}_{\mathsf{s}}(t\#_{\mathsf{s}}s) &:= \mathsf{as}_{\mathsf{s}}^{\#}(\mathsf{as}_{\mathsf{t}}(t), \mathsf{as}_{\mathsf{s}}(s)) \\ \mathsf{as}_{\mathsf{t}}(\tau) &:= \mathsf{as}_{\mathsf{t}}^{\tau} & \mathsf{as}_{\mathsf{t}}(\langle\rangle_X(s)t) &:= \mathsf{as}_{\mathsf{t}}^{\lozenge}(X, \mathsf{as}_{\mathsf{s}}(s), \mathsf{as}_{\mathsf{t}}(t)) \\ \mathsf{as}_{\mathsf{t}}(\langle_x t) &:= \mathsf{as}_{\mathsf{t}}^{\langle}(x, \mathsf{as}_{\mathsf{t}}(t)) & \mathsf{as}_{\mathsf{t}}(\rangle_x t) &:= \mathsf{as}_{\mathsf{t}}^{\rangle}(x, \mathsf{as}_{\mathsf{t}}(t)) \end{aligned}$$

for $h \in \mathsf{HAR}^{\mathsf{ls}}$, $t \in \mathsf{TAR}$, $s \in \mathsf{TAR}^*$, and $X \subseteq \mathcal{X}$. The right-hand sides of the above recursion equations are defined as follows:

$$\begin{aligned}
\mathsf{as}_\mathsf{h}^\varepsilon &:= \emptyset^6\\
\mathsf{as}_\mathsf{h}^\#(\sigma, X, (\sigma', X')) &:= (\sigma, X) \parallel (\sigma', X')\\
\mathsf{as}_\mathsf{s}^\varepsilon &:= \emptyset^5\\
\mathsf{as}_\mathsf{s}^\#(\sigma, \sigma') &:= \sigma \parallel \sigma'\\
\mathsf{as}_\mathsf{t}^\tau &:= \emptyset^5\\
\mathsf{as}_\mathsf{t}^\between(X, \sigma', (r, g_r, u, a, g_a)) &:= \sigma' \parallel (r, X \times r \cup g_r, X \cup u, a, g_a)\\
\mathsf{as}_\mathsf{t}^\langle(x, (r, g_r, u, a, g_a)) &:= \begin{cases} (r, g_r, u, \{x\} \cup a, \{x\} \times u \cup g_a) & \text{if } x \notin a\\ \bot & \text{otherwise}\end{cases}\\
\mathsf{as}_\mathsf{t}^\rangle(x, (r, g_r, u, a, g_a)) &:= (\{x\} \cup r, g_r, u, a, g_a)
\end{aligned}$$

The following lemma shows that the as-function computes the expected result:

Lemma 4.17. *For any lock-a/r-hedge $h \in \mathsf{HAR}^{\mathsf{ls}}$, we have:*

$$\mathsf{as}(h) \neq \bot \implies \mathsf{as}(h) = (\mathsf{rel}(h), \mathsf{E}^{\mathsf{rel}}(h), \mathsf{used}(h), \mathsf{acq}(h), \mathsf{E}^{\mathsf{acq}}(h), \mathsf{set}(h|_2)) \tag{1}$$

$$\mathsf{as}(h) = \bot \iff h \text{ violates (C1)} \tag{2}$$

Before we prove this lemma, we intuitively explain the recursion equations: The acquisition structure of a leaf is obviously $\emptyset^5$, as no locks have been released, used, or acquired so far. On an acquisition-node, $\mathsf{as}_\mathsf{t}^\langle$ first checks whether the lock has already been acquired. If this is the case, (C1) is violated and the result is undefined. Otherwise, the acquired lock is added to the acquisition-set a, and edges from the acquired lock to all locks u that are used after the acquisition are added to the acquisition-graph g_a. On a release-node, $\mathsf{as}_\mathsf{t}^\rangle$ just adds the released lock to the release-set r. On a use-node, first, as_s computes the parallel composition of the acquisition structures of the spawned threads. Then, $\mathsf{as}_\mathsf{t}^\between$ adds the set of used locks to the u-component of the local thread's acquisition structure. Moreover, edges from each used lock to each released lock are added to the release-graph. The functions $\mathsf{as}_\mathsf{s}^\varepsilon$ and $\mathsf{as}_\mathsf{s}^\#$ compute the parallel composition of the acquisition structures of a list of spawned threads, and the functions $\mathsf{as}_\mathsf{h}^\varepsilon$ and $\mathsf{as}_\mathsf{h}^\#$ compute the parallel composition of the hedge acquisition structures of a lock-a/r-hedge.

Proof of Lemma 4.17. Proposition 1 is shown by induction on the structure of h. We show for $h \in \mathsf{HAR}^{\mathsf{ls}}$ and $t \in \mathsf{TAR}$ and $t_1 \dots t_n \in \mathsf{TAR}^*$:

$$\begin{aligned}
\mathsf{as}_\mathsf{h}(h) \neq \bot &\implies \mathsf{as}_\mathsf{h}(h) = (\mathsf{rel}(h), \mathsf{E}^{\mathsf{rel}}(h), \mathsf{used}(h), \mathsf{acq}(h), \mathsf{E}^{\mathsf{acq}}(h), \mathsf{set}(h|_2))\\
\mathsf{as}_\mathsf{s}(t_1 \dots t_n) \neq \bot &\implies \mathsf{as}_\mathsf{s}(t_1 \dots t_n) = \mathsf{as}_\mathsf{t}(t_1) \parallel \dots \parallel \mathsf{as}_\mathsf{t}(t_n)\\
\mathsf{as}_\mathsf{t}(t) \neq \bot &\implies \mathsf{as}_\mathsf{t}(t) = (\mathsf{rel}(t), \mathsf{E}^{\mathsf{rel}}(t), \mathsf{used}(t), \mathsf{acq}(t), \mathsf{E}^{\mathsf{acq}}(t))
\end{aligned}$$

We demonstrate the cases $t = \langle_x t'$ and $t = \langle\rangle_X(t_1 \dots t_n)t'$. The other cases are analogous or straightforward. In the case $t = \langle_x t'$, we clearly have $\mathsf{rel}(t) = \mathsf{rel}(t')$, $\mathsf{E}^{\mathsf{rel}}(t) = \mathsf{E}^{\mathsf{rel}}(t')$, $\mathsf{used}(t) = \mathsf{used}(t')$, and $\mathsf{acq}(t) = \{x\} \cup \mathsf{acq}(t')$. Moreover, the dependence-graph contains an edge from the acquisition-node at the root of t to each use-node in t'. This corresponds to the edges $\{x\}\times\mathsf{used}(t')$ in the acquisition-graph. Thus, we have $\mathsf{E}^{\mathsf{acq}}(t) = \{x\}\times\mathsf{used}(t') \cup \mathsf{E}^{\mathsf{acq}}(t')$. Unfolding the definition of $\mathsf{as}_\mathsf{t}^{\langle}$ and applying the induction hypothesis completes the case.

In the case $t = \langle\rangle_X(t_1 \dots t_n)t'$, we first observe that, due to well-formedness of t, the trees $t_1 \dots t_n$ contain no release-nodes. Thus, all release-nodes in t are from t', i.e., $\mathsf{rel}(t) = \mathsf{rel}(t')$. Moreover, we obviously have:

$$\begin{aligned}
\mathsf{used}(t) &= X \cup \mathsf{used}(t_1) \cup \dots \cup \mathsf{used}(t_n) \cup \mathsf{used}(t')\\
\mathsf{acq}(t) &= \mathsf{acq}(t_1) \cup \dots \cup \mathsf{acq}(t_n) \cup \mathsf{acq}(t')\\
\mathsf{E}^{\mathsf{acq}}(t) &= \mathsf{E}^{\mathsf{acq}}(t_1) \cup \dots \cup \mathsf{E}^{\mathsf{acq}}(t_n) \cup \mathsf{E}^{\mathsf{acq}}(t')
\end{aligned}$$

The dependence-graph contains an edge from the $\langle\rangle_X$-node at the root of t to every release-node in t'. In the release-graph, this corresponds to the edges $X\times\mathsf{rel}(t')$, and we have $\mathsf{E}^{\mathsf{rel}}(t) = X\times\mathsf{rel}(t') \cup \mathsf{E}^{\mathsf{rel}}(t')$. Using the induction hypothesis for as_t and as_s, and unfolding the definition of $\|$, we observe that these are exactly the sets computed by $\mathsf{as}_\mathsf{t}(t)$.

For the $\Longleftarrow$-direction of Proposition 2, assume that h violates (C1). We have to show that $\mathsf{as}(h)$ is undefined. As h violates (C1), there are two final acquisitions of the same lock x in h. There are three cases how the two final acquisitions may be distributed in h. In the first case, there is a use-node $\langle\rangle_X(t_1 \dots t_n)t_{n+1}$, such that $x \in \mathsf{acq}(t_i) \cap \mathsf{acq}(t_j)$ for $1 \leq i < j \leq n + 1$. In the second case, there is an acquisition-node $\langle_x t'$ such that $x \in \mathsf{acq}(t')$, and in the third case, there are lock-a/r-trees $(t_1, X_1), (t_2, X_2)$ in h such that $x \in \mathsf{acq}(t_1) \cap \mathsf{acq}(t_2)$. The first case is handled by the $\|$-operator, which is only defined if the acquisition-sets of the operands are disjoint. In the first case, $\mathsf{as}_\mathsf{s}(t_1 \dots t_n)$ is the parallel composition of the acquisition structures of $t_1 \dots t_n$. Hence, if $j \leq n$, it is undefined. If $j = n+1$, x is contained in the acquisition component of $\mathsf{as}_\mathsf{s}(t_1 \dots t_n)$, and also in that of $\mathsf{as}_\mathsf{t}(t_{n+1})$. Thus, the parallel composition in $\mathsf{as}_\mathsf{t}^{\langle\rangle}(X, \mathsf{as}_\mathsf{s}(t_1 \dots t_n), \mathsf{as}_\mathsf{t}(t_{n+1}))$ is not defined. Analogously, in the third case, the function $\mathsf{as}_\mathsf{h}(h)$ computes the parallel composition of the (hedge) acquisition structures of the lock-a/r-trees. As the acquisition-sets are not disjoint, it is undefined. Finally, in the second case, $\mathsf{as}_\mathsf{t}^{\langle}(x, \mathsf{as}_\mathsf{t}(t'))$ is undefined, as the side condition $x \notin a$ is not satisfied.

For the $\Longrightarrow$-direction of Proposition 2, assume that $\mathsf{as}(h)$ is undefined. We have to show that h violates (C1). First, we spot a node in h where the function is defined for all successor nodes, but undefined for that node. As the function is always defined for leaf-nodes (i.e., $\mathsf{as}_\mathsf{h}(\varepsilon_\mathsf{h}) = \emptyset^6$, $\mathsf{as}_\mathsf{s}(\varepsilon_\mathsf{s}) = \mathsf{as}_\mathsf{t}(\tau) = \emptyset^5$), such a node exists. Analogously to the $\Longleftarrow$-direction, there are three cases for this node: It may be a $\langle\rangle_X$-node where the same lock is acquired in two subtrees, an $\langle_x t'$-node, where x is acquired in t', or an $(t, X)\#_\mathsf{s}h$-node, where the same lock is acquired in t and h. In any of these cases, there are two final acquisitions of the same lock in h, i.e., (C1) is violated. $\square$

In order to check (C2) and (C3), we define the set of *consistent* hedge acquisition structures

$$\mathsf{Cons} = \{(r, g_r, u, a, g_a, X) \mid (u \cup a) \setminus r \cap X = \emptyset \text{ and } g_r \text{ acyclic and } g_a \text{ acyclic}\}.$$

Finally, we prove the following theorem:

Theorem 4.18. *A lock-a/r-hedge $h \in \mathsf{HAR}^{\mathsf{ls}}$ is schedulable, if and only if its acquisition structure is consistent:*

$$\mathsf{sched}_{\mathsf{ar}}(h) \iff \mathsf{as}(h) \in \mathsf{Cons}$$

Proof. Lemma 4.14 states that schedulability of h is equivalent to h satisfying (C1)–(C3). From Lemma 4.16, we know that (C3) is equivalent to the acquisition and release-graph being acyclic. Moreover, from Lemma 4.17, we know that $\mathsf{as}(h)$ computes the sets of acquired, released, used, and initially held locks, as well as the acquisition- and release-graphs, and is undefined iff (C1) is violated. As the set Cons contains only defined hedge acquisition structures that satisfy (C2) and (C3), we get the proposition. $\square$

4.3.3.1 Constructing the Tree Automaton

From Theorem 4.18 and the inductive definition of $\mathsf{as}(h)$, it is straightforward to construct a bottom-up deterministic tree automaton that accepts exactly the schedulable lock-a/r-hedges:

Definition 4.19 (Tree Automaton for Schedulable Lock-A/R-Hedges)**.** *We define the automaton $\mathsf{A}_{\mathsf{Cons}} = (Q, F, \delta)$, where the states are hedge acquisition structures and acquisition structures, and the final states are the consistent hedge acquisition structures, i.e., $Q = \mathsf{AS} \,\dot\cup\, \mathsf{AS}_{\mathsf{h}}$ and $F = \mathsf{Cons}$. The automaton has the following rules δ:*

$$\begin{array}{ll}
\varepsilon_{\mathsf{h}} \rightarrow_\delta \mathsf{as}_{\mathsf{h}}^{\varepsilon} & (q_t, X)\#_{\mathsf{h}}q \rightarrow_\delta \mathsf{as}_{\mathsf{h}}^{\#}(q_t, X, q) \\
\varepsilon_{\mathsf{s}} \rightarrow_\delta \mathsf{as}_{\mathsf{s}}^{\#} & q_t\#_{\mathsf{s}}q_s \rightarrow_\delta \mathsf{as}_{\mathsf{s}}^{\#}(q_t, q_s) \\
\tau \rightarrow_\delta \mathsf{as}_{\mathsf{t}}^{\tau} & \langle\rangle_X(q_s)q_t \rightarrow_\delta \mathsf{as}_{\mathsf{t}}^{\langle\rangle}(X, q_s, q_t) \\
\langle_x q_t \rightarrow_\delta \mathsf{as}_{\mathsf{t}}^{\langle}(x, q_t) & \rangle_x q_t \rightarrow_\delta \mathsf{as}_{\mathsf{t}}^{\rangle}(x, q_t)
\end{array}$$

for all $q_s, q_t \in \mathsf{AS}$, $q \in \mathsf{AS}_{\mathsf{h}}$, $X \subseteq \mathcal{X}$, and $x \in \mathsf{locks}$, such that the function on the right-hand side of the rule is defined.

Lemma 4.20. *The automaton* $\mathsf{A}_{\mathsf{Cons}}$ *accepts exactly the lock-a/r-hedges with a consistent acquisition structure:*

$$\mathsf{L}(\mathsf{A}_{\mathsf{Cons}}) = \{h \in \mathsf{HAR}^{\mathsf{ls}} \mid \mathsf{as}(h) \in \mathsf{Cons}\}.$$

It can be constructed in time $2^{O(|\mathcal{X}|^2)}$*, and its size is* $|\mathsf{A}_{\mathsf{Cons}}| = 2^{O(|\mathcal{X}|^2)}$*.*

Proof. The first proposition is obvious from the definition of as and Theorem 4.18. Formally, it is shown by a straightforward induction over the structure of lock-a/r-hedges.

The alphabet of $\mathsf{A}_{\mathsf{Cons}}$ consists of the constructors for lock-a/r-hedges, these are

$$\{\tau, \varepsilon_{\mathsf{s}}, \varepsilon_{\mathsf{h}}, \#_{\mathsf{s}}\} \cup \{\#_{\mathsf{h}}(X) \mid X \subseteq \mathcal{X}\} \cup \{\langle_x \mid x \in \mathcal{X}\} \cup \{\rangle_x \mid x \in \mathcal{X}\} \cup \{\langle\rangle_X \mid X \subseteq \mathcal{X}\}.$$

Hence, the alphabet has $O(2^{|\mathcal{X}|})$ different symbols. Obviously, there are

$$(2^{3|\mathcal{X}|})(2^{2|\mathcal{X}|^2}) = 2^{O(|\mathcal{X}|^2)}$$

different acquisition structures, and

$$(2^{4|\mathcal{X}|})(2^{2|\mathcal{X}|^2}) = 2^{O(|\mathcal{X}|^2)}$$

different hedge acquisition structures. Thus, the automaton has $2^{O(|\mathcal{X}|^2)}$ different states, and we have

$$|\mathsf{A}_{\mathsf{Cons}}| = \mathsf{poly}(2^{|\mathcal{X}|} 2^{O(|\mathcal{X}|^2)}) = 2^{O(|\mathcal{X}|^2)}.$$

By instantiating the rule-templates for all possible values, the automaton can be constructed in time $2^{O(|\mathcal{X}|^2)}$. □

We summarize the results of this chapter by the following corollary:

Corollary 4.21. *A lock-execution-hedge is schedulable, if and only if its lock-a/r-hedge is accepted by* $\mathsf{A}_{\mathsf{Cons}}$*:*

$$\forall h \in \mathsf{H}^{\mathsf{ls}}.\ \mathsf{sched}_{\mathsf{ls}}(h) \neq \emptyset \iff h \in \mathsf{ar}^{-1}(\mathsf{L}(\mathsf{A}_{\mathsf{Cons}})).$$

Proof. Straightforward combination of the results of this chapter yields:

$$\begin{aligned}
&\mathsf{sched}_{\mathsf{ls}}(h) \neq \emptyset && \\
\iff &\mathsf{sched}_{\mathsf{ar}}(\mathsf{ar}(h)) && \text{by Theorem 4.5} \\
\iff &\mathsf{as}(\mathsf{ar}(h)) \in \mathsf{Cons} && \text{by Theorem 4.18} \\
\iff &\mathsf{ar}(h) \in \mathsf{L}(\mathsf{A}_{\mathsf{Cons}}) && \text{by Lemma 4.20} \\
\iff &h \in \mathsf{ar}^{-1}(\mathsf{L}(\mathsf{A}_{\mathsf{Cons}})) && \text{by definition of } \cdot^{-1}
\end{aligned}$$

□

4.4 Summary and Related Work

In this chapter, we introduced lock-a/r-hedges that summarize matching acquisitions and releases of lock-execution-hedges, and contain no reentrant operations on locks. In order to relate schedulability of lock-execution-hedges to schedulability of lock-a/r-hedges (Theorem 4.5), we introduced the concept of disciplined schedules, and showed that any schedule on a lock-execution-hedge can be reordered to a disciplined schedule. We developed the concept of disciplined schedules in [78], where we used it to prune executions that reach data-races, and to compute acquisition histories of those executions in an abstract interpretation [31, 32] framework. We described schedulability of lock-a/r-hedges by acyclicity of dependence-graphs, and checked this acyclicity in finite state by using acquisition structures (Theorem 4.18). Thus, we obtained a characterization of the schedulable lock-execution-hedges by a regular set of lock-a/r-hedges (Corollary 4.21).

In the remainder of this section, we first discuss the relation of our theory of movers, which we use to reorder schedules to disciplined schedules, to the original theory of Lipton [80]. Then, we discuss the development of acquisition structures, and, finally, we highlight the advantages of our modular approach, which uses lock-a/r-hedges as an intermediate model, over the direct approach that we pursue in [79].

Movers To reorder arbitrary schedules to disciplined schedules, we have developed a theory of movers for lock-sensitive schedules (cf. Subsection 4.2.1). This was inspired by the theory of movers presented by Lipton [80].

The programs in [80] have `parbegin...parend`-blocks for expressing parallel execution, a finite set of semaphores $a_1, \ldots, a_n$ for synchronization, and shared variables. While we regard abstracted programs where we essentially replace conditionals by nondeterministic choice, the programs in Lipton [80] are concrete programs.

Semaphores are a well-known concept to ensure mutual exclusion [33]. A semaphore is essentially a shared variable that holds a positive number. The operation $P(a_i)$ decrements the semaphore and is only executable if the semaphore is greater than zero. The operation $V(a_i)$ increments the semaphore. The operations P and V are executed atomically.

A lock can be modeled by a semaphore that is initialized with 1: The P-operation acquires the lock, and the V-operation releases the lock.

Lipton [80] establishes the notion of *left movers* and *right movers.* For a parallel program, a statement f is a right mover if for any execution $\alpha f h$ of the program, such that h and f are in different threads, also $\alpha h f$ is a valid execution that leads to the same configuration.

Symmetrically, a statement g is a left mover if for any execution $\alpha h g$, such that h and g are in different threads, also $\alpha g h$ is a valid execution that leads to the same configuration.

The main theorem concerning movers states that P-operations are always right movers, and V-operations are always left movers.

In Subsection 4.2.2, we adopted this theory to construct disciplined schedules from arbitrary schedules. Regarding movers, the main difference of the program models is that we have dynamic thread creation, such that a statement cannot move left beyond the creation of the thread executing this statement, although the thread-creation-statement and the first statement of the created thread are executed in different threads. Hence, we introduced the notion of unrelated statements: Two consecutive statements of an execution are unrelated if they are in different threads and the first one did not create the thread of the second one.

Moreover, for our purpose, we do not only need to move left release-statements, but also statements unrelated to locking, as well as usages of locks, i.e., statements that atomically execute a block of multiple statements between a matched acquisition and release. Due to our abstraction, statements unrelated to locking are independent of the other threads' configurations, and thus can always be moved left or right.

Moving left usages is not always possible: As an example, regard the execution $\rangle_x \, \langle_x\rangle_x$, i.e., a release of x followed by a usage of x, and assume that the release and usage are in different threads. Clearly, $\langle_x\rangle_x \, \rangle_x$ is not a valid execution, as the lock that shall be used is acquired by the other thread.

Thus, we classified statements (and sequences of statements executed atomically, like usages) as increasing and decreasing, and only allowed moving over increasing and decreasing statements. Our mover-lemma (Lemma 4.8) essentially states that one can swap an increasing statement followed by an unrelated decreasing statement. Hence, decreasing statements can be seen as constrained left movers: They can be moved left only over increasing statements. Symmetrically, increasing statements can be seen as constrained right movers: They can be moved right only over decreasing statements.

Acquisition Structures The concept of acquisition-graphs was invented by Kahlon et al. [57], called *acquisition histories* there. Release-graphs have first been used by Kahlon and Gupta [55], called *backwards acquisition histories* there. In [55, 57] and also [56], (backwards) acquisition histories are applied to analyze reachability properties of *parallel pushdown systems* with well-nested, non-reentrant locks (Lock-PPDS). A parallel pushdown system (PPDS) consists of multiple pushdown systems running in parallel. DPNs can be seen as a generalization of PPDS to additionally allow dynamic thread creation.

In [57], pairwise reachability properties are checked. This allows limiting the analysis to just two pushdown systems running in parallel (2PDS), and also makes the acquisition-graphs simpler, as one has to check only for cycles of length two. Also in the settings of [55] and [56], only cycles of length two need to be considered. In [78], we use acquisition structures to decide pairwise reachability properties in programs with reentrant monitors and dynamic thread creation. Although we consider dynamic thread creation there, the restriction to pairwise reachability properties allows us to only consider cycles of length two, such that we can use the acquisition histories of [57]. Finally, in [79], we consider lock-sensitive predecessor set computation for DPNs with well-nested, non-reentrant locks. Here, we introduce the concept of execution-trees, and compute acquisition structures over execution-trees. Moreover, as we consider arbitrary regular sets of configurations instead of pairwise reachability, we had to generalize the concept of (backwards) acquisition histories to check for arbitrary long cycles (of course, bounded by the number of locks).

When analyzing parallel pushdown systems with locks, the analysis can be implemented in a modular fashion, computing information for each of the PDSs separately, and combining the information at the end of the analysis. This avoids constructing the state-space of the whole system, thus bypassing the state-space explosion problem [57]. When regarding DPNs, the modularity of our method is not immediately obvious, as the distinct pushdown systems in a DPN are connected by dynamic thread creation. However, we also do not construct the state-space of the whole system, but combine the acquisition structures as late as possible, i.e., at the use-nodes or when combining the acquisition structures of the distinct trees of the lock-a/r-hedge.

Advantages of Lock-A/R-Hedges Our paper [79] is a direct predecessor of the work presented in this thesis. Instead of reentrant monitors, we study well-nested, non-reentrant locks there. Like in this thesis, we use acquisition structures to characterize lock-sensitive schedulability of lock-execution-hedges. And, as in this thesis, we reduce lock-sensitive reachability to lock-insensitive reachability, by characterizing the schedulable executions as a tree automaton, and building a cross-product of the program to be analyzed and that tree automaton.

An important difference between this thesis and [79] is the way how the characterization of schedulable lock-execution-hedges is obtained. In this thesis, we use lock-a/r-hedges as an abstraction of lock-execution-hedges. Lock-a/r-hedges contain just the necessary information to decide schedulability: Reentrant locking is already eliminated, and matched acquisitions and releases are summarized to use-nodes. On this level, schedulability can easily and intuitively be explained via acyclicity of the dependence-graph, and the acquisition- and release-graph is used as a tool to check acyclicity of the dependence-graph with a (finite state) tree automaton. To establish the connection between schedulability of lock-execution-hedges and lock-a/r-hedges, we reorder the schedule to form a disciplined schedule, and then perform reentrance elimination. In this thesis, these concepts are explained in a modular fashion, making the proofs simpler and reusable. For example, adapting our methods to well-nested, non-reentrant locking, as briefly discussed in Chapter 8, does not affect the characterization of schedulable lock-a/r-hedges.

In contrast, the approach taken in [79] directly characterizes the set of schedulable lock-execution-hedges by a tree automaton. The tree automaton computes acquisition structures bottom-up, doing the matching of acquisitions and releases by checking whether the acquired lock is released in the subtree that follows the acquisition. If this is the case, the lock is removed from the release-set and the release-graph, and is added to the use-set. The proof of correctness is done by induction over the lock-execution-hedge. It involves all of the steps described above: Reordering of schedules, checking acyclicity of dependence-graphs via acquisition- and release-graphs, and characterizing schedulability by acyclicity of dependence-graphs. This makes the proof quite involved. Indeed, for [79], the proof was first developed on paper, and then mechanically verified with the interactive theorem prover Isabelle/HOL [94]. During the verification, we discovered some cases that we had not considered in the first version of the proof, which required rather complicated arguments. The proof script can be found in [72].

5 Automata Constructions

In the last chapter, we have characterized the set of schedulable lock-a/r-hedges by the tree automaton $\mathsf{A}_{\mathsf{Cons}}$. In this chapter, we construct a *cross-product* DPN. Its execution-hedges correspond to those lock-execution-hedges of the Monitor-DPN whose lock-a/r-hedges are accepted by $\mathsf{A}_{\mathsf{Cons}}$—i.e., to the schedulable lock-execution-hedges of the Monitor-DPN. In the next chapter, we then use the cross-product DPN to reduce lock-sensitive predecessor set computation to lock-insensitive predecessor set computation.

The construction is done in two steps: In Section 5.1, we transform a tree automaton for lock-a/r-hedges into a *DPN-Acceptor* for lock-execution-hedges. In Section 5.2, we describe a cross-product construction between a Monitor-DPN and a DPN-Acceptor, yielding a new DPN. Its execution-hedges correspond to the lock-execution-hedges of the Monitor-DPN that are accepted by the DPN-Acceptor. Finally, in Section 5.3, we briefly summarize the results of this chapter and discuss related work.

5.1 A DPN-Acceptor for Schedulable Execution-Hedges

In Chapter 4, we have characterized the schedulable lock-a/r-hedges by the tree automaton $\mathsf{A}_{\mathsf{Cons}}$. In this section, we show how to translate a tree automaton for lock-a/r-hedges to a *DPN-Acceptor* for lock-execution-hedges.

5.1.1 DPN-Acceptors

Our goal is to derive a DPN that *accepts* exactly the schedulable lock-execution-hedges. For this, we have to define a notion of acceptance for DPNs. The intuition is to fix a set of initial and final configurations, and define all execution-hedges between these sets of configurations as accepted. In order to handle locks, we use Monitor-DPNs, i.e., we assume that the locks are bound to the stack. Moreover, for our special purpose, we assume that any non-bottom stack-symbol has a lock, and that non-release-rules of the DPN do not depend on the stack.

Definition 5.1 (DPN-Acceptor). *A* DPN-Acceptor $D = (\mathsf{A}_{C_\mathsf{I}}, \mathsf{A}_{C_\mathsf{F}}, M)$ *consists of an* initial automaton $\mathsf{A}_{C_\mathsf{I}}$*, a final automaton* $\mathsf{A}_{C_\mathsf{F}}$*, and a Monitor-DPN* $M = (P, \Gamma, \Gamma_\perp, \mathsf{Act}, \mathcal{X}, \Delta, \mathsf{locks})$*, such that all non-bottom stack-symbols of* M *are bound* *to locks (1), non-release-rules of the DPN do not depend on the stack (2,3), and* $\mathsf{A}_{C_\mathsf{I}}$ *accepts only valid configurations (4):*

$$\forall\gamma \in \Gamma \setminus \Gamma_\perp.\ \mathsf{locks}(\gamma) \neq \emptyset \tag{1}$$

$$p\gamma \overset{o}{\hookrightarrow} p'w\gamma' \in \Delta \implies \gamma' = \gamma \wedge \forall\gamma'' \in \Gamma.\ p\gamma'' \overset{o}{\hookrightarrow} p'w\gamma'' \in \Delta \tag{2}$$

$$p\gamma \overset{o}{\hookrightarrow} p_s w_s \sharp p'w\gamma' \in \Delta \implies \gamma' = \gamma \wedge \forall\gamma'' \in \Gamma.\ p\gamma'' \overset{o}{\hookrightarrow} p_s w_s \sharp p'w\gamma'' \in \Delta \tag{3}$$

$$\mathsf{L}(\mathsf{A}_{C_\mathsf{I}}) \subseteq \mathsf{valid} \tag{4}$$

For a state q of $\mathsf{A}_{C_\mathsf{I}}$*, the set $D(q)$ of lock-execution-hedges* accepted in state q *is defined as*

$$D(q) := \{h \times \mathsf{ls}(c) \mid \exists c'.\ c \in \mathsf{A}_{C_\mathsf{I}}(q) \wedge c \overset{h}{\Rightarrow}_M c' \wedge c' \in \mathsf{L}(\mathsf{A}_{C_\mathsf{F}})\},$$

and the language of D *is defined as the set of lock-execution-hedges between* $\mathsf{L}(\mathsf{A}_{C_\mathsf{I}})$ *and* $\mathsf{L}(\mathsf{A}_{C_\mathsf{F}})$*:*

$$\mathsf{L}(D) := \{h \times \mathsf{ls}(c) \mid \exists c'.\ c \in \mathsf{L}(\mathsf{A}_{C_\mathsf{I}}) \wedge c \overset{h}{\Rightarrow}_M c' \wedge c' \in \mathsf{L}(\mathsf{A}_{C_\mathsf{F}})\}.$$

The additional restrictions of the DPN of a DPN-Acceptor are crucial for the cross-product construction presented in Section 5.2. On the other hand, DPN-Acceptors are expressive enough to accept the set of schedulable lock-execution-hedges, as we show in the next subsection.

5.1.2 A DPN-Acceptor for Regular Sets of A/R-Hedges

Given a tree automaton A over lock-a/r-hedges, we show how to construct a DPN-Acceptor D_A such that $\mathsf{L}(D_A) = \mathsf{ar}^{-1}(\mathsf{L}(A))$, i.e., D_A accepts exactly those lock-execution-hedges whose corresponding lock-a/r-hedges are accepted by A. When setting $A := \mathsf{A}_{\mathsf{Cons}}$, the DPN-Acceptor $D_{\mathsf{A}_{\mathsf{Cons}}}$ accepts exactly the schedulable lock-execution-hedges.

In order to simulate an automaton over the corresponding lock-a/r-hedge, the DPN-Acceptor has to identify matching acquisitions and releases. Moreover, it must identify reentrant acquisitions and releases. For this purpose, the stack of a thread simulates the lockstack. The stack is updated on acquisition- and release-operations only. In order to distinguish final acquisitions from usages, upon an acquisition, the DPN-Acceptor decides nondeterministically whether this acquisition is final, or has a matched release-node. However, it has to ensure that the decision is correct, and, otherwise, the hedge must not be accepted. When deciding for a final acquisition, a flag in the control-state is set, indicating that no more initial releases must be accepted. Thus, if there should be a release-node after an acquisition-node that has been recognized as final, the DPN-Acceptor has no rule to accept this release-node.

When deciding for a usage, the DPN-Acceptor remembers in its control-state that it is currently inside a usage, and marks the first lock of the usage on the stack. The usage is left when the marked stack-symbol is popped by the matching release. Thus, if an acquisition is recognized as a usage, and there is no matching release, the control-state after accepting the whole hedge indicates that the DPN-Acceptor is still inside a usage, and waits for the matching release. The final automaton $\mathsf{A}_{C_\mathsf{F}}$ excludes configurations with such control-states. Moreover, the initial automaton $\mathsf{A}_{C_\mathsf{I}}$ ensures that all control-states are initially outside usages.

In order to identify reentrant acquisitions and releases, the DPN-Acceptor stores the set of currently acquired locks in its control-state, and flags locks on the stack as reentrant or non-reentrant. Using these flags, the set of currently acquired locks can be updated correctly on initial releases. Inside usages, only the set of used locks needs to be computed. For this purpose, it is not required to update the set of acquired locks inside usages, nor to flag the used locks as reentrant on the stack.

The initial automaton $\mathsf{A}_{C_\mathsf{I}}$ ensures that the flagging of locks as reentrant is done correctly on the initial lockstacks, and that the sets of acquired locks in the control-states are initialized correctly.

Finally, the DPN-Acceptor has to simulate the automaton A over lock-a/r-hedges. For this purpose, it maintains the state of A in its control-state. As the DPN-Acceptor identifies usages and reentrance, it has enough information to correctly update the state of A. In order to identify usages and reentrance, we have regarded the DPN-Acceptor from a top-down perspective. However, the tree automaton A over lock-a/r-hedges works bottom-up. Hence, we regard the DPN-Acceptor from a bottom-up perspective for the following explanation.

The final automaton $\mathsf{A}_{C_\mathsf{F}}$ ensures that all states of the automaton are initialized to states that may be accepted by leaf-rules. Outside usages, updating the states of the tree automaton is straightforward, as each rule of the DPN matches a node in the a/r-tree. On the last release of a usage, the control-state of the DPN switches to a usage-state. This state stores the old state of the automaton. Moreover, it keeps track of the set of used locks and the state of the automaton over the list of spawned threads. This information is updated while accepting the nodes inside the usage. The actual use-rule of the automaton is applied when accepting the first acquisition of the usage, using the stored old state of the automaton, the computed set of used locks, and the computed state for the list of spawned threads. Finally, the initial automaton $\mathsf{A}_{C_\mathsf{I}}$ computes the state of A over the hedge, and ensures that it is an initial state.

Following the ideas presented above, for an automaton A over lock-a/r-hedges, we define the DPN-Acceptor D_A, such that $\mathsf{L}(D_A) = \mathsf{ar}^{-1}(\mathsf{L}(A))$. In order to keep the presentation simpler, we additionally assume that A does not depend on use-nodes with empty lockset and no spawned threads ($\langle\rangle_\emptyset(\varepsilon)$). This assumption is obviously true for $\mathsf{A}_{\mathsf{Cons}}$.

Definition 5.2. *Let $A = (Q, F, \delta)$ be an automaton over lock-a/r-hedges, such that*

$$\langle\rangle_{\emptyset}(\varepsilon)q \to_{\delta}^{*} q' \iff q = q'.$$

We define the DPN-Acceptor $D_A = (\mathsf{A}_{C_\mathsf{I}}, \mathsf{A}_{C_\mathsf{F}}, M)$ with the Monitor-DPN $M = (P, \Gamma, \{\bot\}, \mathsf{Act}, \mathcal{X}, \Delta, \mathsf{locks})$, where

$$\begin{aligned} P &:= \{(\mathsf{p}^b, X, q) \mid b \in \mathbb{B}, X \subseteq \mathcal{X}, q \in Q\} \\ &\quad\ \dot\cup\ \{(\mathsf{u}^{b,\tilde q}, X, u, q) \mid b \in \mathbb{B},\ \tilde q, q \in Q,\ X, u \subseteq \mathcal{X}\} \\ \Gamma &:= \{x^b \mid x \in \mathcal{X}, b \in \mathbb{B}\} \,\dot\cup\, \{\bot\} \\ \mathsf{locks}(x^b) &:= \{x\} \qquad \mathsf{locks}(\bot) := \emptyset \end{aligned}$$

The rules Δ of the DPN are the following:

$$(\mathsf{p}^b, X, q)\gamma \overset{\Box_a}{\hookrightarrow} (\mathsf{p}^b, X, q)\gamma \qquad \text{(base)}$$

$$(\mathsf{p}^b, X, q)\gamma \overset{\Box_a}{\hookrightarrow} (\mathsf{p}^\bot, \emptyset, q_1)\bot\#(\mathsf{p}^b, X, q_2)\gamma \qquad \textit{if } \langle\rangle_{\emptyset}([q_1])q_2 \to_{\delta}^{*} q \qquad \text{(spawn)}$$

$$(\mathsf{p}^b, X, q)\gamma \overset{\langle_x}{\hookrightarrow} (\mathsf{p}^\bot, X, q)x^\top\gamma \qquad \textit{if } x \in X \qquad \text{(acq-r)}$$

$$(\mathsf{p}^b, X, q)\gamma \overset{\langle_x}{\hookrightarrow} (\mathsf{p}^\bot, \{x\} \cup X, q')x^\bot\gamma \qquad \textit{if } x \notin X \textit{ and } \langle_x q' \to_\delta q \qquad \text{(acq-n)}$$

$$(\mathsf{p}^\top, X, q)x^\top \overset{\rangle_x}{\hookrightarrow} (\mathsf{p}^\top, X, q) \qquad \text{(rel-r)}$$

$$(\mathsf{p}^\top, X, q)x^\bot \overset{\rangle_x}{\hookrightarrow} (\mathsf{p}^\top, X \setminus \{x\}, q') \qquad \textit{if } \rangle_x q' \to_\delta q \qquad \text{(rel-n)}$$

$$(\mathsf{u}^{b,\tilde q}, X, u, q)\gamma \overset{\Box_a}{\hookrightarrow} (\mathsf{u}^{b,\tilde q}, X, u, q)\gamma \qquad \text{(u-base)}$$

$$(\mathsf{u}^{b,\tilde q}, X, u, q)\gamma \overset{\Box_a}{\hookrightarrow} (\mathsf{p}^\bot, \emptyset, q_1)\bot\#(\mathsf{u}^{b,\tilde q}, X, u, q_2)\gamma \qquad \textit{if } q_1\#_{\mathsf{s}}q_2 \to_\delta q \qquad \text{(u-spawn)}$$

$$(\mathsf{u}^{b,\tilde q}, X, \{x\} \setminus X \cup u, q)\gamma \overset{\langle_x}{\hookrightarrow} (\mathsf{u}^{b,\tilde q}, X, u, q)x^\bot\gamma \qquad \text{(u-acq)}$$

$$(\mathsf{u}^{b,\tilde q}, X, u, q)x^\bot \overset{\rangle_x}{\hookrightarrow} (\mathsf{u}^{b,\tilde q}, X, u, q) \qquad \text{(u-rel)}$$

$$(\mathsf{p}^b, X, q)\gamma \overset{\langle_x}{\hookrightarrow} (\mathsf{u}^{b,\tilde q}, X, u, q')x^\top\gamma \qquad \textit{if } \langle\rangle_{\{x\}\setminus X \cup u}(q')\tilde q \to_\delta q \qquad \text{(u-begin)}$$

$$(\mathsf{u}^{b,\tilde q}, X, \emptyset, q)x^\top \overset{\rangle_x}{\hookrightarrow} (\mathsf{p}^b, X, \tilde q) \qquad \textit{if } \varepsilon_{\mathsf{s}} \to_\delta q \qquad \text{(u-end)}$$

for all $b \in \mathbb{B}$, $X, u \subseteq \mathcal{X}$, $x \in \mathcal{X}$, $a \in \mathsf{Act}$, $\gamma \in \Gamma$, and $q, q_1, q_2, q', \tilde q \in Q$.

The initial automaton is

$$\mathsf{A}_{C_\mathsf{I}} := (\{\mathsf{s}_0\}{\times}Q \,\dot\cup\, \{\mathsf{s}_1\}{\times}2^{\mathcal{X}}{\times}Q, \{\mathsf{s}_0\}{\times}F, \{(\mathsf{s}_0, q) \mid \varepsilon_{\mathsf{h}} \to_\delta q\}, \delta_{\mathsf{I}}).$$

Its transition relation δ_I *is the least solution of the following constraints:*

$$(\mathsf{s}_1, q, \emptyset) \xrightarrow{\perp}_{\delta_\mathsf{I}} (\mathsf{s}_0, q) \qquad \text{(stack-bot)}$$

$$(\mathsf{s}_1, q, \{x\} \cup X) \xrightarrow{x^b}_{\delta_\mathsf{I}} (\mathsf{s}_1, q, X) \qquad \textit{for } b = (x \in X) \qquad \text{(stack)}$$

$$(\mathsf{s}_0, q) \xrightarrow{(\mathsf{p}^\top, X, q_t)}_{\delta_\mathsf{I}} (\mathsf{s}_1, q', X) \qquad \textit{if } (q_t, X)\#_\mathsf{h} q' \to_\delta q \qquad \text{(ctrl)}$$

for all $q, q', q_t \in Q$, $X \subseteq \mathcal{X}$, *and* $x \in \mathcal{X}$.

The final automaton is $\mathsf{A}_{C_\mathsf{F}} := (\{s\}, \{s\}, \{s\}, \delta_\mathsf{F})$. *It has just the single state* s, *and its transition relation* δ_F *is the least solution of the following constraints:*

$$s \xrightarrow{(\mathsf{p}^b, X, q)}_{\delta_\mathsf{F}} s \qquad \textit{if } b \in \mathbb{B},\ X \subseteq \mathcal{X},\ \textit{and } \tau \to_\delta q$$

$$s \xrightarrow{\gamma}_{\delta_\mathsf{F}} s \qquad \textit{if } \gamma \in \Gamma$$

One easily verifies that the DPN M of the DPN-Acceptor is a Monitor-DPN (cf. Definition 3.11). Moreover, all non-bottom stack-symbols are associated with locks, non-release-rules do not depend on the stack, and we have $\mathsf{L}(\mathsf{A}_{C_\mathsf{I}}) \subseteq \mathsf{valid}$. Thus, D_A is a well-defined DPN-Acceptor.

The DPN implements the intuition sketched at the beginning of this subsection: Outside usages, it uses states of the form (p^b, X, q). The flag b indicates whether initial releases are allowed ($b = \top$) or not ($b = \perp$). The set X is the set of currently acquired locks, and q is the current state of the tree automaton A. The stack consists of the $\perp$-symbol and locks that are flagged with Boolean values. A stack entry $x^\perp$ stands for a non-reentrant lock, i.e., there is no other x on the stack below this entry. An entry $x^\top$ stands for a reentrant lock.

The (base)-rule accepts base-nodes in the execution-tree. As base-nodes correspond to $\langle\rangle_\emptyset(\varepsilon)$-nodes in the a/r-tree, it does not change the state of A. The (spawn)-rule accepts a spawn-node. The spawned thread is initialized with an empty set of currently acquired locks, and initial releases are not allowed ($p^\perp$). The spawn-node corresponds to a $\langle\rangle_\emptyset(t_s)$-node in the a/r-tree, and the state of A is updated accordingly. The (acq-r)-rule accepts a reentrant final acquisition. It does not modify the state of A, as the reentrant acquisition corresponds to a $\langle\rangle_\emptyset(\varepsilon)$-node in the lock-a/r-tree. The (acq-n)-rule accepts a non-reentrant final acquisition. A non-reentrant final acquisition corresponds to a $\langle_x$-node in the a/r-tree, and the state of A is updated accordingly. Both, (acq-r)- and (acq-n)-rules accept final acquisitions, and thus set the initial-release flag to $\perp$, allowing no more initial releases to be accepted. Analogously, the (rel-r)- and (rel-n)-rules accept initial releases. They can only be applied if the initial-release flag is still $\top$. Finally, the (u-begin)-rule switches to a use-state. The outermost lock of the usage is flagged with $\top$. The state of A is updated according to a use-rule, using the set of used locks and the state of A for the spawned threads that was computed bottom-up during processing the nodes of the usage. The (u-begin)-rule can be applied instead of an (acq-r)- or (acq-n)-rule, modeling the nondeterministic decision whether an acquisition is final or the beginning of a usage.

Inside usages, states of the form $(\mathsf{u}^{b,\tilde{q}}, X, u, q)$ are used. The tuple $(b, \tilde{q})$ contains the initial-release flag and the tree automaton's state after the usage, which are passed through unchanged. The set X is the set of currently acquired locks before the usage. This set is also passed through unchanged, and used to avoid adding reentrant locks to the set u of used locks. Finally, the state q keeps track of the automaton's state for the list of spawned threads. The (u-base)-rule does not change the state at all, the (u-spawn)-rule updates the state for the list of spawned threads according to a rule for $\#_\mathsf{s}$-nodes of A. The (u-acq)-rule pushes the acquired lock on the stack, flagged with $\perp$, indicating that this lock is an inner lock of a usage. Moreover, if the acquired lock is non-reentrant (w.r.t. the set X of locks held before the usage), it is added to the set of used locks. Note that, for an acquisition that reenters a lock of another non-reentrant acquisition inside the same usage, adding the acquired lock to the set u of used locks has no effect, as it is already contained in u. The (u-rel)-rule pops an inner lock of the usage. Finally, the (u-end)-rule initializes the used-set and the state of A for the list of spawned threads, as well as the initial-release flag b and the state $\tilde{q}$ of A after the usage.

The automaton $\mathsf{A}_{C_\mathsf{I}}$ uses states of the form (s_0, q) and (s_1, X, q). For the following explanation, assume that it reads a configuration backwards, starting at a final state, and accepting in an initial state. It first reads the bottommost stack-symbol of a thread-configuration, then the other stack-symbols, and finally the control-state. The (s_0, q)-states are used just before a new stack is read. The (stack-bot)-rule ensures that the bottommost symbol of this stack is the $\perp$-symbol. The (s_1, X, q)-states are used during reading the stack. The set X collects the set of locks on the stack, and is used by the (stack)-rule to ensure a correct reentrance marking. If the stack is completely read, the (ctrl)-rule reads the control-state, and ensures correct initialization. The q-component of the state keeps track of A's state for the hedge. It is passed through unchanged while accepting the stack, and only changed by the (ctrl)-rule. The final states of $\mathsf{A}_{C_\mathsf{I}}$ ensure that q is initialized to a state with which the empty hedge can be accepted, and the initial states ensure that q is a final state of the tree automaton A.

The automaton $\mathsf{A}_{C_\mathsf{F}}$ ensures that all control-states of the DPN are outside usages, and that the encoded states of the tree automaton A are states with which the empty tree can be accepted.

The correctness of the construction is stated by the following theorem:

Theorem 5.3. *Let $A = (Q, F, \delta)$ be an automaton over lock-a/r-hedges that does not depend on empty use-nodes:*

$$\langle\rangle_\emptyset(\varepsilon)q \rightarrow^*_\delta q' \iff q = q'.$$

Then, D_A accepts exactly the lock-execution-hedges whose corresponding lock-a/r-

hedges are accepted by A:

$$\mathsf{L}(D_A) = \mathsf{ar}^{-1}(\mathsf{L}(A)).$$

The size of D_A is polynomial in the size of A and the number of base-actions, and exponential in the size of locks:

$$|D_A| = \mathsf{poly}(|A||\mathsf{Act}|)2^{O(|\mathcal{X}|)}.$$

Moreover, D_A can be constructed in time $\mathsf{poly}(|A||\mathsf{Act}|)2^{O(|\mathcal{X}|)}$.

Proof. After unfolding the definitions of L, $\mathsf{A}_{C_\mathsf{I}}$, and ar^{-1}, we have to show for all lock-execution-hedges $h \in \mathsf{H}^{\mathsf{ls}}$:

$$\begin{aligned} &\exists q \in F, c \in \mathsf{A}_{C_\mathsf{I}}(\mathsf{s}_0, q), c' \in \mathsf{L}(\mathsf{A}_{C_\mathsf{F}}).\ c \overset{h|_1}{\Longrightarrow} c' \wedge \mathsf{ls}(c) = h|_2 \\ \Longleftrightarrow\ &\exists q \in F.\ \mathsf{ar}(h) \rightarrow_\delta^* q \end{aligned}$$

By induction on the structure of lock-execution-hedges we show the following more general statement: For all lock-execution-hedges $h \in \mathsf{H}^{\mathsf{ls}}$, lock-execution-trees $(t,\mu) \in \mathsf{T}^{\mathsf{ls}}$, and well-nested execution-trees $t_s \in \mathsf{T}^{\mathsf{wn}}$ with $\mu \overset{t_s}{\rightharpoonup} \varepsilon$, we have:

$$\exists c \in \mathsf{A}_{C_\mathsf{I}}(\mathsf{s}_0, q), c' \in \mathsf{L}(\mathsf{A}_{C_\mathsf{F}}).\ c \overset{h|_1}{\Longrightarrow} c' \wedge \mathsf{ls}(c) = h|_2 \iff \mathsf{ar}_\mathsf{h}(h) \rightarrow_\delta^* q \tag{1}$$

$$\begin{aligned} &\exists w \in \Gamma^*, c' \in \mathsf{L}(\mathsf{A}_{C_\mathsf{F}}).\ (\mathsf{p}^b, \mathsf{set}(\mu), q)w \overset{t}{\Rightarrow} c' \wedge w \sim \mu \\ \Longleftrightarrow\ &\mathsf{ar}_\mathsf{t}(t,\mu) \rightarrow_\delta^* q \wedge (b = \top \vee t \in \mathsf{T}^{\mathsf{nr}}) \end{aligned} \tag{2}$$

$$\begin{aligned} &\exists c' \in \mathsf{L}(\mathsf{A}_{C_\mathsf{F}}).\ (\mathsf{u}^{b,\tilde{q}}, X, u, q)\mu^\perp x^\top w \overset{t_s}{\Rightarrow} c'[\varphi] \\ \Longleftrightarrow\ &\exists u', u'' \subseteq \mathcal{X}, q' \in Q, s \in \mathsf{TAR}^*.\ \mathsf{ar}_\mathsf{s}(t_s, X) = (u'', s) \\ &\wedge \varphi = (\mathsf{u}^{b,\tilde{q}}, X, u', q')x^\top w \wedge u = u' \cup u'' \wedge s\#_\mathsf{s}q' \rightarrow_\delta^* q \end{aligned} \tag{3}$$

where $w \sim \mu$ means that w is a valid stack with correct reentrance marking and lockstack μ. It is defined in terms of the automaton $\mathsf{A}_{C_\mathsf{I}}$:

$$w \sim \mu \ :\Longleftrightarrow\ \forall \tilde{q}.\ (\mathsf{s}_1, \mathsf{set}(\mu), \tilde{q}) \xrightarrow{w}{}_{\delta_\mathsf{I}}^* (\mathsf{s}_0, \tilde{q}) \wedge \mathsf{ls}(w) = \mu.$$

Note that it is straightforward to show that $\mathsf{A}_{C_\mathsf{I}}$ correctly computes the lockset and does not change the state of the simulated tree automaton while accepting a stack. Thus, we have

$$(\mathsf{s}_1, X, \tilde{q}) \xrightarrow{w}{}_{\delta_\mathsf{I}}^* (\mathsf{s}_0, q') \implies q' = \tilde{q} \wedge X = \mathsf{set}(\mathsf{ls}(w)) \wedge w \sim \mathsf{ls}(w). \tag{*1}$$

Moreover, the notation $s\#_\mathsf{s}q' \rightarrow_\delta^* q$ means that the list[1] s of a/r-trees is accepted in state q, when starting in state q':

$$s\#_\mathsf{s}q' \rightarrow_\delta^* q \ :\Longleftrightarrow\ \begin{cases} e_1\#_\mathsf{s}\ldots\#_\mathsf{s}e_n\#_\mathsf{s}q' \rightarrow_\delta^* q & \text{if } s = e_1\#_\mathsf{s}\ldots\#_\mathsf{s}e_n\#_\mathsf{s}\varepsilon_\mathsf{s} \text{ for } n > 0 \\ q = q' & \text{if } s = \varepsilon_\mathsf{s} \end{cases}$$

[1] We identify lists of a/r-trees (TAR^*) and terms over $\#_\mathsf{s}$ and ε_s here.

Intuitively, Statement (1) means that D_A accepts a lock-execution-hedge h in state (s_0, q), if and only if A accepts the corresponding lock-a/r-hedge in state q. Statement (2) is the same statement for lock-execution-trees, generalized to capture the meaning of the $\mathsf{p}^\perp$-state that no initial releases are accepted any more. Finally, Statement (3) captures the behavior when processing a usage. It is generalized to also apply for the trees occurring inside a usage, where the locks from μ are released, and to handle arbitrary states q' with which the processing of the list of spawned threads starts and arbitrary sets u' of used locks at the end of a usage. Here, $\mu^\perp$ means that every element of μ is flagged with $\perp$, i.e., for $\mu = x_1 \dots x_n$, we define $\mu^\perp := x_1^\perp \dots x_n^\perp$. Note that, in Statement (3), the stack w that describes the stack below the usage is implicitly all-quantified over the whole statement.

We now demonstrate the cases of the induction:

Cases of Statement (1) In the case $h = \varepsilon$, we have $\mathsf{ar_h}(h) = \varepsilon_\mathsf{h}$, and get

$$\begin{aligned} &\exists c, c'.\ c \in \mathsf{A}_{C_\mathsf{I}}(\mathsf{s}_0, q) \wedge c' \in \mathsf{L}(\mathsf{A}_{C_\mathsf{F}}) \wedge c \overset{\varepsilon}{\Rightarrow} c' \wedge \mathsf{ls}(c) = \varepsilon \\ \iff\ & \varepsilon \in \mathsf{A}_{C_\mathsf{I}}(\mathsf{s}_0, q) && (1.1) \\ \iff\ & \varepsilon_\mathsf{h} \to_\delta q && (1.2) \end{aligned}$$

Equivalence (1.1) is due to $c \overset{\varepsilon}{\Rightarrow} c' \iff c = c' = \varepsilon$, $\varepsilon \in \mathsf{L}(\mathsf{A}_{C_\mathsf{F}})$, and $\mathsf{ls}(\varepsilon) = \varepsilon$. Equivalence (1.2) is due to the definition of the final states of $\mathsf{A}_{C_\mathsf{I}}$.

In the case $h = (t, \mu)h_2$, we have $\mathsf{ar_h}(h) = (\mathsf{ar_t}(t, \mu), \mathsf{set}(\mu))\mathsf{ar_h}(h_2)$. For the $\Longrightarrow$-direction, we assume

$$c \in \mathsf{A}_{C_\mathsf{I}}(\mathsf{s}_0, q) \wedge c' \in \mathsf{L}(\mathsf{A}_{C_\mathsf{F}}) \wedge c \xRightarrow{[t]h_2|_1} c' \wedge \mathsf{ls}(c) = [\mu]h_2|_2.$$

By definition of $\mathsf{A}_{C_\mathsf{F}}$, we obviously have

$$\forall c_1, c_2.\ c_1 c_2 \in \mathsf{L}(\mathsf{A}_{C_\mathsf{F}}) \iff c_1 \in \mathsf{L}(\mathsf{A}_{C_\mathsf{F}}) \wedge c_2 \in \mathsf{L}(\mathsf{A}_{C_\mathsf{F}}) \qquad (*2)$$

By unfolding the definition of $\xRightarrow{[t]h_2|_1}$, splitting the run of $\mathsf{A}_{C_\mathsf{I}}$ accordingly, and applying (*1) and (*2), we obtain $q_t, q_2 \in Q$, $w \in \Gamma^*$, $c_2 \in \mathsf{Conf}$, and $c_1', c_2' \in \mathsf{L}(\mathsf{A}_{C_\mathsf{F}})$ such that

$$\begin{aligned} & c = (\mathsf{p}^\top, \mathsf{set}(\mu), q_t)wc_2 \wedge c' = c_1'c_2' \wedge \mathsf{ls}(w) = \mu \wedge \mathsf{ls}(c_2) = h_2|_2 \\ \wedge\ & (\mathsf{p}^\top, \mathsf{set}(\mu), q_t)w \overset{t}{\Rightarrow} c_1' \wedge c_2 \xRightarrow{h_2|_1} c_2' \\ \wedge\ & (\mathsf{s}_0, q) \xrightarrow{(\mathsf{p}^\top, \mathsf{set}(\mu), q_t)}_{\delta_\mathsf{I}} (\mathsf{s}_1, \mathsf{set}(\mu), q_2) \xrightarrow{w}^*_{\delta_\mathsf{I}} (\mathsf{s}_0, q_2) \wedge c_2 \in \mathsf{A}_{C_\mathsf{I}}(\mathsf{s}_0, q_2) \wedge w \sim \mu. \end{aligned}$$

Applying the induction hypothesis to t and h_2 yields

$$\mathsf{ar_t}(t, \mu) \to_\delta^* q_t \wedge \mathsf{ar_h}(h_2) \to_\delta^* q_2.$$

Moreover, due to the definition of δ_l, we have $(q_t, \mathsf{set}(\mu))\#_\mathsf{h} q_2 \rightarrow_\delta q$, and together we get $\mathsf{ar}(h) \rightarrow_\delta^* q$.

For the $\Longleftarrow$-direction, we assume $\mathsf{ar_h}(h) \rightarrow_\delta^* q$. Analyzing the last rule applied by the run of the tree automaton, and using $\mathsf{ar_h}(h) = (\mathsf{ar_t}(t,\mu), \mathsf{set}(\mu))\#_\mathsf{h}\mathsf{ar_h}(h_2)$, we obtain $q_t, q_2 \in Q$, such that

$$\mathsf{ar_t}(t,\mu) \rightarrow_\delta^* q_t \wedge \mathsf{ar_h}(h_2) \rightarrow_\delta^* q_2 \wedge (q_t, \mathsf{set}(\mu))\#_\mathsf{h} q_2 \rightarrow_\delta q.$$

Applying the induction hypothesis, we obtain $w \in \Gamma^*$, $c_2 \in \mathsf{A}_{C_\mathsf{I}}(\mathsf{s}_0, q_2)$, and $c_1', c_2' \in \mathsf{L}(\mathsf{A}_{C_\mathsf{F}})$ with

$$(\mathsf{p}^\top, \mathsf{set}(\mu), q_t)w \overset{t}{\Rightarrow} c_1' \wedge w \sim \mu \wedge c_2 \overset{h_2|_1}{\Longrightarrow} c_2' \wedge \mathsf{ls}(c_2) = h_2|_2.$$

We set $c := (\mathsf{p}^\top, \mathsf{set}(\mu), q_t)wc_2$. Unfolding the definition of $\sim$, and using the (ctrl)-rule of $\mathsf{A}_{C_\mathsf{I}}$, we get $c \in \mathsf{A}_{C_\mathsf{I}}(\mathsf{s}_0, q)$. Moreover, we have $\mathsf{ls}(c) = h|_2$. With (*2), we get $c_1'c_2' \in \mathsf{L}(\mathsf{A}_{C_\mathsf{F}})$. Combining the executions of t and $h_2|_1$ yields $c \overset{h|_1}{\Longrightarrow} c_1'c_2'$, which completes the case.

Cases of Statement (2) In the case $t = \tau$, we have $\mathsf{ar_t}(t,\mu) = \tau$. Moreover, we have

$$\begin{aligned} &\exists w \in \Gamma^*, c' \in \mathsf{L}(\mathsf{A}_{C_\mathsf{F}}).\ (\mathsf{p}^b, \mathsf{set}(\mu), q)w \overset{\tau}{\Rightarrow} c' \wedge w \sim \mu \\ \Longleftrightarrow\ &\exists w.\ (\mathsf{p}^b, \mathsf{set}(\mu), q)w \in \mathsf{L}(\mathsf{A}_{C_\mathsf{F}}) \wedge w \sim \mu. \end{aligned}$$

For the $\Longrightarrow$-direction, we unfold the definition of $\mathsf{A}_{C_\mathsf{F}}$, and get $\tau \rightarrow_\delta q$, which implies the proposition.

For the $\Longleftarrow$-direction, we observe that it is straightforward to find a stack $w \in \Gamma^*$ with $w \sim \mu$, by simply adding the correct reentrance marking to μ, and appending a $\perp$-symbol. Moreover, due to $\tau \rightarrow_\delta q$, we have $(\mathsf{p}^b, \mathsf{set}(\mu), q)w \in \mathsf{L}(\mathsf{A}_{C_\mathsf{F}})$, which implies the proposition.

In the case $t = \Box_a t_1$, we have $\mathsf{ar_t}(t,\mu) = \langle\rangle_\emptyset(\varepsilon)\mathsf{ar_t}(t_1,\mu)$. As base-steps have no effect and are executable from all states (cf. (base)-constraint), we have $(\mathsf{p}^b, \mathsf{set}(\mu), q)w \overset{\Box_a t_1}{\Longrightarrow} c'$ iff $(\mathsf{p}^b, \mathsf{set}(\mu), q)w \overset{t_1}{\Rightarrow} c'$. Similar, we have $\Box_a t_1 \in \mathsf{T}^\mathsf{nr}$ iff $t_1 \in \mathsf{T}^\mathsf{nr}$. Moreover, as A does not depend on use-nodes, we have $\langle\rangle_\emptyset(\varepsilon)q \rightarrow_\delta q'$ iff $q = q'$. Applying the induction hypothesis completes the case.

In the case $t = \rhd_a(t_1)t_2$, we have $\mathsf{ar_t}(t,\mu) = \langle\rangle_\emptyset([\mathsf{ar_t}(t_1,\varepsilon)])\mathsf{ar_t}(t_2,\mu)$. For the $\Longrightarrow$-direction, we assume

$$c' \in \mathsf{L}(\mathsf{A}_{C_\mathsf{F}}) \wedge (\mathsf{p}^b, \mathsf{set}(\mu), q)w \overset{t}{\Rightarrow} c' \wedge w \sim \mu.$$

Analyzing the rules of D_A, the first step must have been derived by a (spawn)-rule. Using (*2), we obtain $q_1, q_2 \in Q$ and $c_1', c_2' \in \mathsf{L}(\mathsf{A}_{C_\mathsf{F}})$ such that:

$$c' = c_1'c_2' \wedge \langle\rangle_\emptyset([q_1])q_2 \rightarrow_\delta^* q \wedge (\mathsf{p}^\perp, \emptyset, q_1)\perp \overset{t_1}{\Rightarrow} c_1' \wedge (\mathsf{p}^b, \mathsf{set}(\mu), q_2)w \overset{t_2}{\Rightarrow} c_2'.$$

Obviously, we have $\bot \sim \varepsilon$. Thus, we can apply the induction hypothesis on both, t_1 and t_2, and get:

$$\mathsf{ar_t}(t_1, \varepsilon) \rightarrow_\delta^* q_1 \wedge t_1 \in \mathsf{T^{nr}} \wedge \mathsf{ar_t}(t_2, \mu) \rightarrow_\delta^* q_2 \wedge (b = \top \vee t_2 \in \mathsf{T^{nr}}).$$

We have $t \in \mathsf{T^{nr}}$ if and only if $t_2 \in \mathsf{T^{nr}}$. Moreover, using the rule $\langle\rangle_\emptyset([q_1])q_2 \rightarrow_\delta^* q$, we get $\mathsf{ar_t}(t, \mu) \rightarrow_\delta^* q$, which completes the $\Longrightarrow$-direction of the spawn-case.

For the $\Longleftarrow$-direction, we assume

$$\mathsf{ar_t}(t) \rightarrow_\delta^* q \wedge (b = \top \vee t \in \mathsf{T^{nr}}).$$

Analyzing the last rule applied by $\rightarrow_\delta^*$ and unfolding the definition of $\mathsf{ar_t}$, we obtain $q_1, q_2 \in Q$ such that

$$\langle\rangle_\emptyset([q_1])q_2 \rightarrow_\delta^* q \wedge \mathsf{ar_t}(t_1, \varepsilon) \rightarrow_\delta^* q_1 \wedge \mathsf{ar_t}(t_2, \mu) \rightarrow_\delta^* q_2.$$

As t is well-nested, we have $t_1 \in \mathsf{T^{nr}}$. Applying the induction hypothesis on both, t_1 and t_2, and using that $w \sim \varepsilon$ iff $w = [\bot]$, and that $t \in \mathsf{T^{nr}}$ iff $t_2 \in \mathsf{T^{nr}}$, we obtain $w \in \Gamma^*$ and $c_1', c_2' \in \mathsf{L}(\mathsf{A}_{C_\mathsf{F}})$ such that

$$(\mathsf{p}^\bot, \emptyset, q_1)\bot \overset{t_1}{\Rightarrow} c_1' \wedge (\mathsf{p}^b, \mathsf{set}(\mu), q_2)w \overset{t_2}{\Rightarrow} c_2' \wedge w \sim \mu.$$

Using the (spawn)-rule of D_A and (*2) completes the case.

In the case of a non-reentrant final acquisition (i.e., $t = \langle_x t_1$ with $t_1 \in \mathsf{T^{nr}}$ and $x \notin \mathsf{set}(\mu)$), we have $\mathsf{ar_t}(t, \mu) = \langle_x \mathsf{ar_t}(t_1, x\mu)$. For the $\Longrightarrow$-direction, we first have to show that the root-node of t is accepted by the (acq-n)-rule, and not by the (u-begin)-rule that also matches. This is done by contradiction. So assume the root-node is accepted by a (u-begin)-rule. Then, we obtain $\tilde{q}$, q', and u such that

$$(\mathsf{u}^{b,\tilde{q}}, \mathsf{set}(\mu), u, q')x^\top w \overset{t_1}{\Rightarrow} c'.$$

By a straightforward induction on t_1, using the case distinction of Lemma 4.2, we show that for all $w \in \Gamma^*$, $u \subseteq \mathcal{X}$, and $q \in Q$ we have

$$(\mathsf{u}^{b,\tilde{q}}, X, u, q)w \overset{t_1}{\Rightarrow} c' \wedge t_1 \in \mathsf{T^{nr}} \implies \exists u', q', w', c_1'.\ c' = c_1'[(\mathsf{u}^{b,\tilde{q}}, X, u', q')w'w].$$

Intuitively, the non-returning tree t_1 does not pop any symbol from the stack, and thus the (u-end)-rule that leaves the use-state is never applied. Hence, the thread is still in a use-state after executing t_1. In our case, we obtain u', q'', and c_1' such that $c' = c_1'[(\mathsf{u}^{b,\tilde{q}}, \mathsf{set}(\mu), u', q'')]$. However, this contradicts $c' \in \mathsf{L}(\mathsf{A}_{C_\mathsf{F}})$.

Thus, the root-node of t is accepted by an (acq-n)-rule, and we obtain $q' \in Q$ such that

$$\langle_x q' \rightarrow_\delta q \wedge (p^\bot, \mathsf{set}(x\mu), q')x^\bot w \overset{t_1}{\Rightarrow} c'.$$

By unfolding the definition of $\sim$, we also have $x^\bot w \sim x\mu$. Analogously to the above cases, we apply the induction hypothesis to t_1, and infer the proposition.

For the $\Longleftarrow$-direction, we assume $\langle_x \mathsf{ar_t}(t_1, x\mu) \rightarrow^*_\delta q$. Hence, we obtain $q' \in Q$ such that $\langle_x q' \rightarrow_\delta q$ and $\mathsf{ar_t}(t_1, x\mu) \rightarrow^*_\delta q'$. As we have $t_1 \in \mathsf{T^{nr}}$, we can apply the induction hypothesis and obtain $w \in \Gamma^*$ and $c' \in \mathsf{L}(\mathsf{A}_{C_\mathsf{F}})$ such that

$$(\mathsf{p}^\perp, \mathsf{set}(x\mu), q')w \overset{t_1}{\Rightarrow} c' \wedge w \sim x\mu.$$

By unfolding the definition of $\sim$, using $x \notin \mathsf{set}(\mu)$, and analyzing the rules of $\mathsf{A}_{C_\mathsf{I}}$, we obtain w' such that $w = x^\perp w'$ and $w' \sim \mu$. With the (acq-n)-rule we get $(\mathsf{p}^b, \mathsf{set}(\mu), q)w' \overset{t}{\Rightarrow} c'$ for all $b \in \mathbb{B}$, which completes the case.

Now assume that the root-node of t is a reentrant final acquisition or a reentrant initial release:

$$(t = \langle_x t_1 \wedge t_1 \in \mathsf{T^{nr}} \wedge x \in \mathsf{set}(\mu)) \vee (t = \rangle_x t_1 \wedge \mu = x\mu' \wedge x \in \mathsf{set}(\mu')).$$

We have $\mathsf{ar}(t, \mu) = \langle\rangle_\emptyset(\varepsilon)\tilde{t}_1$, where $\tilde{t}_1 = \mathsf{ar}(t_1, x\mu)$ or $\tilde{t}_1 = \mathsf{ar}(t_1, \mu')$, respectively. These cases are proved analogously to the case $t = \Box_a t_1$, and for a reentrant final acquisition, we show—analogously to the above case—that the root-node of t is accepted by the (acq-r)-rule rather than by the (u-begin)-rule.

In the case of a non-reentrant initial release (i.e., $t = \rangle_x t_1$, $\mu = x\mu'$, and $x \notin \mathsf{set}(\mu')$), we have $\mathsf{ar_t}(t, \mu) = \rangle_x \mathsf{ar_t}(t_1, \mu')$. The only matching rule of D_A is the (rel-n)-rule, and the proposition is shown analogously to the cases above.

Finally, in the case of a usage (i.e., $t = \langle_x t_1 \rangle_x t_2$ with $t_1 \in \mathsf{T^{sl}}$), we obtain $u \subseteq \mathcal{X}$ and $s \in \mathsf{TAR}^*$ with

$$\mathsf{ar_t}(t, \mu) = \langle\rangle_{\{x\}\setminus \mathsf{set}(\mu) \cup u}(s)\mathsf{ar_t}(t_2, \mu) \text{ and } \mathsf{ar_s}(t_1, \mathsf{set}(\mu)) = (u, s).$$

For the $\Longrightarrow$-direction, we assume

$$c' \in \mathsf{L}(\mathsf{A}_{C_\mathsf{F}}) \wedge (\mathsf{p}^b, \mathsf{set}(\mu), q)w \xRightarrow{\langle_x t_1 \rangle_x t_2} c' \wedge w \sim \mu.$$

We first show that the $\langle_x$-node is accepted by a (u-begin)-rule, and not by an (acq-r)- or (acq-n)-rule, which also match. The proof is, again, done by contradiction. So assume the $\langle_x$-node is accepted by an (acq-r)-rule (The proof for an (acq-n)-rule is analogous). Then we have

$$(\mathsf{p}^\perp, \mathsf{set}(x\mu), q)x^\top w \xRightarrow{t_1 \rangle_x t_2} c'.$$

It is straightforward to show that after execution of the same-level tree t_1, we are in a $\mathsf{p}^\perp$-state again, and the stack is $x^\top w$, i.e., we obtain c_1' and q' with

$$(\mathsf{p}^\perp, \mathsf{set}(x\mu), q)x^\top w \overset{t_1}{\Rightarrow} c_1'(\mathsf{p}^\perp, \mathsf{set}(x\mu), q')x^\top w \xRightarrow{\rangle_x t_2} c'.$$

However, there is no rule to accept the release-node $\rangle_x$ from a $\mathsf{p}^\perp$-state, in contradiction to the above execution. Thus, the root-node of t is accepted by a (u-begin)-rule, and we obtain $\tilde{q}, q' \in Q$ and $\tilde{u} \subseteq \mathcal{X}$ with

$$\langle\rangle_{\{x\}\setminus \mathsf{set}(\mu) \cup \tilde{u}}(q')\tilde{q} \rightarrow_\delta q \wedge (\mathsf{u}^{b,\tilde{q}}, \mathsf{set}(\mu), \tilde{u}, q')x^\top w \xRightarrow{t_1 \rangle_x t_2} c'.$$

Further splitting the execution of $t_1\rangle_x t_2$, and using (*2), we obtain $c'_1, c'_2 \in \mathsf{L}(\mathsf{A}_{C_\mathsf{F}})$ and $\varphi, \varphi' \in P\Gamma^*$, such that

$$c' = c'_1 c'_2 \wedge (\mathsf{u}^{b,\tilde{q}}, \mathsf{set}(\mu), \tilde{u}, q')x^\top w \overset{t_1}{\Rightarrow} c'_1\varphi \wedge \varphi \overset{\rangle_x}{\Rightarrow} \varphi' \wedge \varphi' \overset{t_2}{\Rightarrow} c'_2.$$

Applying Equivalence (3) of the induction hypothesis and using $\mathsf{ar}(t_s, \mathsf{set}(\mu)) = (u, s)$, we obtain u' and q'' such that

$$\varphi = (\mathsf{u}^{b,\tilde{q}}, \mathsf{set}(\mu), u', q'')x^\top w \wedge \tilde{u} = u' \cup u \wedge s\#_\mathsf{s} q'' \rightarrow^*_\delta q'.$$

Hence, the only applicable rule to accept the $\rangle_x$-node is the (u-end)-rule, and we have

$$u' = \emptyset \wedge \varepsilon_\mathsf{s} \rightarrow_\delta q'' \wedge \varphi' = (\mathsf{p}^b, \mathsf{set}(\mu), \tilde{q})w,$$

and thus $u = \tilde{u}$ and $s \rightarrow^*_\delta q'$. Moreover, we can apply the induction hypothesis for t_2, and obtain

$$\mathsf{ar}(t_2, \mu) \rightarrow^*_\delta \tilde{q} \wedge (b = \top \vee t_2 \in \mathsf{T}^{\mathsf{nr}}).$$

Together, we get $\mathsf{ar}(t, \mu) \rightarrow^*_\delta q$. Moreover, we have $t \in \mathsf{T}^{\mathsf{nr}}$ iff $t_2 \in \mathsf{T}^{\mathsf{nr}}$, which completes the $\Longrightarrow$-direction of the use-case.

For the $\Longleftarrow$-direction, we assume

$$\langle\rangle_{\{x\}\setminus\mathsf{set}(\mu)\cup u}(s)\mathsf{ar}_\mathsf{t}(t_2, \mu) \rightarrow^*_\delta q \wedge (b = \top \vee \langle_x t_1\rangle_x t_2 \in \mathsf{T}^{\mathsf{nr}}).$$

Hence, we obtain $q', \tilde{q} \in Q$ such that

$$\langle\rangle_{\{x\}\setminus\mathsf{set}(\mu)\cup u}(q')\tilde{q} \rightarrow_\delta q \wedge s \rightarrow^*_\delta q' \wedge \mathsf{ar}_\mathsf{t}(t_2, \mu) \rightarrow^*_\delta \tilde{q}.$$

Splitting the run $s \rightarrow^*_\delta q'$ after the first rule, we obtain q'' such that $\varepsilon \rightarrow_\delta q'' \wedge sq'' \rightarrow^*_\delta q'$. Applying the induction hypothesis on t_2 yields $w \in \Gamma^*$ and $c'_2 \in \mathsf{L}(\mathsf{A}_{C_\mathsf{F}})$ such that $(\mathsf{p}^b, \mathsf{set}(\mu), \tilde{q})w \overset{t_2}{\Rightarrow} c'_2$ and $w \sim \mu$. With the (u-end)-rule, we get

$$(\mathsf{u}^{b,\tilde{q}}, \mathsf{set}(\mu), \emptyset, q'')x^\top w \overset{\rangle_x}{\Rightarrow} (\mathsf{p}^b, \mathsf{set}(\mu), \tilde{q})w.$$

Applying Equivalence (3) of the induction hypothesis on t_1, instantiating $\mu = \varepsilon$ and $u' = \emptyset$, yields $c'_1 \in \mathsf{L}(\mathsf{A}_{C_\mathsf{F}})$ such that

$$(\mathsf{u}^{b,\tilde{q}}, \mathsf{set}(\mu), u, q')x^\top w \overset{t_1}{\Rightarrow} c'_1[(\mathsf{u}^{b,\tilde{q}}, \mathsf{set}(\mu), \emptyset, q'')x^\top w].$$

Finally, with the (u-begin)-rule, we get

$$(\mathsf{p}^b, \mathsf{set}(\mu), q)w \overset{\langle_x}{\Rightarrow} (\mathsf{u}^{b,\tilde{q}}, \mathsf{set}(\mu), u, q')x^\top w,$$

and with (*2), we have $c'_1 c'_2 \in \mathsf{L}(\mathsf{A}_{C_\mathsf{F}})$. Combining the executions derived above completes the case.

Cases of Statement (3) In the case $t_s = \tau$, we have $\mu = \varepsilon$ and $\mathsf{ar_s}(t_s, X) = (\emptyset, \varepsilon)$. For the $\Longrightarrow$-direction, we assume

$$(\mathsf{u}^{b,\tilde{q}}, X, u, q)x^\top w \overset{\tau}{\Rightarrow} c'\varphi \wedge c' \in \mathsf{L}(\mathsf{A}_{C_\mathsf{F}}).$$

Hence we have $c' = \varepsilon$ and $\varphi = (\mathsf{u}^{b,\tilde{q}}, X, u, q)x^\top w$. The statements $u = u \cup \emptyset$ and $\varepsilon_\mathsf{s} \#_\mathsf{s} q \to_\delta^* q$ are trivial, which completes the $\Longrightarrow$-direction.

For the $\Longleftarrow$-direction, we assume

$$\varphi = (\mathsf{u}^{b,\tilde{q}}, X, u', q')x^\top w \wedge u = \emptyset \cup u' \wedge \varepsilon \#_\mathsf{s} q \to_\delta q'.$$

Hence, we have $u = u'$ and $q = q'$. We choose $c' := \varepsilon$. By definition of $\mathsf{A}_{C_\mathsf{F}}$, we have $c' \in \mathsf{L}(\mathsf{A}_{C_\mathsf{F}})$, which completes the case.

In the case $t_s = \Box_a t_1$, we have $\mathsf{ar_s}(t_s, X) = \mathsf{ar_s}(t_1, X)$, and the proposition is proved straightforwardly by applying the induction hypothesis and using the definition of the (u-base)-rule that does not change the state or stack.

In the case $t_s = \rhd_a(t_1)t_2$, we have

$$\mathsf{ar_s}(t_s, X) = (u'', [\mathsf{ar_t}(t_1, \varepsilon)]s) \text{ with } \mathsf{ar_s}(t_2, X) = (u'', s).$$

For the $\Longrightarrow$-direction, we assume

$$(\mathsf{u}^{b,\tilde{q}}, X, u, q)\mu^\perp x^\top w \xRightarrow{\rhd_a(t_1)t_2} c'[\varphi] \wedge c' \in \mathsf{L}(\mathsf{A}_{C_\mathsf{F}}).$$

The $\rhd_a$-node was accepted by a (u-spawn)-rule, and with (*2) we obtain $c'_1, c'_2 \in \mathsf{L}(\mathsf{A}_{C_\mathsf{F}})$ and $q_1, q_2 \in Q$ such that

$$c' = c'_1 c'_2 \wedge q_1 \#_\mathsf{s} q_2 \to_\delta q \wedge (\mathsf{p}^\perp, \emptyset, q_1)\perp \overset{t_1}{\Rightarrow} c'_1 \wedge (\mathsf{u}^{b,\tilde{q}}, X, u, q_2)\mu^\perp x^\top w \overset{t_2}{\Rightarrow} c'_2[\varphi].$$

We have $\perp \sim \varepsilon$, and thus can apply Equivalence (2) of the induction hypothesis on t_1, and get $\mathsf{ar_t}(t_1, \varepsilon) \to_\delta^* q_1$. Applying Equivalence (3) of the induction hypothesis to t_2, we obtain $u \subseteq \mathcal{X}$ and $q' \in Q$ such that

$$\varphi = (\mathsf{u}^{b,\tilde{q}}, X, u', q')x^\top w \wedge u = u' \cup u'' \wedge s \#_\mathsf{s} q' \to_\delta^* q_2.$$

Together, we get $[\mathsf{ar_t}(t_1, \varepsilon)]sq' \to_\delta^* q$, which completes the $\Longrightarrow$-direction.

For the $\Longleftarrow$-direction, we assume

$$\varphi = (\mathsf{u}^{b,\tilde{q}}, X, u', q')\mu^\perp x^\top w \wedge [\mathsf{ar_t}(t_1, \varepsilon)]s \#_\mathsf{s} q' \to_\delta^* q \wedge u = u' \cup u''.$$

Hence, we obtain $q_1, q_2 \in Q$ such that

$$q_1 \#_\mathsf{s} q_2 \to_\delta q \wedge \mathsf{ar_t}(t_1, \varepsilon) \to_\delta^* q_1 \wedge s \#_\mathsf{s} q' \to_\delta^* q_2.$$

Due to well-nestedness, we have $t_1 \in \mathsf{T}^{\mathsf{nr}}$, and applying Equivalence (2) of the induction hypothesis on t_1 yields $w \in \Gamma^*$ and $c'_1 \in \mathsf{L}(\mathsf{A}_{C_\mathsf{F}})$ with

$$(\mathsf{p}^\perp, \emptyset, q_1) \overset{t_1}{\Rightarrow} c'_1 \wedge w \sim \varepsilon.$$

Applying Equivalence (3) of the induction hypothesis on t_2, we obtain $c'_2 \in \mathsf{L}(\mathsf{A}_{C_\mathsf{F}})$ such that

$$(\mathsf{u}^{b,\tilde{q}}, X, u, q_2)\mu^\perp x^\top w \overset{t_2}{\Rrightarrow} c'_2[\varphi].$$

With the (u-spawn)-rule, we get $(\mathsf{u}^{b,\tilde{q}}, X, u, q)\mu^\perp x^\top w \xRightarrow{\rhd_a(t_1)t_2} c'_1c'_2[\varphi]$, and due to (*2) we have $c'_1c'_2 \in \mathsf{L}(\mathsf{A}_{C_\mathsf{F}})$, which completes the case.

In the case $t_s = \langle_y t_1$, we have $\mathsf{ar_s}(t_s, X) = (\{y\} \setminus X \cup u'', s)$ where $\mathsf{ar_s}(t_1, X) = (u'', s)$. Here, we need the generalization that allowed arbitrary lockstacks μ above the outermost lock of the usage: By definition of $\rightharpoonup$, we have $y\mu \overset{t_1}{\rightharpoonup} \varepsilon$. This allows us to apply the induction hypothesis on t_1. The following calculation completes the case:

$$\begin{aligned}
&\exists c' \in \mathsf{L}(\mathsf{A}_{C_\mathsf{F}}).\ (\mathsf{u}^{b,\tilde{q}}, X, u, q)\mu^\perp x^\top w \xRightarrow{\langle_y t_1} c'[\varphi] \\
\iff &\exists u_1 \subseteq \mathcal{X}, c' \in \mathsf{L}(\mathsf{A}_{C_\mathsf{F}}).\ (\mathsf{u}^{b,\tilde{q}}, X, u_1, q)(y\mu)^\perp x^\top w \overset{t_1}{\Rrightarrow} c'[\varphi] \wedge u = \{y\} \setminus X \cup u_1 \quad (3.1) \\
\iff &\exists u', u'' \subseteq \mathcal{X}, q' \in Q.\ \varphi = (\mathsf{u}^{b,\tilde{q}}, X, u', q')x^\top w \wedge s\#_\mathsf{s}q' \rightarrow^*_\delta q \quad (3.2) \\
&\quad \wedge u_1 = u' \cup u'' \wedge u = u' \cup (\{y\} \setminus X \cup u'')
\end{aligned}$$

Here, Equivalence (3.1) is due to the fact that the only rule that matches the acquisition-node is (u-acq), and Equivalence (3.2) is due to the induction hypothesis.

Finally, in the case $t_s = \rangle_y t_1$ we have $\mathsf{ar_s}(\rangle_y t_1, X) = \mathsf{ar_s}(t_1, X)$, and the proof is done by straightforward application of the induction hypothesis, using that (u-rel) is the only rule to accept the $\rangle_x$-node.

Size Estimation For the size estimation, we treat the components of D_A separately. We set

$$D_A = (\mathsf{A}_{C_\mathsf{I}}, \mathsf{A}_{C_\mathsf{F}}, M) \text{ with } M = (P, \Gamma, \Gamma_\perp, \mathsf{Act}, \mathcal{X}, \Delta, \mathsf{locks}).$$

For the control-states and stack-symbols, we have:

$$\begin{aligned}
|P| &= 2|Q|2^{|\mathcal{X}|} + 2|Q|^2 2^{2|\mathcal{X}|} = \mathsf{poly}(|A|)2^{O(|\mathcal{X}|)} \\
|\Gamma| &= 2|\mathcal{X}| + 1 = O(|\mathcal{X}|)
\end{aligned}$$

The automaton $\mathsf{A}_{C_\mathsf{I}}$ is defined over the alphabet $P \cup \Gamma$, and we have

$$|P \cup \Gamma| = \mathsf{poly}(|A|)2^{O(|\mathcal{X}|)}.$$

Moreover, $\mathsf{A}_{C_\mathsf{I}}$ has

$$|Q| + 2^{|\mathcal{X}|}|Q| = \mathsf{poly}(|A|)2^{O(|\mathcal{X}|)}$$

states, hence its size is

$$|\mathsf{A}_{C_\mathsf{I}}| = \mathsf{poly}(\mathsf{poly}(|A|)2^{O(|\mathcal{X}|)}\mathsf{poly}(|A|)2^{O(|\mathcal{X}|)}) = \mathsf{poly}(|A|)2^{O(|\mathcal{X}|)}.$$

The automaton $\mathsf{A}_{C_\mathsf{F}}$ is also defined over the alphabet $P \cup \Gamma$, and has one state. Hence, its size is

$$|\mathsf{A}_{C_\mathsf{F}}| = \mathsf{poly}(\mathsf{poly}(|A|)2^{O(|\mathcal{X}|)}) = \mathsf{poly}(|A|)2^{O(|\mathcal{X}|)}.$$

Finally, the size of the Monitor-DPN M is

$$|M| = \mathsf{poly}(|P||\Gamma||\mathsf{Act}|) = \mathsf{poly}(|A||\mathsf{Act}|)2^{O(|\mathcal{X}|)},$$

and together we get

$$\begin{aligned}|D_A| &= |\mathsf{A}_{C_\mathsf{I}}| + |\mathsf{A}_{C_\mathsf{F}}| + |M| \\ &= \mathsf{poly}(|A|)2^{O(|\mathcal{X}|)} + \mathsf{poly}(|A|)2^{O(|\mathcal{X}|)} + \mathsf{poly}(|A||\mathsf{Act}|)2^{O(|\mathcal{X}|)} \\ &= \mathsf{poly}(|A||\mathsf{Act}|)2^{O(|\mathcal{X}|)}.\end{aligned}$$

□

Combining Corollary 4.21, which states that the schedulable lock-execution-hedges are characterized by $\mathsf{ar}^{-1}(\mathsf{L}(\mathsf{A}_{\mathsf{Cons}}))$, and Theorem 5.3, which states that $\mathsf{L}(D_A) = \mathsf{ar}^{-1}(\mathsf{L}(A))$ for a tree automaton A, we get the main result of this chapter:

Theorem 5.4. *The DPN-Acceptor $D_{\mathsf{A}_{\mathsf{Cons}}}$ is well-defined and accepts exactly the schedulable lock-execution-hedges. Its size is polynomial in the number of base-actions, and exponential in the number of locks:*

$$h \in \mathsf{L}(D_{\mathsf{A}_{\mathsf{Cons}}}) \iff \mathsf{sched}_{\mathsf{ls}}(h) \neq \emptyset \text{ and } |D_{\mathsf{A}_{\mathsf{Cons}}}| = \mathsf{poly}(|\mathsf{Act}|)2^{O(|\mathcal{X}|^2)}.$$

Proof. From the definition of $\mathsf{A}_{\mathsf{Cons}}$ (Definition 4.19), it is straightforward to show that $\mathsf{A}_{\mathsf{Cons}}$ does not depend on empty use-nodes. Thus, $D_{\mathsf{A}_{\mathsf{Cons}}}$ is well-defined, and the first part of the proposition follows from Corollary 4.21 and Theorem 5.3.

The size estimation follows from the size estimation for $\mathsf{A}_{\mathsf{Cons}}$ (cf. Lemma 4.20) and that for D_A (cf. Theorem 5.3). □

5.2 Cross-Product Construction

In the last section, we have constructed the DPN-Acceptor $D_{\mathsf{A}_{\mathsf{Cons}}}$ that accepted exactly the schedulable lock-execution-hedges. In this section, we combine $D_{\mathsf{A}_{\mathsf{Cons}}}$ with the Monitor-DPN to be analyzed. We do a general construction for arbitrary DPN-Acceptors: Given a Monitor-DPN M_1 and a DPN-Acceptor D_2, we construct a lock-insensitive *cross-product* DPN $M_\times$ and automata $\mathsf{A}_{C_{\mathsf{I}\times}}$ and $\mathsf{A}_{C_{\mathsf{F}\times}}$, such that the execution-hedges of $M_\times$ between $\mathsf{L}(\mathsf{A}_{C_{\mathsf{I}\times}})$ and $\mathsf{L}(\mathsf{A}_{C_{\mathsf{F}\times}})$ match exactly the lock-execution-hedges of M_1 that are accepted by D_2. Note that $(\mathsf{A}_{C_{\mathsf{I}\times}}, \mathsf{A}_{C_{\mathsf{F}\times}}, M_\times)$ can intuitively be seen as a DPN-Acceptor, however, it does not satisfy the additional restrictions that we imposed on DPN-Acceptors (i.e., that all stack-symbols are bound to locks and that non-release-rules are independent from the stack).

In the remainder of this section, we prove the following theorem:

Theorem 5.5. *Let M_1 be a Monitor-DPN and D_2 be a DPN-Acceptor with*

$$\begin{aligned} M_1 &= (P_1, \Gamma_1, \Gamma_{\perp,1}, \mathsf{Act}, \mathcal{X}, \Delta_1, \mathsf{locks}_1), \\ D_2 &= (\mathsf{A}_{C_\mathsf{I}}, \mathsf{A}_{C_\mathsf{F}}, M_2), \text{ and} \\ M_2 &= (P_2, \Gamma_2, \Gamma_{\perp,2}, \mathsf{Act}, \mathcal{X}, \Delta_2, \mathsf{locks}_2). \end{aligned}$$

Then, a (lock-insensitive) DPN $M_\times = (P_\times, \Gamma_\times, \mathsf{Act}_\mathcal{X}, \Delta_\times)$, automata $\mathsf{A}_{C_{\mathsf{I}\times}}$ and $\mathsf{A}_{C_{\mathsf{F}\times}}$, and a projection $\pi_1 : (P_\times \cup \Gamma_\times)^* \to (P_1 \cup \Gamma_1)^*$ can be constructed, such that, for all execution-hedges $h \in \mathsf{H}$ and sequences $c, c' \in (P_1 \cup \Gamma_1)^*$, we have

$$\begin{aligned} & c \in \mathsf{Conf}_1^{\mathsf{ls}} \cap \mathsf{valid}_1 \wedge c \overset{h}{\Rightarrow}_1 c' \wedge h \times \mathsf{ls}(c) \in \mathsf{L}(D_2) \\ \iff & \exists c_\times \in \mathsf{L}(\mathsf{A}_{C_{\mathsf{I}\times}}), c'_\times \in \mathsf{L}(\mathsf{A}_{C_{\mathsf{F}\times}}).\ c_\times \overset{h}{\Rightarrow}_\times c'_\times \wedge \pi_1(c_\times) = c \wedge \pi_1(c'_\times) = c'. \end{aligned}$$

The automaton $\mathsf{A}_{C_{\mathsf{I}\times}}$ can be constructed in time $\mathsf{poly}(|M_1||D_2|)2^{O(|\mathcal{X}|)}$, and $\mathsf{A}_{C_{\mathsf{F}\times}}$ and $M_\times$ can be constructed in time $\mathsf{poly}(|M_1||D_2|)$. Their sizes are

$$|\mathsf{A}_{C_{\mathsf{I}\times}}| = \mathsf{poly}(|M_1||D_2|)2^{O(|\mathcal{X}|)} \text{ and } |\mathsf{A}_{C_{\mathsf{F}\times}}|, |M_\times| = \mathsf{poly}(|M_1||D_2|).$$

Note that we sometimes write $\times, 1, 2$ instead of $M_\times$, M_1, M_2, e.g. $\Rightarrow_\times$ instead of $\Rightarrow_{M_\times}$.

The construction exploits that the stacks of the Monitor-DPN M_1 and the DPN-Acceptor's DPN M_2 are almost synchronized: On acquisition-actions, both DPNs push a stack-symbol, on release-actions, both DPNs pop a stack-symbol, and on spawn-actions, both DPNs preserve the height of the stack. Only on base-actions, M_1 may push or pop a stack-symbol, or preserve the height of the stack, while M_2 always preserves the height of the stack.

We use a cross-product construction, where the control-states of the new DPN are pairs of control-states from P_1 and P_2. The stack-symbols are either pairs of stack-symbols from Γ_1 and Γ_2, or just symbols from Γ_1. The pairs are pushed on acquisition-steps, the single symbols are pushed on base-steps. As non-release-steps of M_2 do not depend on the stack (by definition of DPN-Acceptors), the cross-product DPN does not need the topmost stack-symbol of M_2's stack in order to perform non-release-steps.

To account for the independence of the stack, we use the following abbreviations for rules from Δ_2:

$$\begin{aligned} p_2 \overset{o}{\hookrightarrow} p'_2 w_2 \in \Delta_2 &:= \exists \gamma_2.\ p_2\gamma_2 \overset{o}{\hookrightarrow} p'_2 w_2 \gamma_2 \in \Delta_2 && \text{for } w_2 \in \Gamma_2^* \\ p_2 \overset{o}{\hookrightarrow} \hat{p}_2\hat{\gamma}_2 \sharp p'_2 w_2 \in \Delta_2 &:= \exists \gamma_2.\ p_2\gamma_2 \overset{o}{\hookrightarrow} \hat{p}_2\hat{\gamma}_2 \sharp p'_2 w_2 \gamma_2 \in \Delta_2 && \text{for } w_2 \in \Gamma_2^* \end{aligned}$$

As sketched above, the control-states of the new DPN are $P_\times := P_1 \times P_2$, and the stack-symbols are

$$\begin{aligned}\Gamma_\times := &\{\gamma_1 \in \Gamma_1 \setminus \Gamma_{\perp,1} \mid \mathsf{locks}(\gamma_1) = \emptyset\}\\ &\cup \{(\gamma_1,\gamma_2) \in \Gamma_1 \times \Gamma_2 \mid \mathsf{locks}(\gamma_1) = \mathsf{locks}(\gamma_2) \neq \emptyset\}\\ &\cup \Gamma_{\perp,1} \times \Gamma_{\perp,2}.\end{aligned}$$

The projections π_1 and π_2 are defined by

$$\begin{array}{lll}\pi_1(p_1,p_2) = p_1 & \pi_1(\gamma_1) = \gamma_1 & \pi_1(\gamma_1,\gamma_2) = \gamma_1\\ \pi_2(p_1,p_2) = p_2 & \pi_2(\gamma_1) = \varepsilon & \pi_2(\gamma_1,\gamma_2) = \gamma_2\end{array}$$

for $p_1 \in P_1$, $p_2 \in P_2$, $\gamma_1 \in \Gamma_1$, and $\gamma_2 \in \Gamma_2$. Note that π_1 and π_2 are homomorphisms. They are extended to sequences and sets of sequences in the natural way. The rules $\Delta_\times$ of the new DPN are defined as the least solution of the following constraints:

$$(p_1,p_2)\gamma_\times \overset{\Box_a}{\hookrightarrow} (p_1',p_2')\gamma_\times' \in \Delta_\times \tag{base}$$
$$\text{if } p_1\pi_1(\gamma_\times) \overset{\Box_a}{\hookrightarrow} p_1'\pi_1(\gamma_\times') \in \Delta_1 \wedge p_2 \overset{\Box_a}{\hookrightarrow} p_2' \in \Delta_2 \wedge \pi_2(\gamma_\times) = \pi_2(\gamma_\times')$$

$$(p_1,p_2)\gamma_\times \overset{\Box_a}{\hookrightarrow} (\hat{p}_1,\hat{p}_2)(\hat{\gamma}_1,\hat{\gamma}_2)\#(p_1',p_2')\gamma_\times' \in \Delta_\times \tag{spawn}$$
$$\text{if } p_1\pi_1(\gamma_\times) \overset{\Box_a}{\hookrightarrow} \hat{p}_1\hat{\gamma}_1\#p_1'\pi_1(\gamma_\times') \in \Delta_1 \wedge p_2 \overset{\Box_a}{\hookrightarrow} \hat{p}_2\hat{\gamma}_2\#p_2' \in \Delta_2 \wedge \pi_2(\gamma_\times) = \pi_2(\gamma_\times')$$

$$(p_1,p_2)\gamma_\times \overset{\Box_a}{\hookrightarrow} (p_1',p_2')\hat{\gamma}_1\gamma_\times' \in \Delta_\times \tag{push}$$
$$\text{if } p_1\pi_1(\gamma_\times) \overset{\Box_a}{\hookrightarrow} p_1'\hat{\gamma}_1\pi_1(\gamma_\times') \in \Delta_1 \wedge p_2 \overset{\Box_a}{\hookrightarrow} p_2' \in \Delta_2 \wedge \pi_2(\gamma_\times) = \pi_2(\gamma_\times')$$

$$(p_1,p_2)\gamma_1 \overset{\Box_a}{\hookrightarrow} (p_1',p_2') \in \Delta_\times \tag{pop}$$
$$\text{if } p_1\gamma_1 \overset{\Box_a}{\hookrightarrow} p_1' \in \Delta_1 \wedge p_2 \overset{\Box_a}{\hookrightarrow} p_2' \in \Delta_2$$

$$(p_1,p_2)\gamma_\times \overset{\langle x}{\hookrightarrow} (p_1',p_2')(\hat{\gamma}_1,\hat{\gamma}_2)\gamma_\times' \in \Delta_\times \tag{acq}$$
$$\text{if } p_1\pi_1(\gamma_\times) \overset{\langle x}{\hookrightarrow} p_1'\hat{\gamma}_1\pi_1(\gamma_\times') \in \Delta_1 \wedge p_2 \overset{\langle x}{\hookrightarrow} p_2'\hat{\gamma}_2 \in \Delta_2 \wedge \pi_2(\gamma_\times) = \pi_2(\gamma_\times')$$

$$(p_1,p_2)(\gamma_1,\gamma_2) \overset{\rangle x}{\hookrightarrow} (p_1',p_2') \in \Delta_\times \tag{rel}$$
$$\text{if } p_1\gamma_1 \overset{\rangle x}{\hookrightarrow} p_1' \in \Delta_1 \wedge p_2\gamma_2 \overset{\rangle x}{\hookrightarrow} p_2' \in \Delta_2$$

where $p_1, \hat{p}_1, p_1' \in P_1$, $p_2, \hat{p}_2, p_2' \in P_2$, $\gamma_1, \hat{\gamma}_1 \in \Gamma_1$, $\gamma_2, \hat{\gamma}_2 \in \Gamma_2$, $\gamma_\times, \gamma_\times' \in \Gamma_\times$, $x \in \mathcal{X}$, and $a \in \mathsf{Act}$. This obviously defines a DPN, i.e., the sets $P_\times$ and $\Gamma_\times$ are finite.

Analogously to the notion of valid stacks for Monitor-DPNs, we define the notion of valid stacks for the cross-product DPN:

$$\Gamma_{\perp,\times} := \Gamma_{\perp,1} \times \Gamma_{\perp,2} \qquad \mathsf{valid}_\times := (\Gamma_\times \setminus \Gamma_{\perp,\times})^*\Gamma_{\perp,\times}.$$

We extend $\mathsf{valid}_\times$ to configurations in the obvious way. It is straightforward to show that validity is preserved by steps of $M_\times$, and that valid configurations of $M_\times$ are projected to valid configurations of M_1 and M_2, respectively:

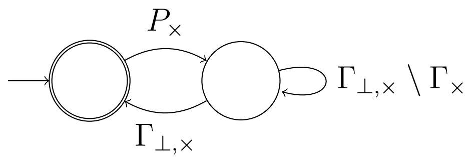
$$\begin{aligned} c_\times \in \mathsf{valid}_\times \wedge c_\times \rightarrow_\times c'_\times &\implies c'_\times \in \mathsf{valid}_\times \\ c_\times \in \mathsf{valid}_\times &\implies \pi_1(c_\times) \in \mathsf{valid}_1 \wedge \pi_2(c_\times) \in \mathsf{valid}_2 \end{aligned} \tag{*1}$$

The automaton $\mathsf{A}_{C_{\mathsf{I}\times}}$ checks that the configuration is valid, that the encoded lockstacks are consistent, and that the projection to the second component is an initial configuration of D_2, i.e., it is accepted by $\mathsf{A}_{C_\mathsf{I}}$. The automaton $\mathsf{A}_{C_{\mathsf{F}\times}}$ checks that the second component is a final configuration of D_2, i.e., it is accepted by $\mathsf{A}_{C_\mathsf{F}}$. We define

$$\mathsf{A}_{C_{\mathsf{I}\times}} := \mathsf{A}_{\mathsf{valid}_\times} \cap \pi_1^{-1}(\mathsf{A}_{\mathsf{Conf}_1^{\mathsf{ls}}}) \cap \pi_2^{-1}(\mathsf{A}_{C_\mathsf{I}}) \text{ and } \mathsf{A}_{C_{\mathsf{F}\times}} := \pi_2^{-1}(\mathsf{A}_{C_\mathsf{F}}),$$

where the automaton $\mathsf{A}_{\mathsf{valid}_\times}$ accepts the valid configurations of $M_\times$ and the automaton $\mathsf{A}_{\mathsf{Conf}_1^{\mathsf{ls}}}$ accepts the consistent configurations of M_1.

The automaton $\mathsf{A}_{\mathsf{valid}_\times}$ is obtained straightforwardly from the definition of $\mathsf{valid}_\times$. Its graphical representation is:



We obviously have $\mathsf{L}(\mathsf{A}_{\mathsf{valid}_\times}) = \mathsf{valid}_\times \cap \mathsf{Conf}_\times$.

The automaton $\mathsf{A}_{\mathsf{Conf}_1^{\mathsf{ls}}}$ is defined as

$$\mathsf{A}_{\mathsf{Conf}_1^{\mathsf{ls}}} := (P_1 \cup \Gamma_1, 2^{\mathcal{X}} \times 2^{\mathcal{X}}, \{(\emptyset,\emptyset)\}, (2^{\mathcal{X}}, 2^{\mathcal{X}}), \delta_{\mathsf{ls}}),$$

where the transition relation δ_{ls} is the least solution of the following constraints:

$$(X,Y) \xrightarrow{p}_{\delta_{\mathsf{ls}}} (\emptyset, X \cup Y) \qquad \text{for } p \in P_1 \tag{ctrl}$$

$$(X,Y) \xrightarrow{\gamma_1}_{\delta_{\mathsf{ls}}} (\mathsf{locks}_1(\gamma) \cup X, Y) \qquad \text{for } \gamma_1 \in \Gamma_1 \text{ if } \mathsf{locks}_1(\gamma_1) \cap Y = \emptyset \tag{stack}$$

A state (X,Y) of the automaton stores the set X of locks on the current lockstack, and the set Y of locks already seen on other stacks. When reading a control-symbol, the (ctrl)-rule joins the two sets. When reading a stack-symbol, the (stack)-rule updates the current lockset. It is only applicable if the lock associated to the stack-symbol is not in the set of locks already seen on other lockstacks. It is straightforward to show that

$$\mathsf{L}(\mathsf{A}_{\mathsf{Conf}_1^{\mathsf{ls}}}) \cap \mathsf{Conf}_1 = \mathsf{Conf}_1^{\mathsf{ls}}.$$

Together, we have

$$\mathsf{L}(\mathsf{A}_{C_{\mathsf{I}\times}}) = \mathsf{valid}_\times \cap \pi_1^{-1}(\mathsf{Conf}_1^{\mathsf{ls}}) \cap \pi_2^{-1}(\mathsf{L}(\mathsf{A}_{C_\mathsf{I}})) \text{ and } \mathsf{L}(\mathsf{A}_{C_{\mathsf{F}\times}}) = \pi_2^{-1}(\mathsf{L}(\mathsf{A}_{C_\mathsf{F}})).$$

Given valid configurations $c_1 \in \mathsf{valid}_1$ and $c_2 \in \mathsf{valid}_2$ with the same lockstacks, it is straightforward to construct a valid configuration $c_\times \in \mathsf{valid}_\times$, whose projections are the given configurations. Vice versa, it is obvious that the projections of a valid configuration $c_\times \in \mathsf{valid}_\times$ have the same lockstacks:

$$\begin{aligned} & c_1 \in \mathsf{valid}_1 \wedge c_2 \in \mathsf{valid}_2 \wedge \mathsf{ls}(c_1) = \mathsf{ls}(c_2) \\ \Longleftrightarrow\ & \exists c_\times \in \mathsf{valid}_\times.\ \pi_1(c_\times) = c_1 \wedge \pi_2(c_\times) = c_2 \end{aligned} \tag{*2}$$

Example 5.6. *We give an example how to construct a configuration $c_\times$ out of two given configurations $c_1 \in \mathsf{valid}_1$ and $c_2 \in \mathsf{valid}_2$. So assume we have $c_1 = p\gamma_1 \dots \gamma_8$ and $c_2 = \hat{p}\hat{\gamma}_1 \dots \hat{\gamma}_5$ with $\gamma_8 \in \Gamma_{\perp,1}$ and $\hat{\gamma}_5 \in \Gamma_{\perp,2}$. The* locks*-functions are:*

locks_1:	$\gamma_1 \mapsto x_1,$	$\gamma_2 \mapsto x_2,$	$\gamma_3 \mapsto \emptyset,$	$\gamma_4 \mapsto x_3,$	$\gamma_5 \mapsto \emptyset,$	$\gamma_6 \mapsto x_4,$	$\gamma_7, \gamma_8 \mapsto \emptyset$
locks_2:	$\hat{\gamma}_1 \mapsto x_1,$	$\hat{\gamma}_2 \mapsto x_2,$		$\hat{\gamma}_3 \mapsto x_3,$		$\hat{\gamma}_4 \mapsto x_4,$	$\hat{\gamma}_5 \mapsto \emptyset$

Then, both configurations are valid and have the same lockstacks. The corresponding configuration $c_\times$ is

$$c_\times = (p, \hat{p})(\gamma_1, \hat{\gamma}_1)(\gamma_2, \hat{\gamma}_2)\gamma_3(\gamma_4, \hat{\gamma}_3)\gamma_5(\gamma_6, \hat{\gamma}_4)\gamma_7(\gamma_8, \hat{\gamma}_5).$$

Finally, we show the following relation between the two component DPNs M_1, M_2 and the cross-product DPN $M_\times$:

$$\forall c_\times, c'_\times \in \mathsf{valid}_\times.\ \pi_1(c_\times) \overset{h}{\Rightarrow}_1 \pi_1(c'_\times) \wedge \pi_2(c_\times) \overset{h}{\Rightarrow}_2 \pi_2(c'_\times) \iff c_\times \overset{h}{\Rightarrow}_\times c'_\times. \tag{*3}$$

Proof. With Theorem 3.8, the proposition is shown by straightforward induction, using the following statement over single lock-actions $o \in \mathsf{Act}_\mathcal{X}$, thread-configurations $(p_1, p_2)w_\times, (p'_1, p'_2)w'_\times \in \mathsf{valid}_\times$, and configurations $c'_\times \in \mathsf{valid}_\times$:

$$\begin{aligned} & p_1\pi_1(w_\times) \xrightarrow{o}_1 \pi_1(c'_\times)p'_1\pi_1(w'_\times) \wedge p_2\pi_2(w_\times) \xrightarrow{o}_2 \pi_2(c'_\times)p'_2\pi_2(w'_\times) \\ \Longleftrightarrow\ & (p_1, p_2)w_\times \xrightarrow{o}_\times c'_\times(p'_1, p'_2)w'_\times. \end{aligned}$$

For the $\Longrightarrow$-direction, we assume

$$p_1\pi_1(w_\times) \xrightarrow{o}_1 \pi_1(c'_\times)p'_1\pi_1(w'_\times) \wedge p_2\pi_2(w_\times) \xrightarrow{o}_2 \pi_2(c'_\times)p'_2\pi_2(w'_\times).$$

We make a case distinction over the step $\xrightarrow{o}_1$. First, consider the case of a base-step. We have $o = \Box_a$ and $c'_\times = \varepsilon$. As π_1 projects each symbol to exactly one symbol, we obtain $\gamma_\times \in \Gamma_\times$ and $v_\times, r_\times \in \Gamma_\times^*$ with $w_\times = \gamma_\times r_\times$, $w'_\times = v_\times r_\times$, and $p_1, \pi_1(\gamma_\times) \overset{\Box_a}{\hookrightarrow} p'_1, \pi_1(v_\times) \in \Delta_1$. As non-release-rules of M_2 are independent of the stack, we have $p_2 \overset{\Box_a}{\hookrightarrow} p'_2$ and $\pi_2(w_\times) = \pi_2(w'_\times)$. We proceed with a case distinction over $v_\times$, i.e., we distinguish whether the step of M_1 was a base-, push- or pop-step. In case of a base-step (i.e., $v_\times = [\gamma'_\times]$ for some $\gamma'_\times \in \Gamma_\times$), we get $\pi_2(\gamma_\times) = \pi_2(\gamma'_\times)$. With the (base)-constraint, we get

$$(p_1, p_2)\gamma_\times \overset{\Box_a}{\hookrightarrow} (p'_1, p'_2)\gamma'_\times \in \Delta_\times,$$

and thus $(p_1, p_2)w_\times \xrightarrow{\Box_a}_\times (p'_1, p'_2)w'_\times$.

In case of a push-step (i.e., $v_\times = [\hat{\gamma}_\times, \gamma'_\times]$ for some $\hat{\gamma}_\times, \gamma'_\times \in \Gamma_\times$), we have $\mathsf{locks}(\pi_1(\hat{\gamma}_\times)) = \emptyset$ and $\pi_1(\hat{\gamma}_\times) \notin \Gamma_{\bot,1}$. Thus, we have $\hat{\gamma}_\times \in \Gamma_1$ which implies $\pi_2(\hat{\gamma}_\times) = \varepsilon$. With $\pi_2(w_\times) = \pi_2(w'_\times)$, we get $\pi_2(\gamma_\times) = \pi_2(\gamma'_\times)$. The (push)-constraint yields

$$(p_1, p_2)\gamma_\times \overset{\Box_a}{\hookrightarrow} (p'_1, p'_2)\hat{\gamma}_\times\gamma'_\times \in \Delta_\times,$$

and thus we have $(p_1, p_2)w_\times \xrightarrow{\Box_a}_\times (p'_1, p'_2)w'_\times$.

In case of a pop-step (i.e., $v_\times = \varepsilon$), we have $\mathsf{locks}(\pi_1(\gamma_\times)) = \emptyset$ and $\gamma_\times \notin \Gamma_{\bot,\times}$. Thus, we have $\gamma_\times \in \Gamma_1$, and the (pop)-constraint yields

$$(p_1, p_2)\gamma_\times \overset{\Box_a}{\hookrightarrow} (p'_1, p'_2) \in \Delta_\times,$$

and thus we have $(p_1, p_2)w_\times \xrightarrow{\Box_a}_\times (p'_1, p'_2)w'_\times$.

We continue the case distinction over the step $\xrightarrow{o}_1$: Consider the case of a spawn-step. We have $o = \Box_a$ and $c'_\times = (\hat{p}_1, \hat{p}_2)(\hat{\gamma}_1, \hat{\gamma}_2)$ with $(\hat{\gamma}_1, \hat{\gamma}_2) \in \Gamma_{\bot,\times}$. Analogously to the previous case, we obtain $\gamma_\times, \gamma'_\times \in \Gamma_\times$ and $r_\times \in \Gamma^*_\times$ with $\pi_2(\gamma_\times) = \pi_2(\gamma'_\times)$ and

$$\begin{aligned} &w_\times = \gamma_\times r_\times && w'_\times = \gamma'_\times r_\times \\ &p_1, \pi_1(\gamma_\times) \overset{\Box_a}{\hookrightarrow} \hat{p}_1\hat{\gamma}_1\sharp p'_1\pi_1(\gamma'_\times) \in \Delta_1 && p_2 \overset{\Box_a}{\hookrightarrow} \hat{p}_2\hat{\gamma}_2\sharp p'_2 \in \Delta_2. \end{aligned}$$

The (spawn)-constraint yields

$$(p_1, p_2)\gamma_\times \overset{\Box_a}{\hookrightarrow} (\hat{p}_1, \hat{p}_2)(\hat{\gamma}_1, \hat{\gamma}_2)\sharp(p'_1, p'_2)\gamma'_\times \in \Delta_\times,$$

and thus we have $(p_1, p_2)w_\times \xrightarrow{\Box_a}_\times c'_\times[(p'_1, p'_2)w'_\times]$.

In case of an acquisition-step, we have $o = \langle_x$ and $c'_\times = \varepsilon$. Analogously to the previous cases, we obtain $\gamma_\times, (\hat{\gamma}_1, \hat{\gamma}_2), \gamma'_\times \in \Gamma_\times$ and $r_\times \in \Gamma^*_\times$ with $\mathsf{locks}(\hat{\gamma}_1) = \mathsf{locks}(\hat{\gamma}_2) = \{x\}$, $\pi_2(\gamma_\times) = \pi_2(\gamma'_\times)$, and

$$\begin{aligned} &w_\times = \gamma_\times r_\times && w'_\times = (\hat{\gamma}_1, \hat{\gamma}_2)\gamma'_\times r_\times \\ &p_1, \pi_1(\gamma_\times) \overset{\langle_x}{\hookrightarrow} p'_1\hat{\gamma}_1\pi_1(\gamma'_\times) \in \Delta_1 && p_2 \overset{\langle_x}{\hookrightarrow} p'_2\hat{\gamma}_2 \in \Delta_2. \end{aligned}$$

With the (acq)-constraint, we get

$$(p_1, p_2)\gamma_\times \overset{\langle_x}{\hookrightarrow} (p'_1, p'_2)\hat{\gamma}_\times\gamma'_\times \in \Delta_\times,$$

and thus $(p_1, p_2)w_\times \xrightarrow{\langle_x}_\times (p'_1, p'_2)w'_\times$.

In case of a release-step, we have $o = \rangle_x$ and $c'_\times = \varepsilon$. We obtain $(\gamma_1, \gamma_2) \in \Gamma_\times$ with $\mathsf{locks}(\gamma_1) = \mathsf{locks}(\gamma_2) = \{x\}$, $w_\times = (\gamma_1, \gamma_2)w'_\times$, and

$$p_1\gamma_1 \overset{\rangle_x}{\hookrightarrow} p'_1 \in \Delta_1 \qquad\qquad p_2\gamma_2 \overset{\rangle_x}{\hookrightarrow} p'_2.$$

With the (rel)-constraint, we get

$$(p_1, p_2)\gamma_\times \overset{\rangle_x}{\hookrightarrow} (p'_1, p'_2) \in \Delta_\times,$$

and thus $(p_1, p_2)w_\times \xrightarrow{\rangle_x}_\times (p'_1, p'_2)w'_\times$.

The proof of the $\Longleftarrow$-direction is done symmetrically. $\square$

Now we are prepared to prove Theorem 5.5:

Proof of Theorem 5.5. The DPN $M_\times$, the automata $\mathsf{A}_{C_{\mathsf{I}\times}}$ and, $\mathsf{A}_{C_{\mathsf{F}\times}}$, and the projection π_1 are constructed as described above.

For the $\Longrightarrow$-direction, we assume

$$c_1 \in \mathsf{Conf}_1^{\mathsf{ls}} \cap \mathsf{valid}_1 \wedge c_1 \overset{h}{\Rightarrow}_1 c'_1 \wedge h{\times}\mathsf{ls}(c_1) \in \mathsf{L}(D_2).$$

By unfolding the definition of $\mathsf{L}(D_2)$, we obtain $c_2 \in \mathsf{L}(\mathsf{A}_{C_\mathsf{I}})$ and $c'_2 \in \mathsf{L}(\mathsf{A}_{C_\mathsf{F}})$ with

$$c_2 \overset{h}{\Rightarrow}_2 c'_2 \wedge \mathsf{ls}(c_1) = \mathsf{ls}(c_2).$$

As D_2 is a DPN-Acceptor, we have $\mathsf{L}(\mathsf{A}_{C_\mathsf{I}}) \subseteq \mathsf{valid}_2$, and thus $c_2 \in \mathsf{valid}_2$. As validity is preserved by executions, we also have $c'_1 \in \mathsf{valid}_1$ and $c'_2 \in \mathsf{valid}_2$. As the lockstacks of a final configuration only depends on the lockstacks of the initial configuration and on the execution-hedge, which are the same for both executions, we have $\mathsf{ls}(c'_1) = \mathsf{ls}(c'_2)$. With (*2), we construct valid configurations $c_\times, c'_\times \in \mathsf{valid}_\times$ with

$$c_1 = \pi_1(c_\times) \wedge c'_1 = \pi_1(c'_\times) \wedge c_2 = \pi_2(c_\times) \wedge c'_2 = \pi_2(c'_\times).$$

With (*3) we get $c_\times \overset{h}{\Rightarrow}_\times c'_\times$. From $c_1 \in \mathsf{Conf}_1^{\mathsf{ls}} \cap \mathsf{valid}_1$, we get $c_\times \in \pi_1^{-1}(\mathsf{Conf}_1^{\mathsf{ls}})$, and from $c_2 \in \mathsf{L}(\mathsf{A}_{C_\mathsf{I}})$, we get $c_\times \in \mathsf{L}(\pi_2^{-1}(\mathsf{A}_{C_\mathsf{I}}))$. Analogously, we get $c'_\times \in \mathsf{L}(\pi_2^{-1}(\mathsf{A}_{C_\mathsf{F}}))$. Due to $c_\times \in \mathsf{valid}_\times$, we have $c_\times \in \mathsf{A}_{\mathsf{valid}_\times}$, and by unfolding the definitions of $\mathsf{A}_{C_{\mathsf{I}\times}}$ and $\mathsf{A}_{C_{\mathsf{F}\times}}$, we get $c_\times \in \mathsf{L}(\mathsf{A}_{C_{\mathsf{I}\times}})$ and $c'_\times \in \mathsf{L}(\mathsf{A}_{C_{\mathsf{F}\times}})$, which completes the case.

For the $\Longleftarrow$-direction, we assume

$$c_\times \in \mathsf{L}(\mathsf{A}_{C_{\mathsf{I}\times}}) \wedge c'_\times \in \mathsf{L}(\mathsf{A}_{C_{\mathsf{F}\times}}) \wedge c_\times \overset{h}{\Rightarrow}_\times c'_\times.$$

By definition of $\mathsf{A}_{C_{\mathsf{I}\times}}$, we have $c_\times \in \mathsf{valid}_\times$, $\pi_1(c_\times) \in \mathsf{Conf}_1^{\mathsf{ls}}$, and $\pi_2(c_\times) \in \mathsf{L}(\mathsf{A}_{C_\mathsf{I}})$. By (*1) we also have $c'_\times \in \mathsf{valid}_\times$. Hence, by (*3), we get

$$\pi_1(c_\times) \overset{h}{\Rightarrow}_1 \pi_1(c'_\times) \wedge \pi_2(c_\times) \overset{h}{\Rightarrow}_2 \pi_2(c'_\times).$$

With (*2), we have $\pi_1(c_\times) \in \mathsf{valid}_1$, $\pi_2(c_\times) \in \mathsf{valid}_2$, and $\mathsf{ls}(\pi_1(c_\times)) = \mathsf{ls}(\pi_2(c_\times))$. By definition of $\mathsf{A}_{C_{\mathsf{F}\times}}$, we also have $\pi_2(c'_\times) \in \mathsf{L}(\mathsf{A}_{C_\mathsf{F}})$. Together, by folding the definition of $\mathsf{L}(D_2)$, we get $h \times \mathsf{ls}(\pi_1(c_\times)) \in \mathsf{L}(D_2)$, which completes the case. $\square$

5.3 Summary and Related Work

In this chapter, we have mapped tree automata on lock-a/r-hedges to DPN-Acceptors on lock-execution-hedges (Theorem 5.3). Using this construction, we obtained the DPN-Acceptor $D_{\mathsf{A}_{\mathsf{Cons}}}$ that accepts exactly the schedulable lock-execution-hedges (Theorem 5.4). Then, we constructed a cross-product between a Monitor-DPN and a DPN-Acceptor (Theorem 5.5). These results are used in the next chapter to reduce lock-sensitive to lock-insensitive predecessor set computation.

In [79], we construct a cross-product of a DPN and a *hedge automaton*. A hedge automaton in [79] is the stackless analogon to a DPN-Acceptor: A hedge automaton consists of an initial automaton and tree automata rules. An execution-hedge is accepted if its trees are accepted by a sequence of states that is accepted by the initial automaton. Note that we do not need a final automaton in [79]. The cross-product construction is simpler than the one presented here, as only one stack is involved. The more complicated construction in this thesis is required because the set of schedulable lock-execution-hedges is not regular for reentrant monitors, while it is regular for the well-nested, non-reentrant locks considered in [79].

The cross-product construction relies on the fact that the stacks of the Monitor-DPN and the DPN-Acceptor are updated nearly synchronously: On acquisition-operations, a stack-symbol is pushed on both stacks, and on release-operations, a stack-symbol is popped from both stacks. Alur and Madhusudan [2] introduce the concept of *visibly pushdown automata*, which they further develop in [1, 3, 4]. Intuitively, in a visibly pushdown automaton, the labels of the rules “make visible” the effect of the rule on the stack by indicating whether the rule pushes or pops a stack-symbol, or does not alter the height of the stack. Like regular languages, visibly pushdown languages are closed under union, intersection, concatenation, Kleene-$*$, and complement [2]. Our notion of Monitor-DPNs and DPN-Acceptors can be seen as a generalization of visibly pushdown automata to DPNs: For DPN-Acceptors, rules that push stack-symbols are labeled by acquisition-operations, rules that pop stack-symbols are labeled by release-operations, and rules that do not alter the height of the stack are labeled by base-operations. Thus, their effect on the stack is fully visible in the execution-tree. However, for Monitor-DPNs, push- and pop-rules are labeled by base-actions. Thus, their effect on the stack is not visible. For this reason, we further restricted DPN-Acceptors such that non-release-rules are independent from the stack. This restriction is strong enough that the cross-product construction works, and weak enough that reentrant monitors can still be modeled. It remains future research to explore whether the theory of visibly pushdown languages yields a more elegant alternative to our cross-product construction.

Kidd et al. [60] have taken a similar approach to handle reentrant monitors. Roughly speaking, they model programs with reentrant monitors as communicating pushdown systems [14], where they have one PDS per lock, and one PDS per thread. Using intersection of visibly pushdown automata, they replace the pushdown systems for locks by regular languages. This accelerates their model-checking procedure, as the number of PDSs is reduced. Moreover, it increases precision, as PDSs are over-approximated by their model-checker, while regular languages are handled precisely.

6 Lock-Sensitive Predecessor Sets

In the last chapter, we have characterized schedulable lock-execution-hedges by the DPN-Acceptor $D_{\mathsf{A_{Cons}}}$, and combined the Monitor-DPN to be analyzed with $D_{\mathsf{A_{Cons}}}$, yielding a cross-product DPN, whose lock-insensitive executions correspond to lock-sensitive executions of the Monitor-DPN. In this chapter, we use these results to reduce lock-sensitive predecessor set computation to lock-insensitive predecessor set computation on the cross-product DPN. The latter is then performed by the algorithm of Bouajjani et al. [16].

This chapter is organized as follows: In Section 6.1, we define predecessor and immediate predecessor sets. In Section 6.2, we recall the algorithm for lock-insensitive predecessor set computation of Bouajjani et al. [16], which we use in Section 6.3 to construct an algorithm for lock-sensitive predecessor set computation. In Section 6.4, we discuss applications of our algorithm to various analysis problems. Finally, in Section 6.5, we summarize the results of this chapter and discuss related work.

6.1 Definitions

In this section we define the predecessor and immediate predecessor set of a set of configurations. Intuitively, the predecessor set of C is the set of configurations from that a configuration in C can be reached. Analogously, the immediate predecessor set of C is the set of configurations from which a configuration in C can be reached in *exactly one* step.

Definition 6.1 (Predecessor Sets). *Given a DPN M and a set of configurations $C \subseteq \mathsf{Conf}$, we define:*

$$\begin{aligned}\mathsf{pre}_M(C) &:= \{c \in \mathsf{Conf} \mid \exists c' \in C.\ c \rightarrow_M c'\}\\ \mathsf{pre}^*_M(C) &:= \{c \in \mathsf{Conf} \mid \exists c' \in C.\ c \rightarrow^*_M c'\}\end{aligned}$$

When clear from the context, we omit the index M and write $\mathsf{pre}(C)$ *and* $\mathsf{pre}^*(C)$*.*

The set $\mathsf{pre}(C)$ *is called the* immediate predecessor set *of C, the set* $\mathsf{pre}^*(C)$ *is called the* predecessor set *of C.*

For Monitor-DPNs, we make a similar definition:

Definition 6.2 (Lock-Sensitive Predecessor Sets). *Given a Monitor-DPN M and a set of consistent and valid configurations $C \subseteq \mathsf{Conf}^{\mathsf{ls}} \cap \mathsf{valid}$, we define:*

$$\begin{aligned}\mathsf{pre}_{\mathsf{ls},M}(C) &:= \{c \in \mathsf{Conf}^{\mathsf{ls}} \cap \mathsf{valid} \mid \exists c' \in C.\ c \rightarrow_{\mathsf{ls},M} c'\}\\ \mathsf{pre}^*_{\mathsf{ls},M}(C) &:= \{c \in \mathsf{Conf}^{\mathsf{ls}} \cap \mathsf{valid} \mid \exists c' \in C.\ c \rightarrow^*_{\mathsf{ls},M} c'\}\end{aligned}$$

Again, we omit the index M when the DPN is clear from the context.

The set $\mathsf{pre}_{\mathsf{ls}}(C)$ is called the lock-sensitive immediate predecessor set *of C, the set $\mathsf{pre}^*_{\mathsf{ls}}(C)$ is called the* lock-sensitive predecessor set *of C.*

6.2 Lock-Insensitive Predecessor Set Computation

In the last section, we have defined predecessor and immediate predecessor sets. In this section, we sketch the lock-insensitive predecessor set computation by Bouajjani et al. [16], slightly adapted to our notation.

The predecessor set computation takes as input a DPN $M = (P, \Gamma, \Delta, \mathsf{Act})$ with $P \cap \Gamma = \emptyset$, and a finite automaton $A = (P \cup \Gamma, Q, I, F, \delta)$. The first step is to convert this automaton into an *M-automaton.* This is done for technical reasons, as it makes description of the predecessor set computation much simpler.

Definition 6.3. *An automaton $A = (P \cup \Gamma, Q, I, F, \delta)$ with epsilon-transitions is an M-automaton, iff*

1. *The states Q can be partitioned into the sets $Q = Q_c \mathbin{\dot{\cup}} Q_s$,*
2. *for every $q \in Q_c$ and $p \in P$, there is a unique and distinguished state $q_p \in Q_s$ and a transition $q \xrightarrow{p}_{\delta} q_p$.*
3. *any other transition in δ is either of the form*

$$\begin{aligned}&q \xrightarrow{\gamma}_{\delta} q' &&\text{for } q \in Q_s,\ \gamma \in \Gamma, \text{ and } q' \in Q_s \setminus \{q_p \mid q \in Q_c, p \in P\}, \text{ or}\\ &q \xrightarrow{\varepsilon}_{\delta} q' &&\text{for } q \in Q_s \text{ and } q' \in Q_c, \text{ and}\end{aligned}$$

4. *the initial states are in Q_c, i.e., $I \subseteq Q_c$.*

For any automaton A, an M-automaton A' with $\mathsf{L}(A') = \mathsf{L}(A) \cap \mathsf{Conf}$ can be constructed [16]. In [16], no procedure for this construction is specified. We sketch a possible construction here, with the goal of estimating the size of the resulting M-automaton.

1. First, we intersect the automaton with an automaton for valid configurations. As such an automaton has two states, this doubles the number of states. Then, we remove states from that no final state is reachable. This ensures that initial states have no outgoing Γ-transitions, nor there is an ε-path from an initial state to a state with an outgoing Γ-transition.

2. Next, we copy each state q of the automaton to obtain two copies q_1 and q_2. Both copies get the same incoming edges as q, but q_1 only gets outgoing ε-transitions and Γ-transitions, and q_2 only gets outgoing P-transitions. The q_2-states will become the states in Q_c. Then, we propagate the initial states over ε-transitions, i.e., every state that is reachable from an initial state via an ε-path becomes initial, too. Finally, we delete all initial states that are not q_2-states. As the automaton only accepts valid configurations, this does not change the language of the automaton.

3. Then, for each q_2-state, we create a state q_2', and a transition $q_2' \xrightarrow{\varepsilon} q_2$, and redirect all incoming transitions of q_2 to q_2'. This step ensures that states from Q_c have only incoming ε-transitions.

4. Next, by copying states, we ensure that each P-transition is the only incoming transition of its target-state. Then, we merge the target-states of P-transitions from the same source state that are labeled with the same control-symbol. This ensures that we can identify unique and distinguished q_p-states for each state with outgoing P-transitions.

5. Finally, we add missing q_p-states and p-transitions.

The first step restricts the language of A to valid configurations. The other steps do not change the language of the automaton. The first 3 steps blow up the number of states of the automaton only by a constant factor, the last two steps blow up the number of states by a factor of at most $|P|$. Thus, a conservative estimate for the size $|A'|$ of the resulting M-automaton is $|A'| = |A|\mathsf{poly}(|M|)$.

In the following, we assume that A is an M-automaton. We define the automaton $\mathsf{pre}^*_M(A) := (Q, I, F, \delta')$, where δ' is the least relation that contains δ and satisfies the following conditions:

$$q_p \xrightarrow{\gamma}_{\delta'} q' \qquad \text{if } p\gamma \overset{o}{\hookrightarrow} p_1w_1 \in \Delta \text{ and } q \xrightarrow{p_1w_1}{}^{*}_{\delta'} q' \tag{R1}$$

$$q_p \xrightarrow{\gamma}_{\delta'} q' \qquad \text{if } p\gamma \overset{o}{\hookrightarrow} p_1w_1\sharp p_2w_2 \in \Delta \text{ and } q \xrightarrow{p_1w_1p_2w_2}{}^{*}_{\delta'} q' \tag{R2}$$

The relation δ' can be computed by repeatedly adding transitions required by (R1) or (R2) to δ, until both conditions are satisfied. As no new states are added, this procedure terminates after polynomially many addition steps, and each addition step needs polynomial time.

This procedure is called *saturation*, and, accordingly, (R1) and (R2) are called *saturation-rules*. The intuition behind this procedure is the following: If there is a configuration of the form $c_1p'wrc_2$, and a rule $p\gamma \overset{o}{\hookrightarrow} p'w \in \Delta$, then the DPN admits the transition $c_1p\gamma rc_2 \xrightarrow{o}_M c_1p'wrc_2$. Hence, the configuration $c_1p\gamma rc_2$ is a predecessor of the configuration $c_1p'wrc_2$. This is exactly the intuition of the saturation-rule (R1). The rule (R2) works similar for spawn-steps.

We summarize this section by the following Theorem:

Theorem 6.4 (Predecessor Sets)**.** *For every DPN M and regular set of configurations $C \subseteq \mathsf{Conf}$, the sets $\mathsf{pre}^*_M(C)$ and $\mathsf{pre}_M(C)$ are regular. Given an automaton A, automata $\mathsf{pre}^*_M(A)$ and $\mathsf{pre}_M(A)$ with*

$$\mathsf{L}(\mathsf{pre}^*_M(A)) = \mathsf{pre}^*_M(\mathsf{L}(A) \cap \mathsf{Conf}) \text{ and } \mathsf{L}(\mathsf{pre}_M(A)) = \mathsf{pre}_M(\mathsf{L}(A) \cap \mathsf{Conf}).$$

can be effectively constructed in time $\mathsf{poly}(|M||A|)$.

The sizes of the automata can be estimated by

$$|\mathsf{pre}^*_M(A)| = |A|\mathsf{poly}(|M|) \text{ and } |\mathsf{pre}_M(A)| = |A|\mathsf{poly}(|M|).$$

Proof. The correctness of the saturation procedure is proved in the appendix of [17]. The modification of the algorithm to compute immediate predecessor sets is described in [16, 17].

For the size estimation, observe that the number of states does not increase during the saturation procedure. For the immediate predecessor algorithm, the number of states is doubled. $\square$

We conclude with a short note on implementing the saturation algorithm: One should use a representation of the M-automaton that does not explicitly store q_p-states that have no outgoing edges and are not final. In particular for DPNs with many control-states, like the cross-product DPNs used in this thesis, this optimization is essential.

6.3 Reduction to Lock-Insensitive Predecessor Set Computation

In Chapter 3, we have shown the correspondence between interleaving and tree-semantics (Theorem 3.8 with Corollary 3.9 for the lock-insensitive case, and Theorem 3.25 with Corollary 3.26 for the lock-sensitive case). In Chapter 4, we characterized the schedulable lock-execution-hedges by their lock-a/r-hedges, and in Chapter 5, we have shown how to combine this characterization with the Monitor-DPN to be analyzed, yielding a cross-product DPN whose lock-insensitive executions correspond to the lock-sensitive executions of the original Monitor-DPN (Theorems 5.4 and 5.5). In the last section, we have shown how to compute lock-insensitive predecessor sets. In this section, we apply these results to reduce lock-sensitive predecessor set computation on the original Monitor-DPN to lock-insensitive predecessor set computation on the cross-product DPN.

Given a Monitor-DPN M, and configurations $c, c' \in \mathsf{Conf}$, we have:

$$
\begin{aligned}
& c \rightarrow^*_{\mathsf{ls},M} c' \wedge c \in \mathsf{Conf}^{\mathsf{ls}} \cap \mathsf{valid} \\
\iff & \exists h \in \mathsf{H}.\ c \overset{h}{\Rightarrow}_M c' \wedge \mathsf{sched}_{\mathsf{ls}}(h \times \mathsf{ls}(c)) \neq \emptyset \wedge c \in \mathsf{Conf}^{\mathsf{ls}} \cap \mathsf{valid} && \text{(Cor. 3.26)} \\
\iff & \exists h \in \mathsf{H}.\ c \overset{h}{\Rightarrow}_M c' \wedge (h \times \mathsf{ls}(c)) \in \mathsf{L}(D_{\mathsf{A}_{\mathsf{Cons}}}) \wedge c \in \mathsf{Conf}^{\mathsf{ls}} \cap \mathsf{valid} && \text{(Thm. 5.4)} \\
\iff & \exists h \in \mathsf{H}, c_\times \in \mathsf{L}(\mathsf{A}_{C_{\mathsf{I}\times}}), c'_\times \in \mathsf{L}(\mathsf{A}_{C_{\mathsf{F}\times}}).\ c_\times \overset{h}{\Rightarrow}_\times c'_\times \wedge \pi_1(c_\times) = c \wedge \pi_1(c'_\times) = c' \\
& && \text{(Thm. 5.5)} \\
\iff & \exists c_\times \in \mathsf{L}(\mathsf{A}_{C_{\mathsf{I}\times}}), c'_\times \in \mathsf{L}(\mathsf{A}_{C_{\mathsf{F}\times}}).\ c_\times \rightarrow^*_\times c'_\times \wedge \pi_1(c_\times) = c \wedge \pi_1(c'_\times) = c' \\
& && \text{(Cor. 3.9)}
\end{aligned}
$$

Here, $(\mathsf{A}_{C_{\mathsf{I}\times}}, \mathsf{A}_{C_{\mathsf{F}\times}}, M_\times)$ is the cross-product of M and $D_{\mathsf{A}_{\mathsf{Cons}}}$. Now, let $C \subseteq \mathsf{Conf}$ be a set of configurations of M. We do the following calculation:

$$
\begin{aligned}
& \mathsf{pre}^*_{\mathsf{ls},M}(C \cap \mathsf{Conf}^{\mathsf{ls}} \cap \mathsf{valid}) \\
&= \{c \mid c \in \mathsf{Conf}^{\mathsf{ls}} \cap \mathsf{valid} \wedge \exists c' \in C \cap \mathsf{Conf}^{\mathsf{ls}} \cap \mathsf{valid}.\ c \rightarrow^*_{\mathsf{ls},M} c'\} && (1) \\
&= \{c \mid c \in \mathsf{Conf}^{\mathsf{ls}} \cap \mathsf{valid} \wedge \exists c' \in C.\ c \rightarrow^*_{\mathsf{ls},M} c'\} && (2) \\
&= \{\pi_1(c_\times) \mid c_\times \in \mathsf{L}(\mathsf{A}_{C_{\mathsf{I}\times}}) \wedge \exists c'_\times \in \mathsf{L}(\mathsf{A}_{C_{\mathsf{F}\times}}).\ \pi_1(c'_\times) \in C \wedge c_\times \rightarrow^*_\times c'_\times\} && (3) \\
&= \pi_1(\mathsf{pre}^*_\times(\pi_1^{-1}(C) \cap \mathsf{L}(\mathsf{A}_{C_{\mathsf{F}\times}})) \cap \mathsf{L}(\mathsf{A}_{C_{\mathsf{I}\times}})) && (4)
\end{aligned}
$$

Here, Equation (1) is due to unfolding the definition of $\mathsf{pre}^*_{\mathsf{ls}}$ (Definition 6.2). Equation (2) is due to the fact that consistency and validity of configurations is preserved by the lock-sensitive interleaving semantics. Equation (3) uses the calculation from above, and Equation (4) is by folding the definition of pre^* (Definition 6.1). Summarized, we have

$$\mathsf{pre}^*_{\mathsf{ls},M}(C \cap \mathsf{Conf}^{\mathsf{ls}} \cap \mathsf{valid}) = \pi_1(\mathsf{pre}^*_\times(\pi_1^{-1}(C) \cap \mathsf{L}(\mathsf{A}_{C_{\mathsf{F}\times}})) \cap \mathsf{L}(\mathsf{A}_{C_{\mathsf{I}\times}})).$$

Using the lock-insensitive predecessor set computation, and standard operations on automata, an automaton for the right-hand side of this equation can be effectively computed.

Together, we get the main theorem of this thesis that generalizes Theorem 6.4 to Monitor-DPNs:

Theorem 6.5 (Computing Lock-Sensitive Predecessor Sets for Monitor-DPNs). *Given a Monitor-DPN M and a regular set of valid, consistent configurations $C \subseteq \mathsf{Conf}^{\mathsf{ls}} \cap \mathsf{valid}$, the sets $\mathsf{pre}^*_{\mathsf{ls},M}(C)$ and $\mathsf{pre}_{\mathsf{ls},M}(C)$ are regular. Given an automaton A, automata $\mathsf{pre}^*_{\mathsf{ls},M}(A)$ and $\mathsf{pre}_{\mathsf{ls},M}(A)$ with*

$$
\begin{aligned}
\mathsf{L}(\mathsf{pre}^*_{\mathsf{ls},M}(A)) &= \mathsf{pre}^*_{\mathsf{ls},M}(\mathsf{L}(A) \cap \mathsf{Conf}^{\mathsf{ls}} \cap \mathsf{valid}) \\
\mathsf{L}(\mathsf{pre}_{\mathsf{ls},M}(A)) &= \mathsf{pre}_{\mathsf{ls},M}(\mathsf{L}(A) \cap \mathsf{Conf}^{\mathsf{ls}} \cap \mathsf{valid})
\end{aligned}
$$

can be effectively constructed in time $\mathsf{poly}(|M||A|)2^{O(|\mathcal{X}|^2)}$*. The sizes of the automata are:*

$$|\mathsf{pre}^*_{\mathsf{ls},M}(A)| = |A|\mathsf{poly}(|M|)2^{O(|\mathcal{X}|^2)}$$
$$|\mathsf{pre}_{\mathsf{ls},M}(A)| = |A|\mathsf{poly}(|M|)2^{O(|\mathcal{X}|^2)}$$

Proof. Using Theorem 6.4, and the results from above, we construct:

$$\mathsf{pre}^*_{\mathsf{ls},M}(A) := \pi_1(\mathsf{pre}^*_{\times}(\pi_1^{-1}(A) \cap \mathsf{A}_{C_{\mathsf{F}\times}}) \cap \mathsf{A}_{C_{\mathsf{I}\times}})$$
$$\mathsf{pre}_{\mathsf{ls},M}(A) := \pi_1(\mathsf{pre}_{\times}(\pi_1^{-1}(A) \cap \mathsf{A}_{C_{\mathsf{F}\times}}) \cap \mathsf{A}_{C_{\mathsf{I}\times}})$$

Note that the construction for immediate predecessor sets is correct, as single steps of the cross-product DPN correspond to single steps of the Monitor-DPN.

The time and size estimation follows straightforwardly from the estimations for the components: The automaton $\mathsf{A}_{\mathsf{Cons}}$ can be constructed in time $2^{O(|\mathcal{X}|^2)}$, and its size is $2^{O(|\mathcal{X}|^2)}$ (Lemma 4.20). Hence, the DPN-Acceptor $D_{\mathsf{A}_{\mathsf{Cons}}}$ can be constructed in time

$$\mathsf{poly}(2^{O(|\mathcal{X}|^2)}|\mathsf{Act}|)2^{O(|\mathcal{X}|)} = \mathsf{poly}(|\mathsf{Act}|)2^{O(|\mathcal{X}|^2)},$$

and its size is $\mathsf{poly}(|\mathsf{Act}|)2^{O(|\mathcal{X}|^2)}$ (Theorem 5.3). Finally, the cross-product can be constructed in time

$$\mathsf{poly}(|M|\mathsf{poly}(|\mathsf{Act}|)2^{O(|\mathcal{X}|^2)})2^{O(|\mathcal{X}|)} = \mathsf{poly}(|M|)2^{O(|\mathcal{X}|^2)},$$

and its size is $\mathsf{poly}(|M|)2^{O(|\mathcal{X}|^2)}$ (Theorem 5.5). The remaining operations (intersection, projection, inverse projection, and lock-insensitive predecessor set computation) are all polynomial time. Moreover, the number of states of the resulting automaton depends linearly on the number of states of the input automata, which completes the proposition. □

Note that the algorithm is polynomial in the size of the automaton and the DPN, and exponential only in the number of locks. For Monitor-DPNs derived from typical programs, we expect the number of locks to be much smaller than the DPN itself.

6.4 Applications

In this section, we show how predecessor set computation can be used for program analysis and verification. We first generalize the definition of predecessor sets:

Definition 6.6. *Let M be a DPN, and $S \subseteq \mathsf{Act}^*$ be a set of sequences of actions. Then, we define*

$$\mathsf{pre}_M[S](C) := \{c \in \mathsf{Conf} \mid \exists c' \in C, \bar{a} \in S.\ c \xrightarrow{\bar{a}}{}^*_M c'\}.$$

Similarly, we define for a Monitor-DPN M and a set $S \subseteq (\mathsf{Act}_\mathcal{X})^*$:

$$\mathsf{pre}_{\mathsf{ls},M}[S](C) := \{c \in \mathsf{Conf} \mid \exists c' \in C, \bar{o} \in S.\ c \xrightarrow{\bar{o}}{}^{*}_{\mathsf{ls},M} c'\}.$$

In order to get an effective predecessor set computation, we need to restrict S to rather simple sets. For example, consider the set S given by the regular expression $(a\bar{a} + b\bar{b} + \tau)^*$. Then, the statement $p_0\gamma_0 \in \mathsf{pre}_M[S](C)$ is true, if and only if there is an execution from the start configuration to a configuration in C that is properly synchronized w.r.t. rendezvous-communication. However, reachability problems for pushdown systems communicating via rendezvous are undecidable [104]. Hence, we only regard sets of the form $S = A$ and $S = A^*$ for a set $A \subseteq \mathsf{Act}_\mathcal{X}$.

Obviously, we have $\mathsf{pre} = \mathsf{pre}[\mathsf{Act}]$ and $\mathsf{pre}^* = \mathsf{pre}[\mathsf{Act}^*]$. In order to compute $\mathsf{pre}[A]$ or $\mathsf{pre}[A^*]$ for some set $A \subseteq \mathsf{Act}$, we modify the DPN by removing all rules with actions not in A, and then compute pre or pre^* for the modified DPN. For a DPN M and a set $A \subseteq \mathsf{Act}$, we write $M|_A$ for the DPN that results from M when removing all rules with actions not in A. Thus, we have $\mathsf{pre}_M[A] = \mathsf{pre}_{M|_A}$ and $\mathsf{pre}_M[A^*] = \mathsf{pre}^*_{M|_A}$, and, analogously, $\mathsf{pre}_{\mathsf{ls},M}[A] = \mathsf{pre}_{\mathsf{ls},M|_A}$ and $\mathsf{pre}_{\mathsf{ls},M}[A^*] = \mathsf{pre}^*_{\mathsf{ls},M|_A}$.

Many interesting properties can be expressed by predecessor sets. For example, if C characterizes some set of undesired configurations (e.g. configurations that have a data-race or are deadlocked), a program has this undesired property if and only if

$$p_0\gamma_0 \in \mathsf{pre}[\mathsf{Act}^*](C),$$

where $p_0\gamma_0$ is the start configuration of the program.

Also, more complex properties can be checked. For example, a variable x is live at program point u if and only if

$$p_0\gamma_0 \in \mathsf{pre}[\mathsf{Act}^*](\mathsf{at}_u \cap \mathsf{pre}[(\neg\mathsf{write}(x))^*](\mathsf{pre}[\mathsf{read}(x)](\mathsf{Conf}))),$$

where at_u is the set of configurations that have at least one thread that is at program point u, $\neg\mathsf{write}(x)$ is the set of actions that do not write to variable x, and $\mathsf{read}(x)$ is the set of actions that read from variable x.

6.4.1 Atomic-Set Serializability Violation

Another analysis that can be expressed by predecessor sets is detection of atomic-set serializability violations [6, 117]. An atomic-set serializability violation indicates a *high-level data-race* [6]. Intuitively, high-level data-races are a generalization of data-races to regions of memory that are assumed to be updated atomically. As an example[1], consider a program that has a vector (with an x- and y-coordinate) shared between two threads. One thread sets the vector to

[1]Taken from [6].

zero, and the other thread reads the vector. Setting the vector to zero is done component-wise, and the update of each component is protected by a lock, as well as the read access to each component of the vector. Obviously, there is no data-race, as the accesses to both the x- and y-component are properly protected. However, when the second thread is executed after the first component has been updated, but before the second component has been updated, it will read some inconsistent vector. Such behaviors are called *high-level data-races.*

We briefly sketch the approach of [117] to describe high-level data-races: The programmer (or some heuristics) has to specify *atomic sets* of data and *units of work* in the code. The data in an atomic set is assumed to belong together, like the x- and y-component of the vector in the above example, and a unit of work is assumed to perform an operation on the data, e.g. setting the vector to zero. An execution is *serializable*, if its projection to each atomic set is equivalent to an execution in which the units of work occur in serial order. Thus, a violation of atomic-set serializability indicates a high-level data-race. Vaziri et al. [117] identify 11 problematic access patterns that are complete in the sense that the program violates atomic-set serializability if and only if it has an execution with such a problematic pattern[2]. The access patterns are sequences of read- and write-actions to elements of an atomic set l. For example, the pattern $W_u(l); R_{u'}(l); W_u(l)$ means that unit of work u writes to l, then unit of work u' reads from l, and, finally, u writes to l again. In this case, the thread executing u' will observe an intermediate state of the data, as u is currently updating it. In the example above, u would be the setting of the vector to zero, and u' would be the reading of the vector.

Checking whether the program has an execution that has one of those problematic access patterns can be done by iterated predecessor set computation. Intuitively, the predecessor set computation is iterated for each access in the pattern, ensuring that the intermediate configurations execute the corresponding access. In practice, one has to additionally identify the units of work. In [61, 63], we have implemented atomic-set serializability violation detection for parallel pushdown systems with locks, which are extracted from Java programs using the random isolation method [62]. The detection of the problematic access patterns described there can be adapted straightforwardly to predecessor set computations for DPNs.

6.4.2 EF-Formula

Lugiez and Schnoebelen [83] show how to decide the branching time logic EF for PA-processes, using predecessor set computations. These results transfer immediately to DPNs and Monitor-DPNs. We briefly recapture the procedure of

[2]Under the assumption that every unit of work that writes to one element of an atomic set writes to all elements of this atomic set.

Lugiez and Schnoebelen [83] here, slightly adapted to our model. The logic EF has the following syntax:

$$\varphi ::= P \mid \neg\varphi \mid \varphi \wedge \varphi' \mid \mathsf{EX}\langle A\rangle\varphi \mid \mathsf{EF}\langle A\rangle\varphi$$

The atomic propositions P are regular sets of configurations, for which automata A_P are given, and the constraints A are sets of actions. We also write EX instead of $\mathsf{EX}\langle\mathsf{Act}\rangle$, and EF instead of $\mathsf{EF}\langle\mathsf{Act}\rangle$. Given a Monitor-DPN M and a consistent and valid configuration $c \in \mathsf{Conf}^{\mathsf{ls}} \cap \mathsf{valid}$, the semantics of EF-logic is defined as follows:

$$\begin{aligned}
c \models P &:\Longleftrightarrow c \in P \cap \mathsf{Conf}^{\mathsf{ls}} \cap \mathsf{valid}\\
c \models \neg\varphi &:\Longleftrightarrow c \in \mathsf{Conf}^{\mathsf{ls}} \cap \mathsf{valid} \wedge c \not\models \varphi\\
c \models \varphi \wedge \varphi' &:\Longleftrightarrow c \models \varphi \wedge c \models \varphi'\\
c \models \mathsf{EX}\langle A\rangle\varphi &:\Longleftrightarrow \exists c' \in \mathsf{Conf}^{\mathsf{ls}} \cap \mathsf{valid}, o \in A.\ c' \models \varphi \wedge c \xrightarrow{o}_{\mathsf{ls},M} c'\\
c \models \mathsf{EF}\langle A\rangle\varphi &:\Longleftrightarrow \exists c' \in \mathsf{Conf}^{\mathsf{ls}} \cap \mathsf{valid}, \bar{o} \in A^*.\ c' \models \varphi \wedge c \xrightarrow{\bar{o}}{}^{*}_{\mathsf{ls},M} c'
\end{aligned}$$

Additionally, we define the connective $\vee$ via De Morgan's law:

$$\varphi \vee \varphi' := \neg(\neg\varphi \wedge \neg\varphi').$$

With EF-logic, the reachability and live variable properties from the beginning of this section can be expressed more succinctly: Reachability of an undesired configuration from a regular set C can be expressed as $p_0\gamma_0 \models \mathsf{EF}\ C$, and liveness of a variable can be expressed as $p_0\gamma_0 \models \mathsf{EF}(\mathsf{at}_u \wedge \mathsf{EF}\langle\neg\mathsf{write}(x)\rangle\mathsf{EX}\langle\mathsf{read}(x)\rangle\mathsf{Conf})$.

In order to decide EF-logic via predecessor set computation, we define

$$[\![\varphi]\!] := \{c \in \mathsf{Conf} \mid c \models \varphi\}.$$

Obviously, the following holds:

$$\begin{aligned}
[\![P]\!] &= \mathsf{L}(\mathsf{A}_P) \cap \mathsf{Conf}^{\mathsf{ls}} \cap \mathsf{valid} & [\![\neg\varphi]\!] &= (\mathsf{Conf}^{\mathsf{ls}} \cap \mathsf{valid}) \setminus [\![\varphi]\!]\\
[\![\varphi \wedge \varphi']\!] &= [\![\varphi]\!] \cap [\![\varphi']\!] & [\![\varphi \vee \varphi']\!] &= [\![\varphi]\!] \cup [\![\varphi']\!]\\
[\![\mathsf{EX}\langle A\rangle\varphi]\!] &= \mathsf{pre}_{\mathsf{ls},M}[A]([\![\varphi]\!]) & [\![\mathsf{EF}\langle A\rangle\varphi]\!] &= \mathsf{pre}_{\mathsf{ls},M}[A^*]([\![\varphi]\!])
\end{aligned}$$

Thus, using the results of this chapter, $[\![\varphi]\!]$ is regular and an automaton for $[\![\varphi]\!]$ can be computed.

In order to do the actual computation, it is easier to compute an automaton A_φ with $\mathsf{L}(\mathsf{A}_\varphi) \cap \mathsf{Conf}^{\mathsf{ls}} \cap \mathsf{valid} = [\![\varphi]\!]$, i.e., we allow that the language of the result automaton may contain inconsistent and invalid configurations. The automaton A_φ is computed as follows:

$$\begin{aligned}
\mathsf{A}_P &= \mathsf{A}_P & \mathsf{A}_{\neg\varphi} &= \neg(\mathsf{A}_\varphi)\\
\mathsf{A}_{\varphi\wedge\varphi'} &= \mathsf{A}_\varphi \cap \mathsf{A}_{\varphi'} & \mathsf{A}_{\varphi\vee\varphi'} &= \mathsf{A}_\varphi \cup \mathsf{A}_{\varphi'}\\
\mathsf{A}_{\mathsf{EX}\langle A\rangle\varphi} &= \mathsf{pre}'_{\mathsf{ls},M}[A](\mathsf{A}_\varphi) & \mathsf{A}_{\mathsf{EF}\langle A\rangle\varphi} &= \mathsf{pre}'^{*}_{\mathsf{ls},M}[A](\mathsf{A}_\varphi)
\end{aligned}$$

where we assume that an atomic proposition P is specified as an automaton, and $\mathsf{pre}'_{\mathsf{ls},M}$ and $\mathsf{pre}'^{*}_{\mathsf{ls},M}$ are the optimized predecessor set computations that may return inconsistent configurations. They will be discussed in Section 7.2 (cf. Theorem 7.2).

For each negation, this construction complements the automaton, resulting in a worst-case exponential blowup. As an optimization, negation can be pushed inwards, but not crossing EX- or EF-operations. It is straightforward to show that the following holds:

$$\begin{aligned} [\![\neg\neg\varphi]\!] &= [\![\varphi]\!] \\ [\![\neg(\varphi \wedge \varphi')]\!] &= [\![\neg\varphi \vee \neg\varphi']\!] \qquad\qquad [\![\neg(\varphi \vee \varphi')]\!] = [\![\neg\varphi \wedge \neg\varphi']\!] \end{aligned}$$

Thus we assume that the considered formula φ is in normal form w.r.t. the above rules, i.e., negations are pushed inwards as far as possible. The size of the automaton A_φ (and also the time required to compute it) can be estimated recursively over the formula φ. For the size $s(\varphi)$ of A_φ, we get:

$$\begin{aligned} s(P) &= |P| & s(\neg\varphi) &= 2^{O(s(\varphi))} \\ s(\varphi \wedge \varphi') &\leq s(\varphi)s(\varphi') & s(\varphi \vee \varphi') &\leq s(\varphi) + s(\varphi') + 1 \\ s(\mathsf{EX}\langle A\rangle\varphi) &= s(\varphi)\mathsf{poly}(|M|) & s(\mathsf{EF}\langle A\rangle\varphi) &= s(\varphi)\mathsf{poly}(|M|)2^{O(|\mathcal{X}|^2)} \end{aligned}$$

If there are no negations in the formula, and we estimate $s(\varphi) + s(\varphi') + 1 \leq s(\varphi)s(\varphi')$ (valid if both automata have more than two states), the size of the resulting automaton is the product of the sizes of the automata for the atomic propositions and a factor of $\mathsf{poly}(|M|)$ for each EX and $\mathsf{poly}(|M|)2^{O(|\mathcal{X}|^2)}$ for each EF. We define $n_{\mathsf{P}}(\varphi)$ to be the number of atomic propositions, $s_{\mathsf{P}}(\varphi)$ to be the maximum size of the automaton for an atomic proposition, $n_{\mathsf{EX}}(\varphi)$ to be the number of EX-operators, and $n_{\mathsf{EF}}(\varphi)$ to be the number of EF-operators. Then, in the negation-free case, we can estimate the size of A_φ by

$$|\mathsf{A}_\varphi| = s_{\mathsf{P}}(\varphi)^{n_{\mathsf{P}}(\varphi)}\mathsf{poly}(M)^{n_{\mathsf{EX}}(\varphi)+n_{\mathsf{EF}}(\varphi)}2^{O(|\mathcal{X}|^2)n_{\mathsf{EF}}(\varphi)}.$$

If we have additional negations, the size increases exponentially for each negation that has to be applied. Like in [82], we define the alternation depth $n_{\mathsf{alt}}(\varphi)$ of negations in φ. Assuming that φ is in normal form, we define:

$$\begin{aligned} n_{\mathsf{alt}}(P) &= 0 & n_{\mathsf{alt}}(\neg\varphi) &= 1 + n_{\mathsf{alt}}(\varphi) \\ n_{\mathsf{alt}}(\varphi \wedge \varphi') &= \max(n_{\mathsf{alt}}(\varphi), n_{\mathsf{alt}}(\varphi')) & n_{\mathsf{alt}}(\varphi \vee \varphi') &= \max(n_{\mathsf{alt}}(\varphi), n_{\mathsf{alt}}(\varphi')) \\ n_{\mathsf{alt}}(\mathsf{EX}\langle A\rangle\varphi) &= n_{\mathsf{alt}}(\varphi) & n_{\mathsf{alt}}(\mathsf{EF}\langle A\rangle\varphi) &= n_{\mathsf{alt}}(\varphi) \end{aligned}$$

Then, the size of the result automaton can be estimated by

$$|\mathsf{A}_\varphi| = \mathsf{tower}(n_{\mathsf{alt}}(\varphi), O(s_{\mathsf{P}}(\varphi)^{n_{\mathsf{P}}(\varphi)})\mathsf{poly}(M)^{n_{\mathsf{EX}}(\varphi)+n_{\mathsf{EF}}(\varphi)}2^{O(|\mathcal{X}|^2)n_{\mathsf{EF}}(\varphi)}),$$

where $\mathsf{tower}(n, m)$ is a tower of n 2s, the innermost exponent being m:

$$\mathsf{tower}(0, m) := m \qquad\qquad \mathsf{tower}(n + 1, m) := 2^{\mathsf{tower}(n,m)}.$$

For the above estimation, we use the fact that $x2^{O(x)} = 2^{O(x)}$, in order to shift the factors upwards in the tower.

6.4.3 Bounded Model-Checking

We already discussed bounded model-checking for DPNs with shared global state [15] in Section 3.4. There, only executions up to a bounded number of *contexts* are regarded. In each context, only a single thread may access the global state. However, all threads may make steps and new threads may be created. Thus, the number of contexts limits the number of communications via the global state. The method described in [15] works by computing iterated predecessor sets. After each predecessor set computation, the result automaton is modified to reflect the context switch (i.e., change the thread that may access the shared memory). This method can also be used with lock-sensitive predecessor set computation. This increases the precision of the analysis, as locks are handled precisely, while they must be emulated via the global state in the original method. However, it also increases the complexity of the analysis, as we replace the polynomial lock-insensitive predecessor set computation by the lock-sensitive predecessor set computation that is exponential in the number of locks.

6.5 Summary and Related Work

In this chapter, we combined the results of the previous chapters to describe a lock-sensitive predecessor set computation that works by reduction of the problem to lock-insensitive predecessor set computation. Then, we sketched applications of our algorithm to program analysis and model-checking.

We used the saturation algorithm of Bouajjani et al. [16] to compute lock-insensitive predecessor sets for DPNs (cf. Theorem 6.5). Regarding pushdown automata and, in general, prefix rewrite systems, the fact that the set of reachable configurations is regular is well-known, and dates back to Büchi [19]. In [22, 23], this result is extended by showing that the relation on configurations induced by runs of a prefix rewrite system is a rational transduction, and giving a construction of the corresponding transducer. In [11, 13, 41], these results are applied to model-checking of linear and branching time logics for pushdown automata, leveraging the automata-theoretic approach to model-checking [115, 116]. In [37], an efficient implementation for linear time logic is provided. Lugiez and Schnoebelen [82, 83] show how to compute predecessor sets for the process algebra PA [8, 12]. They show that predecessor sets of regular sets of process terms are, again, regular and can be computed in polynomial time, and apply this result to model-checking EF-logic. Esparza and Podelski [36] provide an efficient and simple implementation of this algorithm. Finally, Bouajjani et al. [16] generalize predecessor set computation to DPNs and CDPNs.

While the algorithm for DPNs [16] computes predecessor sets only for paths of the form A or A^* for some set of actions A, in [82], a computation for predecessor sets w.r.t. any *decomposable* [109] set of paths is described. Recall that predecessor set computation that is constrained with more complex sets of paths (e.g. arbitrary regular sets) is undecidable, as it can be used to synchronize two pushdown processes. Exploring whether predecessor sets w.r.t. arbitrary decomposable sets can also be computed for DPNs is left to future research.

Interestingly, when regarding execution-hedges instead of interleaved executions, we can compute predecessor sets that are constrained with languages of arbitrary DPN-Acceptors. The languages of DPN-Acceptors include, in particular, all regular languages. In [79], we used a similar technique to compute predecessor sets constrained with regular sets of execution-hedges.

In [35] predecessor and successor set computation for pushdown systems is applied to interprocedural data flow analysis. In [36], this is extended to PA. In [39], an approach to bitvector-analysis of parallel pushdown systems (PPDS) with locks is presented[3], which is also based on acquisition histories. While bitvector-analysis is less general than the temporal logics considered in [55, 56], or the phase-automata of [61, 63], or what can be expressed by lock-sensitive predecessor set computation [79], the analysis of [39] is optimized to bitvector-analysis, allowing a more efficient approach, in particular when computing a solution for every control point in the program. Like our algorithm, the algorithm in [39] has a worst-case runtime that is polynomial in the program size and exponential[4] in the number of locks.

In Section 6.4, we considered live variables analysis as an application for predecessor sets. Regarding dataflow analysis of concurrent programs, in particular kill/gen-analysis, there is a line of research that considers fixed point based techniques rather than automata based techniques. Knoop et al. [65] describe precise intraprocedural kill/gen-analyses for programs with `parbegin...parend`-blocks. The basic idea is to compute the *possible interference* at each control location, i.e., the set of statements that can possibly be executed in parallel. This approach has been extended to interprocedural analysis of programs with parallel calls [110]. In [70], we extend the approach of [65] to interprocedural analysis of programs with dynamic thread creation, and, in [76], to programs with both, dynamic thread creation and parallel calls.

In [78], we use fixed point based techniques and abstract interpretation [31] to compute acquisition structures of programs with procedures and dynamic thread

[3]The paper actually presents the analysis without supporting procedures, but claims that generalization to procedures is straightforward.

[4]The original paper states “double exponential”, but, in personal communication, the authors confirmed that it should be (single) exponential.

creation. Reps et al. [105, 106] introduce *weighted pushdown systems* (WPDS), which combine automata based and fixed point based techniques. Intuitively, a weighted pushdown system is a pushdown system where rules are annotated by weights from a semiring. Given two regular sets of configurations, one can compute the least upper bound of the weights of all paths between the two sets of configurations. The algorithm is similar to the saturation algorithm for predecessor set computation in [11, 37]. Weighted pushdown systems have been extended to support handling of local variables [67]. In [120, 121], weighted dynamic pushdown networks are introduced, leveraging the ideas of WPDSs to DPNs.

In Section 6.4.2, we considered the branching time logic EF. For PA- and PAD-Processes, this logic has been shown decidable by Mayr [85, 86]. The elegant application of predecessor set computation to decide EF-logic has been highlighted by Lugiez and Schnoebelen [83]. For parallel pushdown systems communicating with locks, Kahlon and Gupta [55, 56] have investigated several classes of linear time and branching time logics, and obtained decidability and undecidability results. The atomic propositions regarded there are *double-indexed* properties, fixing the simultaneous state of two processes, while our atomic propositions are arbitrary regular sets of configurations. Kahlon and Gupta [55, 56] show[5] decidability of the linear time logics LTL\X (in [55]), LTL(X,F,F^∞), LTL(X,G), and the branching time alternation free mu-calculus (all in [56]). While deciding EF-logic via predecessor set computations is straightforward, those logics require more subtle algorithms. We leave it for future research to investigate whether they can be transferred to DPNs with locks.

Atomic-set serializability violation detection has been implemented for Java [47] programs [61–63] and applied to some programs of the ConTest benchmark suite [38], yielding promising results. The Java programs are abstracted to parallel pushdown systems (PPDSs) with locks, using the *random isolation* method [62]. While the original analysis of the PPDS [62] is based on an over-approximation by communicating pushdown systems [14], we present a decision procedure that is based on acquisition histories in [61, 63]. This decision procedure has been implemented symbolically with BDDs, and resulted in a significant speedup compared to the approximation by communicating pushdown systems.

[5] The published versions of these papers have a flaw: The algorithms do not properly keep track of the correlation between the different threads that are induced by atomic propositions, leading to incorrect results. The flaw has been confirmed by the authors in personal communication, and can be fixed at the cost of increased complexity. For a detailed description of this flaw, see [59, Sec. 7.9]

7 Optimizations

In the last chapter, we presented an algorithm for lock-sensitive predecessor set computation and discussed its applications. In this chapter, we present optimizations that increase the efficiency of our algorithm for specific applications. Then, we illustrate how our algorithm is used to detect the data-race in the program of Example 1.1. With the help of this example, we identify problems that an implementation of our algorithm has to solve, and discuss possible solutions.

This chapter is organized as follows: In Section 7.1, we introduce a notation for DPNs as pseudocode programs, which is more convenient than specifying sets of transition rules in some cases. In Section 7.2, we discuss optimizations that can be applied to the typical case of querying whether the result of a predecessor set computation contains the start configuration. In Section 7.3, we investigate optimizations for deadlock-free DPNs. In Section 7.4, we illustrate the application of our algorithm to data-race detection by means of a simple example program, and discuss issues that have to be addressed by an implementation. Finally, in Section 7.5, we briefly summarize the results of this chapter and discuss related work.

7.1 Pseudocode

Instead of specifying a DPN by its set of states, stack-symbols, actions, and rules, it is sometimes more convenient to specify a program in pseudocode. The translation of these programs to DPNs is straightforward: Each label and each unlabeled statement in the program corresponds to a stack-symbol. Also, sync-blocks correspond to stack-symbols. The DPN has just one control-state \$ and one base-action a. Each statement is translated into a rule that changes the topmost stack-symbol accordingly.

Example 7.1 (Translation of Pseudocode to DPN)**.** *As an example, consider the following program:*

```
p_1:  sync (x_1){spawn p_2; u: skip};
p_2:  sync (x_2){}; v: skip;
```

To translate this program, we first add labels to all unlabeled statements and at the end of non-returning procedures and synchronized blocks.

```
p_1:  sync (x_1){l_1: spawn p_2; u: skip; l_2: }; l_3:
p_2:  sync (x_2){l_4: }; v: skip; l_5:
```

Then, each label becomes a stack-symbol, and the rules are added according to the statements in the program. The resulting Monitor-DPN is

$$M = (\{\$\}, \Gamma, \Gamma_\perp, \{a\}, \mathcal{X}, \Delta, \mathsf{locks}),$$

where

$$\begin{aligned}
\Gamma &= \{p_1, l_1, l_2, l_3, l_4, l_5, p_2, u, v\} \\
\Gamma_\perp &= \{p_1, l_3, p_2, v, l_5\} \\
\mathcal{X} &= \{x_1, x_2\} \\
\mathsf{locks} &= p_1, l_3, p_2, v, l_5 \mapsto \emptyset, \quad l_1, l_2, u \mapsto \{x_1\}, \quad l_4 \mapsto \{x_2\} \\
\Delta &= \{\$p_1 \overset{\langle x}{\hookrightarrow} \$l_1 l_3, \$l_1 \overset{a}{\hookrightarrow} \$p_2\#\$u, \$u \overset{a}{\hookrightarrow} \$l_2, \$l_2 \overset{\rangle x}{\hookrightarrow} \$, \\
&\quad \$p_2 \overset{\langle x}{\hookrightarrow} \$l_4 v, \$l_4 \overset{\rangle x}{\hookrightarrow} \$, \$v \overset{a}{\hookrightarrow} \$l_5\}
\end{aligned}$$

Instead of sync-blocks, we may also use $\mathsf{acq}\ x$- and $\mathsf{rel}\ x$- statements.

7.2 Queries from the Start Configuration

Predecessor sets are often used to describe executions from the start configuration (cf. the data-race detection and live variables analysis examples in Section 6.4). In those cases, the result of a predecessor set computation is the input to the next iteration, or just queried whether it contains the start configuration. In this section, we describe optimizations that are possible in this case.

7.2.1 Consistency Check

As executions preserve consistency of configurations, we can pull intersections with $\mathsf{A}_{\mathsf{Conf}^{\mathsf{ls}}}$ out of predecessor set computations. We define

$$\mathsf{pre}'^*_{\mathsf{ls}}(A) := \pi_1(\mathsf{pre}^*_\times(\pi_1^{-1}(A) \cap \mathsf{A}_{C_{\mathsf{F}\times}}) \cap \mathsf{A}_{\mathsf{valid}_\times} \cap \pi_2^{-1}(\mathsf{A}_{C_\mathsf{I}})).$$

Then, due to the definitions of $\mathsf{A}_{C_{\mathsf{I}\times}}$ (cf. Section 5.2) and $\mathsf{pre}^*_{\mathsf{ls}}(A)$ (cf. Theorem 6.5), we have

$$\begin{aligned}
&\mathsf{L}(\mathsf{pre}^*_{\mathsf{ls}}(A)) \\
&= \mathsf{L}(\pi_1(\mathsf{pre}^*_\times(\pi_1^{-1}(A) \cap \mathsf{A}_{C_{\mathsf{F}\times}}) \cap \mathsf{A}_{\mathsf{valid}_\times} \cap \pi_1^{-1}(\mathsf{A}_{\mathsf{Conf}^{\mathsf{ls}}}) \cap \pi_2^{-1}(\mathsf{A}_{C_\mathsf{I}}))) \\
&= \mathsf{L}(\mathsf{pre}'^*_{\mathsf{ls}}(A)) \cap \mathsf{Conf}^{\mathsf{ls}},
\end{aligned}$$

where the first equality is by unfolding the definitions of $\mathsf{pre}^*_{\mathsf{ls}}(A)$ and $\mathsf{A}_{C_{\mathsf{I}\times}}$, and the second equality by $\pi_1 \circ \pi_1^{-1} = \mathsf{id}$ and $\mathsf{L}(\mathsf{A}_{\mathsf{Conf}^{\mathsf{ls}}}) = \mathsf{Conf}^{\mathsf{ls}}$.

Thus, when querying whether a consistent configuration $c_0 \in \mathsf{Conf}^{\mathsf{ls}}$ (e.g. the start configuration) is contained in the predecessor set, we can exploit the following optimization:

$$c_0 \in \mathsf{pre}^*_{\mathsf{ls}}(A) \iff c_0 \in \mathsf{pre}'^*_{\mathsf{ls}}(A).$$

As $\mathsf{A}_{\mathsf{Conf}^{\mathsf{ls}}}$ has exponentially many states in the number of locks, the automaton $\mathsf{pre}'^*_{\mathsf{ls}}(A)$—that performs no intersection with $\mathsf{A}_{\mathsf{Conf}^{\mathsf{ls}}}$—is considerably smaller than $\mathsf{pre}^*_{\mathsf{ls}}(A)$ in typical cases.

We defined the automaton $\mathsf{pre}^*_{\mathsf{ls}}(A)$ for arbitrary automata A, such that $\mathsf{pre}^*_{\mathsf{ls}}(A)$ only considers consistent and valid configurations in $\mathsf{L}(A)$ (cf. Theorem 6.5). Thus, when computing iterated predecessor sets, there is no need to intersect the result of the inner computation with $\mathsf{A}_{\mathsf{Conf}^{\mathsf{ls}}}$, and the following optimization applies:

$$\mathsf{L}(\mathsf{pre}^*_{\mathsf{ls}}(A_1 \cap \mathsf{pre}^*_{\mathsf{ls}}(A_2))) = \mathsf{L}(\mathsf{pre}^*_{\mathsf{ls}}(A_1 \cap \mathsf{pre}'^*_{\mathsf{ls}}(A_2))).$$

Similar optimizations apply for immediate predecessor set computations, and in the next subsection we show an polynomial algorithm to compute an automaton $\mathsf{pre}'_{\mathsf{ls}}(A)$. In many cases no intersection with $\mathsf{A}_{\mathsf{Conf}^{\mathsf{ls}}}$ is required at all, e.g. in the data-race detection and live variables analysis examples. We also applied these optimizations to model-checking EF-logic in Subsection 6.4.2.

7.2.2 Immediate Predecessor Sets

When computing immediate predecessor sets, the lock-execution-hedge h contains exactly one non-leaf-node. Thus, also the lock-a/r-hedge $\mathsf{ar}(h)$ has exactly one non-leaf-node. Hence, its dependence-graph is trivially acyclic, and it cannot contain two final acquisitions of the same lock. Thus, criteria (C1) and (C3) are trivially satisfied, and it remains to check Criterion (C2):

$$((\mathsf{used}(\mathsf{ar}(h)) \cup \mathsf{acq}(\mathsf{ar}(h))) \setminus \mathsf{rel}(\mathsf{ar}(h))) \cap \mathsf{ar}(h)|_2 = \emptyset.$$

As a use-node in $\mathsf{ar}(h)$ with a non-empty set of used locks corresponds to at least two non-leaf-nodes in h, there are no such use-nodes in $\mathsf{ar}(h)$, and we have $\mathsf{used}(\mathsf{ar}(h)) = \emptyset$. Moreover, $\mathsf{ar}(h)$ contains exactly one node, hence it does not contain both, a release- and an acquisition-node. Thus, (C2) simplifies to

$$\mathsf{acq}(\mathsf{ar}(h)) = \emptyset \ \vee\ (\exists x.\ \mathsf{acq}(\mathsf{ar}(h)) = \{x\} \wedge x \notin \mathsf{ar}(h)|_2).$$

This can be checked by a polynomial size tree automaton. Also the translation of this tree automaton to a DPN-Acceptor can be done in polynomial time: Here, the reentrance elimination caused the exponential blowup. However, if there is just one node, reentrance elimination can be done in polynomial time, e.g. by nondeterministically guessing the lock x of the node. Thus, except the intersection with $\mathsf{A}_{\mathsf{Conf}^{\mathsf{ls}}}$, immediate predecessor sets can be computed in polynomial time. In the last section we argued that this intersection can be omitted in many cases. However, in Section 9.2.1.1, we show that deciding whether a regular set of configurations contains a consistent configuration is NP-hard, and thus also immediate predecessor set computation with intersection with $\mathsf{A}_{\mathsf{Conf}^{\mathsf{ls}}}$ is NP-hard. This matches the fact that $\mathsf{A}_{\mathsf{Conf}^{\mathsf{ls}}}$ has exponential size in the number of locks, and thus the result automaton of the immediate predecessor set computation may blow up exponentially upon intersection with $\mathsf{A}_{\mathsf{Conf}^{\mathsf{ls}}}$.

We summarize the results by the following theorem:

Theorem 7.2. *Given a Monitor-DPN M and an automaton A, we can compute automata $\mathsf{pre}'^{*}_{\mathsf{ls},M}(A)$ and $\mathsf{pre}'_{\mathsf{ls},M}(A)$, such that*

$$\mathsf{L}(\mathsf{pre}'^{*}_{\mathsf{ls},M}(A)) \cap \mathsf{Conf}^{\mathsf{ls}} = \mathsf{pre}^{*}_{\mathsf{ls},M}(\mathsf{L}(A) \cap \mathsf{Conf}^{\mathsf{ls}} \cap \mathsf{valid})$$
$$\mathsf{L}(\mathsf{pre}'_{\mathsf{ls},M}(A)) \cap \mathsf{Conf}^{\mathsf{ls}} = \mathsf{pre}_{\mathsf{ls},M}(\mathsf{L}(A) \cap \mathsf{Conf}^{\mathsf{ls}} \cap \mathsf{valid}).$$

The automaton $\mathsf{pre}'^{*}_{\mathsf{ls},M}(A)$ can be computed in time $\mathsf{poly}(|M||A|)2^{O(|\mathcal{X}|^2)}$ and has size $|A|\mathsf{poly}(|M|)2^{O(|\mathcal{X}|^2)}$. The automaton $\mathsf{pre}'_{\mathsf{ls},M}(A)$ can be computed in time $\mathsf{poly}(|M||A|)$ and has size $|A|\mathsf{poly}(|M|)$. □

7.2.3 Initial Releases

Moreover, the start configuration usually holds no locks. In this case, the execution that belongs to the outermost predecessor set computation contains no initial releases. Consequently, the release-set and the release-graph of execution-hedges from the start configuration are always empty. This simplifies the construction of the automaton $\mathsf{A}_{\mathsf{Cons}}$, as the release-set and release-graph components of the acquisition structure may be omitted. Also, the initial lockstack is empty, and thus the component of the hedge acquisition structure that collects the initially held locks may be omitted, and Criterion (C2) becomes trivial, as there are no initially held locks. (Recall that (C2) (cf. Lemma 4.14) checks that the set of used and acquired locks that are not released is disjoint from the set of initially held locks.) Moreover, in the construction of $D_{\mathsf{A}_{\mathsf{Cons}}}$ (cf. Section 5.1.2), we may omit the initial-release flag.

As a final note, we mention that for iterated predecessor set computations like

$$p_0\gamma_0 \in \mathsf{pre}^{*}_{\mathsf{ls}}(A_1 \cap \mathsf{pre}^{*}_{\mathsf{ls}}(A_2)),$$

the release-set for the inner predecessor set computation is, in general, not empty. Thus, this optimization applies only to the outermost predecessor set computation.

7.3 Deadlocks

In the last section, we identified some optimizations that may be applied in common cases. In this section, we investigate the correspondence between deadlocks and cycles in the dependence-graph, and derive optimizations for certain classes of deadlock-free DPNs. We aim at finding classes of DPNs where the dependence-graph is always acyclic, and thus the acquisition- and release-graphs are not required for the analysis. Note that the acquisition- and release-graphs are sets over pairs of locks, i.e., there are $2^{|\mathcal{X}|^2}$ different acquisition-graphs. All other sets involved in our analysis are just sets of locks. Thus, when omitting the acquisition and release-graphs, the exponential factor in the complexity of our analysis reduces from $2^{O(|\mathcal{X}|^2)}$ to $2^{O(|\mathcal{X}|)}$.

Intuitively, a deadlock is a configuration where some threads mutually wait on each other to release a lock, or where a thread is terminated still holding a lock, and thus blocking another thread. We distinguish a deadlock from a terminated configuration, where threads may still hold locks, but do not block other threads. Formally, we characterize a deadlock by a non-empty set of threads. Each thread in a deadlock is either terminated and holds a lock, or all its outgoing steps are acquisitions that are blocked by another thread from the deadlock. In order to distinguish a deadlocked from a terminated configuration, we require at least one thread in the deadlock to have outgoing acquisition-steps.

Definition 7.3 (Deadlocked Configuration)**.** *Let M be a Monitor-DPN. First, we define the function* $\mathsf{act} : P\Gamma^* \to 2^{\mathsf{Act}_{\mathcal{X}}}$ *that maps a thread-configuration to the set of* possible actions *from this configuration:*

$$\mathsf{act}(pw) := \{o \mid \exists c.\ pw \xrightarrow{o} c\}.$$

A configuration $c = p_1w_1 \dots p_nw_n \in \mathsf{Conf}^{\mathsf{ls}}$ *is called* deadlocked*, iff*

$$\begin{aligned}&\exists \emptyset \subset D \subseteq \{1,\dots,n\}.\ \forall i \in D.\\ &\qquad (\mathsf{act}(p_iw_i) = \emptyset \wedge \mathsf{ls}(w_i) \neq \varepsilon)\\ &\qquad \vee (\mathsf{act}(p_iw_i) \neq \emptyset \wedge \forall o \in \mathsf{act}(p_iw_i).\ \exists j \in D, x \in \mathsf{ls}(w_j).\ i \neq j \wedge o = \langle_x)\\ &\quad \wedge \exists i \in D.\ \mathsf{act}(p_iw_i) \neq \emptyset\end{aligned}$$

The set of deadlocked configurations is denoted by C_{dead}.

A DPN M with a start configuration $c_0 \in \mathsf{Conf}^{\mathsf{ls}}$ is called deadlock-free*, if no deadlocked configuration can be reached from c_0.*

Assuming that the analyzed DPN is deadlock-free is realistic, if a separate analysis for possible deadlocks is done before the reachability analysis. If this prior analysis proves the DPN deadlock-free, we may exploit this knowledge in our analysis.

The reason why we examine deadlock-free DPNs is the observation that cycles in the dependence-graph (cf. Section 4.3.1) indicate deadlock-like conditions, where a subset of threads is mutually waiting on each other to release a lock.

First of all, we give some examples of deadlock-free DPNs, where acquisition- and release-graphs are required for precise reachability analysis. Then, we discuss additional restrictions with which acquisition- and release-graphs are not required for precise analysis. Consider the following program:

```
p:  acq x                         q:  acq y
    acq y OR goto l                   acq x
    rel y                             rel x
      u: read                           v: write
    l: rel x                          rel y
```

Here, OR means nondeterministic choice—i.e., the corresponding DPN has two rules that both can be applied, one acquiring the lock y, and the other continuing the execution at label l. This program has no deadlock, as the thread p can always decide to continue at l, and thus break the deadlock.

Moreover, the program has no data-race: The only lock-a/r-hedge that reaches u and v simultaneously when starting with two threads, one at p and one at q, is $h = [(t_1, \emptyset), (t_2, \emptyset)]$, with

$$t_1 = \langle_x \rangle_{\{y\}}(\varepsilon)\tau_u \quad \text{and} \quad t_2 = \langle_y \rangle_{\{x\}}(\varepsilon)\tau_v$$

For better readability, we explicitly wrote the τ-nodes, and indexed them with the label that is reached by the corresponding thread. Obviously, h satisfies (C1) and (C2), but violates (C3). In our example, h has a cyclic acquisition-graph. However, an analogous example is possible where h has a cyclic release-graph. Hence, this example shows that both the acquisition- and release-graph are required for precise reachability analysis, even for deadlock-free programs.

7.3.1 Inescapable Locks

The fact that the above program has no deadlock results from allowing to non-deterministically choose not to acquire the lock y, but jump to label l instead, and thus break a possible deadlock. However, in typical programs, there is no such option: One has to call some `lock()` function, and this function will not return until the lock is acquired. An exception are libraries that support time-outs for locking (i.e., the lock-operation returns with an error after waiting a certain time for the lock), or `tryLock()`-operations (i.e., the lock-operation immediately returns with an error if the lock is not free). Such operations can be over-approximated by nondeterministically deciding whether to acquire the lock or not.

For example, the `java.util.concurrent.locks`-package supports both, time-outs and `tryLock()`-operations, while the original Java locking mechanism via synchronized methods does not support either concept.

Locks that cannot be "escaped" by nondeterministically choosing another transition are called *inescapable*. Accordingly, a Monitor-DPN where all locks are inescapable is called *inescapable* Monitor-DPN.

Definition 7.4 (Inescapable Monitor-DPNs). *Let*

$$M = (P, \Gamma, \Gamma_\perp, \mathsf{Act}, \mathcal{X}, \Delta, \mathsf{locks})$$

be a Monitor-DPN. M is called inescapable, *if there are no alternatives to entering a monitor:*

$$p\gamma \overset{\langle_x}{\hookrightarrow} p_1 w_1 \in \Delta \wedge p\gamma \overset{o}{\hookrightarrow} p_2 w_2 \in \Delta \implies o = \langle_x$$

Example 7.5. *Consider the following example program:*

```
p:  spawn q            q:  acq y              r:  acq x
    acq x                    spawn r              rel x
      acq y                  v: skip              w: skip
      rel y                rel y
      u: skip
    rel x
```

The execution starts with a single thread at p that holds no locks. The corresponding DPN is obviously inescapable and deadlock-free. We want to check whether the labels u,v, and w are simultaneously reachable. The only lock-a/r-hedge that reaches a configuration that is simultaneously at u,v, and w is $h = [(t, \emptyset)]$ with

$$t = \langle\rangle_\emptyset(\langle_y \langle\rangle_\emptyset(\langle\rangle_{\{x\}}(\varepsilon)\tau_w)\tau_v)\langle_x\langle\rangle_{\{y\}}(\varepsilon)\tau_u$$

The tree t has the following shape:

$$\begin{tikzcd}
 & & \langle\rangle_\emptyset \arrow[dl, dash] \arrow[dr, dash] & \\
 & \langle_y \arrow[d, dash] & & \langle_x \arrow[d, dash] \\
 & \langle\rangle_\emptyset \arrow[dl, dash] \arrow[dr, dash] & & \langle\rangle_{\{y\}} \arrow[d] \\
\langle\rangle_{\{x\}} \arrow[d, dash] & & \tau_v & \tau_u \\
\tau_w & & &
\end{tikzcd}$$

Obviously, h has no multiple final acquisitions of the same lock, and thus satisfies (C1). As no locks are initially held by h, it trivially satisfies (C2). However, the dependence-graph of h has a cycle:

$$\langle_x \rightarrow \langle\rangle_{\{y\}} \rightarrow \langle_y \rightarrow \langle\rangle_\emptyset \rightarrow \langle\rangle_{\{x\}} \rightarrow \langle_x$$

Hence, the labels u, v, and w are not simultaneously reachable.

The cycle in the dependence-graph leads to a cyclic acquisition-graph. Thus, this example shows that we need acquisition-graphs for precise analysis of inescapable, deadlock-free DPNs.

However, we do not need release-graphs: We show that any lock-a/r-hedge from a reachable configuration of an inescapable, deadlock-free DPN has an acyclic release-graph:

Lemma 7.6. *Let M be an inescapable, deadlock-free DPN with start configuration $c_0 \in \mathsf{Conf}^{\mathsf{ls}}$. Let $c \in \mathsf{Conf}^{\mathsf{ls}}$ be a reachable configuration (i.e., $c_0 \rightarrow_{\mathsf{ls}}^{*} c$) and h an execution-hedge from c (i.e., $\exists c'.\ c \overset{h}{\Rightarrow} c'$). Moreover, assume that the corresponding lock-a/r-hedge $\mathsf{ar}(h{\times}\mathsf{ls}(c))$ satisfies (C1) and (C2). Then, its release-graph $\mathsf{E}^{\mathsf{rel}}(\mathsf{ar}(h{\times}\mathsf{ls}(c)))$ is acyclic.*

In the proof of this lemma, we will construct a partial schedule of a lock-execution-hedge that leads to a deadlock. In order to connect the partial schedule of the lock-execution-hedge to a run of the interleaving semantics, we need the following lemma:

Lemma 7.7. *Let $h \in \mathsf{H}$ be an execution-hedge, $c \in \mathsf{Conf}^{\mathsf{ls}}$ be a consistent configuration, $c' \in \mathsf{Conf}$ be a configuration, and $h_s{\times}\bar{\mu}_s \in \mathsf{H}^{\mathsf{ls}}$ be a lock-execution-hedge. Then, we have*

$$\begin{aligned} & c \overset{h}{\Rightarrow} c' \wedge h{\times}\mathsf{ls}(c) \rightsquigarrow_{\mathsf{ls}}^{*} h_s{\times}\bar{\mu}_s \\ \Longrightarrow\ & \exists \tilde{c}, \tilde{h}.\ c \overset{\tilde{h}}{\Rightarrow} \tilde{c} \overset{h_s}{\Rightarrow} c' \wedge \mathsf{sched}_{\mathsf{ls}}(\tilde{h}{\times}\mathsf{ls}(c)) \neq \emptyset \wedge \mathsf{ls}(\tilde{c}) = \bar{\mu}_s. \end{aligned}$$

Proof. The lemma is shown by induction on $\rightsquigarrow_{\mathsf{ls}}^{*}$. If $\rightsquigarrow_{\mathsf{ls}}^{*}$ makes no steps, we have $h = h_s$ and $\mathsf{ls}(c) = \bar{\mu}_s$. Moreover, we have $c \overset{\tau^{|h|}}{\Longrightarrow} c \overset{h}{\Rightarrow} c'$ and $\mathsf{sched}_{\mathsf{ls}}(\tau^{|h|}{\times}\mathsf{ls}(c)) = \{\varepsilon\}$, where $\tau^{|h|}$ is the execution-hedge that consists of $|h|$ leaf-nodes. Choosing $\tilde{c} := c$ and $\tilde{h} := \tau^{|h|}$ completes the case.

In the case that $\rightsquigarrow_{\mathsf{ls}}^{*}$ makes at least one step, we obtain $\hat{h}{\times}\hat{\bar{\mu}} \in \mathsf{H}^{\mathsf{ls}}$, such that

$$h{\times}\mathsf{ls}(c) \rightsquigarrow_{\mathsf{ls}} \hat{h}{\times}\hat{\bar{\mu}} \rightsquigarrow_{\mathsf{ls}}^{*} h_s{\times}\bar{\mu}_s.$$

If the first step is no spawn, we split h,c,$\hat{h}$, and $\hat{\bar{\mu}}$ according to the first step. We obtain $h_1, h_2, o, \hat{t}, \hat{\mu}, c_1$, and c_2 with

$$h = h_1(o;\hat{t})h_2,\ \ c = c_1\varphi c_2,\ \ \hat{h} = h_1\hat{t}h_2, \text{ and }\ \hat{\bar{\mu}} = \mathsf{ls}(c_1)\hat{\mu}\mathsf{ls}(c_2),$$

such that o can be scheduled in the context of $\mathsf{ls}(c_1c_2)$. From $c \overset{h}{\Rightarrow} c'$, we obtain $\hat{\varphi}$, c_1', c_2', and c_φ' such that

$$\varphi \overset{o}{\Rightarrow} \hat{\varphi} \overset{\hat{t}}{\Rightarrow} c_\varphi',\ \ c_1 \overset{h_1}{\Rightarrow} c_1', \text{ and }\ c_2 \overset{h_2}{\Rightarrow} c_2'.$$

As the operation o has the same effect on the lockstack, whether executed by the scheduler or by the Monitor-DPN, we have $\hat{\mu} = \mathsf{ls}(\hat{\varphi})$. Together, we get

$$\hat{h}{\times}\mathsf{ls}(c_1\hat{\varphi}c_2) \rightsquigarrow_{\mathsf{ls}}^{*} h_s{\times}\bar{\mu}_s \wedge c_1\hat{\varphi}c_2 \overset{\hat{h}}{\Rightarrow} c'.$$

The induction hypothesis yields $\tilde{c}$ and $\tilde{h}$ such that

$$c_1\hat{\varphi}c_2 \overset{\tilde{h}}{\Rightarrow} \tilde{c} \overset{h_s}{\Rightarrow} c' \wedge \mathsf{sched}_{\mathsf{ls}}(\tilde{h}\times\mathsf{ls}(c_1\hat{\varphi}c_2)) \neq \emptyset \wedge \mathsf{ls}(c_1\hat{\varphi}c_2) = \bar{\mu}_s.$$

Splitting $\tilde{h}$, we obtain $\tilde{h}_1, \tilde{h}_2$, $\tilde{t}$, $\tilde{c}_1, \tilde{c}_\varphi$, and $\tilde{c}_2$ such that

$$\tilde{h} = \tilde{h}_1\tilde{t}\tilde{h}_2 \qquad \tilde{c} = \tilde{c}_1\tilde{c}_\varphi\tilde{c}_2 \qquad c_1 \overset{\tilde{h}_1}{\Rightarrow} \tilde{c}_1 \qquad \hat{\varphi} \overset{\tilde{t}}{\Rightarrow} \tilde{c}_\varphi \qquad c_2 \overset{\tilde{h}_2}{\Rightarrow} \tilde{c}_2.$$

Prepending the execution $\varphi \overset{o}{\Rightarrow} \hat{\varphi}$, we get $\varphi \overset{o;\tilde{t}}{\Rightarrow} \tilde{c}_\varphi$, and thus $c \xRightarrow{\tilde{h}_1(o;\tilde{t})\tilde{h}_2} \tilde{c}_1\tilde{c}_\varphi\tilde{c}_2$. As o is schedulable in the context of $\mathsf{ls}(c_1c_2)$, we also have

$$(\tilde{h}_1(o;\tilde{t})\tilde{h}_2)\times\mathsf{ls}(c) \rightsquigarrow_{\mathsf{ls}} \tilde{h}\times\mathsf{ls}(c_1\hat{\varphi}c_2),$$

which completes the case.

The case of a spawn-step is proved analogously. □

Note that the above proof could be done more conceptually with the notion of a prefix of an execution-hedge: Intuitively, we can prove the lemma as follows: The scheduler schedules a *prefix* of the hedge h, leaving the suffix h_s. Let this prefix be $\tilde{h}$. Then, we split the execution $c \overset{h}{\Rightarrow} c'$ into the execution of the prefix $\tilde{h}$ and the suffix h_s, obtaining a configuration $\tilde{c}$ with $c \overset{\tilde{h}}{\Rightarrow} \tilde{c} \overset{h_s}{\Rightarrow} c'$. As the scheduler and the rules of the Monitor-DPN alter the lockstack in the same way, we have $\bar{\mu}_s = \mathsf{ls}(\tilde{c})$. As the scheduler schedules the complete prefix $\tilde{h}\times\mathsf{ls}(c)$ of $h\times\mathsf{ls}(c)$, we have $\mathsf{sched}_{\mathsf{ls}}(\tilde{h}\times\mathsf{ls}(c)) \neq \emptyset$. □

However, formalizing the notion of a prefix of an execution-hedge and connecting this notion with the hedge semantics and the scheduler requires quite a large formal overhead. As this is the only place in this thesis where such a notion is needed, we avoided this overhead, and performed a more direct (but less conceptual) induction proof here.

Now, we are prepared to prove Lemma 7.6:

Proof of Lemma 7.6. By contraposition, we may assume that $\mathsf{E}^{\mathsf{rel}}(\mathsf{ar}(h\times\mathsf{ls}(c)))$ is cyclic and construct a schedule to a deadlocked configuration. Let $\tilde{h} := \mathsf{ar}(h\times\mathsf{ls}(c))$ be the corresponding lock-a/r-hedge.

First, we schedule an initial part of $\tilde{h}$: We repeatedly select a minimal node in the dependence-graph and schedule it, until there are no minimal nodes left. As $\tilde{h}$ satisfies the criteria (C1) and (C2), and (C1) and (C2) are preserved by scheduling steps, minimal nodes in the dependence-graph can always be scheduled. We get a lock-a/r-hedge $\tilde{h}_1$ and a schedule $\tilde{h} \rightsquigarrow^* \tilde{h}_1$, such that $\mathsf{g}^{\mathsf{dep}}(\tilde{h}_1)$ contains no minimal nodes.

All non-terminated threads in $\tilde{h}_1$ start with an acquisition- or a use-node, as release-nodes have no incoming additional edges in $\mathsf{g}^{\mathsf{dep}}(h_1)$, and thus would be minimal. Moreover, as the release-graph was cyclic, at least one[1] thread starts with a use-node.

Let $D \subset \mathbb{N}$ be the set of threads that start with a use-node. (Threads are identified by their position in the list $\tilde{h}_1$.) Each thread $i \in D$ starts with a use-node $\langle\rangle_X$ that has an incoming edge in the dependence-graph. This incoming edge must result from a release-node $\rangle_x$ of one of the used locks $x \in X$. This release-node cannot be in one of the threads outside D, as they are terminated or start with final acquisitions, and due to well-formedness, initial releases do not occur after final acquisitions. Let the index of the thread with the $\rangle_x$-node be $j \in D$. Due to well-nestedness, the initially released lock must be on the lockstack, i.e., we have $x \in \mu_j$. Moreover, we have $i \neq j$, as otherwise the usage of x would be reentrant. Hence each thread in D starts with a usage that uses a lock that is currently acquired by some other thread in D.

At this point, we have constructed a deadlock w.r.t. the scheduler for lock-a/r-hedges. However, deadlocks are defined on the more fine grained interleaving semantics. Thus, we transfer the deadlock to the scheduler for lock-execution-hedges and then to the interleaving semantics: The schedule $\tilde{h} \rightsquigarrow^* \tilde{h}_1$ corresponds to a schedule on lock-execution-hedges. We get $h_1 \times \bar{\mu}_1 \in \mathsf{H}^{\mathsf{ls}}$ such that $\tilde{h}_1 = \mathsf{ar}(h_1 \times \bar{\mu}_1)$ and $h \times \mathsf{ls}(c) \rightsquigarrow_{\mathsf{ls}}^* h_1 \times \bar{\mu}_1$. We continue scheduling $h_1 \times \bar{\mu}_1$ with the lock-sensitive scheduler. For each thread in D, this schedule must get stuck before completing the first usage, just before an acquisition of a lock that is currently held by another thread in D. We get $h_2 \times \bar{\mu}_2 \in \mathsf{H}^{\mathsf{ls}}$ such that $h \times \mathsf{ls}(c) \rightsquigarrow_{\mathsf{ls}}^* h_2 \times \bar{\mu}_2$, and for each thread in D, the first node of the corresponding tree in h_2 is an acquisition of a lock held by some other thread in D. Using Lemma 7.7, we obtain c_p, h_p such that $c \overset{h_p}{\Longrightarrow} c_p \overset{h_2}{\Longrightarrow} c'$, $\mathsf{sched}_{\mathsf{ls}}(h_p \times \mathsf{ls}(c)) \neq \emptyset$, and $\mathsf{ls}(c_p) = \bar{\mu}_2$. By Theorem 3.25, we also have $c \rightarrow_{\mathsf{ls}}^* c_p$, thus, as c is reachable, c_p is also a reachable configuration. Moreover, in $h_2 \times \bar{\mu}_2$, each thread in D starts with an acquisition of a lock that is currently held by some other thread in D. These acquisitions correspond to acquisition-rules of M that are applicable to the threads in c_p, and as M is inescapable, there are no alternatives to these acquisition-rules. Hence, c_p is deadlocked. $\square$

The previous lemma implies that predecessor sets of inescapable, deadlock-free DPNs can be computed precisely without using release-graphs, as long as only (lock-insensitively) reachable configurations are considered.

7.3.2 No Spawn inside Monitors

In the last subsection, we have shown that we do not need a release-graph when analyzing deadlock-free, inescapable DPNs. However, as was illustrated in Example 7.5, the acquisition-graph is still required for this class of DPNs. In this

[1] Actually, cycles in the release-graph have a minimum length of two, thus at least two threads start with a use-node.

subsection, we regard Monitor-DPNs that allow spawn-operations only when the spawning thread is outside any monitor. Typically, the operations inside monitors just access some protected data. Hence, this assumption may be realistic for many programs.

We show that we do need neither acquisition- nor release-graphs for precise reachability analysis of inescapable, deadlock-free DPNs without spawn-operations inside monitors.

Lemma 7.8. *Let M be a Monitor-DPN with start configuration $c_0 \in \mathsf{Conf}^{\mathsf{ls}}$, such that spawn-operations may only be performed outside locks. Formally, we require that in each lock-a/r-hedge from c_0, there must be no path from an acquisition-node to a use-node with a non-empty list of spawned threads. Moreover, let $c = p_1w_1 \dots p_nw_n$ be a reachable lock-configuration, and $\tilde{h}$ be a lock-a/r-hedge from c, i.e., we have a lock-execution-hedge h and a configuration $c' \in \mathsf{Conf}$ such that*

$$c_0 \rightarrow_{\mathsf{ls}}^{*} c \wedge c \overset{h}{\Rightarrow} c' \wedge \tilde{h} = \mathsf{ar}(h \times \mathsf{ls}(c)).$$

Finally, assume that $\tilde{h}$ satisfies (C1) and (C2).

Then, the dependence-graph $\mathsf{g}^{\mathsf{dep}}(\tilde{h})$ of $\tilde{h}$ is acyclic.

Proof. First note that the precondition implies that lock-a/r-hedges from *any reachable* configuration, in particular $\tilde{h}$, contain no paths from acquisition-nodes to use-nodes that spawn threads.

The proof is done similar to the proof of Lemma 7.6—i.e., we assume that the dependence-graph is cyclic, and construct a schedule to a deadlocked configuration.

As in the proof of Lemma 7.6, we repeatedly schedule minimal nodes of $\mathsf{g}^{\mathsf{dep}}(\tilde{h})$ until there are no minimal nodes left, and get a schedule $\tilde{h} \rightsquigarrow^{*} \tilde{h}_1$, such that $\mathsf{g}^{\mathsf{dep}}(\tilde{h}_1)$ contains no minimal nodes, $\tilde{h}_1$ satisfies (C1) and (C2), and we have a corresponding schedule of the lock-execution-hedge, i.e., $h \times \mathsf{ls}(c) \rightsquigarrow_{\mathsf{ls}}^{*} h_1 \times \bar{\mu}_1$ and $\tilde{h}_1 = \mathsf{ar}(h_1 \times \bar{\mu}_1)$.

All non-terminated threads in $\tilde{h}_1$ start with an acquisition- or a use-node. If there is a thread starting with a use-node, we deduce the contradiction analogously to the proof of Lemma 7.6. So assume all non-terminated threads in $\tilde{h}_1$ start with an acquisition-node. Thus $\tilde{h}_1$ contains no use-nodes that spawn threads.

Let D be the set of non-terminated threads in $\tilde{h}_1$. In the proof of Lemma 7.6, any schedule of $\tilde{h}_1$ got stuck in a deadlock. In this case, however, we have to carefully construct the schedule to a deadlock.

In order to reach a deadlock, we do not need to schedule $\tilde{h}_1$ completely. We remove leaf-nodes[2] from $\tilde{h}_1$. Removing nodes may break cycles in the dependence-graph. We repeatedly remove leaf-nodes until there are no more leaf-nodes that

[2]In this context, a leaf-node is a node that has τ as its successor

can be removed without making the dependence-graph acyclic. We arrive at a lock-a/r-hedge $\tilde{h}_2$ that is a prefix of $\tilde{h}_1$. Let h_2 be the corresponding lock-execution-hedge, i.e., we have $\tilde{h}_2 = \mathsf{ar}(h_2 \times \mathsf{ls}(\bar{\mu}_1))$.

In $\tilde{h}_2$, each thread from D ends with a use-node, as all other nodes would have been removed without breaking a cycle. Let $\tilde{h}_2 = (t_1, X_1) \dots (t_n, X_n)$, and consider a thread $i \in D$. The tree t_i spawns no threads, i.e., it is actually a list. The last node is a use-node that is part of a cycle in the dependence-graph, i.e., it has an outgoing edge to some acquisition-node in a thread $j \in D$. As the use-node is non-reentrant, we have $j \neq i$. Hence, each thread in $\tilde{h}_2$ ends with a use-node that uses some lock that is finally acquired in some other thread of D.

Next, we remove all the use-nodes at the leafs of $\tilde{h}_2$, arriving at a lock-a/r-hedge $\tilde{h}_3$ with corresponding execution-hedge h_3, i.e., $\tilde{h}_3 = \mathsf{ar}(h_3 \times \mathsf{ls}(\bar{\mu}_1))$. The dependence-graph of $\tilde{h}_3$ is acyclic, i.e., it satisfies (C3). Moreover, as $\tilde{h}_1$ contains no initial release-nodes and satisfies (C1) and (C2), and $\tilde{h}_3$ contains less nodes than $\tilde{h}_1$, $\tilde{h}_3$ also satisfies (C1) and (C2). Hence, we can completely schedule $\tilde{h}_3$. Thus, also the corresponding lock-execution-hedge $h_3 \times \mathsf{ls}(\bar{\mu}_1)$ is schedulable. As h_3 is a prefix of h_1, we can combine this with the schedule $h \times \mathsf{ls}(c) \rightsquigarrow^*_{\mathsf{ls}} h_1 \times \bar{\mu}_1$, resulting in a schedule $h \times \mathsf{ls}(c) \rightsquigarrow^*_{\mathsf{ls}} h_s \times \bar{\mu}_s$ for a lock-execution-hedge $h_s \times \bar{\mu}_s$. By construction of h_3, in $h_s \times \bar{\mu}_s$, each thread $i \in D$ starts with the usage of a lock that was finally acquired by another thread $j \in D$ during the schedule of the prefix of h. Thus, in $\bar{\mu}_s$, thread j has this lock on its lockstack. Analogously to the proof of Lemma 7.6, we construct an execution of the interleaving semantics to a deadlocked configuration. □

7.3.3 Deadlock Detection

In the last subsections, we have shown that we may omit the release-graph when considering inescapable, deadlock-free DPNs. Moreover, if the DPN additionally spawns no threads while inside monitors, we may omit both, the acquisition and the release-graph.

Checking whether a DPN is inescapable is straightforward by examining the rules. Checking whether a DPN spawns no threads inside monitors is also straightforward: Whether a lock-a/r-hedge has a path from an acquisition- to a use-node that spawns threads is a regular property, and the methods of Chapter 5 can be used to check whether the original DPN has an execution with such a path.

In order to check whether a DPN is deadlock-free, we observe that the set C_{dead} of deadlocked configurations is regular. For any set S of actions, the set $\{pw \mid \mathsf{act}(pw) = S\}$ of configurations with that set of actions is regular. Whether a rule is applicable depends only on the control-state and the topmost stack-symbol. Hence, we can write the set as follows:

$$\{pw \mid \mathsf{act}(pw) = S\} = \{p\gamma\tilde{w} \mid \tilde{w} \in \Gamma^* \wedge S = \{o \mid \exists c.\ p\gamma \overset{o}{\hookrightarrow} c \in \Delta\}\}.$$

The set on the right-hand side of this equation is obviously regular. Moreover, it is sufficient for a deadlock to only regard threads that currently hold locks, plus at most one thread that does not hold a lock. Hence, for any deadlocked configuration, we find a set D such that $|D| \leq |\mathcal{X}|+1$ according to Definition 7.3. Thus, C_{dead} can be characterized as follows:

$$\begin{aligned} C_{\mathsf{dead}} = \bigcup \{ & \mathsf{Conf}^{\mathsf{ls}}\{[(p_1 w_1)]\}\mathsf{Conf}^{\mathsf{ls}} \ldots \mathsf{Conf}^{\mathsf{ls}}\{[(p_n, w_n)]\}\mathsf{Conf}^{\mathsf{ls}} \mid 0 < n \leq |\mathcal{X}| + 1 \\ & \wedge \forall 0 < i \leq n.\ (\mathsf{act}(p_i w_i) = \emptyset \wedge \mathsf{ls}(w_i) \neq \varepsilon) \\ & \quad \vee (\exists S \neq \emptyset.\ \mathsf{act}(p_i w_i) = S \\ & \qquad \wedge \forall o \in S.\ \exists 0 < j \leq n, x \in \mathsf{ls}(w_j).\ i \neq j \wedge o = \langle_x) \\ & \wedge \exists 0 < i \leq n.\ \mathsf{act}(p_i w_i) \neq \emptyset \} \cap \mathsf{Conf}^{\mathsf{ls}}. \end{aligned}$$

The set on the right-hand side is obviously regular, and we can decide whether a DPN with start configuration c_0 is deadlock-free, by deciding

$$c_0 \notin \mathsf{pre}^*_{\mathsf{ls}}(C_{\mathsf{dead}}).$$

However, an automaton for C_{dead} has exponential size in the number of locks. Thus, a precise deadlock analysis may not be worth the effort. However, there are sufficient criteria for a DPN to be deadlock-free that are simpler to check, e.g. global lock ordering.

Global lock ordering is a well-known criterion for ensuring absence of deadlocks. We adapt this notion to Monitor-DPNs here: We define a relation $< \subseteq \mathcal{X} \times \mathcal{X}$, such that $x < y$ iff there is an execution from the start configuration in that a thread acquires y while holding x. If $<$ is acyclic, the DPN adheres to a *global lock ordering.*

Lemma 7.9. *If a Monitor-DPN M has a global lock ordering, and each thread that is inside a monitor has a possible action (i.e., threads may not terminate while inside monitors), then M is deadlock-free.*

Proof. Assume the DPN reaches a deadlocked configuration. W.l.o.g. let this configuration be $c = p_1 w_1 \ldots p_n w_n$. By definition of a deadlock (cf. Definition 7.3), we obtain a non-empty set $D \subseteq \{1, \ldots, n\}$, such that

$$\forall i \in D.\ \mathsf{act}(p_i w_i) \neq \emptyset \wedge \forall o \in \mathsf{act}(p_i w_i).\ \exists j \in D, x \in \mathsf{ls}(w_j).\ i \neq j \wedge o = \langle_x.$$

Note that, by assumption, there are no terminated threads that hold locks. Thus we omitted the case $\mathsf{act}(p_i w_i) = \emptyset \wedge \mathsf{ls}(w_i) \neq \varepsilon$.

We now construct a cycle in $<$. If we remove the threads that hold no locks from D, the reduced set D still satisfies the deadlock condition. (Note that the existential quantification $\exists j \in D$ only considers threads that hold at least one lock.) Moreover, D contains at least one thread that holds a lock. Thus, we may safely assume that all threads in D hold at least one lock. In order to construct the cycle, we start with some thread $i_1 \in D$. By definition of $<$, we have $x_1 < x_2$ for each lock $x_1 \in \mathsf{ls}(w_{i_1})$ and x_2 with $\langle_{x_2} \in \mathsf{act}(p_{i_1} w_{i_1})$. As both sets are not empty, we can pick such a pair x_1, x_2, and find a thread $i_2 \neq i_1$ with $x_2 \in \mathsf{ls}(w_{i_2})$. Continuing this argumentation inductively, we get a chain $x_1 < x_2 < x_3 < \ldots$. However, as there are only finitely many locks, this implies a cycle in $<$. □

Note that the other direction of Lemma 7.9 is not true in general, i.e., there are deadlock-free programs that do not adhere to a global lock ordering. For example, consider the following program:

```
p:  spawn q          q:  acq x
    acq x                acq z
    acq y                acq y
    acq z                rel y
    rel z                rel z
    rel y                rel x
    rel x
```

Clearly, we have $y < z < y$. However, the program has no deadlock: Intuitively, the lock x prevents that two threads simultaneously reach the location where $<$ is cyclic. Thus, a global lock ordering is a sufficient, but not a necessary criterion for absence of deadlocks.

Checking whether each (lock-insensitive) execution of a Monitor-DPN adheres to a global lock ordering, as well as checking whether threads may terminate while holding locks, can be done by a straightforward, polynomial time analysis. Such an analysis could be done before the actual predecessor set computation, in order to determine whether the DPN meets the criteria to omit the release- and/or acquisition-graph.

7.4 Example

In this section, we resume the example from the introduction (Example 1.1). Although the scope of this thesis is limited to the analysis of Monitor-DPNs, this example covers all steps from the Java program to the actual predecessor set computation. However, the abstraction step from the Java program to the Monitor-DPN is done with an ad-hoc technique.

We start with the following program written in Java [47]:

```
public class Example implements Runnable {
  private static void write_terminal(String s) {
    for (int i=0;i<s.length();++i) {
      System.out.print(s.charAt(i));
      try {
        Thread.yield();
      } catch (Exception e) {}
    }
    System.out.flush();
  }

  private synchronized void write(String s) {
    g5: write_terminal(s);
    g6: {}
  }

  @Override
  public void run() {
    g3: write("Hello");
    g4: {}
  }

  public static void main(String args[]) {
    g0: new Thread(new Example()).start();
    g1: new Thread(new Example()).start();
    g2: {}
  }
}
```

Note that the labels `g0,g1,...` have been added to illustrate the connection of this program and its translation to a DPN that is presented below. The program resembles the one from Example 1.1, except that we have instrumented the `write_terminal()` method with a call to `Thread.yield()`, in order to increase the probability that the concurrency error manifests itself.

The main method of this program spawns two instances of the `Example`-thread. The `run`-method of this thread calls the `write`-method. This is a synchronized method that accesses the terminal to print a string. The access to the terminal is done by the `write_terminal`-method that models an unsynchronized access to the terminal. It may be interrupted after each letter is printed. Additionally, we hint the scheduler to actually interrupt the thread after each letter, using the `yield`-method. This increases the probability that the data-race actually manifests itself in erroneous output. Probably, the programmer expected this program to output "HelloHello". However, when running the program a few times, one observes different outputs. On the author's machine, for example, the most frequent output was "HHeelllloo", and sometimes, the author got "HHelloello", or even the correct output "HelloHello". Obviously, the above program contains a concurrency error. Deeper inspection of the program quickly reveals the error: The `write`-method is synchronized, however it synchronizes on the

`Example`-object of the current thread. As each thread has its own instance of `Example`, the two threads synchronize `write` on two different monitors, resulting in a data-race on the write-access to the terminal. Note that this concurrency error occurs at a very small probability, when omitting the call to `yield()` after output of each letter, and may easily be missed by testing. Thus, this program is a very simple example that motivates the use of automatic verification methods for concurrent programs. Moreover, it is well-suited to illustrate our analysis, point out possible implementation issues, and propose solutions to them, while keeping the intermediate results at a suitable size for manual calculation.

The first step of the analysis is the translation of the Java program to a Monitor-DPN. We abstract from the `write_terminal`-method, and replace it by a base-action labeled w (for write-access). We construct the Monitor-DPN

$$M = (P, \Gamma, \Gamma_\perp, \mathsf{Act}, \mathcal{X}, \Delta, \mathsf{locks})$$

with

$$\begin{aligned}
P &= \{p\} & \mathsf{Act} &= \{a, w\} \\
\Gamma &= \{\gamma_5^1, \gamma_5^2, \gamma_6^1, \gamma_6^2\} \cup \Gamma_\perp & \Gamma_\perp &= \{\gamma_0, \gamma_1, \gamma_2, \gamma_3^1, \gamma_3^2, \gamma_4^1, \gamma_4^2\} \\
\mathcal{X} &= \{x^1, x^2\} \\
\mathsf{locks} &= \gamma_0, \gamma_1, \gamma_2, \gamma_3^1, \gamma_3^2, \gamma_4^1, \gamma_4^2 \mapsto \emptyset, \quad \gamma_5^1, \gamma_6^1 \mapsto x^1, \quad \gamma_5^2, \gamma_6^2 \mapsto x^2
\end{aligned}$$

and the rules:

$$\begin{aligned}
&p\gamma_0 \overset{\Box_a}{\hookrightarrow} p\gamma_3^1 \sharp p\gamma_1 && p\gamma_1 \overset{\Box_a}{\hookrightarrow} p\gamma_3^2 \sharp p\gamma_2 && \text{(main)} \\
&p\gamma_3^1 \overset{\langle_{x^1}}{\hookrightarrow} p\gamma_5^1\gamma_4^1 && p\gamma_3^2 \overset{\langle_{x^2}}{\hookrightarrow} p\gamma_5^2\gamma_4^2 && (\text{run}^1 \text{ and run}^2) \\
&p\gamma_5^1 \overset{\Box_w}{\hookrightarrow} p\gamma_6^1 && p\gamma_6^1 \overset{\rangle_{x^1}}{\hookrightarrow} p && (\text{write}^1) \\
&p\gamma_5^2 \overset{\Box_w}{\hookrightarrow} p\gamma_6^2 && p\gamma_6^2 \overset{\rangle_{x^2}}{\hookrightarrow} p && (\text{write}^2)
\end{aligned}$$

The stack-symbols $\gamma_0, \gamma_1, \ldots$ correspond to the labels `g0,g1,`... in the Java program. The run methods and the main method have no return-rules, thus matching our constraint that the bottommost stack-symbol must not be popped. The two instances of the `Example`-class are resolved by duplication of the methods and locks. We added the superscripts $\cdot^1$ and $\cdot^2$ to the affected stack-symbols and locks. For the spawn-rules, we introduced a dummy action a that has no further meaning, but is syntactically required, as any rule must have a label. One can easily verify that M is a Monitor-DPN.

In general, translation of arbitrary Java programs to Monitor-DPNs is not as easy as for this simple example. A major issue is pointer analysis, and its implications for locks: While DPNs only support boundedly many directly addressed locks, Java programs may have unboundedly many locks that are addressed by reference. When abstracting a Java program to a Monitor-DPN, a monitor from Java can only be translated to a monitor in the DPN if pointer analysis has not summarized different objects. If, for example, pointer analysis would summarize the two instances of the `Example`-object in our program, we could not identify the summarized instance with a single lock, as the translated DPN would have no data-race any more, and the abstraction would be unsound. General methods for abstraction of concurrent Java programs to Monitor-DPNs are beyond the scope of this thesis.

The next step of our analysis is to specify the error condition to be checked, and to translate it to a predecessor set computation. We want to check whether more than one thread can simultaneously call `write_terminal`. On the DPN, this translates to checking whether more than one thread can do a $\Box_w$-action simultaneously. As only rules from $p\gamma_5^1$ and $p\gamma_5^2$ may perform a $\Box_w$-action, this is equivalent to checking whether a configuration from the regular set

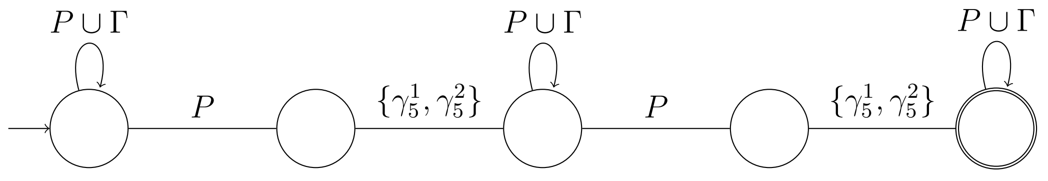
$$C := (P \cup \Gamma)^* P\{\gamma_5^1, \gamma_5^2\}(P \cup \Gamma)^* P\{\gamma_5^1, \gamma_5^2\}(P \cup \Gamma)^*$$

is reachable from the start configuration $p\gamma_0$. Hence, the DPN cannot reach two $\Box_w$-actions simultaneously, if and only if

$$\mathsf{pre}^*_{\mathsf{ls}}(C) \cap p\gamma_0 = \emptyset.$$

In general the abstraction from the Java program to the DPN may introduce spurious executions. Thus, if the above statement holds, the Java program definitely cannot call `write_terminal` simultaneously from two threads. However, if the above statement does not hold, this only indicates a *possible* error: Either, the Java program has a real error, or the error is due to some spurious execution introduced by the abstraction. Note that, in our simple example, the abstraction is precise, i.e., it introduces no spurious executions.

The set C is accepted by the following automaton A:



As we check for reachability from the start configuration, the optimizations sketched in Section 7.2 can be applied: We may omit the release-set, the release-graph, and the set of initially held locks, and we need not intersect the result with $\mathsf{Conf}^{\mathsf{ls}}$. Additionally, we may omit the initial-release flag in $D_{\mathsf{A_{Cons}}}$.

Following the construction from the proof of Theorem 6.5, we first construct the automaton $\mathsf{A_{Cons}}$ on lock-a/r-hedges. Then, we translate $\mathsf{A_{Cons}}$ to the DPN-Acceptor $D_{\mathsf{A_{Cons}}}$, which has the components $D_{\mathsf{A_{Cons}}} = (\mathsf{A}_{C_\mathsf{I}}, \mathsf{A}_{C_\mathsf{F}}, M_2)$ with $M_2 =$

$(P_2, \Gamma_2, \Gamma_{\perp,2}, \mathsf{Act}, \mathcal{X}, \Delta_2, \mathsf{locks}_2)$. Next, we do a cross-product construction between M and $D_{\mathsf{A}_{\mathsf{Cons}}}$, yielding the DPN $M_\times = (P_\times, \Gamma_\times, \mathsf{Act}_\mathcal{X}, \Delta_\times)$, automata $\mathsf{A}_{C_{\mathsf{I}\times}}$ and $\mathsf{A}_{C_{\mathsf{F}\times}}$, and the projections π_1, π_2. Finally, we compute the automaton

$$\mathsf{pre}'^*_{\mathsf{ls}}(A) = \pi_1(\mathsf{pre}^*_\times(\pi_1^{-1}(A) \cap \mathsf{A}_{C_{\mathsf{F}\times}}) \cap \mathsf{A}_{\mathsf{valid}_\times} \cap \pi_2^{-1}(\mathsf{A}_{C_\mathsf{I}})),$$

and check $p_0\gamma_0 \in \mathsf{pre}'^*_{\mathsf{ls}}(A)$.

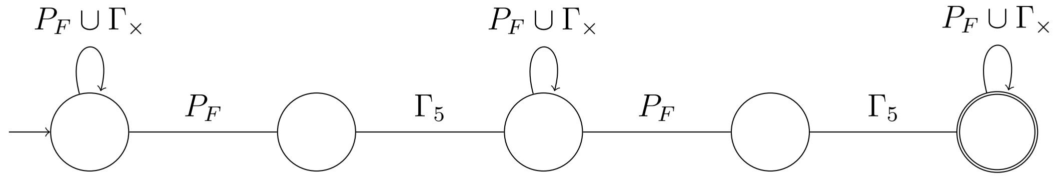
The control-symbols in $P_\times$ have the form

$$(p, (\mathsf{p}, X, (u, a, g_a))) \text{ or } (p, (\mathsf{u}^{(\tilde{u},\tilde{a},\tilde{g}_a)}, X, Y, (u, a, g_a))),$$

where p is the single control-state of M, X is the set of currently acquired locks, Y is the set that accumulates used locks[3], (u, a, g_a) and $(\tilde{u}, \tilde{a}, \tilde{g}_a)$ are acquisition structures without release-sets and release-graphs. In order to make the presentation more readable, we omit the control-state p of M, and just write $(\mathsf{p}, X, (u, a, g_a))$ and $(\mathsf{u}^{(\tilde{u},\tilde{a},\tilde{g}_a)}, X, Y, (u, a, g_a))$.

The stack-symbols in $\Gamma_\times$ have the form (γ, x^b) with $\gamma \in \Gamma$ and $\mathsf{locks}(\gamma) = \{x\}$ or $(\gamma_\perp, \perp)$ with $\gamma_\perp \in \Gamma_\perp$. Note that, in our special case, all symbols are either bottom stack-symbols or are associated with a lock, such that there are no stack-symbols γ with $\gamma \in \Gamma \setminus \Gamma_\perp$ and $\mathsf{locks}(\gamma) = \emptyset$.

First, we construct the automaton $\pi_1^{-1}(A) \cap \mathsf{A}_{C_{\mathsf{F}\times}} = \pi_1^{-1}(A) \cap \pi_2^{-1}(\mathsf{A}_{C_\mathsf{F}})$. It has the following graph:

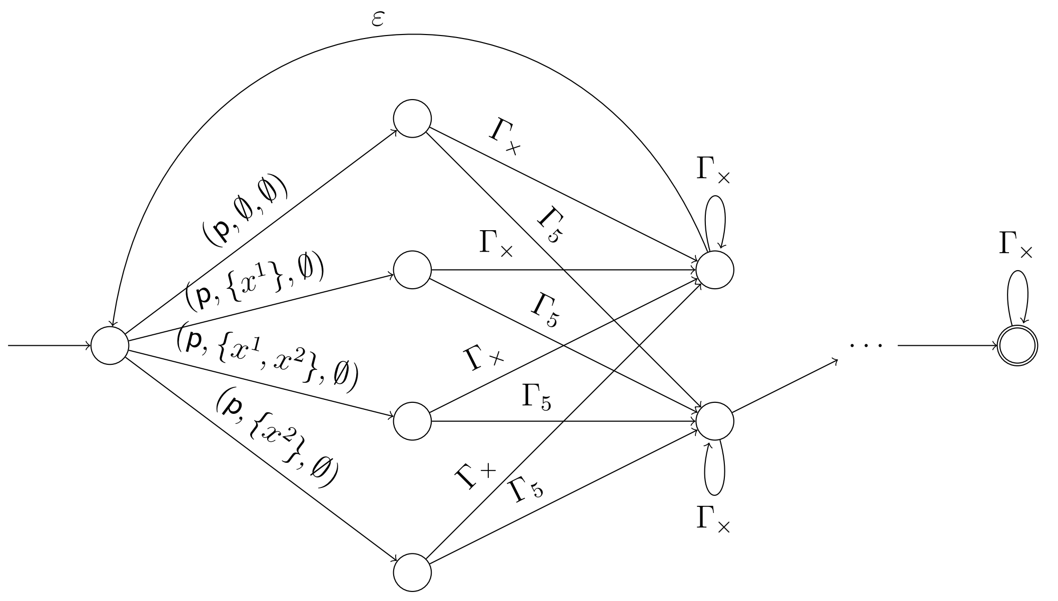


Here, we use the abbreviations

$$\begin{aligned} P_F &:= \{(\mathsf{p}, X, \emptyset^3) \mid X \subseteq \mathcal{X}\}, \\ \Gamma_5 &:= \{(\gamma_5^i, x^{i,b}) \mid i \in \{1,2\}, b \in \mathbb{B}\}, \text{ and} \\ \emptyset^3 &:= (\emptyset, \emptyset, \emptyset). \end{aligned}$$

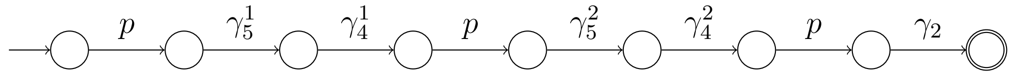
Next, we convert this automaton to an M-automaton. This is achieved by duplicating some states. Note that, after this conversion, words that do not represent configurations (i.e., that do not start with a control-symbol) are not accepted any more. The resulting M-automaton has the following shape:

[3]In Definition 5.2, this set is named u. However, this name would clash with the u-component of the acquisition structure.



This reveals a problem: Due to the M-automata condition, we have to create a separate node for each control-state. For edges not present in the automaton, we may omit these nodes and add them on demand during the saturation. However, for edges present in the automaton, we have to actually create the separate states. This results in an exponential blowup in the number of locks, as we have to add a state for every possible set of allocated locks. Note that the automaton accepts configurations with arbitrary stack, thus all sets of locks may actually occur on some stack.

However, when inspecting our example DPN more deeply, we observe that not all of these configurations are reachable, even lock-insensitively. Actually, only a single configuration in $\mathsf{L}(A)$ is reachable lock-insensitively: $p\gamma_5^1\gamma_4^1p\gamma_5^2\gamma_4^2p\gamma_2$. Using this information, we transform A to the following automaton A':



The automaton for $\pi_1^{-1}(A') \cap \pi_2^{-1}(\mathsf{A}_{C_\mathsf{F}})$ is constructed accordingly. However, we do not include configurations where the lockstack does not match the set of acquired locks stored in the DPNs control-state. This is a correct optimization, as $\mathsf{A}_{C_\mathsf{I}}$ ensures that the locks match the stored sets in the initial configuration, and this property is invariant under transitions of M_2. As our automaton A' accepts only one configuration, there is only one set of locks for each control-state, and we get the following automaton:

$$\begin{tikzcd}
{} \arrow[r] & q_1 \arrow[r, "(\mathsf{p}, \{x^1\}, \emptyset^3)"] & q_2 \arrow[r, "(\gamma_5^1, x^{1,\perp})"] & q_3 \arrow[r, "(\gamma_4^1, \perp)"] & q_4 \arrow[dlll, "\varepsilon"'] \\
& q_5 \arrow[r, "(\mathsf{p}, \{x^2\}, \emptyset^3)"] & q_6 \arrow[r, "(\gamma_5^2, x^{2,\perp})"] & q_7 \arrow[r, "(\gamma_4^2, \perp)"] & q_8 \arrow[dlll, "\varepsilon"'] \\
& q_9 \arrow[r, "(\mathsf{p}, \emptyset, \emptyset^3)"] & q_{10} \arrow[r, "(\gamma_2, \perp)"] & \boxed{q_{11}} &
\end{tikzcd}$$

Note that we already introduced the ε-transitions necessary to make this automaton an M-automaton. We now apply the saturation algorithm. For this, we show how to instantiate the rule-templates of $\mathsf{A}_{\mathsf{Cons}}$ (Definition 4.19), $D_{\mathsf{A}_{\mathsf{Cons}}}$ (Definition 5.2), and the cross-product construction (Section 5.2) to obtain the rules of $M_\times$ that are used for the saturation.

1. Regard the rule
$$p\gamma_3^1 \overset{\langle_{x^1}}{\hookrightarrow} p\gamma_5^1\gamma_4^1 \in \Delta$$
and the rule
$$(\mathsf{p}, \emptyset, \mathsf{as}_{\mathsf{t}}^{\langle}(x^1, \emptyset^3))\bot \overset{\langle_{x^1}}{\hookrightarrow} (\mathsf{p}, \{x^1\} \cup \emptyset, \emptyset^3)x^{1,\perp}\bot \in \Delta_2.$$
The latter one is obtained by instantiating the (acq-n)-rule of $D_{\mathsf{A}_{\mathsf{Cons}}}$ with the acquisition-rule of $\mathsf{A}_{\mathsf{Cons}}$, the empty set of acquired locks, and the empty acquisition structure $\emptyset^3$. These rules are paired by the (acq)-rule of the cross-product construction, yielding the rule
$$(\mathsf{p}, \emptyset, (\emptyset, \{x^1\}, \emptyset))(\gamma_3^1, \bot) \overset{\langle_{x^1}}{\hookrightarrow} (\mathsf{p}, \{x^1\}, \emptyset^3)(\gamma_5^1, x^{1,\perp})(\gamma_4^1, \bot).$$
The right-hand side of this rule is recognized by the automaton from state q_1 to q_4. Hence, the saturation algorithm adds a shortcut using the left-hand side. As there is no transition from q_1 for the control-state $(\mathsf{p}, \emptyset, (\emptyset, \{x^1\}, \emptyset))$ yet, we create a state q_2^1, and add the transitions
$$q_1 \xrightarrow{(\mathsf{p},\emptyset,(\emptyset,\{x^1\},\emptyset))} q_2^1 \text{ and } q_2^1 \xrightarrow{(\gamma_3^1,\bot)} q_4$$
to the automaton.

2. Similar as in Step 1, we create a state q_6^1 and add the transitions
$$q_5 \xrightarrow{(\mathsf{p},\emptyset,(\emptyset,\{x^2\},\emptyset))} q_6^1 \text{ and } q_6^1 \xrightarrow{(\gamma_3^2,\bot)} q_8.$$

3. Next, regard the rule
$$p\gamma_1 \overset{\Box_a}{\hookrightarrow} p\gamma_3^2 \sharp p\gamma_2 \in M$$
and the rule
$$(\mathsf{p}, \emptyset, (\emptyset, \{x^2\}, \emptyset))\bot \overset{\Box_a}{\hookrightarrow} (\mathsf{p}, \emptyset, (\emptyset, \{x^2\}, \emptyset))\bot\sharp(\mathsf{p}, \emptyset, \emptyset^3)\bot \in D_{\mathsf{A_{Cons}}},$$
that is obtained by instantiating the (spawn)-rule of $D_{\mathsf{A_{Cons}}}$ with
$$\langle\rangle_\emptyset([(\emptyset, \{x^2\}, \emptyset)])\emptyset^3 \rightarrow^*_{\mathsf{A_{Cons}}} (\emptyset, \{x^2\}, \emptyset),$$
which, in turn, is easily derived from the rules of $\mathsf{A_{Cons}}$. The (spawn)-rule of the cross-product construction pairs these rules, yielding
$$(\mathsf{p}, \emptyset, (\emptyset, \{x^2\}, \emptyset))(\gamma_1, \bot) \overset{\Box_a}{\hookrightarrow} (\mathsf{p}, \emptyset, (\emptyset, \{x^2\}, \emptyset))(\gamma_3^2, \bot)\sharp(\mathsf{p}, \emptyset, \emptyset^3)(\gamma_2, \bot).$$
The right-hand side of this rule is recognized by the automaton, using the sequence of states $q_5 q_6^1 q_8 q_9 q_{10} q_{11}$. Hence, we have to add the left-hand side as a shortcut from state q_5 to q_{11}. There is already the state q_6^1 for the control-symbol $(\mathsf{p}, \emptyset, (\emptyset, \{x^2\}, \emptyset))$, hence we only add the transition
$$q_6^1 \xrightarrow{(\gamma_1, \bot)} q_{11}.$$

4. Similar to Step 3, we add the state q_2^2 and the transitions
$$q_1 \xrightarrow{(\mathsf{p}, \emptyset, (\emptyset, \{x^1, x^2\}, \emptyset))} q_2^2 \text{ and } q_2^2 \xrightarrow{(\gamma_0, \bot)} q_{11}$$
to the automaton.

At this point, the saturation is finished. We call the saturated automaton $\mathsf{A_{sat}}$. Next, we have to compute
$$p_0\gamma_0 \in \mathsf{L}(\pi_1(\mathsf{A_{sat}} \cap \mathsf{A_{valid_\times}} \cap \pi_2^{-1}(\mathsf{A}_{C_\mathsf{I}}))).$$
This is equivalent to
$$\pi_1^{-1}(p\gamma_0) \cap \mathsf{A_{sat}} \cap \mathsf{A_{valid_\times}} \cap \pi_2^{-1}(\mathsf{A}_{C_\mathsf{I}}) \neq \emptyset.$$
Intersection of $\mathsf{A_{sat}}$ with $\pi_1^{-1}(p\gamma_0)$ yields the following automaton:

$$\begin{tikzcd}
{} \arrow[r] & q_1 \arrow[r, "{(\mathsf{p}, \emptyset, (\emptyset, \{x^1, x^2\}, \emptyset))}"] & q_2^2 \arrow[r, "{(\gamma_0, \bot)}"] & q_{11}
\end{tikzcd}$$

Intersection with $\mathsf{A}_{\mathsf{valid}_\times}$ does not change this automaton. So it remains to intersect the automaton with $\pi_2^{-1}(\mathsf{A}_{C_\mathsf{I}})$. The automaton $\mathsf{A}_{C_\mathsf{I}}$ (Definition 5.2) computes, within its states, the hedge acquisition structure. It only accepts configurations with consistent hedge acquisition structures.

However, in our setting, the stack-symbol γ_0 of the initial configuration has no locks. Moreover, the initial configuration has just one thread. Thus, intersection with $\pi_2^{-1}(\mathsf{A}_{C_\mathsf{I}})$ can be done by removing all edges that are labeled with cyclic acquisition-graphs. The acquisition-graph of $(\mathsf{p}, \emptyset, (\emptyset, \{x^1, x^2\}, \emptyset))$ is empty, and thus acyclic. Hence, intersection with $\pi_2^{-1}(\mathsf{A}_{C_\mathsf{I}})$ also does not change the automaton.

The language of the automaton is obviously not empty, and thus a data-race can be reached from the initial configuration. When adding some bookkeeping functionality to the saturation algorithm, one can reconstruct the executions associated with each configuration in the saturated automaton [111, 121]. This could be used to construct an error trace.

7.4.1 Discussion

In this section, we have illustrated—with the help of a very simple example—how our model-checking algorithm works. We performed some ad-hoc optimizations that made the analysis simpler. We now comment on these optimizations and their general applicability.

The first optimizations (i.e., omitting the release-set, the release-graph, the initial-release flag, the intersection of the result with $\mathsf{Conf}^{\mathsf{ls}}$, and the set of initially held locks) have been discussed in Section 7.2. They can always be applied for non-iterated reachability queries from a start configuration that holds no locks, make the involved data structures simpler, and avoid unnecessary blowup when constructing the M-automaton for $\pi_1^{-1}(A) \cap \pi_2^{-1}(\mathsf{A}_{C_\mathsf{F}})$. Moreover, if the start configuration consists of just a single process, intersection of the saturated automaton with $\pi_2^{-1}(\mathsf{A}_{C_\mathsf{I}})$ reduces to removing edges labeled with cyclic acquisition-graphs.

Note that the example program is deadlock-free, as its global lock ordering relation is empty. Moreover, the locks are inescapable and threads are not spawned inside monitors. Thus, we could also have omitted the acquisition-graph (cf. Subsection 7.3).

The automaton $\pi_2^{-1}(\mathsf{A}_{C_\mathsf{F}})$ admits exponentially many control-states in the number of locks. A crucial optimization was to additionally intersect the automaton $\pi_1^{-1}(A) \cap \pi_2^{-1}(\mathsf{A}_{C_\mathsf{F}})$ with the set of those configurations where the set of locks stored in the control-state matches the actual set of locks on the stack. This optimization does not work if A, and thus $\pi_1^{-1}(A)$, contains all possible stacks, as it is the case in our example. Thus, we used forward reachability information to intersect A with the lock-insensitively reachable configurations. In our case, this reduced the number of configurations accepted by A to a single configuration. However, lock-sensitive successor sets of regular sets (and even of singleton sets) are, in general, not regular [16]. However, they are still context-free [16]. Thus, intersecting A with the set of reachable configurations cannot be done precisely, but regular approximations of the precise context-free set may be used.

A simpler alternative would be to compute an over-approximation of the reachable locksets by a simple static analysis. When constructing $\mathsf{A}_{C_\mathsf{F}}$, only reachable locksets have to be considered.

The above optimization tackles a fundamental problem when implementing our analysis: The predecessor set computation explores all states that are backward-reachable from the target configurations. These typically include many states that are not forward reachable. Computing the acquisition histories and locksets for all those unreachable states may cause a significant blowup—in our case, we had to explore stacks with exponentially many different (unreachable) locksets. The optimizations try to restrict the states that are explored to forward reachable states.

Another crucial optimization was the demand driven construction of the rules of $M_\times$. There are exponentially many rules in $M_\times$, but, in our example, only few of them are actually needed for the saturation. We expect that the set of acquisition histories belonging to actual executions of the program is quite small in typical programs. Thus, for typical programs, it should be profitable to generate the rules of $M_\times$ on demand, using symbolic techniques to compactly represent $M_\times$.

The last optimization was to do the intersection of the saturated automaton with the start configuration as early as possible. This reduced the number of states in the automaton. This optimization is profitable for simple sets of start configurations, where intersection is expected to reduce the number of edges and states. Typical start configurations consists of just a single control-state and a single stack-symbol, and thus are suited for this optimization.

7.5 Summary and Related Work

In this chapter, we have discussed optimizations of the lock-sensitive predecessor set computation and presented an example how our analysis finds a data-race.

In Section 7.2, we presented some simple optimizations for reachability queries from the start configuration. In Section 7.3, we discussed the relation between deadlocks and the acquisition and release-graph, with the result that acquisition and release-graphs are not required for inescapable, deadlock-free DPNs without spawn-operations inside monitors. A preliminary result of this kind has been developed in our group quite early [111]. There, inescapable, deadlock-free DPNs where all spawn-operations are performed before the first monitor-operation are regarded. Our criterion (i.e., no spawn-operations inside monitors) is a generalization of this criterion. In Section 7.4, we presented an example of our analysis and indicated problems that have to be solved by an implementation of the analysis. Despite the worst-case exponential blowup in the number of locks, acquisition history based methods have been successfully implemented [39, 40, 57, 61, 63].

Our optimizations aim at reducing the unreachable executions for that acquisition structures are computed. Weighted DPNs [120, 121] are DPNs where the edges are annotated with weights. Wenner [120, 121] presents an extended saturation algorithm for predecessor set computation that yields the automaton for the predecessor set, and, additionally, a constraint system that associates edges of the automaton with weights. Using this constraint system, one can compute the least upper bound of the weights of all executions between two regular sets of configurations. There is evidence that sets of acquisition structures can be used as weights. For reachability analysis, the solution of the generated constraint system needs only to be computed for the start configuration, thus avoiding the expensive computation of acquisition structures for unreachable states. An elaboration and evaluation of this approach is left to future research.

Another approach, which is complementary to predecessor set computation, is successor set computation via *regular execution-trees* [44, 74]. While the set of execution-trees of a DPN is not regular in general, one can add more structure to execution-trees such that this set becomes regular. The idea is to distinguish whether a pushed stack-symbol is popped during the execution or not. In the former case, the push-rule is associated to a binary node in the regular execution-tree, where the left successor describes the execution up to the matching pop-rule, and the right successor describes the remaining execution after the pop-rule. From the rules of a DPN, a tree automaton that accepts the regular execution-trees of the DPN can be efficiently constructed. Also the set of regular execution-trees that have a lock-sensitive schedule is regular, as well as the set of regular execution-trees that reach a configuration accepted by a given automaton. Thus, lock-sensitive reachability analysis can be implemented by an emptiness check of the intersection of three tree automata. There is evidence that this emptiness check can be implemented in a way such that only acquisition structures for reachable executions are generated, thus avoiding the blowup that we encountered for predecessor set computation. While we have already described lock-sensitive reachability analysis via regular execution-trees [44] (even for Join-Lock-DPNs), an implementation of this method and its extension to more powerful analyses (e.g. bitvector-analysis and atomic-set serializability violation detection) is subject of current research.

The problem of abstracting Java programs to DPNs is out of scope of this thesis. From our point of view, the most interesting work in this direction is the random isolation method [62], that can be used to abstract concurrent Java programs to parallel pushdown systems with monitors. Applied on the ConTest benchmark suite by Eytani et al. [38], this approach yields promising results [59, 61, 63] in combination with acquisition history based methods. Leveraging the random isolation method to abstract to Monitor-DPNs is left to future research.

8 Non-Monitor Locking Disciplines

In the last chapters, we described lock-sensitive predecessor set computation for Monitor-DPNs. In this chapter, we briefly discuss other locking disciplines.

Restricting analyses to Monitor-DPNs can be justified by the fact that common high-level programming languages (e.g. Java [47]) use monitors rather than arbitrary locking.[1]

However, we are also able to analyze DPNs that use locks not adhering to a monitor-discipline. In this case, we have to restrict to well-nested, non-reentrant locks: If locks are used in a non-well-nested fashion, reachability problems become undecidable, even for non-reentrant locks [57]. For DPNs with well-nested, reentrant locks, we do not know whether reachability properties are decidable. We discuss this problem in Section 8.1. In Section 8.2, we briefly discuss how to modify the methods of this thesis such that they apply to DPNs with well-nested, non-reentrant locks. Moreover, we sketch a polynomial time algorithm to verify that a given DPN uses locks only in a well-nested, non-reentrant fashion. Finally, in Section 8.3, we briefly summarize the results of this chapter and discuss related work.

8.1 Well-Nested, Reentrant Locks

We do not know whether reachability properties of DPNs with well-nested, reentrant locks are decidable. The cross-product construction (cf. Section 5.2) heavily relies on the fact that the lockstack and the callstack of a Monitor-DPN are dependent: The locks are bound to stack-symbols, and thus the lockstack can be extracted from the callstack by projecting the stack-symbols to their associated locks, as done by the ls-function. However, if locks may be acquired and released independently from the callstack, we do not have this dependence any more: The stack of the DPN M to be analyzed and the stack of the DPN-Acceptor $D_{\mathsf{A}_{\mathsf{Cons}}}$, which accepts schedulable lock-execution-hedges, are independent. Thus, the cross-product construction would have to intersect two DPNs with independent stacks. However, as DPNs are a more general model than pushdown systems,

[1]Since Java version 5, the possibility for arbitrary locking has been introduced by the package `java.util.concurrent.locks` However, monitors are more tightly integrated within the syntax of Java (`synchronized`-keyword), while locks are realized as library functions.

and already the intersection of pushdown systems is not effective, intersection of DPNs is not effective, too.

On the other hand, we cannot immediately derive an undecidability result from this observation, as we are not forced to use the method of this thesis: There may exist other methods for reachability analysis that do not have this problem. We have not been able to obtain undecidability results for reachability properties of DPNs with well-nested, reentrant locks, and leave this problem open for future research.

8.2 Well-Nested, Non-Reentrant Locks

In the last section, we sketched that our method does not transfer to well-nested, reentrant locks. However, it does transfer to well-nested, *non-reentrant* locks.

In this section, we briefly sketch how to adapt our lock-sensitive predecessor set computation to DPNs with well-nested, non-reentrant locks (*Lock-DPNs*). As this thesis is focused on reentrant monitors, we remain on an intuitive level here, omitting formal definitions. For a complete description of an acquisition structure based predecessor set computation for Lock-DPNs, we refer to our previous work [79]. As an additional contribution over [79], we sketch a polynomial algorithm that checks whether a DPN uses locks only in a well-nested, non-reentrant fashion. Such an algorithm is important to check that the DPN to be analyzed is actually well-nested and non-reentrant, as our predecessor set computation is unsound for DPNs that violate the well-nestedness or non-reentrance assumption.

The interleaving semantics of Lock-DPNs can be defined as a labeled transition system on configurations paired with lockstacks. We have to explicitly keep track of lockstacks, while, for Monitor-DPNs, the lockstacks are encoded into the callstacks. The tree-semantics, as well as lock-a/r-hedges and the definition of $\mathsf{A}_{\mathsf{Cons}}$, remain the same. Also, the construction of the DPN-Acceptor $D_{\mathsf{A}_{\mathsf{Cons}}}$ (cf. Section 5.1.2) does not change. However, the heights of the stacks used by $D_{\mathsf{A}_{\mathsf{Cons}}}$ are bounded by the number of locks. Thus, the stacks can be encoded into the control-states of $D_{\mathsf{A}_{\mathsf{Cons}}}$, such that the DPN of $D_{\mathsf{A}_{\mathsf{Cons}}}$ effectively becomes a tree automaton. We call the resulting structure a *stackless DPN-Acceptor*. In [79], we describe a cross-product construction between a *hedge automaton* and a DPN. It is straightforward to adapt this construction to cross-products of Lock-DPNs and stackless DPN-Acceptors. Then, like for Monitor-DPNs, lock-sensitive predecessor set computation is reduced to lock-insensitive predecessor set computation on the cross-product DPN.

As there are more than $|\mathcal{X}|!$ different non-reentrant lockstacks for the set $\mathcal{X}$ of locks, the hedge automaton for $D_{\mathsf{A}_{\mathsf{Cons}}}$ has more than $|\mathcal{X}|!$ states. However, if we assume that every execution of the DPN uses locks only in a well-nested, non-reentrant fashion, the lockstacks in the configurations can be replaced by locksets. Also for $D_{\mathsf{A}_{\mathsf{Cons}}}$ (cf. Definition 5.2), we do not require the lockstacks to eliminate reentrance any more, nor to exclude non-well-nested executions. The summarization of usages, which is implemented by using flags on the lockstack, can also be realized by storing the outermost lock of the usage in the control-state. This way, the number of states of $D_{\mathsf{A}_{\mathsf{Cons}}}$ can be reduced to $2^{O(|\mathcal{X}|^2)}$—the same bound as for monitors.

Checking that all executions of a given DPN use locks only in a well-nested, non-reentrant fashion can be done in polynomial time. In the remainder of this section, we present a polynomial time algorithm to verify that all executions of a given DPN use locks only in a well-nested, non-reentrant fashion. It can be used prior to the predecessor set computation, in order to refuse to analyze a DPN that does not satisfy the well-nestedness or non-reentrance assumption.

Given a DPN M with locks $\mathcal{X}$, and a start configuration $p_0\gamma_0$ that holds no locks, we check whether every execution-tree from the start configuration is non-reentrant and well-nested. The idea is to do the check separately for each lock. Given a lock $x \in \mathcal{X}$, we construct an automaton A_x that accepts execution-trees where the acquisition- and release-operations between an acquisition and a release of x are balanced, ignoring the actual locks, and where x is only used in a non-reentrant fashion. For example, A_x accepts $\langle_x \langle_y \rangle_y \rangle_x$, but also $\langle_x \langle_y \rangle_z \rangle_x$. However, it does not accept $\langle_x \langle_y \rangle_x$, $\langle_x \langle_y \langle_x$, nor $\langle_z \langle_y \rangle_y \rangle_x$. The idea is to count the number of locks that are above x on the lockstack. As locks are non-reentrant, this number is bounded by the number of locks. For each lock $x \in \mathcal{X}$, we define the automaton $\mathsf{A}_x := (Q, F, \delta)$ with

$$Q = \{q_0^b \mid b \in \mathbb{B}\} \cup \{(q_1, n) \mid 0 \leq n < |\mathcal{X}|\} \qquad F = \{q_0^b \mid b \in \mathbb{B}\}$$

and the rules

$$\begin{array}{llrl}
\tau \to_\delta q_0^\perp & \text{(leaf)} & \Box_a q \to_\delta q & \text{(base)} \\
\rhd_a(q_0^b)q \to_\delta q & \text{(spawn)} & \langle_y q_0^b \to_\delta q_0^b & \text{(acq-y0)} \\
\rangle_y q_0^b \to_\delta q_0^b & \text{(rel-y0)} & \langle_y (q_1, n+1) \to_\delta (q_1, n) & \text{(acq-y1)} \\
\rangle_y (q_1, n) \to_\delta (q_1, n+1) & \text{(rel-y1)} & \langle_x (q_1, 0) \to_\delta q_0^\top & \text{(acq-x)} \\
\rangle_x (q_0^b) \to_\delta (q_1, 0) & \text{(rel-x)} & \langle_x q_0^\perp \to_\delta q_0^\top & \text{(acq-xf)}
\end{array}$$

for any $q \in Q$, $y \in \mathcal{X} \setminus \{x\}$, $b \in \mathbb{B}$, and $0 \leq n < |\mathcal{X}| - 1$.

Intuitively, the q_0^b-states indicate that we are outside an acquisition of x. The b-flag indicates whether there is an acquisition of x in the local branch of the current subtree. This is required to correctly accept an unmatched final acquisition of x. Hence, the state $q_0^\perp$ can be interpreted as *maybe outside* x, while the state $q_0^\top$ can be interpreted as *definitely outside* x. The (q_1, n)-states indicate that we are inside an acquisition of x, and there are n more locks on the lockstack above x.

Initially, the (leaf)-rule starts in state $q_0^\perp$ (i.e., we may be outside x or inside a final acquisition of x). The (base)- and (spawn)-rules do not change the state. The (spawn)-rule additionally ensures that the spawned thread is not currently inside x, as this would violate well-nestedness. If outside (or maybe outside) an acquisition of x, the (acq-y0)- and (rel-y0)-rules accept acquisitions and releases of other locks than x, without changing the state. If inside an acquisition of x, the (acq-y1)- and (rel-y1)-rules update the nesting counter. They can only be applied if the counter would not overflow. Note that the counter can only overflow on reentrant executions. The (acq-x)- and (rel-x)-rules accept an acquisition or release of x. The (rel-x)-rule initializes the nesting counter to 0, and the (acq-x)-rule requires the nesting counter to be 0 again, as otherwise the release and acquisition of x would be mismatched. Finally, the (acq-xf)-rule accepts an unmatched final acquisition, and sets the state to $q_0^\top$, indicating that we are now definitely outside an acquisition of x.

We now show that non-reentrant and well-nested execution-trees can be characterized by A_x as follows:

t is non-reentrant and well-nested, if and only if $\forall x \in \mathcal{X}.\ t \in \mathsf{L}(\mathsf{A}_x)$.

Proof. With the intuition described above, it is straightforward to show that, for all $x \in \mathcal{X}$, A_x accepts any well-nested and non-reentrant execution-tree. This implies the $\Longrightarrow$-direction.

For the $\Longleftarrow$-direction, we assume that t is reentrant or not well-nested, and show that there is an $x \in \mathcal{X}$ such that t is not accepted by A_x.

If t is reentrant, it contains two acquisitions of a lock x in the same thread, without a release of x in between. After accepting the second acquisition, A_x is in state $q_0^\top$. As there is no release of x, A_x still is in state $q_0^\top$ when arriving at the first acquisition. However, there is no rule to accept $\langle_x q_0^\top$, and thus t is not accepted by A_x.

If t is not well-nested, it has an unmatched release of a lock x, or an acquisition of a lock y that matches a release of a lock $x \neq y$. In the former case, in A_x, a subtree at a spawn-node or t itself is accepted with a (q_1, n)-state. However, there is no rule to accept a $\triangleright_a(q_1, n)$-node, nor is (q_1, n) a final state. Thus, t is not accepted by A_x. In the latter case, A_x accepts the release-node of x in the state $(q_1, 0)$. When arriving at the mismatched acquisition of y, the state is again $(q_1, 0)$. As there is no rule to accept $\langle_y(q_1, 0)$ in A_x, the tree t is not accepted. □

Obviously, the size of A_x is polynomial in the size of M. Moreover, as A_x is a bottom-up deterministic automaton, it can be complemented by making it complete and complementing the set of final states, yielding the automaton $\overline{\mathsf{A}_x}$, whose size is also polynomial in the size of M. The cross-product $(\mathsf{A}^x_{\mathsf{C}_{\mathsf{I}\times}}, \mathsf{A}^x_{\mathsf{C}_{\mathsf{F}\times}}, M^x_\times)$ of M and $\overline{\mathsf{A}_x}$ can be constructed in polynomial time (cf. [79] for details on the cross-product construction between tree automata and DPNs). We have

$$\begin{aligned}
&\forall t, c.\ p_0\gamma_0 \overset{t}{\Rightarrow}_M c \implies t \text{ non-reentrant and well-nested} \\
\iff &\forall t, c.\ p_0\gamma_0 \overset{t}{\Rightarrow}_M c \implies (\forall x.\ t \in \mathsf{L}(\mathsf{A}_x)) \\
\iff &\forall x.\ \neg \left(\exists t, c.\ p_0\gamma_0 \overset{t}{\Rightarrow}_M c \wedge t \in \mathsf{L}(\overline{\mathsf{A}_x})\right) \\
\iff &\forall x.\ p_0\gamma_0 \notin \pi_1(\mathsf{A}^x_{C_{\mathsf{I}\times}} \cap \mathsf{pre}^*_{M^x_\times}(\mathsf{A}^x_{C_{\mathsf{F}\times}}))
\end{aligned}$$

where the first equivalence was shown above, the second equivalence is due to basic rewriting, and the third equivalence is due to the correctness of the cross-product construction. The right-hand side of this equivalence can be checked in polynomial time, iterating over each lock $x \in \mathcal{X}$ and using the lock-insensitive predecessor set computation from [16] (cf. Theorem 6.4) in each iteration. Thus, it can be checked in polynomial time whether M uses locks only in a well-nested and non-reentrant fashion.

8.3 Summary and Related Work

In this chapter, we indicated why our method does not transfer to well-nested, reentrant locks. We left open the decidability for this class of models. However, we briefly sketched how to transfer our method to DPNs with well-nested, non-reentrant locks (Lock-DPNs). Finally, we sketched a polynomial time algorithm to check whether a DPN satisfies the well-nestedness and non-reentrance assumption.

While reachability analysis for arbitrary locking is undecidable [57], it remains decidable for parallel pushdown systems with bounded lock-chains [54]. The bounded lock-chain criterion is a generalization of well-nestedness: Intuitively, in an execution of a single thread, a *lock-chain* is a sequence of acquisitions and releases of locks, such that the ith lock is released between the acquisitions of the $i+1$st and $i+2$nd lock. Thus, well-nested lock-usage corresponds to lock-chains of length at most one. Investigating whether those results transfer to DPNs with locks is left to future research.

In [79], we describe predecessor set computation for DPNs with well-nested, non-reentrant locks. The approach is largely the same as what we described above. Like in this thesis, we define an interleaving semantics, a tree-semantics, and a lock-sensitive scheduler. In contrast to this thesis, we use locksets instead of lockstacks already in the definition of the semantics, assuming that locks are used in a well-nested and non-reentrant fashion. We show that the set of schedulable lock-execution-hedges is regular, and do a cross-product construction of the DPN and the regular set of schedulable lock-execution-hedges. We do not use the concept of lock-a/r-hedges, but characterize the schedulable lock-execution-hedges directly. This results in more complicated definitions and proofs. (Cf. Section 4.4 for a detailed comparison.)

9 Complexity

In Chapter 6, we described an algorithm for lock-sensitive predecessor set computation. It is based on acquisition structures and requires polynomial time in the program size and exponential time in the number of locks. This algorithm can be used to model-check EF-logic and various practical problems that can be encoded into fragments of EF-logic. In Chapter 7, we analyzed an example and discussed methods to avoid the worst-case run time for typical programs. Moreover, we pointed to successful implementations of acquisition structure based methods for parallel pushdown systems (PPDS) [39, 57, 61, 63] that indicate that the problem remains tractable in practice. In this chapter, we discuss the computational complexity of analyzing programs communicating with locks. Our main result is that model-checking negation-free EF-formulas with a fixed number of operators, as well as many practical relevant properties, is NP-complete for Monitor-DPNs. This gives strong evidence that the worst-case exponential runtime cannot be avoided. The NP-hardness results also apply to practical analysis problems like data-race detection and bitvector-analysis, which indicates that our method of using lock-sensitive predecessor set computations is adequate for those problems, as it introduces no additional complexity. Moreover, we show that model-checking problems become harder for more expressive properties or models. This indicates that we have not missed extensions to more general models or properties that could be made without increasing the complexity.

This chapter is organized as follows: In Section 9.1, we introduce the models and properties that we will consider. In Section 9.2, we study the complexity of checking models with monitors or well-nested, non-reentrant locks. In Section 9.3, we consider models with stronger synchronization mechanisms like join and locks or rendezvous-communication. Finally, in Section 9.4, we discuss our results and give pointers to related work.

9.1 Models and Properties

In this section, we introduce the models and properties whose complexity will be studied. The models are derived from DPNs, and the properties are fragments of EF-logic (cf. Section 6.4.2).

A DPN has two dimensions of infinity: Unboundedly many threads may be created, and the stacks may be unboundedly deep. We obtain our models by restricting none, one, or both of these dimensions. A DPN without thread creation is a parallel pushdown system (PPDS). In a PPDS, each thread is described by a pushdown system, and the number of threads does not change during the run of a PPDS, i.e., all threads are already present in the start configuration. If the number of threads is fixed (i.e., there are n threads for a constant n), we get an nPDS. A model that consists of concurrent finite-state machines, where a transition rule may spawn a new thread as a side effect, is called *dynamic finite-state network* (DFN). Similar, in a *parallel finite-state machine* (PFSM), we have multiple finite-state machines that are concurrently executed, and in an nFSM, the number of threads is fixed to n.

For DPNs and DFNs, we assume that the start configuration consists of one thread that has just one symbol on its stack (for DPNs) and holds no locks. For PPDSs and PFSMs, we assume that the start configuration consists of a list of threads, each thread having just one symbol on its stack (for PPDS), and holding no locks.

For models with monitors but without a stack, we require that the lockstack is encoded into the control-states, i.e., we have a function locks from control-states to lockstacks that is compatible with the transition relation.

The properties that we consider are fragments of EF-logic. First, we distinguish between double-indexed and regular atomic propositions. A double-indexed atomic proposition has the form $p_1 \| p_2$, where p_1 and p_2 are control-states. It holds for configurations where one thread is at p_1 and another thread is at p_2. Double-indexed atomic propositions are suited to characterize data-races or atomicity violations. A regular atomic proposition is an automaton that describes a set of configurations. Note that regular atomic propositions are strictly more general than double-indexed atomic propositions.

Second, we distinguish whether path-operators are nested. We regard *pairwise reachability* ($\mathsf{EF}(p_1 \| p_2)$), i.e., whether two control-states p_1 and p_2 can simultaneously be reached from the start configuration; *regular set reachability* ($\mathsf{EF}(A)$); *iterated pairwise reachability* ($\mathsf{EF}(p_1 \| p_2 \wedge \mathsf{EF}(p_3 \| p_4))$); and *iterated regular set reachability* ($\mathsf{EF}(A_1 \wedge \mathsf{EF}(A_2))$). Moreover, we regard EF-formulas with a fixed number of operators and EF-formulas with an unbounded number of operators, as well as EF-formulas with (EF) and without ($\mathsf{EF}^{\setminus \mathsf{neg}}$) negation. Note that, for double-indexed EF-formulas, we use the term *fixed-size* to indicate a formula with a fixed number of operators. We avoid this term for EF-formulas with regular atomic propositions, to clarify that we only fix the number of operators, not the size of the automata for the atomic propositions.

Figure 9.1 illustrates the considered models and properties. An arrow points from a more general to a more special model or property, e.g. DPNs are more general than DFNs and PPDS, and EF-logic is more general than negation-free EF-logic.

$$\begin{tikzcd}
\text{DPN} \arrow[d] \arrow[dr] & \\
\text{PPDS} \arrow[d] \arrow[dr] & \text{DFN} \arrow[d] \\
n\text{PDS} \arrow[dr] & \text{PFSM} \arrow[d] \\
 & n\text{FSM}
\end{tikzcd}$$

$$\begin{tikzcd}
 & \mathsf{EF} \arrow[dl] \arrow[dr] & \\
\mathsf{EF}\ (\text{fixed }\#\text{ops}) \arrow[dr] & & \mathsf{EF}^{\backslash\mathsf{neg}} \arrow[dl] \\
 & \mathsf{EF}^{\backslash\mathsf{neg}}\ (\text{fixed }\#\text{ops}) \arrow[d] & \\
 & \mathsf{EF}(A_1 \wedge \mathsf{EF}(A_2)) \arrow[dl] \arrow[dr] & \\
\mathsf{EF}(A) \arrow[dr] & & \mathsf{EF}(p_1 \| p_2 \wedge \mathsf{EF}(p_3 \| p_4)) \arrow[dl] \\
 & \mathsf{EF}(p_1 \| p_2) &
\end{tikzcd}$$

Figure 9.1: Considered models and properties.

Table 9.1: Complexity results for models with reentrant monitors or well-nested, non-reentrant locks.

	DPN	PPDS	2PDS	DFN	PFSM	nFSM
$\mathsf{EF}(p_1 \| p_2)$	$\text{NP}^{*?}$	$\text{NP}^{\dagger ?}$	$\underline{\text{NP}}^{\dagger ?}$	$\underline{\text{NP}}^{*!}$	P	P
$\mathsf{EF}(A)$	NP	NP	$\text{NP}^{\dagger ?}$	NP	$\underline{\text{NP}}$	P
$\mathsf{EF}(p_1 \| p_2 \wedge \mathsf{EF}(p_3 \| p_4))$	NP	NP	$\underline{\text{NP}}$	$\utilde{\text{NP}}^{*!}$	P	P
$\mathsf{EF}(A_1 \wedge \mathsf{EF}(A_2))$	NP	NP	NP	NP	NP	P
$\mathsf{EF}^{\backslash\mathsf{neg}}$ (fixed #ops)	$\utilde{\text{NP}}$	NP	NP	NP	NP	P
EF (fixed #ops)	$\geq \underline{\text{PSPACE}}^{\ddagger}$			$\geq$NP		P
$\mathsf{EF}^{\backslash\mathsf{neg}}$	$\geq \underline{\text{PSPACE}}^{\ddagger \mathsf{reg}?}$				$\geq \underline{\text{NP}}^{\ddagger}$	P
EF	$\geq \underline{\text{PSPACE}}^{\ddagger}$					$\utilde{\text{P}}$

9.2 Monitors and Well-Nested, Non-Reentrant Locks

In this section, we consider the complexity of checking models with reentrant monitors or well-nested, non-reentrant locks. Table 9.1 shows the complexity results that we will establish in this section. The rows of the table are indexed with the properties, and the columns are indexed with the models. Interestingly, the hardness results already hold for models with non-reentrant monitors. This locking discipline is more special than the reentrant monitor and well-nested, non-reentrant locking disciplines that are typically considered.

The entries in the table are annotated with various additional information that we now explain: A $*$-annotation indicates that the NP-hardness result requires a model that spawns threads inside monitors. Note that this might not be realistic, as monitors are typically used to protect access to shared resources, and the code inside monitors is kept small. Thus, one would usually not spawn threads inside monitors. An $*!$-annotation means that we have a polynomial algorithm if we do not spawn threads inside monitors. An $*?$-annotation means that we do not know the exact complexity of the analysis problem, if we must not spawn threads inside monitors, and the model is additionally deadlock-free and uses inescapable locks.

A †-annotation indicates that the NP-hardness result requires a model that has deadlocks or escapable locks. Allowing escapable locks may not be realistic, as real programming languages are deterministic, and thus locks are inescapable. However, escapable locks are required to model timeouts or `tryLock`-operations, which are often used to avoid deadlocks and are also available for Java [47]. Assuming that the program to be analyzed may have deadlocks is realistic, as deadlocks are a common programming error. However, if a prior analysis step has verified the absence of deadlocks, we may assume that the program is deadlock-free. Unfortunately, we do not know the exact complexity of some problems in the deadlock-free case with inescapable locks, as we indicate with the †?-annotation.

A ‡-annotation means that the hardness result does not require locks at all. Those hardness results are rather strong, as they do not depend on communication between threads.

The reg-annotation in the row for negation-free EF-logic means that we require regular atomic propositions to establish the hardness result, while the hardness result for EF-logic with negation already holds for single-indexed atomic propositions. We leave open the exact complexity for double-indexed EF-logic, as indicated by the reg?-annotation. A lower bound is, of course, NP-hardness.

An underlined entry means that the hardness has to be proved for this entry. The hardness for entries without an underline follows from the hierarchy of the models and properties (cf. Figure 9.1).

An entry with a wavy underline indicates that the easiness has to be proved for this entry. The easiness for entries without a wavy underline follows from the hierarchy of the models and properties (cf. Figure 9.1).

Finally, for entries that are marked with $\geq$, we only show the hardness direction.

The remainder of this section is organized as follows: In Subsection 9.2.1, we establish the lower complexity bounds, and in Subsection 9.2.2, we establish the upper complexity bounds.

9.2.1 Lower Complexity Bounds

In this subsection, we establish the lower complexity bounds for the problems in Table 9.1. We only show the bounds for underlined entries. The other results follow from the hierarchy of the models and properties (cf. Figure 9.1).

The NP-hardness results are obtained by reduction from the 3SAT-problem. For the PSPACE-hardness results, we perform a reduction from the QBF-problem or refer to existing results. Note that all hardness results established in this subsection already hold for models with non-reentrant monitors.

For the next paragraphs, assume that we have an arbitrary 3SAT-instance (V, C) over n variables and m clauses.

NP-Hardness of $\mathsf{EF}(p_1 \| p_2 \wedge \mathsf{EF}(p_3 \| p_4))$ for 2PDS We reduce 3SAT to this problem. Regard the following program:

```
$p_1$:      sync $(v_1)$\{call $p_2$\} OR sync $(\neg v_1)$\{call $p_2$\};
$p_2$:      sync $(v_2)$\{call $p_3$\} OR sync $(\neg v_2)$\{call $p_3$\}; return
...
$p_n$:      sync $(v_n)$\{call $p_{n+1}$\} OR sync $(\neg v_n)$\{call $p_{n+1}$\}; return
$p_{n+1}$:  $a$: skip; return
$q$:
            sync $(l_{11})$\{\} OR sync $(l_{12})$\{\} OR sync $(l_{13})$\{\}
            ...
            sync $(l_{m1})$\{\} OR sync $(l_{m2})$\{\} OR sync $(l_{m3})$\{\}
            $b$: skip
```

Note that the program is given as pseudocode. The symbols $p_1, \ldots, p_{n+1}$ and q denote procedures here, and should not be confused with control-states. We argue on the basis of pseudocode here, a translation to 2PDS is straightforward. The execution starts with two threads at p_1 and q. The program uses monitors v_i and $\neg v_i$ for $1 \leq i \leq n$. Intuitively, the monitors encode a valuation of the variables. Entering the monitor v_i encodes the valuation $v_i = \bot$, and entering the monitor $\neg v_i$ encodes the valuation $v_i = \top$. The first thread, which starts at p_1, can reach its label a with monitors encoding exactly all possible valuations of the variables. Assuming that the first thread is at label a, the second thread, which starts at q, can reach the label b if and only if the valuation chosen by the first thread satisfies all clauses. Hence, to complete the reduction, we have to check whether the first thread, called the *chooser* thread, can reach a, and afterwards, the second thread, called the *checker* thread, can reach b while the chooser thread remains at a. This can be expressed by an iterated pairwise reachability property, and we have that

$$p_1\gamma_1 q\gamma_1 \models \mathsf{EF}(a \| q \wedge \mathsf{EF}(a \| b)),$$

if and only if there is a valuation that satisfies all clauses. Here, γ_1 and γ_2 are the bottommost stack-symbols of the threads, and we assume that the labels are encoded into the control-state. Moreover, the program is obviously deadlock-free, even if the locks are translated to inescapable locks.

NP-Hardness of $\mathsf{EF}(A)$ for PFSM Again, we reduce 3SAT to this problem. The idea is similar to the reduction above. However, we use one copy of the monitor per clause, and use separate threads for each monitor. The regular set to be reached ensures that the copies of the monitors are consistently entered. We use the following program, with chooser threads t_i^j and checker threads c^j for $1 \leq i \leq n$ and $1 \leq j \leq m$.

```
t_i^j:  sync (v_i^j){a_i^j: skip} OR sync (¬v_i^j){b_i^j: skip}
c^j:    sync ((l_j1)^j){c_1^j: skip}
          OR sync ((l_j2)^j){c_2^j: skip}
          OR sync ((l_j3)^j){c_3^j: skip}
```

The regular set to be reached is defined by the regular expression

$$\begin{aligned}\mathsf{L}(A) := {} & ((a_1^1 \dots a_1^m) + (b_1^1 \dots b_1^m)) \dots ((a_n^1 \dots a_n^m) + (b_n^1 \dots b_n^m)) \\ & (c_1^1 + c_2^1 + c_3^1) \dots (c_1^m + c_2^m + c_3^m).\end{aligned}$$

Obviously, A can be defined in size $O(nm)$. We then have

$$t_1^1 \dots t_1^m \ \dots \ t_n^1 \dots t_n^m \ c_1 \dots c_m \models \mathsf{EF}(A),$$

if and only if the 3SAT-formula is satisfiable. Moreover, the program is deadlock-free, as it does not use nested monitors.

At this point, we have shown NP-hard all problems except pairwise reachability. Note that regular set reachability and pairwise reachability is equivalent for 2PDS, and that pairwise reachability problems can be reduced from PPDS to 2PDS, as only two threads need to be considered (cf. [57]).

The problem to apply our reduction to pairwise reachability properties is to ensure the correct sequence of checker and chooser thread. For iterated reachability, ensuring the correct sequence was straightforward. For regular set reachability, we could exploit the regular set to be reached to enforce consistency between many threads. For pairwise reachability, however, we see no other option than using thread-creation inside monitors or deadlocks.

NP-Hardness of $\mathsf{EF}(p_1 \| p_2)$ for 2PDS Again, we reduce 3SAT to this problem. We consider a chooser and a checker thread, like in the reduction for iterated pairwise reachability. However, the whole checker thread is synchronized on an additional lock x, and the chooser thread synchronizes on x before it reaches label a. This way, the checker thread cannot start too early, as it would prevent the chooser thread from reaching a. However, the program can deadlock: If the chooser thread chooses an unsatisfiable configuration, and the checker thread starts before the chooser thread has synchronized on x, both threads will get stuck. Regard the following program:

```
$p_1$:      sync $(v_1)${call $p_2$} OR sync $(\neg v_1)${call $p_2$};
$p_2$:      sync $(v_2)${call $p_3$} OR sync $(\neg v_2)${call $p_3$}; return
...
$p_n$:      sync $(v_n)${call $p_{n+1}$} OR sync $(\neg v_n)${call $p_{n+1}$}; return
$p_{n+1}$:  sync $(x)${}; $a$: skip; return
$q$:        sync $(x)${
            sync $(l_{11})${} OR sync $(l_{12})${} OR sync $(l_{13})${}
            ...
            sync $(l_{m1})${} OR sync $(l_{m2})${} OR sync $(l_{m3})${}
            $b$: }
```

With the arguments sketched above, we have that $p_1\gamma_1 q\gamma_2 \models \mathsf{EF}(a\|b)$, if and only if the 3SAT-formula is satisfiable.

NP-Hardness of $\mathsf{EF}(p_1\|p_2)$ for DFN We reduce 3SAT to this problem. Here, we use thread creation inside monitors to ensure the proper sequence of checker and chooser threads. We regard the following program:

```
$t_0$:      sync $(x)${spawn $t_1$; $a$: skip}
$t_1$:      sync $(v_1)${spawn $t_2$; sync $(x)${}} OR sync $(\neg v_1)${spawn $t_2$; sync $(x)${}}
            ...
$t_n$:      sync $(v_n)${spawn $t_{n+1}$; sync $(x)${}} OR sync $(\neg v_n)${spawn $t_{n+1}$; sync $(x)${}}
$t_{n+1}$:
            sync $(l_{11})${} OR sync $(l_{12})${} OR sync $(l_{13})${}
            ...
            sync $(l_{m1})${} OR sync $(l_{m2})${} OR sync $(l_{m3})${}
            $b$: skip
```

The chooser threads $t_1 \dots t_n$ are spawned in sequence: After entering its monitor, a chooser thread spawns the next chooser thread, and the chooser thread t_n for the last variable spawns the checker thread. Thus, the checker thread runs after the chooser threads have made their choices. The additional lock x ensures that the chooser threads do not leave their monitors too early. Note that the program has no deadlocks: At any time, t_0 can leave its monitor, and then, the chooser threads can use x and leave their monitors. With the arguments sketched above, we have $t_0 \models \mathsf{EF}(a\|b)$, if and only if the 3SAT-formula is satisfiable.

PSPACE-Hardness for EF-Logic For full EF-logic, there are various known hardness results for models without locks. Bouajjani et al. [13] show that model-checking a fixed-size, single-indexed EF-formula is PSPACE-hard already for PDS [13]—a model without concurrency. This implies the PSPACE-hardness results in the EF-row for a fixed number of operators.

A well-known result is that deciding emptiness of the intersection of a given set of automata (INT) is PSPACE-complete [66]. INT can be reduced to model-checking an EF-formula of the form $\mathsf{EF}(\mathsf{L}(A_1) \wedge \ldots \wedge \mathsf{L}(A_n))$, if the model is able to produce arbitrary length configurations. This is the case for any model that is as least as general as DFN or 2PDS. This implies the PSPACE-hardness results in the $\mathsf{EF}^{\setminus \mathsf{neg}}$-row.

Note that this reduction does not work for double-indexed negation-free EF-formulas, and we have to leave open the exact complexity for this case. At least, we obtain NP-hardness for PFSMs without locks: We reduce 3SAT to this problem. Given a 3SAT-instance (V, C), we regard a PFSM with one chooser thread per variable and the rules $p_i \to v_i$ and $p_i \to \neg v_i$ for $1 \leq i \leq n$. The choice of the variables is indicated by the states of the chooser threads. As we work with unbounded size EF-formulas, we can encode the clauses directly into the formula, and have

$$p_1 \ldots p_n \models \mathsf{EF}(\bigwedge_{1 \leq i \leq m} \bigvee_{1 \leq j \leq 3} l_{ij}),$$

if and only if the formula is satisfiable. (Note that this formula is even single-indexed.)

Esparza [34] shows that model-checking EF-logic is PSPACE-hard for *basic parallel processes*. We adopt the idea of his reduction to show PSPACE-hardness of model-checking PFSMs without locks. We reduce QBF to this problem. Given a QBF-instance (V, C) over n variables and m clauses, we use the same PFSM as above, i.e., we have one thread per variable and the rules $p_i \to v_i$ and $p_i \to \neg v_i$. Then, we construct an EF-formula that checks all possibilities of the variable valuations, and tests the clauses for each possibility. Existential quantification is modeled by the EF-operator, and universal quantification is modeled by the AG-operator. Note that we have $\mathsf{AG}\varphi = \neg\mathsf{EF}\neg\varphi$. We regard the following formula:

$$\begin{aligned}\varphi :=& \mathsf{EF}((v_1 \vee \neg v_1) \wedge p_2 \wedge \ldots \wedge p_n \\ &\quad \wedge \mathsf{AG}((v_2 \vee \neg v_2) \wedge p_3 \wedge \ldots \wedge p_n \\ &\quad\quad \implies \ldots \\ &\quad\quad \wedge \mathsf{AG}((v_n \vee \neg v_n) \implies \bigwedge_{1 \leq i \leq m} \bigvee_{1 \leq j \leq 3} l_{ij}) \ldots))\end{aligned}$$

We have $p_1 \ldots p_n \models \varphi$, if and only if the QBF is true. This implies the hardness results in the EF-row.

9.2.1.1 Filtering Consistent Configurations

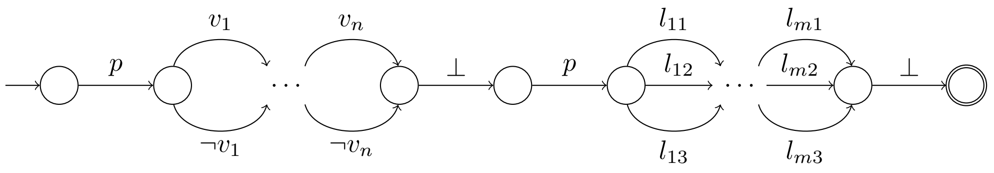
The reductions presented in the previous paragraphs required reachability queries only from consistent start configurations. Thus, our analysis algorithm does not require intersection of the result with the automaton $\mathsf{A}_{\mathsf{Conf}^{\mathsf{ls}}}$ (cf. Chapter 7). However, if we are interested in the set of consistent predecessor configurations or consistent immediate predecessor configurations of some automaton that may also contain inconsistent configurations, we have to intersect the result set with the automaton $\mathsf{A}_{\mathsf{Conf}^{\mathsf{ls}}}$ that accepts the consistent configurations. In this subsection, we show that this makes the problem hard. More precisely, we show that deciding whether an automaton A with $\mathsf{L}(A) \subseteq \mathsf{valid}$ accepts a consistent configuration is NP-hard, and thus also deciding whether $\mathsf{pre}_{\mathsf{ls},M}(\mathsf{L}(A) \cap \mathsf{Conf}^{\mathsf{ls}}) \neq \emptyset$ is NP-hard, although we have a polynomial algorithm to compute an automaton $\mathsf{pre}'_{\mathsf{ls},M}(A)$ with $\mathsf{L}(\mathsf{pre}'_{\mathsf{ls},M}(A)) \cap \mathsf{Conf}^{\mathsf{ls}} = \mathsf{pre}_{\mathsf{ls},M}(\mathsf{L}(A) \cap \mathsf{Conf}^{\mathsf{ls}})$ (cf. Theorem 7.2).

Lemma 9.1. *Given a Monitor-DPN $M = (P, \Gamma, \Gamma_\perp, \mathsf{Act}, \mathcal{X}, \Delta, \mathsf{locks})$, and an automaton A over the alphabet $P \cup \Gamma$ such that $\mathsf{L}(A) \subseteq \mathsf{valid}$. Then, it is NP-hard to decide whether*

$$\mathsf{L}(A) \cap \mathsf{Conf}^{\mathsf{ls}} \neq \emptyset.$$

The problem remains NP-hard, even if we assume that all configurations accepted by A are non-reentrant.

Proof. The proof is, again, based on a reduction from 3SAT. Given a 3SAT-instance (V, C) over n variables and m clauses, we construct a Monitor-DPN M with a single control-state $P = \{p\}$, locks $\mathcal{X} = V \cup \{\neg v \mid v \in V\}$, stack alphabet $\mathcal{X} \dot\cup \{\perp\}$, and $\mathsf{locks} : \perp \mapsto \emptyset, x \mapsto \{x\}$. Note that the other components of the Monitor-DPN are not relevant for our construction, as we just need the DPN to define the set $\mathsf{Conf}^{\mathsf{ls}}$ of consistent configurations. We construct the following automaton A:



Recall that, in a consistent configuration, each lock is held by at most one process. The automaton A now produces configurations that consist of two processes. The first process holds, for each variable v_i, either the lock v_i or $\neg v_i$. The second process holds, for each clause, the lock associated with one of the literals of that clause. If A accepts a consistent configuration, the locks held by the first and second process are disjoint, i.e., when accepting the first process, A chooses a valuation such that, when accepting the second process, each clause can be satisfied. Vice versa, for any satisfying valuation, a corresponding consistent configuration is accepted by A. Obviously we have $\mathsf{L}(A) \subseteq \mathsf{valid}$, which completes the proof.

The reduction can easily be modified such that all configurations accepted by A are non-reentrant, i.e., no lock occurs twice on the same lockstack. For this purpose, we work with $2m$ different locks $v_i^1, \dots, v_i^m$ and $\neg v_i^1, \dots, \neg v_i^m$ for each variable v_i. When choosing the valuation, we ensure that either all v_i^j or all $\neg v_i^j$ locks are on the lockstack. When checking the clauses, a literal l_{jk} is checked by the lock $(l_{jk})^j$. Thus, the lockstacks of both threads are non-reentrant. □

In Section 7.2.2, we sketched a polynomial time algorithm for computing immediate predecessor sets. However, the result could still contain inconsistent configurations. If inconsistent configurations need to be filtered out from the result, but the input automaton may contain inconsistent configurations, the problem is NP-hard.

Theorem 9.2. *Given as input a Monitor-DPN or Lock-DPN M and an automaton A with $\mathsf{L}(A) \subseteq \mathsf{valid}$. Then, it is NP-hard to decide whether:*

$$\mathsf{pre}_{\mathsf{ls},M}(\mathsf{L}(A) \cap \mathsf{Conf}^{\mathsf{ls}}) \neq \emptyset.$$

Proof. We reduce the problem of checking whether a given automaton A with $\mathsf{L}(A) \subseteq \mathsf{valid}$ accepts a consistent configuration (cf. Lemma 9.1) to our problem. We set the rules of M such that there is a rule of the form $p\gamma \overset{a}{\hookrightarrow} p\gamma$ for each $p \in P$, $\gamma \in \Gamma$, and some $a \in \mathsf{Act}$. Thus, steps of the DPN do not change the configuration and are always possible. Hence, A accepts a consistent configuration, if and only if $\mathsf{pre}_{\mathsf{ls},M}(\mathsf{L}(A) \cap \mathsf{Conf}^{\mathsf{ls}}) \neq \emptyset$. □

9.2.2 Upper Complexity Bounds

In the last subsection, we have established the lower complexity bounds of Table 9.1. In this subsection, we establish the upper complexity bounds. We have to provide algorithms for the entries marked with a wavy underline. The other results then follow from the hierarchy of the models and properties illustrated in Figure 9.1.

First of all, note that model-checking CTL for sequential finite-state processes (1FSM) can be done in polynomial time [27]. Moreover, model-checking an nFSM with m states can be reduced to model-checking its asynchronous product, which is a 1FSM with m^n states. As n is fixed, we get a polynomial time algorithm.

The remainder of this subsection is organized as follows: In Subsection 9.2.2.1, we give an alternative characterization of the complexity class NP that is better suited for our purposes. In Subsection 9.2.2.2, we show that model-checking negation-free EF-formulas with a fixed number of operators and regular atomic propositions is in NP for Monitor-DPNs. In Subsection 9.2.2.3, we briefly sketch that this result also transfers to Lock-DPNs. In Subsection 9.2.2.4, we show that model-checking fixed-size, negation-free, double-indexed EF-formulas is in P for Monitor-DFNs.

9.2.2.1 NP via Verification of Certificates

We characterize NP as those problems where membership can be verified by a polynomial size certificate in polynomial time (cf. Papadimitriou [96], Chapter 9):

Theorem 9.3. *A problem L is in NP, if and only if there is a polynomially decidable and polynomially balanced relation R, such that*

$$L := \{x \mid \exists y.\ (x, y) \in R\},$$

where a relation R is polynomially balanced*, iff*

$$(x, y) \in R \implies |y| \leq |x|^k \textit{ for some fixed } k \geq 1.$$

Intuitively, for each element x of the problem L, we find a certificate y that has polynomial size in x, and can be verified in polynomial time.

In the remainder of this subsection, we show how to characterize lock-sensitive reachability problems for Monitor-DPNs in this way, thus proving that they are in NP.

9.2.2.2 Model-Checking negation-free EF-Formulas with a Fixed Number of Operators

We show that the problem is in NP for Monitor-DPNs. In order to obtain an NP-algorithm, we proceed in two steps. In the first step, we develop a polynomial verifier for lock-sensitive predecessor sets. Given a Monitor-DPN M and an automaton A, the verifier takes some certificate C and computes an automaton A' that has polynomial size in M, A, and C, and whose language is a subset of $\mathsf{pre}^*_{\mathsf{ls},M}(\mathsf{L}(A))$. Moreover, we show that, for each configuration $c \in \mathsf{pre}^*_{\mathsf{ls},M}(\mathsf{L}(A))$, there is a certificate of size $\mathsf{poly}(|\mathcal{X}|)$ such that $c \in A'$.

In the second step, this verifier is used to develop an NP-algorithm for model-checking negation-free EF-formulas with a fixed number of operators.

Recall that we use the DPN-Acceptor $D_{\mathsf{A}_{\mathsf{Cons}}}$ to characterize the set of schedulable lock-execution-hedges. We then construct the cross-product of the Monitor-DPN to be analyzed and $D_{\mathsf{A}_{\mathsf{Cons}}}$, and compute lock-insensitive predecessor sets on the cross-product. The result has to be intersected with the set of consistent configurations, described by the automaton $\mathsf{A}_{\mathsf{Conf}^{\mathsf{ls}}}$. The cross-product construction and lock-insensitive predecessor set computation can be done in polynomial time. However, the size of $D_{\mathsf{A}_{\mathsf{Cons}}}$ and $\mathsf{A}_{\mathsf{Conf}^{\mathsf{ls}}}$ is exponential in the number of locks.

The idea of the certificate is to regard the rules of $D_{\mathsf{A}_{\mathsf{Cons}}}$ that are actually required to accept a single lock-execution-hedge that witnesses membership of a single configuration in the predecessor set. After some modifications to $\mathsf{A}_{\mathsf{Cons}}$ and $D_{\mathsf{A}_{\mathsf{Cons}}}$, we show that we only need polynomially many rules to accept a single lock-execution-hedge. Similar, we only need polynomially many rules of $\mathsf{A}_{\mathsf{Conf}^{\mathsf{ls}}}$ to accept a single configuration. Thus, the certificate consists of the rules required to accept the lock-execution-hedge and the configuration, and the verifier performs the cross-product construction, the lock-insensitive predecessor set algorithm, and the required automata-operations. As all those operations are polynomial time, the verifier runs in polynomial time.

We construct the verifier in three steps. First, we modify the automaton $\mathsf{A}_{\mathsf{Cons}}$ that accepts schedulable lock-a/r-hedges, such that every hedge that is accepted requires only polynomially many rules. Second, we modify the translation to the DPN-Acceptor $D_{\mathsf{A}_{\mathsf{Cons}}}$, such that any lock-execution-hedge that is accepted requires only polynomially many rules. Third, we show that only polynomially many rules of $\mathsf{A}_{\mathsf{Conf}^{\mathsf{ls}}}$ are required to accept a single configuration.

Subsumption Order on Acquisition Structures For the first step, we require the *subsumption ordering* on acquisition structures.

Definition 9.4 (Subsumption Order)**.** *We define an ordering $\leq \subseteq \mathsf{AS} \times \mathsf{AS}$ on acquisition structures by pointwise lifting of set-inclusion. For two acquisition structures $s = (r, g_r, u, a, g_a)$ and $s' = (r', g'_r, u', a', g'_a)$, we define*

$$s \leq s' \;:\Longleftrightarrow\; r \supseteq r' \wedge g_r \subseteq g'_r \wedge u \subseteq u' \wedge a \subseteq a' \wedge g_a \subseteq g_a.$$

We also overload $\leq$ to hedge acquisition structures, and define $\leq \subseteq \mathsf{AS_h} \times \mathsf{AS_h}$ by:

$$(s, X) \leq (s', X') \;:\Longleftrightarrow\; s \leq s' \wedge X \subseteq X'.$$

The ordering $\leq$ on (hedge) acquisition structures is called subsumption ordering*.*

Note that we defined the inclusion for release-sets the other way round. Intuitively, this is because a thread that releases more locks is more likely to be executable than a thread that releases fewer locks.

It is straightforward to show that consistency of acquisition histories is monotonic w.r.t. the subsumption ordering:

Lemma 9.5.

$$s \in \mathsf{Cons} \wedge s' \leq s \implies s' \in \mathsf{Cons}$$

Proof. Obvious by definition of Cons. □

Intuitively, if we have $s' \leq s$, the acquisition structure s' is *subsumed* by s, as s' is consistent if s is. In order to verify that an execution-hedge is schedulable, it is sufficient to verify that its acquisition history is subsumed by a consistent acquisition history.

Bounding the Tree Automaton on Lock-A/R-Hedges Regard a schedulable lock-a/r-hedge accepted by $\mathsf{A}_{\mathsf{Cons}}$. It has no two final acquisitions of the same lock and the set of used locks minus the set of initially released locks is disjoint from the set of initially held locks. As the hedge is consistent and well-formed, it contains no two initial releases of the same lock.

Now, regard an accepting run of $\mathsf{A}_{\mathsf{Cons}}$ for this hedge. In its states, $\mathsf{A}_{\mathsf{Cons}}$ computes the acquisition history for the hedge in a bottom-up fashion. The acquisition-graph and the sets of initially released and finally acquired locks (g_a-, r-, and a-components of the acquisition structure) are only changed at acquisition- and release-nodes. Thus, during the run of $\mathsf{A}_{\mathsf{Cons}}$, there are only $O(|\mathcal{X}|)$ different g_a-, r- and a-components.

During processing a tree, the release-graph (g_r-component) only increases, and when joining an acquisition history with a hedge acquisition history at $\#_{\mathsf{h}}$-nodes, only disjoint release-graphs are joined, as no lock is initially released more than once. As there are $|\mathcal{X}|^2$ possible edges in a release-graph, there are only $O(|\mathcal{X}|^2)$ different release-graphs in the states used during the run of $\mathsf{A}_{\mathsf{Cons}}$.

While processing the $\#_{\mathsf{h}}$-nodes of the hedge, the set of initially held locks is collected in the X-component of the hedge acquisition structure. As this set only increases, there are only $O(|\mathcal{X}|)$ different X-components during the run. As the sets of initially held locks of every thread are disjoint, the elements of the hedge have only $O(|\mathcal{X}|)$ different sets of initially held locks. Thus, in order to process all those sets, we only need to instantiate the $\#_{\mathsf{h}}$-rule of $\mathsf{A}_{\mathsf{Cons}}$ for $O(|\mathcal{X}|)$ different X-sets.

However, there may be $2^{|\mathcal{X}|}$ different u-components during the run: As the size of the a/r-hedge is not bounded, it may consist of $2^{|\mathcal{X}|}$ different threads, each using a different set of locks. Thus, exponentially many rules may be required to process a single hedge.

This is where the subsumption ordering comes into play. Instead of precisely computing the use-sets, we nondeterministically guess the use-sets at the leafs of the hedge and at acquisition-nodes. Then, we check that the actual use-sets are below the guessed sets. Moreover, we ensure that the guessed use-sets of spawned threads are the same as the guessed use-set of the spawning thread, such that combination of threads at use-nodes does not involve different use-sets. This way, we can bound the number of use-sets during the run of the automaton to $O(|\mathcal{X}|)$. Moreover, for processing a use-node, we do not need to instantiate the rule for the exact set of used locks, but it is sufficient to specify an upper bound. For this purpose, we introduce rule-templates of the form $\langle\rangle_X(q_1)q_2 \to q\ \forall X \subseteq Y$, and count such a template as a single rule. Later, we show how to translate those templates to a DPN-Acceptor, requiring only polynomially many rules.

We define the automaton $\mathsf{A}'_{\mathsf{Cons}} := (Q, F, \delta)$ with $Q = \mathsf{AS} \mathbin{\dot\cup} \mathsf{AS}_{\mathsf{h}}$, $F = \mathsf{Cons}$, and the following rules:

$$\begin{array}{llr}
\varepsilon_{\mathsf{h}} \rightarrow_\delta \mathsf{as}_{\mathsf{h}}^{\varepsilon}[\mathsf{u} \mapsto u] & \text{for all } u \subseteq \mathcal{X} & \text{(h-empty)} \\
(s_t, X)\#_{\mathsf{h}} s \rightarrow_\delta \mathsf{as}_{\mathsf{h}}^{\#}(s_t, X, s) & \text{if } s_t.\mathsf{u} = s.\mathsf{u} & \text{(h-cons)} \\
\\
\varepsilon_{\mathsf{s}} \rightarrow_\delta \mathsf{as}_{\mathsf{s}}^{\varepsilon}[\mathsf{u} \mapsto u] & \text{for all } u \subseteq \mathcal{X} & \text{(u-empty)} \\
s_t\#_{\mathsf{s}} s_s \rightarrow_\delta \mathsf{as}_{\mathsf{s}}^{\#}(s_t, s_s) & \text{if } s_t.\mathsf{u} = s_s.\mathsf{u} & \text{(u-cons)} \\
\\
\tau \rightarrow_\delta \mathsf{as}_{\mathsf{t}}^{\tau}[\mathsf{u} \mapsto u] & \text{for all } u \subseteq \mathcal{X} & \text{(nil)} \\
\langle\rangle_X(s_s)s_t \rightarrow_\delta \mathsf{as}_{\mathsf{t}}^{\langle\rangle}(X, s_s, s_t)\ \forall X \subseteq s_t.\mathsf{u} & \text{if } s_t.\mathsf{u} = s_s.\mathsf{u} & \text{(use)} \\
\langle_x s_t \rightarrow_\delta \mathsf{as}_{\mathsf{t}}^{\langle}(x, s_t)[\mathsf{u} \mapsto u'] & \text{for all } u' \supseteq s_t.\mathsf{u} & \text{(acq)} \\
\rangle_x s_t \rightarrow_\delta \mathsf{as}_{\mathsf{t}}^{\rangle}(x, s_t) & & \text{(rel)}
\end{array}$$

where we use the notation $s.\mathsf{u}$ for a (hedge) acquisition structure $s \in \mathsf{AS} \cup \mathsf{AS}_{\mathsf{h}}$, to identify the used-set component of s, and $s[\mathsf{u} \mapsto u']$ to replace the used-set component of the (hedge) acquisition structure s by the set u'.

First of all, we show the correctness of $\mathsf{A}'_{\mathsf{Cons}}$, i.e., we have

$$\mathsf{L}(\mathsf{A}'_{\mathsf{Cons}}) = \mathsf{L}(\mathsf{A}_{\mathsf{Cons}}).$$

Proof. It is straightforward to show that any hedge h accepted by $\mathsf{A}'_{\mathsf{Cons}}$ with acquisition structure σ' is accepted by $\mathsf{A}_{\mathsf{Cons}}$ with a smaller acquisition structure $\sigma \leq \sigma'$:

$$h \in \mathsf{A}'_{\mathsf{Cons}}(\sigma') \implies \exists\sigma.\ \sigma \leq \sigma' \wedge h \in \mathsf{A}_{\mathsf{Cons}}(\sigma).$$

As consistency is monotonic w.r.t. the subsumption ordering (Lemma 9.5), we get $\mathsf{L}(\mathsf{A}'_{\mathsf{Cons}}) \subseteq \mathsf{L}(\mathsf{A}_{\mathsf{Cons}})$.

For the $\supseteq$-direction, we show the following statement, which implies $\mathsf{L}(\mathsf{A}'_{\mathsf{Cons}}) \supseteq \mathsf{L}(\mathsf{A}_{\mathsf{Cons}})$: For all $h \in \mathsf{HAR}^{\mathsf{ls}}$, $t \in \mathsf{TAR}$, $l \in \mathsf{TAR}^*$, $\sigma \in \mathsf{AS}$, and $X, u' \subseteq \mathcal{X}$ we have:

$$h \in \mathsf{A}_{\mathsf{Cons}}(\sigma, X) \wedge u' \supseteq \sigma.\mathsf{u} \implies h \in \mathsf{A}'_{\mathsf{Cons}}(\sigma[\mathsf{u} \mapsto u'], X) \tag{1}$$
$$t \in \mathsf{A}_{\mathsf{Cons}}(\sigma) \wedge u' \supseteq \sigma.\mathsf{u} \implies t \in \mathsf{A}'_{\mathsf{Cons}}(\sigma[\mathsf{u} \mapsto u']) \tag{2}$$
$$l \in \mathsf{A}_{\mathsf{Cons}}(\sigma) \wedge u' \supseteq \sigma.\mathsf{u} \implies l \in \mathsf{A}'_{\mathsf{Cons}}(\sigma[\mathsf{u} \mapsto u']) \tag{3}$$

This statement is shown by a rather straightforward induction on the structure of lock-a/r-hedges, i.e., by induction on h, t, and l: The case $h = \varepsilon$ is trivial.

In the case $h = (t, X_t)\hat{h}$, we obtain acquisition structures $\sigma_t \in \mathsf{AS}$ and $(\hat{\sigma}, \hat{X}) \in \mathsf{AS}_{\mathsf{h}}$ such that

$$(\sigma, X) = \mathsf{as}_{\mathsf{h}}^{\#}(\sigma_t, X_t, (\hat{\sigma}, \hat{X})) = (\sigma_t, X_t) \parallel (\hat{\sigma}, \hat{X}) \wedge t \in \mathsf{A}_{\mathsf{Cons}}(\sigma) \wedge \hat{h} \in \mathsf{A}_{\mathsf{Cons}}(\hat{\sigma}, \hat{X}).$$

By unfolding the definition of $\parallel$, we get $\sigma.\mathsf{u} = \sigma_t.\mathsf{u} \cup \hat{\sigma}.\mathsf{u}$, and thus $u' \supseteq \sigma_t.\mathsf{u}$ and $u' \supseteq \hat{\sigma}.\mathsf{u}$. Hence, we can apply the induction hypothesis and get

$$t \in \mathsf{A}'_{\mathsf{Cons}}(\sigma_t[\mathsf{u} \mapsto u']) \wedge \hat{h} \in \mathsf{A}'_{\mathsf{Cons}}(\hat{\sigma}[\mathsf{u} \mapsto u'], \hat{X}).$$

Using the (h-cons)-rule, we get

$$h \in \mathsf{A}'_{\mathsf{Cons}}(\mathsf{as}^{\#}_{\mathsf{h}}(\sigma_t[\mathsf{u} \mapsto u'], X_t, (\hat{\sigma}[\mathsf{u} \mapsto u'], \hat{X}))).$$

Moreover, we have

$$\mathsf{as}^{\#}_{\mathsf{h}}(\sigma_t[\mathsf{u} \mapsto u'], X_t, (\hat{\sigma}[\mathsf{u} \mapsto u'], \hat{X})) = \mathsf{as}^{\#}_{\mathsf{h}}(\sigma_t, X_t, (\hat{\sigma}, \hat{X}))[\mathsf{u} \mapsto u'],$$

which completes the case.

The case $t = \tau$ is, again, trivial.

In the case $t = \langle\rangle_{X_u}(s)\hat{t}$, we obtain acquisition structures $\sigma_s \in \mathsf{AS}$ and $\hat{\sigma} \in \mathsf{AS}$ with

$$\sigma = \mathsf{as}^{\Diamond}_{\mathsf{t}}(X_u, \sigma_s, \hat{\sigma}) \wedge s \in \mathsf{A}_{\mathsf{Cons}}(\sigma_s) \wedge \hat{t} \in \mathsf{A}_{\mathsf{Cons}}(\hat{\sigma}).$$

Hence, we have $\sigma.\mathsf{u} = \sigma_s.\mathsf{u} \cup \hat{\sigma}.\mathsf{u} \cup X_u$, and thus $u' \supseteq \sigma_s.\mathsf{u}$ and $u' \supseteq \hat{\sigma}.\mathsf{u}$ and $u' \supseteq X_u$. The induction hypothesis yields

$$s \in \mathsf{A}'_{\mathsf{Cons}}(\sigma_s[\mathsf{u} \mapsto u']) \wedge \hat{t} \in \mathsf{A}'_{\mathsf{Cons}}(\hat{\sigma}[\mathsf{u} \mapsto u']).$$

By the (use)-rule, we get $t \in \mathsf{A}'_{\mathsf{Cons}}(\mathsf{as}^{\Diamond}_{\mathsf{t}}(X_u, \sigma_s[\mathsf{u} \mapsto u'], \hat{\sigma}[\mathsf{u} \mapsto u']))$. Moreover, we have $\mathsf{as}^{\Diamond}_{\mathsf{t}}(X_u, \sigma_s[\mathsf{u} \mapsto u'], \hat{\sigma}[\mathsf{u} \mapsto u']) = \mathsf{as}^{\Diamond}_{\mathsf{t}}(X_u, \sigma_s, \hat{\sigma})[\mathsf{u} \mapsto u']$, which completes the case.

In the case $t = \langle_x \hat{t}$, we obtain $\hat{\sigma} \in \mathsf{AS}$, such that

$$\sigma = \mathsf{as}^{\langle}_{\mathsf{t}}(x, \hat{\sigma}) \wedge \hat{t} \in \mathsf{A}_{\mathsf{Cons}}(\hat{\sigma}).$$

By induction hypothesis, we have $\hat{t} \in \mathsf{A}'_{\mathsf{Cons}}(\hat{\sigma})$, and with the (acq)-rule, we get $t \in \mathsf{A}'_{\mathsf{Cons}}(\mathsf{as}^{\langle}_{\mathsf{t}}(x, \hat{\sigma})[\mathsf{u} \mapsto u'])$, which completes the case.

The case $t = \rangle_x \hat{t}$ is proved by straightforward application of the induction hypothesis and the (rel)-rule.

The case $l = \varepsilon$ is trivial.

The case $l = t\hat{l}$ is shown analogously to the case $h = (t, X)\hat{h}$. □

Next, we show that, given a hedge $h \in \mathsf{L}(\mathsf{A}'_{\mathsf{Cons}})$, we find a polynomial size subset of the rules of $\mathsf{A}'_{\mathsf{Cons}}$, such that h is also accepted by this subset of rules. As discussed above, we count rule-templates of the form $\langle\rangle_X(q_1, q_2) \rightarrow_\delta q \; \forall X \subseteq Y$ as one rule here. We already argued that, in an accepting run of $\mathsf{A}_{\mathsf{Cons}}$, there occur only $O(|\mathcal{X}|)$ different acquisition-sets (a), release-sets (r), initially held locksets (X), and acquisition-graphs (g_a), as well as $O(|\mathcal{X}|^2)$ different release-graphs. The same argument also works for $\mathsf{A}'_{\mathsf{Cons}}$. Moreover, the u-components of the states of $\mathsf{A}'_{\mathsf{Cons}}$ only change at acquisition-nodes, and each u-component guessed at a leaf-node is either equal to the u-component at an acquisition-node, or to the u-component guessed by the (h-empty)-rule. Thus, there are only $O(|\mathcal{X}|)$ different u-components during a run of $\mathsf{A}'_{\mathsf{Cons}}$. Together, $\mathsf{A}'_{\mathsf{Cons}}$ requires only $\mathsf{poly}(|\mathcal{X}|)$ states to accept h. As we count the use-rule-templates as one rule, and need to instantiate the (h-cons)-rule for only $O(|\mathcal{X}|)$ different sets X, it also requires only $\mathsf{poly}(|\mathcal{X}|)$ rules. Together, we get the following lemma:

Lemma 9.6. *There is a deterministic polynomial time algorithm that takes a certificate C, and constructs an automaton $\mathsf{A}^{C}_{\mathsf{Cons}}$ of size $\mathsf{poly}(|C|)$, such that $\mathsf{L}(\mathsf{A}^{C}_{\mathsf{Cons}}) \subseteq \mathsf{L}(\mathsf{A}_{\mathsf{Cons}})$.*

Moreover, given a schedulable lock-a/r-hedge h, there is a certificate C of size $\mathsf{poly}(\mathcal{X})$, such that $h \in \mathsf{L}(\mathsf{A}^{C}_{\mathsf{Cons}})$.

Proof. The algorithm first checks that the certificate C consists of a subset of rules from $\mathsf{A}'_{\mathsf{Cons}}$ and a single final state $q_i \in \mathsf{Cons}$. If this is not the case, the certificate is considered invalid, and the algorithm returns an empty automaton. Otherwise, it returns the automaton $\mathsf{A}^{C}_{\mathsf{Cons}}$ that consists of the rules from C, with the only final state q_i. We have

$$\mathsf{L}(\mathsf{A}^{C}_{\mathsf{Cons}}) \subseteq \mathsf{L}(\mathsf{A}'_{\mathsf{Cons}}) = \mathsf{L}(\mathsf{A}_{\mathsf{Cons}}).$$

As described above, for every lock-a/r-hedge h accepted by $\mathsf{A}'_{\mathsf{Cons}}$, we find $\mathsf{poly}(|\mathcal{X}|)$ rules of $\mathsf{A}'_{\mathsf{Cons}}$ that are sufficient to accept h. The certificate C for the hedge h consists of these rules and the final state in that h is accepted. □

Note that it is sufficient to have a certificate of polynomial size in the input, in order to show membership in NP. Here, we additionally demonstrate that the size of the certificate only depends polynomially on the number of locks, thus showing that the high complexity is caused by the number of locks, and not by the program size.

With a slightly more complicated argumentation, we could bound the grade of the polynomial, and show that a certificate of size $O(|\mathcal{X}|^2)$ is sufficient. This would match the complexity of the algorithm presented in Chapter 6. The idea is to also guess the edges that will be added to the release-graph at a release-node, and only check that the actual edges are below the guessed set. Moreover, the certificate would not store the rules of the automaton, but only the sets of locks. The instantiation of the rules with the guessed sets of locks can be done by the verifier in polynomial time. This would only require $O(|\mathcal{X}|)$ different sets of locks to be guessed. As each set of locks can be represented by $|\mathcal{X}|$ bits, the certificate would have size $O(|\mathcal{X}|^2)$.

Bounding the DPN-Acceptor In the last paragraph, we have shown that we only need a polynomial size subset of the automaton $\mathsf{A}'_{\mathsf{Cons}}$ to accept a lock-a/r-hedge. In this paragraph, we show the analogous result for lock-execution-hedges and DPN-Acceptors.

Regard a run of the DPN-Acceptor $D_{\mathsf{A}^{C}_{\mathsf{Cons}}}$ (cf. Definition 5.2) that accepts a schedulable lock-execution-hedge h, and assume that $|C| = \mathsf{poly}(|\mathcal{X}|)$. The DPN has control-states of the form (p, X, q) and $(\mathsf{u}^{b,\tilde{q}}, X, u, q)$. The X-components are the sets of currently acquired locks. They are only changed on final acquisitions and initial releases, not while processing usages of locks. As h is consistent, there are only $O(|\mathcal{X}|)$ different sets of initially held locks. As $\mathsf{ar}(h)$ is schedulable, there is at most one final acquisition and one initial release for each lock. Thus, there are only $O(|\mathcal{X}|)$ different X-components during the run of $D_{\mathsf{A}^C_{\mathsf{Cons}}}$. As $\mathsf{A}^C_{\mathsf{Cons}}$ has size $\mathsf{poly}(|\mathcal{X}|)$, there are only $\mathsf{poly}(|\mathcal{X}|)$ different q- and $\tilde{q}$-components. However, there may be $2^{|\mathcal{X}|}$ different u-components during a single run of $D_{\mathsf{A}^C_{\mathsf{Cons}}}$. We had the analogous problem for the automaton $\mathsf{A}_{\mathsf{Cons}}$: There, we had worst-case exponentially many $\langle\rangle_X$-nodes. The problem was solved by introducing rule-templates of the form $\langle\rangle_X(q_1)q_2 \to q \; \forall X \subseteq Y$. In order to check whether an instance of such a rule-template is applicable, the DPN-Acceptor does not need to precisely collect the set of used locks, but only needs to check whether the set of used locks is a subset of Y, i.e., whether each used lock is contained in Y.

We modify the DPN-Acceptor D_A from Definition 5.2 accordingly. We obtain D'_A by replacing the following rules of D_A:

$$(\mathsf{u}^{b,\tilde{q}}, X, u, q)\gamma \overset{\langle x}{\hookrightarrow} (\mathsf{u}^{b,\tilde{q}}, X, u, q)x^{\perp}\gamma \quad \text{if } \{x\} \setminus X \subseteq u \tag{u-acq}$$

$$(\mathsf{p}^{b}, X, q)\gamma \overset{\langle x}{\hookrightarrow} (\mathsf{u}^{b,\tilde{q}}, X, u, q')x^{\top}\gamma \quad \text{if } \langle\rangle_Y(q')\tilde{q} \to_\delta q \;\forall Y \subseteq u \text{ and } \{x\} \setminus X \subseteq u \tag{u-begin}$$

$$(\mathsf{u}^{b,\tilde{q}}, X, u, q)x^{\top} \overset{\rangle x}{\hookrightarrow} (\mathsf{p}^{b}, X, \tilde{q}) \quad \text{if } \varepsilon_{\mathsf{s}} \to_\delta q \tag{u-end}$$

where δ is the transition relation of A. The automata $\mathsf{A}_{C_{\mathsf{F}}}$ and $\mathsf{A}_{C_{\mathsf{I}}}$ are the same as for D_A.

Intuitively, we replace accumulation of the set of used locks by checking whether the used locks are below the upper bound from the template-rule. The u-component in the new set of rules stores this upper bound, and is not modified during a usage. Hence, we do not require more u-components than there are template-rules in A.

Analogously to Theorem 5.3, we have

$$\mathsf{L}(D'_A) = \mathsf{ar}^{-1}(\mathsf{L}(A)).$$

Proof. The proof is analogous to that of Theorem 5.3. For the induction, the statements for hedges (1) and trees outside usages (2) remain the same. For same-level trees t_s and lockstacks μ with $\mu \overset{t_s}{\to} \varepsilon$, we show the following statement instead:

$$\begin{aligned} &\exists c' \in \mathsf{L}(\mathsf{A}_{C_{\mathsf{F}}}).\ (\mathsf{u}^{b,\tilde{q}}, X, u, q)\mu^{\perp}x^{\top}w \overset{t_s}{\Rightarrow} c'[\varphi] \\ \iff &\exists u'' \subseteq \mathcal{X}, q' \in Q, s \in \mathsf{TAR}^*.\ \mathsf{ar}_{\mathsf{s}}(t_s, X) = (u'', s) \\ &\wedge \varphi = (\mathsf{u}^{b,\tilde{q}}, X, u, q')x^{\top}w \wedge u'' \subseteq u \wedge s\#_{\mathsf{s}}q' \to^*_\delta q \end{aligned} \tag{3}$$

The proof of the cases is analogous to that of Theorem 5.3. □

Next, we have to show that we only need polynomially many states and rules for a single run of $D'_{\mathsf{A}^C_{\mathsf{Cons}}}$ that accepts the lock-execution-hedge h. We already argued that we only need $O(|\mathcal{X}|)$ sets of currently acquired locks (X-components). Moreover, each u-component stems from a template-rule of the tree automaton, thus we only need $O(|\mathsf{A}^C_{\mathsf{Cons}}|) = \mathsf{poly}(|\mathcal{X}|)$ such components. The sets X in the states of the initial automaton $\mathsf{A}_{C_\mathsf{I}}$ are exactly the sets of initially held locks of the threads in h, hence we only need $O(|\mathcal{X}|)$ of them. Also, the automaton $\mathsf{A}_{C_\mathsf{F}}$ requires only rules for the $O(|\mathcal{X}|)$ many X-components that actually occur during the run of the DPN. Together, we get the following lemma:

Lemma 9.7. *There is a deterministic algorithm that takes a certificate C, and constructs in time $\mathsf{poly}(|C|)O(|\mathsf{Act}|)$ a DPN-Acceptor $D^C_{\mathsf{A}_{\mathsf{Cons}}}$ such that $\mathsf{L}(D^C_{\mathsf{A}_{\mathsf{Cons}}}) \subseteq \mathsf{L}(D_{\mathsf{A}_{\mathsf{Cons}}})$ and $|D^C_{\mathsf{A}_{\mathsf{Cons}}}| = \mathsf{poly}(|C|)O(|\mathsf{Act}|)$.*

Moreover, for any lock-execution-hedge $h \in \mathsf{L}(D_{\mathsf{A}_{\mathsf{Cons}}})$, there is a certificate C of size $\mathsf{poly}(|\mathcal{X}|)$, such that $h \in \mathsf{L}(D^C_{\mathsf{A}_{\mathsf{Cons}}})$.

Proof. We first check that the certificate has the form $C = (C', \bar{X})$, for a certificate C' for the algorithm from Lemma 9.6, and a set $\bar{X} \subseteq 2^{\mathcal{X}}$ of sets of locks. If the certificate has the wrong format, we return an empty DPN-Acceptor.

Then, we run the algorithm from Lemma 9.6, and obtain $\mathsf{A}^{C'}_{\mathsf{Cons}}$. Next, $D^C_{\mathsf{A}_{\mathsf{Cons}}}$ is constructed by instantiating the rules of $D'_{\mathsf{A}^{C'}_{\mathsf{Cons}}}$ only for X-components from $\bar{X}$. As we have to instantiate the rules for base- and spawn-actions for all possible action labels from Act, we get the extra factor $|\mathsf{Act}|$ in the size and runtime estimation.

The rules of $D^C_{\mathsf{A}_{\mathsf{Cons}}}$ are a subset of the rules of $D'_{\mathsf{A}^{C'}_{\mathsf{Cons}}}$, and thus we have

$$\mathsf{L}(D^C_{\mathsf{A}_{\mathsf{Cons}}}) \subseteq \mathsf{L}(D'_{\mathsf{A}^{C'}_{\mathsf{Cons}}}) = \mathsf{L}(D_{\mathsf{A}^{C'}_{\mathsf{Cons}}}) = \mathsf{ar}^{-1}(\mathsf{L}(\mathsf{A}^{C'}_{\mathsf{Cons}})) \subseteq \mathsf{ar}^{-1}(\mathsf{L}(\mathsf{A}_{\mathsf{Cons}})) = \mathsf{L}(D_{\mathsf{A}_{\mathsf{Cons}}}).$$

Given a lock-execution-hedge $h \in \mathsf{L}(D_{\mathsf{A}_{\mathsf{Cons}}})$, we have $\mathsf{ar}(h) \in \mathsf{L}(\mathsf{A}_{\mathsf{Cons}})$. By Lemma 9.6, we obtain C' with $|C'| = \mathsf{poly}(|\mathcal{X}|)$ such that $\mathsf{ar}(h) \in \mathsf{L}(\mathsf{A}^{C'}_{\mathsf{Cons}})$. Thus, we choose $C := (C', \bar{X})$ where the set $\bar{X}$ contains the X-components that occur in the run of $D'_{\mathsf{A}^{C'}_{\mathsf{Cons}}}$ that accepts h. As argued above, their number is bounded by $\mathsf{poly}(|\mathcal{X}|)$. □

Bounding the Automaton for Consistent Configurations The cross-product construction requires an automaton $\mathsf{A}_{\mathsf{Conf}^{\mathsf{ls}}}$ that accepts the consistent configurations (cf. Section 5.2). It has exponentially many states in the number of locks. However, for accepting a single consistent configuration, only polynomially many states are required. Thus, we get:

Lemma 9.8. *There is an algorithm that, given a certificate C, constructs an automaton $\mathsf{A}^C_{\mathsf{Conf}^{\mathsf{ls}}}$ with $\mathsf{L}(\mathsf{A}^C_{\mathsf{Conf}^{\mathsf{ls}}}) \subseteq \mathsf{Conf}^{\mathsf{ls}}$. The algorithm needs time $\mathsf{poly}(|P||\Gamma||C|)$.*

Moreover, for any consistent configuration $c \in \mathsf{Conf}^{\mathsf{ls}}$, there exists a certificate C of size $\mathsf{poly}(|\mathcal{X}|)$ such that $c \in \mathsf{L}(\mathsf{A}^C_{\mathsf{Conf}^{\mathsf{ls}}})$.

Proof. We use the automaton $\mathsf{A}_{\mathsf{Conf}^{\mathsf{ls}}}$ that was constructed in Section 5.2. The certificate C is interpreted as a set of sets of locks, and the automaton $\mathsf{A}^{C}_{\mathsf{Conf}^{\mathsf{ls}}}$ is constructed by instantiating the rules of $\mathsf{A}_{\mathsf{Conf}^{\mathsf{ls}}}$ only for locksets from C. The time estimation follows from $|\mathcal{X}| = O(|\Gamma|)$ (every lock is bound to a stack-symbol).

In a single accepting run of $\mathsf{A}_{\mathsf{Conf}^{\mathsf{ls}}}$, only $O(|\mathcal{X}|)$ different sets of locks are required, as the sets of locks only increase, and the set of locks for a thread is disjoint from the overall set of locks. These sets of locks are chosen as certificate C. □

NP-Algorithm for Negation-Free EF-Formulas Combining the results of the last paragraphs, we get:

Lemma 9.9. *There is a polynomial time algorithm that, given a Monitor-DPN M, an automaton A, and a certificate C, constructs automata*

$$\mathsf{pre}^{C}_{\mathsf{ls},M}(A) \text{ and } \mathsf{pre}^{*,C}_{\mathsf{ls},M}(A),$$

such that

$$\mathsf{L}(\mathsf{pre}^{C}_{\mathsf{ls},M}(A)) \subseteq \mathsf{pre}_{\mathsf{ls},M}(\mathsf{L}(A) \cap \mathsf{Conf}^{\mathsf{ls}} \cap \mathsf{valid}) \quad |\mathsf{pre}^{C}_{\mathsf{ls},M}(A)| = |A|\mathsf{poly}(|M||C|)$$
$$\mathsf{L}(\mathsf{pre}^{*,C}_{\mathsf{ls},M}(A)) \subseteq \mathsf{pre}^{*}_{\mathsf{ls},M}(\mathsf{L}(A) \cap \mathsf{Conf}^{\mathsf{ls}} \cap \mathsf{valid}) \quad |\mathsf{pre}^{*,C}_{\mathsf{ls},M}(A)| = |A|\mathsf{poly}(|M||C|)$$

Moreover, given a configuration $c \in \mathsf{pre}_{\mathsf{ls},M}(\mathsf{L}(A) \cap \mathsf{Conf}^{\mathsf{ls}} \cap \mathsf{valid})$, there is a certificate C of size $\mathsf{poly}(|\mathcal{X}|)$ such that $c \in \mathsf{L}(\mathsf{pre}^{C}_{\mathsf{ls},M}(A))$. Also, given a configuration $c \in \mathsf{pre}^{*}_{\mathsf{ls},M}(\mathsf{L}(A) \cap \mathsf{Conf}^{\mathsf{ls}} \cap \mathsf{valid})$, there is a certificate C of size $\mathsf{poly}(|\mathcal{X}|)$ such that $c \in \mathsf{L}(\mathsf{pre}^{*,C}_{\mathsf{ls},M}(A))$.

Proof. The algorithm checks whether the certificate has the form $C = (C_1, C_2)$, such that C_1 is a valid certificate for the algorithm from Lemma 9.7 and C_2 is a valid certificate for the algorithm from Lemma 9.8. Otherwise, the empty automaton is returned. Then, it computes in polynomial time the automaton

$$\mathsf{pre}^{*,C}_{\mathsf{ls},M}(A)) := \pi_1(\mathsf{pre}^{*}_{\times}(\pi_1^{-1}(\mathsf{L}(A)) \cap \mathsf{L}(\mathsf{A}_{C_{\mathsf{F}\times}})) \cap \mathsf{A}'_{C_{\mathsf{I}\times}}),$$

where $(\mathsf{A}_{C_{\mathsf{I}\times}}, \mathsf{A}_{C_{\mathsf{F}\times}}, M_\times)$ is the cross-product of M and $D^{C_1}_{\mathsf{A}_{\mathsf{Cons}}} = (\mathsf{A}_{C_{\mathsf{I}}}, \mathsf{A}_{C_{\mathsf{F}}}, M_2)$, and we set

$$\mathsf{A}'_{C_{\mathsf{I}\times}} := \mathsf{valid}_\times \cap \pi_1^{-1}(\mathsf{A}^{C_2}_{\mathsf{Conf}^{\mathsf{ls}}}) \cap \pi_2^{-1}(\mathsf{A}_{C_{\mathsf{I}}}).$$

The size bound for this automaton is shown analogously to Theorem 6.5, using Lemmas 9.7 and 9.8.

The proof is completed by the following calculation: Similar to Section 6.3, we have

$$c \in \mathsf{pre}^{*}_{\mathsf{ls},M}(\mathsf{L}(A) \cap \mathsf{Conf}^{\mathsf{ls}} \cap \mathsf{valid})$$
$$\iff \exists h \in \mathsf{H}, c' \in \mathsf{L}(A).\ c \in \mathsf{Conf}^{\mathsf{ls}} \cap \mathsf{valid} \wedge c \overset{h}{\Rightarrow} c' \wedge (h \times \mathsf{ls}(c)) \in \mathsf{L}(D_{\mathsf{A}_{\mathsf{Cons}}})$$

With Lemma 9.7, we get:

$$\begin{aligned}\iff \exists h \in \mathsf{H}, c' \in \mathsf{L}(A), C_1.\ &|C_1| = \mathsf{poly}(|\mathcal{X}|)\\ &\wedge c \in \mathsf{Conf}^{\mathsf{ls}} \cap \mathsf{valid} \wedge c \overset{h}{\Rightarrow} c' \wedge (h \times \mathsf{ls}(c)) \in \mathsf{L}(D^{C_1}_{\mathsf{A}_{\mathsf{Cons}}})\end{aligned}$$

Using the cross-product construction, we get, analogously to Section 5.2:

$$\iff \exists C_1.\ |C_1| = \mathsf{poly}(|\mathcal{X}|) \wedge c \in \pi_1(\mathsf{pre}^*_{\times}(\pi_1^{-1}(\mathsf{L}(A)) \cap \mathsf{L}(\mathsf{A}_{C\mathsf{F}_\times})) \cap \mathsf{A}_{C\mathsf{I}_\times})$$

where $(\mathsf{A}_{C\mathsf{I}_\times}, \mathsf{A}_{C\mathsf{F}_\times}, M_\times)$ is the cross-product of M and $D^{C_1}_{\mathsf{A}_{\mathsf{Cons}}}$. Unfolding the definition of $\mathsf{A}_{C\mathsf{I}_\times}$, using Lemma 9.8, and folding the definition of $\mathsf{A}'_{C\mathsf{I}_\times}$, we get:

$$\iff \exists C_1, C_2.\ C_1, C_2 = \mathsf{poly}(|\mathcal{X}|) \wedge c \in \pi_1(\mathsf{pre}^*_{\times}(\pi_1^{-1}(\mathsf{L}(A')) \cap \mathsf{L}(\mathsf{A}_{C\mathsf{F}_\times})) \cap \mathsf{A}'_{C\mathsf{I}_\times})$$

The argumentation for immediate predecessor sets is analogous. □

This algorithm is now used to compute the semantics of a negation-free EF-formula.

Theorem 9.10. *Given a negation-free EF-formula φ, a Monitor-DPN M, and a certificate C, there is an algorithm that computes an automaton A^C_φ of size*

$$|\mathsf{A}^C_\varphi| = s_{\mathsf{P}}(\varphi)^{n_{\mathsf{P}}(\varphi)} \mathsf{poly}(M)^{n_{\mathsf{EX}}(\varphi)+n_{\mathsf{EF}}(\varphi)} \mathsf{poly}(C),$$

such that $\mathsf{L}(\mathsf{A}_\varphi) \subseteq [\![\varphi]\!]$. The algorithm is polynomial in $|M|$, $|C|$, and $s_{\mathsf{P}}(\varphi)$, and exponential in the number of operators of φ.

Moreover, for any configuration $c \in [\![\varphi]\!]$, we get a certificate C of size

$$|C| = \mathsf{poly}(|\mathcal{X}|(n_{\mathsf{EF}}(\varphi) + 1)),$$

such that $c \in \mathsf{L}(\mathsf{A}^C_\varphi)$.

Proof. The automaton A^C_φ is computed similar to the algorithm presented in Section 6.4.2. However, we use the nondeterministic predecessor set computation from Lemma 9.9. Thus a certificate of size $\mathsf{poly}(|\mathcal{X}|)$ is required for each of the $n_{\mathsf{EF}}(\varphi)$ EF-operators. (The intermediate predecessor sets are computed with the polynomial algorithm from Section 7.2.2.) Moreover, we intersect the final result with the the set of consistent configurations. Due to Lemma 9.8, this requires another certificate of size $\mathsf{poly}(|\mathcal{X}|)$.

The size of the resulting automaton is shown analogously to Section 6.4.2. □

We immediately get the following corollary, which is the main result of this subsection:

Corollary 9.11. *Model-checking negation-free EF-formulas with a fixed number of operators is in NP for Monitor-DPNs.* □

Finally, note that our result does not show NP-easiness for checking EF-formulas with unboundedly many operators, as the verifier requires exponential time in the number of atomic propositions and path-operators.

9.2.2.3 Well-Nested, Non-Reentrant Locks

In Chapter 8, we briefly discussed how the predecessor set computation that was developed in this thesis can be adapted to DPNs with well-nested, non-reentrant locks (Lock-DPNs). Here, it was sufficient to keep track of the set of currently acquired locks in configurations, and the DPN-Acceptor $D_{\mathsf{A_{Cons}}}$ could be realized without a stack. The same modifications that we did for monitors also apply to to the stackless version of $D_{\mathsf{A_{Cons}}}$. Thus, we get an NP-algorithm also for well-nested, non-reentrant locks. The cross-product construction of a Lock-DPN and a stackless DPN-Acceptor can be done in polynomial time, along the lines of [79]. As this thesis is focused on reentrant monitors, we omit any formal details here, and just state the result:

Theorem 9.12. *Model-checking negation-free EF-formulas with a fixed number of operators is in NP for Lock-DPNs.*

9.2.2.4 Model-Checking DFNs without Spawns inside Monitors

In the last subsection, we have shown that model-checking Monitor-DPNs against negation-free EF-formulas with a fixed number of operators and regular atomic propositions is in NP. In this subsection, we present a polynomial time algorithm for model-checking fixed-size, negation-free, double-indexed EF-formulas for Monitor-DFNs without spawns inside monitors.

The algorithm is based on the observation that we only need to keep track of two simultaneously running threads in order to check a double-indexed atomic proposition. If a thread is spawned, both the spawning and the spawned thread hold no locks, and thus can be discarded if not involved in satisfying any double-indexed atomic proposition. Thus, in order to check a formula φ with n double-indexed atomic propositions, it is sufficient to keep track of $2n$ threads simultaneously.

The first step of the polynomial algorithm is to bound the configurations of the DFN to at most $2n$ threads. When executing a spawn-rule, we nondeterministically choose to drop the spawning or the spawned thread from the configuration or to keep both threads if this would not exceed the limit of $2n$ threads. The second step is to model-check the resulting *bounded DFN*, which can be done in polynomial time.

Let $M = (P, \mathsf{Act}, \mathcal{X}, \Delta, \mathsf{locks})$ be the Monitor-DFN, and $p_0 \in P$ be the start configuration. Note that $\mathsf{locks} : P \to \mathcal{X}^*$ is a function from control-states to lockstacks here, and the rules are assumed to be compatible with this function. The bounded DFN A is a finite-state machine with the states $Q = \{c \in \mathsf{Conf}^{\mathsf{ls}} \mid$

$|c| \leq 2n\}$ and the following rules:

$$\begin{array}{lll}
c_1pc_2 \xrightarrow{o}_A c_1p'c_2 & \text{for } p \overset{o}{\hookrightarrow} p' \in \Delta \text{ and } c_1p'c_2 \in \mathsf{Conf}^{\mathsf{ls}} & \text{(no-spawn)} \\
c_1pc_2 \xrightarrow{o}_A c_1p_sp'c_2 & \text{for } p \overset{o}{\hookrightarrow} p_s\sharp p' \in \Delta,\ |c_1p_sp'c_2| \leq 2n & \text{(spawn-1)} \\
c_1pc_2 \xrightarrow{o}_A c_1p'c_2 & \text{for } p \overset{o}{\hookrightarrow} p_s\sharp p' \in \Delta & \text{(spawn-2)} \\
c_1pc_2 \xrightarrow{o}_A c_1p_sc_2 & \text{for } p \overset{o}{\hookrightarrow} p_s\sharp p' \in \Delta & \text{(spawn-3)}
\end{array}$$

The (no-spawn)-rule simply changes the state of a thread, and ensures that a step can only be applied if the new configuration is consistent. The (spawn-1)-rule spawns a new thread, and keeps both, the spawning and the spawned thread. This is only possible if the number of threads does not exceed $2n$. The (spawn-2)-rule drops the spawned thread, and the (spawn-3)-rule drops the spawning thread. The bounded DFN has less than $|P|^{2n+1}$ different states, thus its size is polynomial in $|P|$ (as n is a constant).

By assumption, spawn-statements are not done inside monitors, and spawned threads hold no monitors initially. Thus, the (spawn-2)- and (spawn-3)-rules only drop threads that hold no locks. Hence, any execution of A corresponds to an execution of M:

$$b \rightarrow_A^* b' \implies \exists c'.\ b' \preceq c' \wedge b \rightarrow_M^* c'.$$

Here, $\preceq$ denotes the subword relation, i.e., we have $c \preceq c'$, iff c can be obtained from c' by deleting some elements. As atomic propositions remain valid when adding additional threads, we have

$$b \models_A \varphi \implies b \models_M \varphi.$$

For the other direction, we show that we do not need to keep more than $2n$ threads in order to satisfy φ. Formally, we have to apply a generalization for the inductive proof. We show

$$\begin{array}{l}
c \models_M \varphi \\
\implies \exists b.\ b \preceq c \wedge |b| \leq \mathsf{idx}(\varphi) \wedge (\forall b'.\ b \preceq b' \preceq c \wedge |b'| \leq 2n \implies b' \models_A \varphi)\,,
\end{array}$$

where $\mathsf{idx}(\varphi)$ is twice the number of atomic propositions in φ. Intuitively, this proposition fixes a set b of threads that are required to satisfy φ, and states that φ is satisfied on the bounded DFN for all sub-configurations of c that contain b. This generalization is required for conjunction and disjunction of formulas, where we have to consider the required threads of both formulas.

The proof is done by rather straightforward induction on the structure of φ: If φ is an atomic proposition, we have $\mathsf{idx}(\varphi) = 2$ and choose b to contain the two threads that satisfy the atomic proposition. If φ is a conjunction or disjunction of φ_1 and φ_2, we choose b to contain those threads required by φ_1 or φ_2. If we have $\varphi = \mathsf{EF}\varphi'$ or $\varphi = \mathsf{EX}\varphi'$, we include a thread in b if it is required by φ', or if it spawns a required thread. During the execution, a spawned thread is dropped if it is not required. Also the spawning thread is dropped if its only purpose was to spawn a required thread. Thus, we do not need to consider more than $\mathsf{idx}(\varphi')$ threads at any point of the execution, and the execution can be simulated by A.

Summarized, we have $p_0 \models_M \varphi$ if and only if $p_0 \models_A \varphi$. As model-checking of CTL for finite-state machines can be done in polynomial time [27], we get the following theorem:

Theorem 9.13. *Given as inputs a DFN M that spawns no threads inside monitors, a control-state p_0, and a fixed-size, negation-free, double-indexed EF-formula φ. Then, there is a polynomial time algorithm that decides $p_0 \models_M \varphi$.* □

9.3 Stronger Synchronization Mechanisms

In the last sections, we discussed the complexity of checking models with locks. We discussed monitors and well-nested, non-reentrant locks. In this section, we discuss stronger synchronization mechanisms. As explained in Section 8.1, we left open the decidability of checking models with well-nested, reentrant locks. Non-well-nested, non-reentrant locks can be used to emulate rendezvous-communication [57]. Moreover, in [44], we analyzed Join-Lock-DPNs, which use well-nested, non-reentrant locks and join-synchronization. Thus we regard models that synchronize via locks and joins, and models that synchronize via rendezvous.

Rendezvous-synchronization allows two threads to wait for each other, and then simultaneously execute two statements. For this purpose, we assume a set of synchronized actions $\{a_1, \bar{a}_1, \dots, a_n, \bar{a}_n\}$ with $n \geq 2$, such that an a_i-step must pair with an $\bar{a}_i$ step such that both steps are executed simultaneously, and, vice versa, an $\bar{a}_i$ step must pair with an a_i-step. Rendezvous-synchronization, usually in the form of synchronous message-passing, is found in many frameworks for distributed computing, like the well-known *Message-Passing Interface* (MPI) (cf. [49]).

Intuitively, join-synchronization allows a thread to wait until some other thread has terminated. Join-synchronization is supported by many programming languages, e.g. Java [47]. For our purpose, we introduce Join-Lock-DPNs. A Join-Lock-DPN is a Lock-DPN, with a special jo-action. A thread t can execute a jo-action only if all threads created by t so far have terminated. Termination of a thread is modeled by reaching a special control-state. We studied Join-Lock-DPNs in [44], and showed that regular set reachability is decidable. Here, we show that already deciding reachability of a single control-state is PSPACE-hard. For a formal semantics of Join-Lock-DPNs, we refer to [44]. However, an intuitive understanding of the semantics is sufficient to follow the reduction presented in this section.

Table 9.2: Complexity of checking pairwise reachability.

$\mathsf{EF}(p_1 \| p_2)$	DPN	PPDS	2PDS	DFN	PFSM	nFSM
Monitors	NP	NP	NP	NP	P	P
Join-Lock	≥PSPACE	—	—	≥PSPACE	—	—
Rendezvous	undec.	undec.	undec.	≥NP	NP	P

Table 9.2 shows the complexity for pairwise reachability properties for models with various synchronization mechanisms. The complexities of the first row have been shown in Section 9.2. The complexities for rendezvous-communication follow straightforwardly from results of Taylor [113] (PFSM) and Ramalingam [104] (2PDS). Note that we do not investigate here whether the NP-easiness results of Taylor [113] also apply to DFNs.

It remains to show the results for Join-Lock-DPNs and Join-Lock-DFNs. Note that join-synchronization only makes sense for models with thread creation. We show, by reduction from QBF, that already deciding reachability of a single control-state from the start configuration is PSPACE-hard for Join-Lock-DFNs. Given a QBF (V, C), we construct the following program:

```
t_1:      sync (x_1){spawn t_2; join } OR sync (¬x_1){spawn t_2; join }; a: terminate
t_2:      sync (x_2){spawn t_3; join }; sync (¬x_2){spawn t_3; join }; terminate
...
t_n:      sync (x_n){spawn t_{n+1}; join }; sync (¬x_n){spawn t_{n+1}; join }; terminate
t_{n+1}:
          sync (l_11){} OR sync (l_12){} OR sync (l_13){}
          ...
          sync (l_m1){} OR sync (l_m2){} OR sync (l_m3){}
          terminate
```

The chooser threads $t_1 \dots t_n$ choose all valuations that have to be checked, one by one. An existentially quantified variable is modeled by nondeterministic choice, and a universally quantified variable is modeled by sequentially producing both valuations. It is straightforward to show that the above program can reach label a if and only if the QBF is true. The above program translates to a Join-Lock-DFN with non-reentrant monitors, and thus deciding single control-state reachability for Join-Lock-DFNs is PSPACE-hard. The result for Join-Lock-DPNs follows from the fact that Join-Lock-DPNs are more expressive than Join-Lock-DFNs.

9.4 Discussion and Related Work

In this chapter, we have established upper and lower complexity bounds for various model-checking problems. The main result is that model-checking negation-free EF-formulas with a fixed number of operators and regular atomic propositions is NP-complete for Monitor-DPNs. The easiness part of this result implies that there are NP-algorithms for various interesting problems that can be mapped to those EF-formulas. These include detection of data-races and atomic-set serializability violations, bitvector-analysis, and bounded model-checking with a constant bound. Moreover, the high complexity of the problem is induced by the number of locks, i.e., the number of nondeterministic choices required by the NP-algorithm only depends on the number of locks, not on the program size. This also matches the complexity of the algorithm that we developed in Chapter 6, which is exponential only in the number of locks.

In order to show that our model-checking algorithm does not solve a too general problem at the cost of increased complexity, we have established lower bounds for rather special problems. These imply that detection of data-races and atomic-set serializability violations, as well as bitvector-analysis and bounded model-checking with a bound of two rendezvous-communications is NP-hard, even for more special models than Monitor-DPNs. Our hardness results also apply to various problems on parallel pushdown systems communicating via well-nested, non-reentrant locks, which have been studied recently without explicitly stating lower complexity bounds: Pairwise reachability properties [57]; model-checking of double-indexed LTL\X-formula [55], LTL(X,F)-formula [56], and alternation free mu-calculus [56]; reachability properties w.r.t. phase-automata [61, 63]; and bitvector-analysis [39].

We also regarded deadlock-free models with inescapable locks and no spawns inside monitors. The assumptions of inescapable locks and no spawning inside monitors are realistic, and assuming deadlock-free models makes sense if a prior analysis is used to verify absence of deadlocks. Interestingly, the complexity changes for this locking discipline: There is a polynomial algorithm for model-checking fixed-size, double-indexed EF-formulas for DFNs, while regular set reachability remains hard. Also, iterated pairwise reachability is hard for models with a stack. However, we had to leave open the complexity of the pairwise reachability problem for models with a stack. This problem is important, as it is equivalent to data-race detection, and a lower bound would also transfer to bitvector-analysis and atomic-set serializability violation detection.

In order to show that our problem is a rather general NP-complete problem, we have shown PSPACE-hardness of some slightly more general problems. When increasing the expressiveness of the properties, the problem gets harder: For EF-formulas with negation, the problem is PSPACE-hard, already for fixed-size, single-indexed formula and 1PDS without locks [13]. For unbounded size, negation-free EF-formulas with regular atomic propositions, the problem is also PSPACE-hard, already for DFNs and 1PDS. However, we had to leave open the exact complexity of model-checking unbounded size, negation-free, double-indexed EF-formulas. Here, the best lower bound that we could establish is NP-hardness for PFSMs without locks.

Also increasing the expressiveness of the model makes the problem harder: Deciding reachability properties for Join-Lock-DPNs [44] is PSPACE-hard. For even more powerful synchronization mechanisms, like shared memory or rendezvous, problems tend to get undecidable for models with a stack [104].

We already mentioned most of the related work: Bouajjani et al. [13] apply predecessor set computation to model-check linear and branching time logics for pushdown systems, and also establish lower complexity bounds. Among other results, they show PSPACE-hardness of model-checking fixed-size (single-indexed) EF-formulas for pushdown systems, which implies our hardness results for 2PDS, PPDS, and DPNs. Esparza [34] studies the complexity of model-checking linear and branching time logics for Petri-nets and basic parallel processes (BPPs). Among other results, he shows that model-checking EF-logic for BPPs is PSPACE-hard. We have transferred this result to PFSMs. Taylor [113] shows that checking various interesting properties for PFSMs with rendezvous-communication is NP-complete, and Ramalingam [104] shows that these problems become undecidable when regarding a model with stacks, e.g. 2PDS.

Regarding program analysis for concurrent programs without locks, interprocedural bitvector-analysis seems to be robustly in polynomial time for many classes of models, among them programs with parallel calls [110] and programs with parallel calls and thread creation [76]. These analysis are based on the idea of computing *possible interference*, which has first been used by Knoop et al. [65] for intraprocedural analysis of parallel programs. However, more complex analysis problems tend to become harder again: Müller-Olm and Seidl [89, 92] show that interprocedural copy-constants and some related problems (like faint variables and slicing) are undecidable for programs with parallel calls. Even the intraprocedural variants of these analysis are PSPACE-complete, and the intraprocedural variants for loop-free programs are still (co)NP-complete. They also show that their hardness results do not depend on the ability of parallel procedure calls to synchronize on termination, and thus the problems are also undecidable for parallel pushdown systems or DPNs. Interestingly, interprocedural copy-constants and faint variables become decidable for programs with parallel calls, when assignments are interpreted non-atomically, i.e., a context switch may occur after reading the right-hand side, but before writing the left-hand side [90, 91]. However, the problem is still co-NP-hard, and there is evidence that it is strictly harder than co-NP [90]. Whether this decidability result transfers to DPNs or even DPNs with locks is left to future research.

Finally, we used a subsumption ordering on acquisition structures for our NP-easiness result. We originally used a similar subsumption ordering as an optimization, reducing sets of collected acquisition structures to their antichains w.r.t. the subsumption ordering [78].

10 Conclusion

Concurrent programming becomes more and more important, as concurrent hardware, like multicore processors, is becoming popular. It is prone to subtle errors, like data-races, that are easily missed by testing. This increases the need for automatic verification methods for concurrent programs. Modern high-level languages, like Java, support concurrency via dynamic thread creation, monitors, and shared memory.

In this thesis, we have studied Monitor-DPNs, an abstract model suited for this type of concurrent programs. It supports procedures, dynamic thread creation, and reentrant monitors. We have presented an algorithm for precise lock-sensitive predecessor set computation that can be used to decide various interesting properties, among them absence of data-races and EF-logic.

Our algorithm requires polynomial time in the size of the program, and exponential time in the number of locks. However, the exponential complexity is a worst-case bound that is not reached by typical programs, as suggested by experimental results on the closely related PPDS-model [39, 57, 63]. Suitable abstractions from real programming languages to Monitor-DPNs are subject of current research.

While automatic verification tries to prove absence of errors, a complementary method is error-detection that tries to find errors. Here, our methods can be applied to increase the precision of bounded model-checking.

From a theoretical point of view, we have extended the decidability boundary for model-checking of concurrent programs with locks. Prior to this research, the state of the art was checking pairwise reachability (viz. double-indexed temporal logics) for parallel pushdown systems with well-nested, non-reentrant locks. We have extended this to reachability between arbitrary regular sets (viz. EF-logic with regular atomic propositions) for DPNs with well-nested, non-reentrant locks (in [79]) or reentrant monitors (in this thesis). Moreover, we have established precise upper and lower complexity bounds for our problems, showing that model-checking negation-free EF-logic with a fixed number of operators is NP-complete, and that the complexity depends on the number of locks rather than on the size of the program. The NP-hardness results already hold for many practical problems like data-race detection and bitvector-analysis.

Our method is based on execution-trees, which are a true-concurrency semantics for DPNs. In addition to encoding the causal order of steps, an execution-tree is ordered, i.e., a spawn-node has a left successor that describes the execution of the spawned thread, and a right successor that describes the remainder of the execution of the spawning thread. The ordering allows us to keep track of the execution of a thread, and enables handling of execution-trees with tree automata based methods.

While reachability analysis with regular constraints on the interleaved executions is undecidable for DPNs, reachability analysis with regular constraints on the execution-trees is efficiently decidable. It remains decidable even for constraints given by the language of DPN-Acceptors, a generalization of tree automata, which may make limited use of a stack. We obtained this result by reducing constrained to unconstrained reachability analysis, using a cross-product construction.

Moreover, we have characterized the set of schedulable execution-trees of a Monitor-DPN by a DPN-Acceptor. This result has been obtained by generalizing the acquisition history method of Kahlon et al. [57] to execution-trees. We used acquire/release-trees as an intermediate model, to get an elegant and modular approach.

Combining these results, we constrain reachability analysis to only consider schedulable execution-trees, and thus reduce lock-sensitive reachability analysis to lock-*insensitive* reachability analysis. The lock-insensitive reachability analysis is then performed by the existing predecessor set algorithm of Bouajjani et al. [16].

Topics of current and future research include the generalization to more expressive but still decidable models, exploration of efficient implementations, exploration of suitable abstractions from real programming languages, and mechanically verified analysis algorithms. We already generalized our methods to reachability analysis for Join-Lock-DPNs [44]. The next step is generalization to CDPNs with locks and to more powerful analyses, e.g. bitvector-analyses and atomic-set serializability violation detection.

Regarding efficient implementations, we are currently exploring lock-sensitive analysis via regular execution-trees [44, 74]. This technique computes successor sets rather than predecessor sets, and thus does not suffer from state-space explosion due to generating acquisition structures for unreachable executions. While we only cover reachability properties from the start configuration in [44, 74], we are currently extending our methods to iterated reachability properties, like bitvector-analysis and atomic-set serializability violation detection. A topic for future research is to use Weighted-DPNs [120, 121] to avoid exploring unreachable executions during predecessor set computation.

For the actual implementation of our algorithms, we are experimenting with logic programming languages derived from PROLOG [28]. They allow for a succinct representation of algorithms that are based on fixed point computations. We may use the DATALOG [43] subset of PROLOG, where predicates are interpreted over finite domains. *Bddbddb* [69, 122] is a symbolic implementation of DATALOG that uses binary decision diagrams (BDDs) [18]. It has been extended in our research group to concepts required for coding acquisition structure based analysis [88]. Evaluation and further improvement of this approach remains future research. We are also planning to evaluate other implementations of PROLOG or related formalisms, like the *succinct solver* [93], which implements the alternation free fragment of least fixed point logic (ALFP) (cf. [93]).

Abstraction from Java [47] to Monitor-DPNs is subject of current research. The greatest challenge is to get precise abstractions of locks, which are referenced via pointers in Java. A promising approach is the random isolation method of Kidd et al. [62] that we already used successfully for parallel pushdown systems [61, 63].

We have mechanically verified most of the results that led to this thesis [72, 77] with the interactive theorem prover Isabelle/HOL [94]. Also the main result of this thesis—i.e., lock-sensitive predecessor set computation for Monitor-DPNs—has been mechanically verified. The formal verification has been completed before this thesis was written, and, consequently, this thesis contains some changes in order to improve the presentation. The main objective of the verification was to show that our methods are correct and effective. Another objective, which is subject of current research, is to derive verified and efficient analysis algorithms from the formalizations. As a first step in this direction, we developed an efficient and mechanically verified tree automata library [71], which is based on our mechanically verified library of efficient collection data structures [73, 75]. We also have (unpublished) formalizations of the saturation algorithm for lock-insensitive predecessor set computation [16, 17] and of the regular execution-tree method [74]. Integrating and extending these results to obtain efficient and mechanically verified lock-sensitive analysis algorithms remains future research.

Bibliography

[1] R. Alur. Marrying words and trees. In *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '07, pages 233–242, New York, NY, USA, 2007. ACM.

[2] R. Alur and P. Madhusudan. Visibly pushdown languages. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, STOC '04, pages 202–211, New York, NY, USA, 2004. ACM.

[3] R. Alur and P. Madhusudan. Adding nesting structure to words. In O. H. Ibarra and Z. Dang, editors, *Developments in Language Theory*, volume 4036 of *Lecture Notes in Computer Science*, pages 1–13. Springer Berlin / Heidelberg, 2006.

[4] R. Alur and P. Madhusudan. Adding nesting structure to words. *J. ACM*, 56:16:1–16:43, May 2009. ISSN 0004-5411.

[5] K. R. Apt. Ten years of Hoare's logic: A survey - part I. *ACM Trans. Program. Lang. Syst.*, 3:431–483, October 1981.

[6] C. Artho, K. Havelund, and A. Biere. High-level data races. *Software Testing, Verification and Reliability*, 13(4):207–227, 2003.

[7] J. Baeten and W. Weijland. Process algebra. *Cambridge Tracts in Theoretical Computer Science*, 18, 1990.

[8] J. Bergstra and J. Klop. Process theory based on bisimulation semantics. In J. de Bakker, W. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *Lecture Notes in Computer Science*, pages 50–122. Springer Berlin / Heidelberg, 1989.

[9] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without bdds. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, pages 193–207, London, UK, 1999. Springer-Verlag.

[10] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. volume 58 of *Advances in Computers*, pages 117 – 148. Elsevier, 2003.

[11] A. Bouajjani and O. Maler. Reachability analysis of pushdown automata. In *Proc. Intern. Workshop on Verification of Infinite-State Systems (Infinity'96)*, 1996.

[12] A. Bouajjani, R. Echahed, and P. Habermehl. Verifying infinite state processes with sequential and parallel composition. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, pages 95–106, New York, NY, USA, 1995. ACM.

[13] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *Proceedings of the 8th International Conference on Concurrency Theory*, pages 135–150, London, UK, 1997. Springer-Verlag. ISBN 3-540-63141-0.

[14] A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '03, pages 62–73, New York, NY, USA, 2003. ACM.

[15] A. Bouajjani, J. Esparza, S. Schwoon, and J. Strejcek. Reachability analysis of multithreaded software with asynchronous communication. In *25th Intern. Conf. on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'05).* LNCS 3821, Springer, 2005.

[16] A. Bouajjani, M. Müller-Olm, and T. Touili. Regular symbolic analysis of dynamic networks of pushdown systems. In *Concurrency Theory. 16th Int. Conf. (CONCUR)*, pages 473–487. LNCS 3653, Springer, 2005.

[17] A. Bouajjani, M. Müller-Olm, and T. Touili. Regular symbolic analysis of dynamic networks of pushdown systems. Available on: `http://cs.uni-muenster.de/sev/publications`, 2005. Extended version with proofs.

[18] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35:677–691, August 1986.

[19] J. R. Büchi. Regular canonical systems. *Arch. Math. Logik Grundlag.*, 6: 91–111, 1964.

[20] J. R. Burch, E. M. Clarke, K. L. Mcmillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proc. of Logic in Computer Science (LICS)*, pages 428–439, 1990.

[21] D. R. Butenhof. *Programming with POSIX Threads.* Addison-Wesley, 1997.

[22] D. Caucal. On the regular structure of prefix rewriting. In A. Arnold, editor, *CAAP '90*, volume 431 of *Lecture Notes in Computer Science*, pages 87–102. Springer Berlin / Heidelberg, 1990.

[23] D. Caucal. On the regular structure of prefix rewriting. *Theoretical Computer Science*, 106(1):61–86, 1992.

[24] E. Clarke. The birth of model checking. In O. Grumberg and H. Veith, editors, *25 Years of Model Checking*, volume 5000 of *Lecture Notes in Computer Science*, pages 1–26. Springer Berlin / Heidelberg, 2008.

[25] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In E. Emerson and A. Sistla, editors, *Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer Berlin / Heidelberg, 2000.

[26] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

[27] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8:244–263, April 1986.

[28] A. Colmerauer, H. Kanoui, P. Roussel, and R. Pasero. Un systeme de communication homme-machine en Francais, 1973. Rapport de Recherche en Intelligence Artificielle, Marseille.

[29] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 2007. release October, 12th 2007.

[30] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[31] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of POPL'77*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York.

[32] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York, NY.

[33] E. W. Dijkstra. Cooperating sequential processes, technical report ewd-123. Technical report, 1965.

[34] J. Esparza. Decidability of model checking for infinite-state concurrent systems. *Acta Informatica*, 34:85–107, 1997.

[35] J. Esparza and J. Knoop. An automata-theoretic approach to interprocedural data-flow analysis. In W. Thomas, editor, *Foundations of Software Science and Computation Structures*, volume 1578 of *Lecture Notes in Computer Science*, pages 642–642. Springer Berlin / Heidelberg, 1999.

[36] J. Esparza and A. Podelski. Efficient algorithms for pre* and post* on interprocedural parallel flow graphs. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '00, pages 1–11, New York, NY, USA, 2000. ACM.

[37] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In E. Emerson and A. Sistla, editors, *Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 232–247. Springer Berlin / Heidelberg, 2000.

[38] Y. Eytani, K. Havelund, S. D. Stoller, and S. Ur. Towards a framework and a benchmark for testing tools for multi-threaded programs: Research articles. *Concurr. Comput. : Pract. Exper.*, 19:267–279, March 2007.

[39] A. Farzan and Z. Kincaid. Compositional bitvector analysis for concurrent programs with nested locks. In *Proceedings of the 17th international conference on Static analysis*, SAS'10, pages 253–270, Berlin, Heidelberg, 2010. Springer-Verlag.

[40] A. Farzan, P. Madhusudan, and F. Sorrentino. Meta-analysis for atomicity violations under nested locking. In *Computer Aided Verification*, pages 248–262. Springer-Verlag, 2009.

[41] A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems (extended abstract). *Electronic Notes in Theoretical Computer Science*, 9:27 – 37, 1997. Infinity'97, Second International Workshop on Verification of Infinite State Systems.

[42] R. W. Floyd. Assigning meanings to programs. *Mathematical aspects of computer science*, 19(19-32):1, 1967.

[43] H. Gallaire, J. Minker, , and J.-M. Nicolas. An overview and introduction to logic and databases. In *Logic and Databases*, pages 123–134. Plenum Press, 1978.

[44] T. M. Gawlitza, P. Lammich, M. Müller-Olm, H. Seidl, and A. Wenner. Join-lock-sensitive forward reachability analysis of concurrent programs with dynamic process creation. In *To appear in Proc. of VMCAI 2011*. Springer, 2011.

[45] D. Gelperin and B. Hetzel. The growth of software testing. *Commun. ACM*, 31:687–695, June 1988.

[46] P. Godefroid. Using partial orders to improve automatic verification methods. In E. Clarke and R. Kurshan, editors, *Computer-Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 176–185. Springer Berlin / Heidelberg, 1991.

[47] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java(TM) Language Specification.* Addison Wesley, 3rd edition edition, 2005. ISBN 978-0321246783.

[48] J. Grabowski. On partial languages. *Annales Societatis Mathematicae Polonae, Fundamenta Informaticae*, IV(2):428–498, 1981.

[49] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI — Portable Parallel Programming with the Message Passing Interface.* MIT Press, 2nd edition edition, 1999.

[50] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12:576–580, October 1969.

[51] C. A. R. Hoare. Monitors: an operating system structuring concept. *Commun. ACM*, 17:549–557, October 1974.

[52] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 3rd edition, 2006.

[53] C. B. Jones. *Development Methods for Computer Programs including a Notion of Interference.* PhD thesis, Oxford University, June 1981. Printed as: Programming Research Group, Technical Monograph 25.

[54] V. Kahlon. Boundedness vs. unboundedness of lock chains: Characterizing decidability of cfl-reachability for threads communicating via locks. In *Proc. of the 24th Annual IEEE Symposium on Logic in Computer Science (LICS), Los Angeles, USA*, August 2009.

[55] V. Kahlon and A. Gupta. An automata-theoretic approach for model checking threads for LTL properties. In *In LICS*, pages 101–110. IEEE Computer Society, 2006.

[56] V. Kahlon and A. Gupta. On the analysis of interacting pushdown systems. In *Proceedings of POPL '07*, pages 303–314, New York, NY, USA, 2007. ACM.

[57] V. Kahlon, F. Ivancic, and A. Gupta. Reasoning about threads communicating via locks. In *Proc. of CAV 2005*, volume 3576 of *LNCS*. Springer, 2005.

[58] R. M. Karp. In complexity of computer computations. *Reducibility Among Combinatorial Problems*, pages 85–103, 1972.

[59] N. Kidd. *Static verification of data-consistency properties.* PhD thesis, Computer Sciences Department, University of Wisconsin, Madison, WI, August 2009. Tech. Rep. TR-1665.

[60] N. Kidd, A. Lal, and T. Reps. Language strength reduction. In M. Alpuente and G. Vidal, editors, *Static Analysis*, volume 5079 of *Lecture Notes in Computer Science*, pages 283–298. Springer Berlin / Heidelberg, 2008.

[61] N. A. Kidd, P. Lammich, T. Touili, , and T. W. Reps. A decision procedure for detecting atomicity violations for communicating processes with locks. In *Proc. of SPIN Workshop on Model Checking of Software (SPIN).* Springer, June 2009.

[62] N. A. Kidd, T. W. Reps, J. Dolby, and M. Vaziri. Finding concurrency-related bugs using random isolation. In *Proc. of Verification, Model Checking, and Abstract Interpretation (VMCAI).* Springer, January 2009.

[63] N. A. Kidd, P. Lammich, T. Touili, , and T. W. Reps. A decision procedure for detecting atomicity violations for communicating processes with locks. *Proc. Int. Journal on Software Tools for Technology Transfer (STTT)*, 13 (1):37–60, 2010.

[64] T. Kleymann. *Hoare Logic and VDM: Machine-Checked Soundness and Completeness Proofs.* PhD thesis, University of Edinburgh, 1998.

[65] J. Knoop, B. Steffen, and J. Vollmer. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. *TOPLAS*, 18(3):268–299, May 1996.

[66] D. Kozen. Lower bounds for natural proof systems. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 254–266, Washington, DC, USA, 1977. IEEE Computer Society.

[67] A. Lal, T. Reps, and G. Balakrishnan. Extended weighted pushdown systems. In *In CAV*, pages 434–448, 2005.

[68] A. Lal, T. Touili, N. Kidd, and T. Reps. Interprocedural analysis of concurrent programs under a context bound. In C. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 282–298. Springer Berlin / Heidelberg, 2008.

[69] M. S. Lam, J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel. Context-sensitive program analysis as database queries. In *Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '05, pages 1–12, New York, NY, USA, 2005. ACM.

[70] P. Lammich. Fixpunkt-basierte optimale Analyse von Programmen mit Thread-Erzeugung. Master's thesis, University of Dortmund, May 2006.

[71] P. Lammich. Tree automata. In G. Klein, T. Nipkow, and L. Paulson, editors, *The Archive of Formal Proofs*. Nov 2009. Formal proof development.

[72] P. Lammich. Isabelle formalization of hedge-constrained pre* and DPNs with locks. Available from `http://cs.uni-muenster.de/sev/publications/`, 2009. Technical Report.

[73] P. Lammich. Collections framework. In G. Klein, T. Nipkow, and L. Paulson, editors, *The Archive of Formal Proofs*. Nov 2009. Formal proof development.

[74] P. Lammich. Tree automata for analyzing dynamic pushdown networks. In *Tagungsband des 15. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS 2009)*, Schriftenreihe des Instituts für Computersprachen. Technische Universität Wien, 2009.

[75] P. Lammich and A. Lochbihler. The isabelle collections framework. In M. Kaufmann and L. Paulson, editors, *Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science*, pages 339–354. Springer Berlin / Heidelberg, 2010.

[76] P. Lammich and M. Müller-Olm. Precise fixpoint-based analysis of programs with thread-creation. In *Proc. of CONCUR 2007*, pages 287–302. Springer, 2007.

[77] P. Lammich and M. Müller-Olm. Formalization of conflict analysis of programs with procedures, thread creation, and monitors. In G. Klein, T. Nipkow, and L. Paulson, editors, *The Archive of Formal Proofs*. Dec 2007. Formal proof development.

[78] P. Lammich and M. Müller-Olm. Conflict analysis of programs with procedures, dynamic thread creation, and monitors. In *Proc. of SAS'08*, volume 5079 of *LNCS*. Springer, 2008. ISBN 978-3-540-69163-1.

[79] P. Lammich, M. Müller-Olm, and A. Wenner. Predecessor sets of dynamic pushdown networks with tree-regular constraints. In *CAV*, pages 525–539, 2009.

[80] R. J. Lipton. Reduction: a method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975. ISSN 0001-0782.

[81] D. Lugiez. Forward analysis of dynamic network of pushdown systems is easier without order. In O. Bournez and I. Potapov, editors, *Reachability Problems*, volume 5797 of *Lecture Notes in Computer Science*, pages 127–140. Springer Berlin / Heidelberg, 2009.

[82] D. Lugiez and P. Schnoebelen. The regular viewpoint on pa-processes. *Theor. Comput. Sci.*, 274:89–115, March 2002.

[83] D. Lugiez and P. Schnoebelen. The regular viewpoint on pa-processes. In *Proceedings of the 9th International Conference on Concurrency Theory*, CONCUR '98, pages 50–66, London, UK, 1998. Springer-Verlag.

[84] R. Mayr. Process rewrite systems. *Information and Computation*, 156: 2000, 1997.

[85] R. Mayr. Model checking pa-processes. In A. Mazurkiewicz and J. Winkowski, editors, *CONCUR '97: Concurrency Theory*, volume 1243 of *Lecture Notes in Computer Science*, pages 332–346. Springer Berlin / Heidelberg, 1997.

[86] R. Mayr. *Decidability and Complexity of Model Checking Problems for Infinite-State Systems.* PhD thesis, TU München, April 1998.

[87] A. Mazurkiewicz. Concurrent program schemes and their interpretations. In *DAIMI Report PB-78*. Department of Computer Science, Aarhus University, Aarhus, Denmark, 1977.

[88] M. Mohr. Ein Datalog-Dialekt mit BDD-Semantik zur symbolischen Analyse paralleler Programme. Master's thesis, WWU Münster, 2011. submitted.

[89] M. Müller-Olm. The complexity of copy constant detection in parallel programs. In *Proceedings of the 18th Annual Symposium on Theoretical Aspects of Computer Science*, STACS '01, pages 490–501, London, UK, 2001. Springer-Verlag.

[90] M. Müller-Olm. *Variations on Constants.* Habilitation thesis, Fachbereich Informatik, Universität Dortmund, August 2002.

[91] M. Müller-Olm. Precise interprocedural dependence analysis of parallel programs. *Theoretical Computer Science (TCS)*, 31:325–388, 2004.

[92] M. Müller-Olm and H. Seidl. On optimal slicing of parallel programs. In *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, STOC '01, pages 647–656, New York, NY, USA, 2001. ACM.

[93] F. Nielson, H. Seidl, and H. R. Nielson. A succinct solver for alfp. *Nordic J. of Computing*, 9:335–372, December 2002.

[94] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[95] S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach. *Commun. ACM*, 19:279–285, May 1976.

[96] C. M. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994. ISBN 0-201-53082-1.

[97] D. Peled. Combining partial order reductions with on-the-fly model-checking. In D. Dill, editor, *Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 377–390. Springer Berlin / Heidelberg, 1994.

[98] C. A. Petri. *Kommunikation mit Automaten.* Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962.

[99] C. A. Petri. Kommunikation mit automaten. *New York: Griffiss Air Force Base, Technical Report RADC-TR-65-377*, 1:1–Suppl. 1, 1966. English translation.

[100] V. Pratt. Modelling concurrency with partial orders. *International Journal of Parallel Programming*, 15:33–71, 1986.

[101] S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *In TACAS*, pages 93–107. Springer, 2005.

[102] S. Qadeer and D. Wu. Kiss: keep it simple and sequential. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, PLDI '04, pages 14–24, New York, NY, USA, 2004. ACM.

[103] J. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In M. Dezani-Ciancaglini and U. Montanari, editors, *International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer Berlin / Heidelberg, 1982.

[104] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *TOPLAS*, 22(2):416–430, 2000.

[105] T. Reps, S. Schwoon, and S. Jha. Weighted pushdown systems and their application to interprocedural dataflow analysis. In R. Cousot, editor, *Static Analysis*, volume 2694 of *Lecture Notes in Computer Science*, pages 1075–1075. Springer Berlin / Heidelberg, 2003.

[106] T. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Sci. Comput. Program.*, 58(1-2):206–263, 2005.

[107] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):pp. 358–366, 1953.

[108] W. J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *J. Comput. Syst. Sci.*, 4:177–192, April 1970.

[109] Ph. Schnoebelen. Decomposable regular languages and the shuffle operator. *EATCS Bulletin*, 67:283–289, Feb. 1999. URL `http://www.lsv.ens-cachan.fr/Publis/PAPERS/PS/Sch-BEATCS99.ps`.

[110] H. Seidl and B. Steffen. Constraint-based inter-procedural analysis of parallel programs. *Nordic Journal of Computing (NJC)*, 7(4):375–400, 2000.

[111] M. Strauch. Realisierung und Anwendung eines automatenbasierten Ansatzes zur Analyse von Programmen mit Threads. Master's thesis, Fachbereich Informatik der Universität Dortmund, 2007.

[112] B. Stroustrup. *The C++ Programming Language (Special Edition)*. Addison Wesley, Reading Mass. USA, 2000.

[113] R. N. Taylor. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Informatica*, 19:57–84, 1983.

[114] A. Valmari. A stubborn attack on state explosion. In *Proceedings of the 2nd International Workshop on Computer Aided Verification*, CAV '90, pages 156–165, London, UK, 1991. Springer-Verlag.

[115] M. Y. Vardi and P. Wolper. Automata theoretic techniques for modal logics of programs: (extended abstract). In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, STOC '84, pages 446–456, New York, NY, USA, 1984. ACM. ISBN 0-89791-133-4.

[116] M. Y. Vardi and P. Wolper. Automata-theoretic techniques for modal logics of programs. *J. Comput. Syst. Sci.*, 32:183–221, April 1986. ISSN 0022-0000.

[117] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '06, pages 334–345, New York, NY, USA, 2006. ACM.

[118] D. von Oheimb. Hoare logic for mutual recursion and local variables. In V. R. C. Pandu Rangan and R. Ramanujam, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 1738 of *LNCS*, pages 168–180. Springer, 1999.

[119] I. Wegener. *Komplexitätstheorie: Grenzen der Effizienz von Algorithmen.* Springer, 2003. ISBN 978-3540001614.

[120] A. Wenner. Optimale Analyse gewichteter dynamischer Push-Down Netzwerke. Master's thesis, WWU Münster, August 2008.

[121] A. Wenner. Weighted dynamic pushdown networks. In A. Gordon, editor, *Programming Languages and Systems*, volume 6012 of *LNCS*, pages 590–609. Springer Berlin / Heidelberg, 2010.

[122] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, PLDI '04, pages 131–144, New York, NY, USA, 2004. ACM.

[123] S. Zilio and D. Lugiez. Xml schema, tree logic and sheaves automata. *Applicable Algebra in Engineering, Communication and Computing*, 17:337–377, 2006.

Tabellarischer Lebenslauf

Peter Lammich,
geboren am 8. Dezember 1980 in Freiburg im Breisgau.
Familienstand: ledig.
Name des Vaters: Siegfried Lammich.
Name der Mutter: Maria Lammich, geb. Perschke.

Schulbildung:	Grundschule von 1987 bis 1991 in Mülheim a. d. Ruhr. Gymnasium von 1991 bis 2000 in Mülheim a. d. Ruhr.
Hochschulreife (*Abitur*):	Am 14. Juni 2000 in Mülheim a. d. Ruhr. Durchschnittsnote: 1.7. Zivildienst von August 2000 bis Juni 2001 in Mülheim a. d. Ruhr.
Studium:	Informatik von Oktober 2001 bis Mai 2006 an der Universität Dortmund.
Promotionsstudiengang:	Ab SoSo 2007 an der Universität Münster.
Prüfungen:	Diplom im Fach Informatik am 18. Mai 2006 an der Universität Dortmund. Gesamturteil: Mit Auszeichnung.
Tätigkeiten:	Studentische Hilfskraft an der Universität Dortmund von April 2002 bis März 2006. Wissenschaftlicher Mitarbeiter an der WWU Münster ab Juni 2006.
Beginn der Dissertation:	April 2007 am Institut für Informatik (Univ. Münster) bei Prof. Dr. Markus Müller-Olm.