Vom Fachbereich Informatik
der Technischen Universität Kaiserslautern
zur Verleihung des akademischen Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
genehmigte Dissertation

Verification Techniques for TSO-Relaxed Programs

Autor: Georgel Ionut Calin

Datum der Disputation: October 14, 2016

Technische Universität
Kaiserslautern

D 386

Abstract

Knowing the extent to which we rely on technology one may think that correct programs are nowadays the norm. Unfortunately, this is far from the truth. Luckily, possible reasons why program correctness is difficult often come hand in hand with some solutions. Consider concurrent program correctness under Sequential Consistency (SC). Under SC, instructions of each program's concurrent component are executed atomically and in order. By using logic to represent correctness specifications, model checking provides a successful solution to concurrent program verification under SC.

Alas, SC's atomicity assumptions do not reflect the reality of hardware architectures. Total Store Order (TSO) is a less common memory model implemented in SPARC and in Intel x86 multiprocessors that relaxes the SC constraints. While the architecturally de-atomized execution of stores under TSO speeds up program execution, it also complicates program verification. To be precise, due to TSO's unbounded store buffers, a program's semantics under TSO might be infinite. This, for example, turns reachability under SC (a PSpace-complete task) into a non-primitive-recursive-complete problem under TSO.

This thesis develops verification techniques targeting TSO-relaxed programs. To be precise, we present under- and over-approximating heuristics for checking reachability in TSO-relaxed programs as well as state-reducing methods for speeding up such heuristics.

In a first contribution, we propose an algorithm to check reachability of TSO-relaxed programs lazily. The under-approximating refinement algorithm uses auxiliary variables to simulate TSO's buffers along instruction sequences suggested by an oracle. The oracle's deciding characteristic is that if it returns the empty sequence then the program's SC- and TSO-reachable states are the same.

Secondly, we propose several approaches to over-approximate TSO buffers. Combined in a refinement algorithm, these approaches can be used to determine safety with respect to TSO reachability for a large class of TSO-relaxed programs. On the more technical side, we prove that checking reachability is decidable when TSO buffers are approximated by multisets with tracked per address last-added-values.

Finally, we analyze how the explored state space can be reduced when checking TSO and SC reachability. Intuitively, through the viewpoint of Shasha-and-Snir-like traces, we exploit the structure of program instructions to explain several state-space reducing methods including dynamic and cartesian partial order reduction.

Acknowledgments

I am, first and foremost, indebted to Prof. Dr. Roland Meyer for his continuous guidance throughout the duration of my PhD. For his insights into the various topics that together encompass concurrency theory — and a part of which are included in this thesis — as well as for his leading by example in terms of dedication, enthusiasm, exigency, and pedagogy, I express to him, my supervisor, my deepest gratitude.

I sincerely thank Prof. Dr. Klaus Schneider for accepting to review this manuscript and for his comments on it. Backtracking chronologically, I am thankful to the handful of friends who read the draft of the thesis and helped with their valuable feedback.

I was very fortunate to interact with and to learn from a small yet select circle of co-authors: Prof. Dr. Ahmed Bouajjani, Dr.-Ing. Egor Derevenetc, Dr. Zhenyue Long, Prof. Dr. Rupak Majumdar, and Prof. Dr. Roland Meyer. I am in particular fond to recall many fascinating discussions with Ahmed, Egor, and Roland during our reciprocal visits in Paris and in Kaiserslautern. Dear colleagues, it was a pleasure to have worked with and to have got to know you all!

I am furthermore grateful to have had such wonderful colleagues and friends during my PhD time in Kaiserslautern. Dear Dan, Daniel, Egor, Emanuele, Florian, Georg, Prof. Madlener, Peter, Reiner, Sebastian, Sebastian, and — foremost — dear Roland, you have all been good friends many a time; I shall never forget it!

Last and without a doubt first in my heart, I am grateful for the continued support and understanding of my intended better half, my parents, and my sisters. Thank you!

Kaiserslautern, January 31, 2016

Georgel Ionuţ Călin

Contents

List of Acronyms

BFS	Breadth-First Search
DFS	Depth-First Search
FIFO	First In, First Out
LTL	Linear Temporal Logic
POR	Partial Order Reduction
SC	Sequential Consistency
SMT	Satisfiability Modulo Theory
SPARC	Scalable Parallel Architecture
TSO	Total Store Order
WQO	Well-quasi-ordering
WSTS	Well-structured Transition System

List of Algorithms

List of Figures

Chapter 1

Introduction

Dekker's solution dates back to 1959 and [...] has been considered a "curiosity", until these issues [...] became relevant for me again and Dekker's solution acted as the basis of my next attempts.

Edsger W. Dijkstra
About the sequentiality of process descriptions (EWD-35)

Half a century after Dijkstra wrote his 35th manuscript, Dekker's mutual exclusion solution is no longer a curiosity. In fact, many protocols can nowadays ensure mutual exclusion in the classical sense: no concurrently executing sequential processes simultaneously reach their critical sections. Nonetheless, it is Dijkstra's seminal work [Dij65] that has since opened the door for an extensive amount of results in the area of concurrent algorithms.

Taking a stride in time, 5 years after Dijkstra received the Turing award in 1972, Pnueli built on earlier work dating back to Prior's tense logic to unify reasoning about the verification of both sequential and concurrent programs using temporal logic specifications [Pnu77]. Two decades later, in 1996, Pnueli also earned the Turing award for his "seminal work introducing temporal logic into computing science".

Shortly afterward, Lamport set the standard for memory consistency models by formalizing sequential consistency [Lam79]. He too received the Turing award in 2013 for his "fundamental contributions to the theory and practice of distributed and concurrent systems".

Under these auspices, Queille and Sifakis [QS82] and, independently, Clarke and Emerson [CE81] proposed the first fully automated algorithms for temporal-logic model checking. The latter three scientists went on to co-receive the Turing award in 2007 for "their roles in developing model checking into a highly effective verification technology".

Fast-forwarding to more recent times — of yet-unknown future Turing awardees — we find that Dijkstra's predicament has once again resurfaced.

Indeed, given the ubiquitous nature of concurrency many "issues" have both grown more complex and multiplied. Meanwhile, unlike half a century ago, Dekker's solution[1] no longer serves as "the basis of next attempts". Instead, it became a way to showcase differences between the — now classical — sequential consistency and other widely-deployed system architectures. To better understand where we stand, consider the original Dekker algorithm in Figure 1.1 — adapted from Dijkstra's EWD-35 manuscript.

All variables used by machines A and B in Figure 1.1 are Boolean — with the convention that $turn \in \{A, B\}$ and $flag_A, flag_B \in \{true, false\}$. Nodes are assumed to be executed atomically. An edge annotated by "yes" is taken out of a diagram node if the node query is positively answered. Conversely, an edge annotated by "no" in the diagram is taken if the corresponding node query is negatively answered. Edges with no labels are unequivocally followed. The loop involving the top two nodes in both A and B set the contention flags to *false*, thus yielding to the other machine. The middle nodes lock the simultaneous access to the bottom nodes that contain the critical sections and that alternate *turn*'s values.

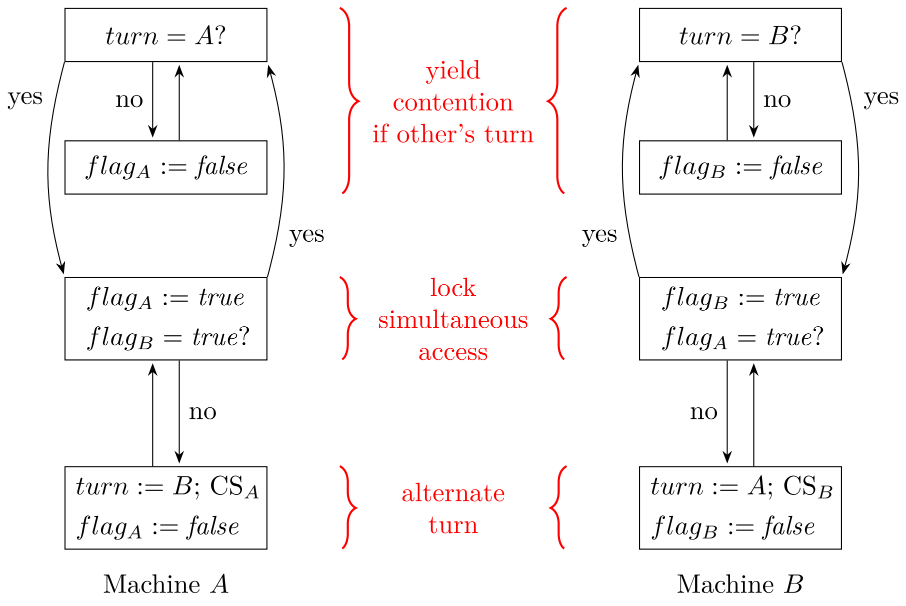


Figure 1.1: Dekker's solution to non-blocking critical-section-exclusion for, as named by Dijkstra, machines A and B. CS$_A$ and CS$_B$ denote the critical sections of A and B and colored annotations indicate the three phases of the algorithm.

As pointed out by Dijkstra, the above Dekker algorithm implements non-blocking critical-section-exclusion. That is, concurrently running A and B does neither block a machine's access to its critical sections nor does it

[1]Dekker's solution will be called *the Dekker algorithm* throughout the manuscript.

witness both machines' simultaneous access to their critical section nodes.

Intuitively, if both A and B were to reach their middle/lock nodes at the same time then they would next move to their top/yield loop to reassess their flag contention. This means that when both machines simultaneously reach their lock nodes neither A nor B moves to the critical section nodes. The machines might, however, reach their critical section nodes by entering them one at a time. Assume, that critical-section-exclusion fails this way. Moreover, wlog due to the algorithm symmetry, assume that A moves to its critical section node first and (without leaving) it is followed by B's move to its critical section node. Then

(1) B is still in its top/yield loop,

(2) $flag_A = \mathit{true}$ holds since A could enter its critical section node, and

(3) if B were to leave its top/yield loop then $\mathit{turn} := B$ must be — or have been — executed in A.

The only way for B to get closer to its critical section node is if (3) occurs. In that case, however, $flag_A = \mathit{true}$ blocks B from entering its critical section node — since by assumption A cannot set $flag_A := \mathit{false}$ and, thus, move out of its critical section node. This proves incorrect the assumption that A and B move one at a time to their critical section nodes and, hence, it shows that critical-section-exclusion holds.

To see that the Dekker algorithm does not block critical section access (in the sense meant by Dijkstra), notice that

(1) A and B cannot be stuck together in their top/yield loop, i.e., the machine whose turn it is can move to its middle/lock node, and

(2) if A and B are simultaneously in their middle/lock nodes then, after they both move to their top/yield loops, one of the machines will stay in its yield loop waiting for the other whose turn it is to reach the critical section node (and, thus, alternate the turn).

Covertly, the critical-section-exclusive behavior of the Dekker algorithm is governed by Lamport's Sequential Consistency (SC) assumption: every concurrent execution is the same as executing operations of the involved machines in some sequential interleaved order and, restrictively, executing operations of each machine adheres to the machine's control flow order.

Nowadays however, the well-behaved SC assumption grows further apart from the reality of microprocessors. For example, SPARC and Intel x86 multiprocessors implement the Total Store Order (TSO) memory model that deliberately buffers write accesses between each concurrent machine and the system's shared memory. These buffered accesses are later flushed into the shared memory, as depicted (for one machine) in Figure 1.2.

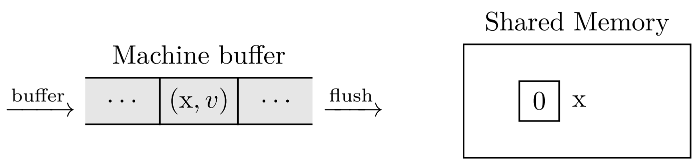


Figure 1.2: Intuitive view of TSO memory for a machine. An assignment $\mathrm{x} := v$ is enqueued as a pair (x, v) in the buffer of the machine that executes it and is later flushed into the shared memory. Variables used in conditionals take their value from the same machine's most recently buffered variable-and-value pair (for the same variable) or from memory if no such pair exists.

On one hand, de-atomizing write accesses under TSO speeds up overall concurrent execution. On the other hand, concurrent behavior under TSO is more difficult to grasp and to reason about. For example, since executions under TSO of concurrent machines are generally a strict superset of their executions under SC, critical-section-exclusion does not always hold for the Dekker algorithm on a TSO-governed microprocessor. To see this, consider the following two-step TSO execution of the machines in Figure 1.1 starting from their top yield nodes with $turn = A$ and $flag_A = flag_B = true$:

(1) machine A places $(flag_A, true)$ in its local buffer and proceeds to its critical section node;

(2) machine B places $(flag_B, true)$ in its local buffer and *since machine A's changes have not been flushed to the shared memory* it can also move to its critical section node.

Hence, we can conclude that — when executed under TSO — the Dekker algorithm is not critical-section-exclusive. In fact, for the Dekker algorithm, critical-section-exclusion does not hold regardless of the execution's initial concurrent state or assumptions on $flag_A$, $flag_B$ and $turn$. This leads to the following architecture-agnostic question:

How is critical-section-exclusion checked for concurrent machines?

To address such questions, automated program verification both has been and continues to be a promising solution. We use model checking [CGP99; BK08], a well known approach to verification, as the starting point to our study. Roughly speaking, model checking asks whether a finite-state system model M satisfies some (given) temporal logic correctness specification φ. In its classical sense, model checking comes with a serious limitation, namely, it targets finite-state models. By contrast, many of the systems that are of interest today are inherently infinite-state. This, in particular, holds for most programs executed under TSO. To be precise, if one were to verify programs executed on specific SPARC or Intel x86 multiprocessors then the chip's exact buffer size would be of consequence and the program's execution model would be finite. However, since the TSO buffer paradigm is the same for different chip variants, it is customary to assume buffers unbounded. This latter assumption produces the blow-up from a finite model for the program state-space under SC to an infinite model for the program state-space under TSO: all that is needed is for some memory writes to be performed within a loop. Nevertheless, this also makes possible using and advancing recent results in the analysis of infinite-state systems.

To conclude, the earlier question gives way to the more specific question that we target in our study:

How does one verify safety of programs under TSO?

We next outline the scientific contribution of the thesis. We then briefly survey related work. Specific connections to closely-related work are highlighted, in situ, throughout the thesis. To sum up the introduction, we end it by delineating the thesis structure.

1.1 Contribution

In this thesis we develop algorithms that verify TSO-relaxed programs. By programs we now mean collections of machines à la Dijkstra. In the manuscript the machines are substituted by, for most means and purposes synonymous, collections of processes or of threads. As already mentioned, the core feature of the TSO memory model is that write accesses are buffered between a program's concurrent machines and the system's shared memory. While the TSO de-atomizing of memory writes speeds up program execution, it also makes program behavior more difficult to understand and analyse. Take for instance the reachability problem for concurrent programs. It is known that for programs under SC the reachability problem is *only* PSpace-complete [Koz77]. Reachability under TSO, on the other hand, was recently proved to be non-primitive-recursive-complete decidable [Ati+10]. The main verification techniques that we develop are actually approximating heuristics targeting this difficult yet decidable problem.

Figure 1.3 depicts the idea behind using approximations to check system correctness under TSO. Roughly speaking, program correctness can often be encoded as a TSO unreachability problem using the specification of the system's bad behaviors. We have already seen a bad behaviors example: the bad behaviors for the Dekker algorithm are those system runs that witness non-exclusive critical section execution.

To address *under-approximating* TSO reachability, we propose the lazy TSO reachability algorithm from [Bou+15]. Intuitively, write buffering is introduced — guided by oracle queries — *only where needed to refute the specification* instead of everywhere in the program. Concretely, the oracle returns sequences of operations in one of the program's machines and satisfies two natural requirements:

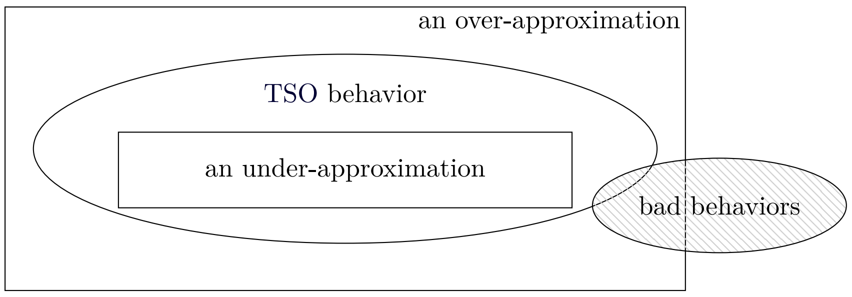


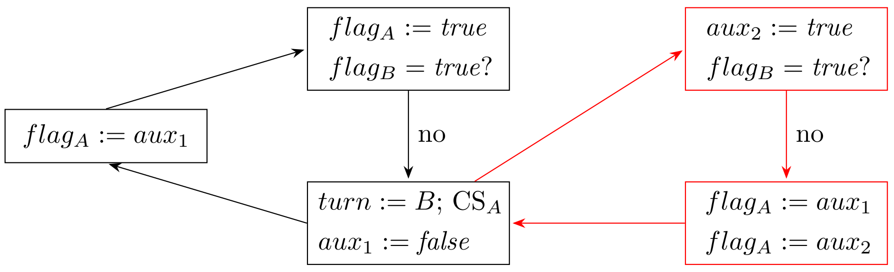
Figure 1.3: Approximating TSO behavior for analyzing system correctness. The system is *incorrect* if an under-approximation contains a bad behavior. The system is *correct* if an over-approximation contains no bad behaviors. In the picture, it cannot be concluded that the system is incorrect with respect to the given bad behaviors using either of the two depicted approximations.

(1) if the oracle returns the empty sequence then the program's SC- and TSO-reachable states are the same;

(2) otherwise, the oracle returns an operation sequence that starts with a write access (as in, e.g., shared variable assignments) and that ends with a read access (as in, e.g., conditional checks).

If the oracle outputs the empty sequence then — according to (1) — the easier SC reachability task produces the same answer as that of checking TSO reachability. If, on the other hand, the oracle outputs some non-empty sequence σ then the TSO delays that σ may produce by buffering its write accesses can be encoded using finitely many auxiliary (and machine local) variables. The under-approximating heuristics can then affirmatively answer TSO reachability by checking SC reachability in the system that includes the encoded delays (the program refined by σ) or it repeats the refinement.

Consider the earlier (Figure 1.1) Dekker algorithm. As discussed, we know that critical-section-exclusion fails under TSO. Assume that the oracle returns the sequence $\sigma =$ "$flag_A := \mathit{false}$";"$flag_A := \mathit{true}$";"$flag_B = \mathit{true}?$" of operations. The bottom nodes in A (i.e., A without yield contention) are modified to additionally encode the TSO delays of $flag_A := \mathit{false}$ and $flag_A := \mathit{true}$ past $flag_B = \mathit{true}?$ using auxiliary variables aux_1 and aux_2:



The added (red colored) nodes have the effect of delaying $flag_A :=$ *false* and $flag_A :=$ *true* as TSO buffers would. Using this modified variant of the Dekker algorithm one can check that a program state where both critical sections are simultaneously accessed is reachable under SC. E.g., A could access its critical section, delay its writes to $flag_A$ using the added behavior until (inclusively) failing the $flag_B =$ *true*? check, B would then be able to go to its critical section and, finally, A would assign the auxiliary variable values (corresponding to flushing TSO buffers) and also access its critical section. This can be seen as an alternative proof to the Dekker algorithm's lack of critical-section-exclusion under TSO.

To address the complementary problem of proving a system correct with respect to TSO unreachability, we propose a few abstractions of TSO buffers as well as a refinement algorithm that exploits them. More precisely, inspired by abstractions from [KVY11], we develop methods for over-approximating a program's TSO behaviors to soundly establish correctness. Furthermore, for over-approximations using multiset-abstracted buffers with per-variable last-added-value information — as depicted in Figure 1.4 — we find that reachability is decidable.

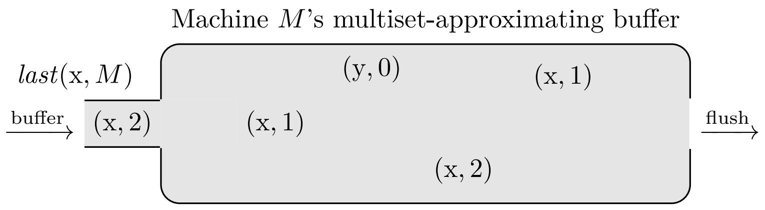


Figure 1.4: Intuitive view of a multiset-approximating buffer. Buffering a new pair $(\mathrm{x}, 3)$ would replace $(\mathrm{x}, 2)$ as variable x's last added value $last(\mathrm{x}, M)$ by pushing it into the depicted "bag". Flushing the pair $(\mathrm{x}, 2)$ from x's last added value position is possible iff no other $(\mathrm{x}, *)$ pair — for the same variable — is in the multiset buffer.

Finally, we show that Partial Order Reduction (POR) approaches, as introduced to address the model checking state-explosion, can be generalized to account for TSO memory. Concretely, we show that both the persistent set approach [God96] and the partial-order semantics of runs [Maz86] benefit from the independence of operations executed by different machines.

Intuitively, (under TSO) atomic components of conditional checks and of variable assignments consist of

- a machine-local operation whose precise interleaving with operations of the other machines is inconsequential to program correctness, and
- a non-local operation whose interleaving with non-local operations of the other machines might alter safe/unsafe program behavior.

Using this insight and the trace-based POR perspective [Maz86; SS88] we survey and adapt several reduction techniques including dynamic and cartesian partial order reduction.

1.2 Related Work

There are various surveys that discuss memory consistency models, viz. Adve and Gharachorloo [AG96], Higham et al. [HKV97], Luchangco [Luc01], Steinke and Nutt [SN04], Arvind and Maessen [AM06], Alglave [Alg10], as well as Alglave et al. [AMT14]. Owing to its longevity and simplicity, the TSO memory model received a lot of attention. Its origins can be tracked to the Scalable Parallel Architecture (SPARC) [WG94] developed at Sun Microsystems starting in the late 1980s. SPARC processors were the first widely-deployed multiprocessors to allow TSO-relaxed program executions and were, more recently, followed by Intel x86 processors [Int06]. Out of the available TSO models — see, e.g., [HKV97; BM08; BP09] — we chose to use the "Intel x86 programmer's memory model" version introduced by Owens et al. [OSS09a; Sew+10].

Coincidentally, there is continuous interest in the research community for advancing verification to checking real — typically C [KR88] — programs. This is a tremendous open-ended task already for concurrent programs in an SC environment. In the context of TSO, the task becomes even more complicated and, sometimes, verification is still intertwined with validation: see, e.g., recent work on linearizability specifications [Bur+12; GMY12; DSD14]. Fortunately, for other correctness criteria like (non-)reachability [Ati+10], robustness [BMM11], persistence [AAN15], or data-race-freedom [Adv+91] specification validation is relatively seamless.

Complexity-wise, Alur et al. [AMP96] showed it is undecidable whether a finite-state memory model implements an SC specification. Soon afterward, Atig et al. [Ati+10] proved that checking reachability under TSO is decidable — non-primitive-recursive-complete to be specific — and also generalized the result to slightly more relaxed memory models [Ati+12]. By contrast, for robustness the situation is more optimistic: both in the case of TSO as well as for the other memory models considered so far, robustness is PSPACE-complete [BMM11; Cal+13; DM14; Der15]. Furthermore, in the context of testing, checking that a given individual computation satisfies a certain memory consistency model was showed to be, typically, between P and NP-complete [GK97; Fur+14].

Most of the correctness criteria mentioned as well as some of the above complexity results are accompanied by verification algorithms. We continue by pointing out closely-related results for algorithmic development towards relaxed-memory program verification.

Hangal et al. [Han+04], Roy et al. [Roy+06], and Baswana et al. [BMP08] implemented algorithms to check that a computation satisfies constraints of a specific memory model while Yang et al. [Yan+04] gave a non-operational (i.e. axiomatic) yet executable specification framework that reduces memory model conformance to either Prolog or SAT constraint solving.

Park and Dill [PD95] and Huyhn and Roychoudhury [HR06] proposed using explicit state model checking with operational finite state models to prove program safety. This was furthered by Burkhardt et al. [BAM07] who reduced consistency checking to SAT solving and, ultimately, by Burkhardt and Musuvathi [BM08] as well as Burnim et al. [BSS11] who implemented monitoring algorithms.

Several tools target state reachability under relaxed memory models. Memorax [Abd+12b] implements a sound and complete decision procedure that combines an automata-based abstraction of the set of feasible program computations with backward reachability analysis. The tool also implements a counterexample-guided fence insertion algorithm that computes fence sets forcing a program's adherence to a given safety specification. A later version of the tool [Abd+12a] uses predicate abstraction to allow for infinite-state program analysis. Remmex [LW11; LW13] implements acceleration to perform forward reachability analysis by exactly representing store buffer contents as finite automata. The bounded model checker CBMC [CKL04] encodes memory model constraints as SAT formulas [AKT13]. Due to its under-approximate nature, CBMC is sound but not complete. Alternative approaches for TSO reachability bound the number of allowed context switches [ABP11] or the size of store buffers [Alg+13] as well as use a scheduler to explore the reduced state-space of bounded programs [Abd+15].

Algorithms have additionally been implemented to check triangular-race freedom [Owe10] and robustness [BDM13], as well as persistence [AAN15] and state-based robustness [Liu+12]. All these approaches yield methods for automatic inference of memory fences, as do the robustness-approximating tools in [BM08; AM11; BSS11; Alg+14]. Other methods for fence inference include those in [KVY10; KVY11] and may cause unnecessary over-fencing. Finally, existing compiler optimization approaches either add memory fences to enforce sequential consistency [Sur+05] or remove redundant memory fences to improve performance [VZ11].

1.3 Thesis Structure

In Chapter 2 we survey related and necessary preliminaries. We start by summarizing a handful of model checking ideas through the introduction of Linear Temporal Logic and Partial Order Reduction. We then explain how the SC and TSO memory models determine different program semantics and we highlight the reachability and robustness correctness specifications. This sets the stage for our theoretical contributions in Chapters 3 and 4.

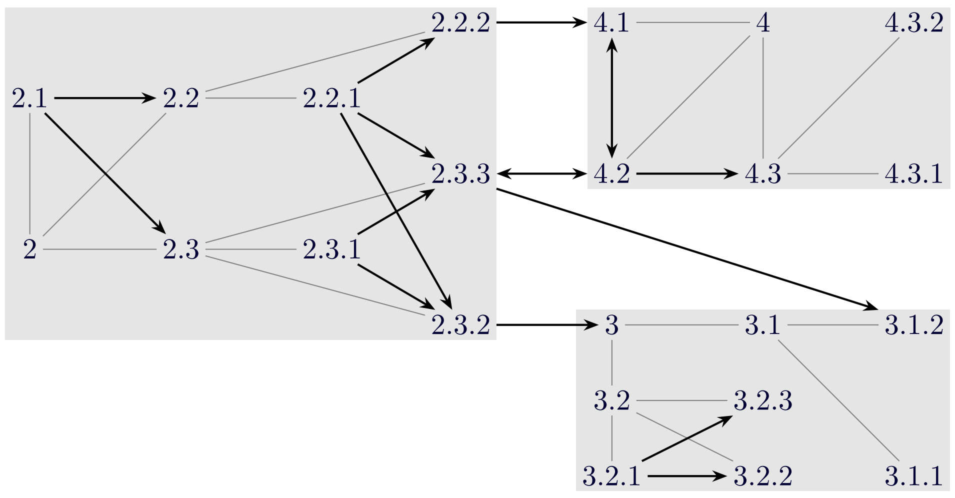
In Chapter 3 we introduce under- and over-approximating heuristics for checking reachability of TSO-relaxed programs.

First, in Section 3.1, we describe how to check TSO reachability lazily: using queries to an oracle, store buffering is introduced *only where needed* *to refute the specification.* On the more technical side, in Section 3.1.1 we prove that lazy TSO reachability yields a semi-decision procedure, i.e., an algorithm that always returns correct answers and that is guaranteed to terminate whenever some goal state is TSO-reachable. Finally, to show that efficient oracles exist, in Section 3.1.2 we explain how the robustness correctness specification can be used to implement a robustness-based oracle.

For over-approximating analysis, in Section 3.2.1 we first describe how buffers can be abstracted by sets. Subsequently, we generalize the basic abstraction to two other set-based abstractions and come up with an abstraction refinement algorithm for checking safety that combines them. In the more technical side of this contribution we prove that reachability is decidable for a multiset buffer abstraction with per-address last-added-value information. To be precise, we prove decidability by showing that the multiset-abstract semantics is a well-structured transition system with effectively computable minimal predecessors and a decidable well-quasi order. For clarity the details of the proof are deferred to Appendix B.

In Chapter 4, to speed up the search both in the concrete as well as in the approximating semantics we revise state-space reduction techniques for TSO-relaxed programs. We first recall, in Section 4.1, the classical persistent set approach [God96] in the context of TSO-relaxed programs. Afterward, in Section 4.2, we use Shasha-and-Snir-like computation traces [SS88] to describe an equivalent understanding of POR under TSO while focusing on the amount of achievable reduction. Finally, through the introduced viewpoint of computation traces, in Section 4.3 we recall two well known exploration techniques for partial order reduction.

Chapter 5 is dedicated to experimental evaluation. Finally, in Chapter 6 we summarize the thesis and discuss some ideas for future work. The picture below depicts dependencies (directed edges) between Chapters 2, 3 and 4.



Chapter 2

Program Verification

Contents

We are interested in verifying programs that implement reactive systems. Known examples include safety-critical embedded systems and operating systems running on servers, systems that are typically non-terminating and interact continuously with their environment. Therefore, their appropriate modeling and meaningful analysis is, more often than not, required.

The importance of modeling is brought forward already in Section 2.1 by introducing a typical imperative language for concurrent programs. We highlight complications that may arise for this idealized program language due to different assumptions concerning the atomicity of instructions. Based on this idealized understanding of programs we then present a few core ideas behind model checking in Section 2.2. Concretely, we describe the standard Linear Temporal Logic (LTL) safety specifications as well as partial order reduction for LTL without the next operator.

Taking a step in the direction of a more concrete model, in Section 2.3 we explain how checking safety generalizes to the infinite-state TSO program semantics. In this context we describe two safety specifications for programs running under TSO: unreachability and robustness.

$$
\begin{aligned}
\langle \textit{prog} \rangle &::= \texttt{program}\ \langle \textit{name} \rangle\ \texttt{variables}\ \langle \textit{vMap} \rangle\ \langle \textit{procList} \rangle \\
\langle \textit{vMap} \rangle &::= \text{a description of the initial shared variable values} \\
\langle \textit{procList} \rangle &::= \varepsilon \mid \langle \textit{proc} \rangle\ ;\ \langle \textit{procList} \rangle \\
\langle \textit{proc} \rangle &::= \texttt{process}\ \langle \textit{name} \rangle\ \texttt{begin}\ \langle \textit{statements} \rangle\ \texttt{end} \\
\langle \textit{statements} \rangle &::= \varepsilon \mid \langle \textit{statement} \rangle\ ;\ \langle \textit{statements} \rangle \\
\langle \textit{statement} \rangle &::= \langle \textit{label} \rangle : \langle \textit{basic} \rangle \\
\langle \textit{basic} \rangle &::= \texttt{skip} \\
&\mid x := \langle \textit{expr} \rangle \text{ — for some } x \in V \\
&\mid \texttt{if}\ \langle \textit{expr} \rangle\ \texttt{then}\ \langle \textit{statements} \rangle\ \texttt{else}\ \langle \textit{statements} \rangle\ \texttt{fi} \\
&\mid \texttt{while}\ \langle \textit{expr} \rangle\ \texttt{do}\ \langle \textit{statements} \rangle\ \texttt{od} \\
\langle \textit{expr} \rangle &::= x \in V \mid v \in D \\
&\mid f_{\text{un}} \langle \textit{expr} \rangle \text{ for some } f_{\text{un}} \in D \to D \\
&\mid f_{\text{bin}} \langle \textit{expr} \rangle\ \langle \textit{expr} \rangle \text{ for some } f_{\text{bin}} \in D \times D \to D
\end{aligned}
$$

Figure 2.1: Syntax of the IMP programming language. We assume that $\langle \textit{name} \rangle$ and $\langle \textit{label} \rangle$ strings uniquely identify process names and labels. For simplicity we don't go into the details of (arithmetic and logical) functions f_{un} and f_{bin}.

2.1 Concurrent Programs

We are concerned with asynchronous concurrent programs and follow the modeling approaches of [MP95; CGP99]. Therefore, we assume a program consists of concurrent processes described through sequential statements. Statements operate on a program's finite set of shared variables V that range over a finite domain $D \supseteq \{0, 1\}$. The statements of disjoint processes are delimited, through labels, by unique entry and exit points.

Figure 2.1 describes our idealized programming language that we call, henceforth, IMP. IMP programs use standard arithmetic and logical functions as well as the notation "`if` $\langle \textit{expr} \rangle$ `then` $\langle \textit{statements} \rangle$ `fi`" instead of the lengthier "`if` $\langle \textit{expr} \rangle$ `then` $\langle \textit{statements} \rangle$ `else skip; fi`".

To illustrate the IMP language and subsequent modeling formalisms we primarily use mutual exclusion algorithms, colloquially called *mutexes*. The formal structure of a mutex process is the following [AKH03]:

```
while true do
  // non-critical section
  // entry section
  // critical section
  // exit section
od
```

```
    program Simplified variables $flag_0 = flag_1 = 0$
    process $P_0$ begin
1:    while $true$ do
                                                    // non-critical section
2:      $flag_0 := 1$;              // $P_0$ wants critical section access
3:      while ($flag_1 = 1$) do                       // busy wait for turn
4:        skip;
        od;
5:      skip;                                           // critical section
6:      $flag_0 := 0$;                       // retract $P_0$'s contention
      od
    end
    process $P_1$ begin
7:    while $true$ do
                                                    // non-critical section
8:      $flag_1 := 1$;              // $P_1$ wants critical section access
9:      while ($flag_0 = 1$) do                       // busy wait for turn
10:       skip;
        od;
11:     skip;                                           // critical section
12:     $flag_1 := 0$;                       // retract $P_1$'s contention
      od
    end
```

Figure 2.2: IMP implementation of a simplified Dekker algorithm. In their entry (red-colored) sections both processes signal they want to access their critical section by setting their flags to 1. This is undone in their exit (blue-colored) sections.

Intuitively, devising a mutual exclusion algorithm requires designing its entry and exit sections such that both critical-section-exclusion as well as starvation-freedom hold. While critical-section-exclusion asks that at most one process is in its critical section at a time, starvation-freedom requires that if some process is in its entry section then that process will eventually access its critical section.

The processes of the Figures 2.2 and 2.3 IMP programs adhere to the above structure of mutexes. As we will see, both programs satisfy critical-section-exclusion (under standard SC and atomicity assumptions) while only the Figure 2.3 program satisfies starvation-freedom.

To compactly describe the model checking framework we additionally use *process graphs* as a visual representation for IMP processes. Apart from being at least as expressive as IMP processes, process graphs allow a few simplifications that make program models smaller. They are, intuitively, an abbreviated version of program graphs from [MP92; BK08].

Formally, given some description `process` P `begin` $\langle statements \rangle$ `end`,

```
     program Dekker variables $turn = flag_0 = flag_1 = 0$
     process $P_0$ begin
1:     while $true$ do
                                                  // non-critical section
2:         $flag_0 := 1$;                // $P_0$ wants critical section access
3:         while $flag_1 = 1$ do             // check for contention from $P_1$
4:             if $turn \neq 0$ then           // check if $P_1$ is given priority
5:                 $flag_0 := 0$;               // yield since $P_1$ has priority
6:                 while $turn \neq 0$ do            // busy wait for $P_0$'s turn
7:                     skip;
                   od;
8:                 $flag_0 := 1$;                 // re-affirm $P_0$'s contention
               fi;
           od;
9:         skip;                                      // critical section
10:        $turn := 1$;                                  // give turn to $P_1$
11:        $flag_0 := 0$;                         // retract $P_0$'s contention
       od
     end
     process $P_1$ begin
12:    while $true$ do
                                                  // non-critical section
13:        $flag_1 := 1$;                // $P_1$ wants critical section access
14:        while $flag_0 = 1$ do             // check for contention from $P_0$
15:            if $turn \neq 1$ then           // check if $P_0$ is given priority
16:                $flag_1 := 0$;               // yield since $P_0$ has priority
17:                while $turn \neq 1$ do            // busy wait for $P_1$'s turn
18:                    skip;
                   od;
19:                $flag_1 := 1$;                 // re-affirm $P_1$'s contention
               fi;
           od;
20:        skip;                                      // critical section
21:        $turn := 0$;                                  // give turn to $P_0$
22:        $flag_1 := 0$;                         // retract $P_1$'s contention
       od
     end
```

Figure 2.3: IMP implementation of the Dekker algorithm. In the entry sections critical-section-contention is resolved, if necessary, in favor of the process whose turn it is. In the exit sections the turn is flipped and the flags are reset.

$$G(\ell\ :\ \texttt{while}\ \langle expr\rangle\ \texttt{do}\ \langle wlist\rangle\ \texttt{od};\ \langle list\rangle) :=$$

$$\begin{tikzcd}[column sep=huge] \underset{entry(\langle list\rangle)}{\circ} & \underset{\ell\, =\, exit(\langle wlist\rangle)}{\circ} \arrow[l, "\neg\langle expr\rangle"'] \arrow[r, "\langle expr\rangle"] & \underset{entry(\langle wlist\rangle)}{\circ} \arrow[l, bend right, dashed, "{[\![\langle wlist\rangle]\!]}"'] \end{tikzcd}$$

$$G(\ell\ :\ \texttt{if}\ \langle expr\rangle\ \texttt{then}\ \langle ilist\rangle\ \texttt{else}\ \langle elist\rangle\ \texttt{fi};\ \langle list\rangle) :=$$

$$\begin{tikzcd}[column sep=huge] & \overset{entry(\langle ilist\rangle)}{\circ} \arrow[dr, dashed, "{[\![\langle ilist\rangle]\!]}"] & \\ \ell\ \circ \arrow[ur, "\langle expr\rangle"] \arrow[dr, "\neg\langle expr\rangle"'] & & \circ\ entry(\langle list\rangle) \\ & \underset{entry(\langle elist\rangle)}{\circ} \arrow[ur, dashed, "{[\![\langle elist\rangle]\!]}"'] & \end{tikzcd}$$

$$G(\ell\ :\ x := \langle expr\rangle;\ \langle list\rangle) :=\ \begin{tikzcd}[column sep=huge] \ell\ \circ \arrow[r, "x := \langle expr\rangle"] & \circ\ entry(\langle list\rangle) \end{tikzcd}$$

$$G(\ell\ :\ \texttt{skip};\ \langle list\rangle) :=\ \begin{tikzcd}[column sep=huge] \ell\ \circ \arrow[r, "\texttt{skip}"] & \circ\ entry(\langle list\rangle) \end{tikzcd}$$

Figure 2.4: Graph translation rules for `process` P `begin` $\langle statements\rangle$ `end`. For any list of statements $\langle list\rangle$ we assume that $entry(\langle list\rangle)$ returns the entry label for $\langle list\rangle$ and that $exit(\langle list\rangle)$ returns the exit label for $\langle list\rangle$ with the excepction of $exit(\langle statements\rangle)$ and the top level $entry(\varepsilon)$ which return a termination label $\ell_\perp$. If $(\ell, \langle expr\rangle, \ell') \in \hookrightarrow$ and $\langle expr\rangle$ contains no $x \in V$ then either $\langle expr\rangle$ evaluates to *true* in which case ℓ and ℓ' are merged or $\langle expr\rangle$ evaluates to *false* in which case the transition and the resulting unreachable graph section are removed; if $(\ell, \texttt{skip}, \ell') \in \hookrightarrow$ then ℓ and ℓ' are merged as well.

the process graph $G(P) := (Loc, Act, \ell_0, \hookrightarrow)$ is the graph with initial location $\ell_0 \in Loc$ identifying the entry label of $\langle statements\rangle$ and transition relation $\hookrightarrow \subseteq Loc \times Act \times Loc$ obtained as described in Figure 2.4. This particularly implies that the transition labels Act are either assignments $x := \langle expr\rangle$ or conditions[1] $\langle expr\rangle$ that contain at least a variable $x \in V$.

Figure 2.5 depicts the process graphs of the IMP program in Figure 2.2 and Figure 2.6 the process graph $G(P_0)$ of the first Dekker algorithm process.

The correspondence between IMP processes and their process graphs should be transparent. Take the simplified version of the Dekker algorithm as example. Its process P_0 performs the following in a loop: (1) it sets $flag_0$ to 1 thus signaling its intent to advance to the critical section; (2) it waits

[1]Any expression $\langle expr\rangle$ is a condition in the following general terms: if $\langle expr\rangle$ evaluates to either 1 or *true* then the expression is *true*; otherwise the expression is *false*.

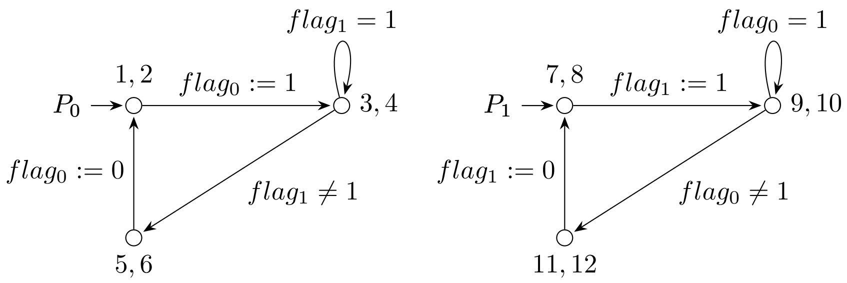


Figure 2.5: Graph processes for the simplified Dekker algorithm.

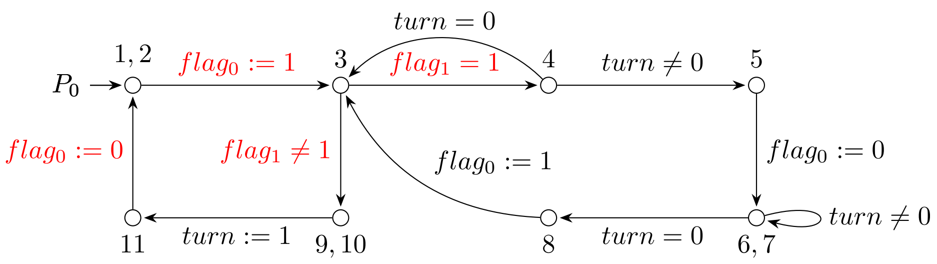


Figure 2.6: Graph process for the first Dekker algorithm process. Colored labels identify transitions preserved in the algorithm's simplified version.

for its turn by checking if the other process also set its flag $flag_1$ to 1; and (3) if $flag_1$ is not set to 1 it advances to its critical section (line label 6).

The common program structure of the Dekker algorithm and of its simplified version should be clearer by looking at the process graphs in Figures 2.5 and 2.6. Indeed, process P_0 for both these algorithms adheres to the looping behavior described above. However, only in the case of the Dekker algorithm can the potential simultaneous busy-wait of P_0 and P_1 be resolved: this is achieved using the extra shared variable *turn*.

So far we used the mutual exclusion problem to introduce two ways to model concurrent systems: IMP programs and process graphs. We rely on transition systems to describe and analyze the systems' semantics/behavior.

A *transition system* is a tuple $(Act, S, \rightarrow, s_0, AP, L)$ where Act is a set of actions, S a set of states, $\rightarrow \subseteq S \times Act \times S$ a transition relation, $s_0 \in S$ an initial state, AP a set of atomic propositions, and $L\colon S \rightarrow 2^{AP}$ a state-labeling function. A transition system is finite if its states S, actions Act, and propositions AP are all finite.

Given some IMP program $\mathcal{P}$, its *transition system semantics* $TS(\mathcal{P})$ describes an interleaved execution model for $\mathcal{P}$. Formally, let $\nu_0 \in V \rightarrow D$ be $\mathcal{P}$'s initial variable valuation and let $G(P_i) := (Loc_i, Act_i, \ell_{0,i}, \hookrightarrow_i)$ be the process graphs for $\mathcal{P}$'s processes P_i (where $i \in [1..N]$ for $N \in \mathbb{N}$). The transition semantics $TS(\mathcal{P}) := (Act, S, \rightarrow, s_0, AP, L)$ is defined by actions $Act := Act_1 \cup \ldots \cup Act_N$ changing states $S := (Loc_1 \times \ldots \times Loc_N) \times (V \rightarrow D)$

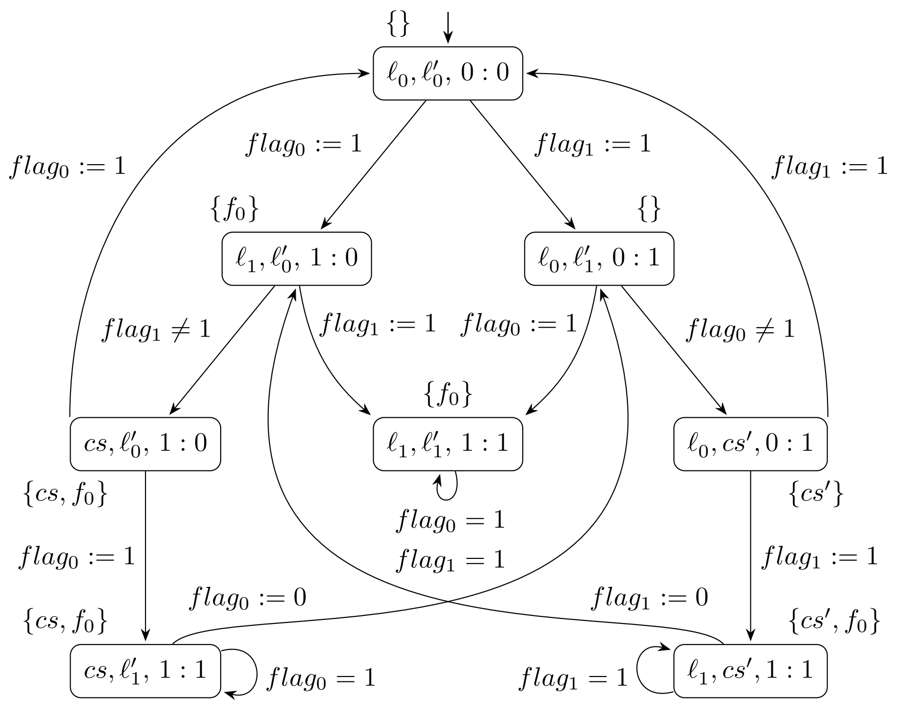


Figure 2.7: Transition system semantics for the simplified Dekker algorithm over the finite atomic propositions set $AP = \{cs, cs', f_0 := (flag_0 = 1)\}$. Valuations have ordered entries for $flag_0$ and $flag_1$ and the labeling function L is indicated by annotated sets. We use $\ell_0 := 1, 2$, $\ell_1 := 3, 4$, and $cs := 5, 6$ for the P_0 locations and $\ell'_0 := 7, 8$, $\ell'_1 := 9, 10$, and $cs' := 11, 12$ for the P_1 ones.

starting from the initial state $s_0 := ((\ell_{0,1}, \ldots, \ell_{0,N}), \nu_0)$.

The transition relation $\rightarrow$ relates $((\ell_1, \ldots, \ell_N), \nu)$ and $((\ell'_1, \ldots, \ell'_N), \nu')$ through $a \in Act$ if $\ell_i \xrightarrow{a} \ell'_i$ for some process graph $G(P_i)$ where $i \in [1..N]$. If $((\ell_1, \ldots, \ell_N), \nu) \xrightarrow{a} ((\ell'_1, \ldots, \ell'_N), \nu')$ and $\ell_i \xrightarrow{a} \ell'_i$ then $\ell_j = \ell'_j$ for all $j \neq i$ and either

- a is a store $x := v$ that updates valuation ν to $\nu' := \nu[x := v]$, or
- a is a condition over V that must hold for variable valuation $\nu = \nu'$.

Atomic propositions $AP \subseteq Cond(V) \cup \bigcup_{i=1}^{N} Loc_i$ consist of conditions $Cond(V)$ over $\mathcal{P}$'s variables and of process labels. The labeling function L provides the AP-labeling for each of $TS(\mathcal{P})$'s states, $L(((\ell_1, \ldots, \ell_N), \nu)) := AP \cap (\{\ell_1, \ldots, \ell_N\} \cup \{\langle expr \rangle \in Cond(V) \mid \langle expr \rangle \text{ evaluates to } true \text{ for } \nu\})$.

Figure 2.7 shows TS(Simplified) for the simplified Dekker algorithm.

As we will show in Section 2.2, a thorough analysis of IMP programs can be performed using the previous transition semantics. However, as classical manuscripts point out [CGP99; BK08], special care should be given to the granularity of program statements.

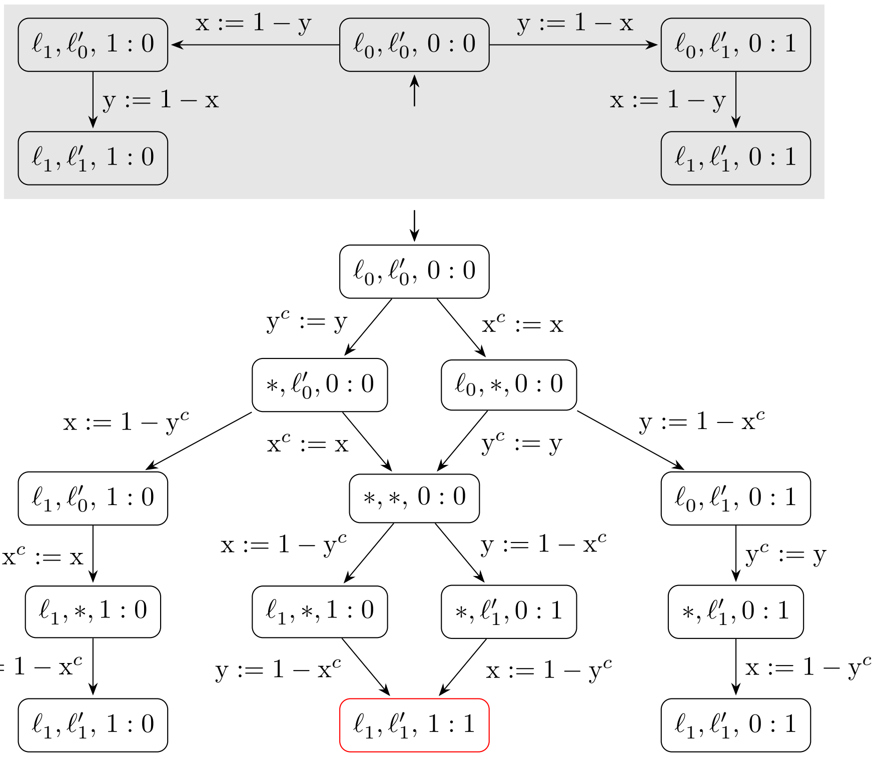


Figure 2.8: Transition system semantics for different atomicity assumptions. The non-standard transition semantics depicted below assumes that assignments are implemented by locally copying the values of right-hand-side variables and then evaluating the assignment expression using these values. For simplicity, the labeling is left out and $*$ symbols mark intermediary locations. In the bottom transition system x^c and y^c are used to copy locally x and y.

Indeed, one typically assumes actions $a \in Act$ are atomic when modeling concurrent systems. This is a natural yet restrictive assumption.

As a hands-on example regarding the importance of atomicity, consider the two processes below that perform symmetric assignments concurrently.

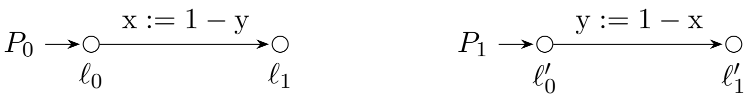


Figure 2.8 shows the transition semantics for this example both when assignments are indivisible (top transition system, on grey background) as well as when they have a simple load-and-store implementation (bottom transition system). The colored state where $x = y = 1$ can only be found in the latter system.

In Section 2.2 we assume all actions to be atomic. Under relaxed memory, (as we will in detail explain in Section 2.3) this is no longer the case.

2.2 Model Checking

In this section we recall a few known results about model checking concurrent systems starting from the transition system semantics of IMP programs. Being such an extensively studied topic, we only present a small subset of existing results that directly relates to our later contributions in the context of programs running under TSO-relaxed memory.

Generally speaking, the model checking problem asks if a finite-state system model M satisfies some temporal logic correctness specification φ. Concretely, a model checking session consists of several phases exploiting the existence of some — typically automated — model checker. The central phases of model checking are enumerated below.

- The *modeling* phase: interpret both the system to be analyzed and the specification to be checked in a way understood by the model checker.

- The *running* phase: check whether the specification holds for the given system model by executing the model checker.

- The *analysis* phase: interpret (as user) the results of the running phase, i.e., handle potential counterexamples found in the running phase by

  (1) using simulation to check the validity of these counterexamples,

  (2) refining the model or the specification to better reflect reality, and

  (3) repeating the entire model-checking procedure if necessary.

Through planning and administering verification by model checking one may add additional phases to the ones described above. For example, the analysis phase may be enriched to account for the running phase stopping due to the model checker running out of memory. This can furthermore be combined with some reasonable model refinement or reduction. A more detailed discussion can be consulted, e.g., in [BK08].

The common understanding of model checking corresponds more closely to its intuitive description. Namely, assuming that the system model and correctness specification are predefined and match the model checker input, the model checking procedure checks whether the specification holds — and it returns a counterexample when this check fails.

So far, in Section 2.1, we described the syntax and transition semantics of concurrent programs. In the following Section 2.2.1 we describe linear time specifications and highlight safety properties. We then present LTL, the classical logic for linear-time property specification. To end our model checking narrative, in Section 2.2.2 we describe POR reduction for transition systems and show that it preserves correctness for the $\mathrm{LTL}_{-\mathcal{X}}$ subset of LTL.

2.2.1 LTL and Safety Specifications

Linear-time properties are an important specification mechanism used to reason about executions of a system. Whenever possible, either a state-based or an action-based approach is followed to analyze concurrent systems. To describe LTL and safety specifications we adopt the commonly-used state-based approach that abstracts actions away by taking only predicates over states into consideration.

In the following, let $TS = (Act, S, \to, s_0, AP, L)$ be a transition system. An execution of TS is a maximal alternating succession of states and actions starting with the initial state. Note that each TS execution corresponds to an execution in the system that TS models.

Formally, an execution of TS is any initial maximal execution fragment. An *execution fragment* is either a *finite* alternating sequence of states and actions ending with a state, i.e.,

$$\rho = s_0 a_1 s_1 \ldots a_n s_n \text{ such that } s_{i-1} \xrightarrow{a_i} s_i \text{ for all } i \in [1..n]$$

or an *infinite* alternating sequence of states and actions, i.e.,

$$\rho = s_0 a_1 s_1 \ldots \text{ such that } s_{i-1} \xrightarrow{a_i} s_i \text{ for all } i \geq 1.$$

An execution fragment is *maximal* if it is infinite or if it ends in a state with no outgoing transitions. An execution fragment is *initial* if it starts with the initial state in TS.

Since we are mainly interested in states visited during executions, instead of some execution $s_0 \xrightarrow{a_1} s_1 \ldots$ we consider sequences $L(s_0)L(s_1)\ldots$ that track atomic propositions valid along such executions. Such sequences of words over the alphabet 2^{AP} are called *traces*.

In the remainder of the subsection we assume that TS has no terminal state, i.e., no state without outgoing transitions. This assumption implies that all traces are infinite words and is not a serious restriction.[2]

We use $trace(\rho)$ for the trace of some execution ρ, $Traces(s)$ for the traces of all execution fragments starting in the state s, and $Traces(TS)$ for the traces of all executions of the transition system TS. Formally, the trace of a finite execution fragment $\rho = s_0 a_1 s_1 \ldots a_n s_n$ is defined as $trace(\rho) := L(s_0)L(s_1)\ldots L(s_n)$ while the trace of an infinite execution fragment $\rho = s_0 a_1 s_1 \ldots$ is defined as $trace(\rho) := L(s_0)L(s_1)\ldots$.

Consider, for example, the transition system $TS_{\text{Simplified}}$ in Figure 2.9. This transition system is the simplification of the transition system semantics $TS(\text{Simplified})$ in Figure 2.7 with transition labels discarded.

[2] A transition system TS with terminal states can be extended by a state s_{deadlock} with $s_{\text{deadlock}} \to s_{\text{deadlock}}$ and such that $s \to s_{\text{deadlock}}$ for each terminal state s in TS.

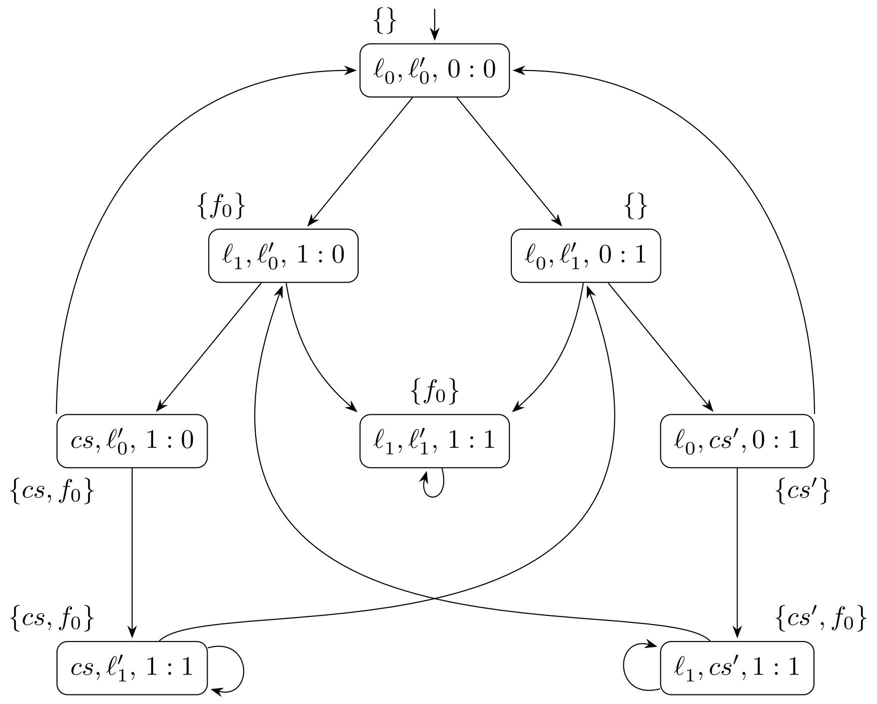


Figure 2.9: Transition system — $TS_{\text{Simplified}}$ — underlying the simplified Dekker algorithm. The atomic proposition f_0 in $AP = \{cs, cs', f_0 := (flag_0 = 1)\}$ signals that process P_0 wants to enter its critical section.

A $TS_{\text{Simplified}}$ execution in which the two processes enter their critical sections in an alternate fashion is the following

$$\begin{aligned}\rho := ((\ell_0, \ell'_0), 0:0) &\to ((\ell_1, \ell'_0), 1:0) \to ((cs, \ell'_0), 1:0) \to ((cs, \ell'_1), 1:1)\\ &\to ((\ell_0, \ell'_1), 0:1) \to ((\ell_0, cs'), 0:1) \to ((\ell_1, cs'), 1:1)\\ &\to ((\ell_1, \ell'_0), 1:0) \dots\end{aligned}$$

The trace of the above execution ρ is the infinite word

$$trace(\rho) = \emptyset\,\{f_0\}\,\{cs, f_0\}\,\{cs, f_0\}\,\emptyset\,\{cs'\}\,\{cs', f_0\}\,\{f_0\}\dots$$

Linear-time properties specify, intuitively, desired behavior of the system under consideration. Formally, a *linear-time property (LT property)* over the set of atomic propositions AP is a subset of $\left(2^{AP}\right)^{\omega}$, the set of all ω-words over 2^{AP}.[3] Let Φ be an LT property and assume TS is a transition system without terminal states over the same set of atomic propositions. We say that *TS satisfies* Φ, denoted by $TS \vDash \Phi$, iff $Traces(TS) \subseteq \Phi$. Similarly, some state *s satisfies* Φ, denoted by $s \vDash \Phi$, iff $Traces(s) \subseteq \Phi$.

[3] $\left(2^{AP}\right)^{\omega}$ denotes all words resulting from infinite concatenations of finite 2^{AP} words.

Consider once more the simplified Dekker algorithm. The critical-section-exclusion condition can be described by the following LT property:

$$\Phi_{\text{mutex}} := \{A_0 A_1 \ldots \in (2^{AP})^{\omega} \mid \{cs, cs'\} \not\subseteq A_i \text{ for all } i \geq 0\}.$$

By a quick analysis one finds that $\{cs, cs'\} \not\subseteq L(s)$ for any $TS_{\text{Simplified}}$ state s. Hence, $TS_{\text{Simplified}} \models \Phi_{\text{mutex}}$, since $\mathit{Traces}(TS_{\text{Simplified}}) \subseteq \Phi_{\text{mutex}}$, and the simplified Dekker algorithm satisfies critical-section-exclusion.

The starvation-freedom condition for process P_0 in the simplified Dekker algorithm can be described by the following LT property:

$$\begin{aligned}\Phi_{\neg\,\text{starving}} := \{A_0 A_1 \ldots \in (2^{AP})^{\omega} \mid\ & \text{if } f_0 \in A_i \text{ and } cs \notin A_i \text{ for some } i \geq 0\\ & \text{then } cs \in A_j \text{ for some } j > i\}.\end{aligned}$$

Since, e.g., $\emptyset\{f_0\}^{\omega}$ belongs to $\mathit{Traces}(TS_{\text{Simplified}})$ but not to $\Phi_{\neg\,\text{starving}}$ we find that $TS_{\text{Simplified}} \not\models \Phi_{\neg\,\text{starving}}$. This means that the simplified Dekker algorithm does not satisfy starvation-freedom.

Safety Specifications Safety intuitively means that "along all traces of the transition system, nothing bad happens ". Many LT properties encode safety and invariants are one of the most well-known type of safety properties — e.g., critical-section-exclusion is an invariant.

Let $\varphi ::= \mathit{true} \mid a \in AP \mid \neg\varphi \mid \varphi \wedge \varphi$ define *propositional logic (PL)* over some set of atomic propositions AP. An LT property Φ_{inv} over AP is an *invariant* if there is a PL formula φ over AP such that

$$\Phi_{\text{inv}} = \{A_0 A_1 \ldots \in (2^{AP})^{\omega} \mid A_i \models \varphi \text{ for all } i \geq 0\}.$$

The formula φ is typically called the invariant- or state-condition of Φ_{inv}. For example, the mutual exclusion property Φ_{mutex} from earlier is an invariant with invariant-condition $\neg(cs \wedge cs')$.

Let $\models$ define the *satisfaction relation* for PL formulas over AP, i.e., for any set $X \subseteq AP$, $X \models \varphi$ if there is a *satisfying* assignment $\chi\colon AP \to \mathbb{B}$ such that $\chi(a) = \mathit{true}$ iff $a \in X$ and $\chi(a) = \mathit{false}$ iff $a \notin X$. Note that

$$\begin{aligned}TS \models \Phi_{\text{inv}} &\text{ iff } \mathit{trace}(\rho) \in \Phi_{\text{inv}} \text{ for all executions } \rho \text{ of } TS\\ &\text{ iff } L(s) \models \varphi \text{ for all states } s \text{ of any } TS \text{ execution}\\ &\text{ iff } L(s) \models \varphi \text{ for all states reachable in } TS.\end{aligned}$$

The notion of invariant can, hence, be explained as follows: Φ_{inv} is an invariant for the transition system TS if its invariant-condition φ holds for all states reachable from the initial state in TS.

Algorithm 2.1 shows a simple Depth-First Search (DFS) algorithm for invariant checking and is similar in nature to Algorithm 2.3 on page 38.

Algorithm 2.1 Naive DFS invariant checker

```
    Input: Finite transition system $TS$ and invariant-condition $\varphi$
    Output: true if $TS$ satisfies “always $\varphi$” and, otherwise, false
    Global Variable: $visited \subseteq S$, initially $visited = \emptyset$

 1: procedure EXPLORER($TS, s$)
 2:     if $s \notin visited$ then               // check that a new $S$ state is explored
 3:         if $L(s) \not\models \varphi$ then
 4:             return false;
 5:         end if
 6:         add $s$ to $visited$;
 7:         for all $s' \in S$ such that $s \to s'$ do
 8:             if $\neg$ EXPLORER($TS, s'$) then
 9:                 return false;
10:             end if
11:         end for
12:     end if
13:     return true;
14: end procedure
```

To turn Algorithm 2.1 into an invariant model-checker one would have to return, additionally to the Boolean value, the current s state at line 4 and, respectively, the current s' state at line 9. Then, whenever Explorer(TS, s_0) would return *false*, the stack of states leading to this outcome would be a counterexample sequence of TS states.

Even though invariants are an important class of properties, not all safety properties are invariants. For example, in the simplified Dekker algorithm, the LT property asking that “the critical-section cs is entered only after process P_0 signaled its contention through f_0” is a safety property although it is not an invariant.

Intuitively, an LT property Φ is a safety property if every infinite word $w \in \left(2^{AP}\right)^{\omega}$ such that $w \not\models \Phi$ contains a *bad prefix.* A *bad prefix* is some finite prefix $w' \in \left(2^{AP}\right)^{*}$ where “something bad has happened” and, hence, no ω-continuation of the bad prefix w' will fulfill Φ.

Formally, an LT property Φ over AP is a *safety property* if for all words $w \in \left(2^{AP}\right)^{\omega} \setminus \Phi$ there exists a finite prefix $w_{\text{fin-pref}}$ of w such that

$$\Phi \cap \left\{w' \in \left(2^{AP}\right)^{\omega} \mid w_{\text{fin-pref}} \text{ is a prefix of } w'\right\} = \emptyset.$$

Any such finite prefix $w_{\text{fin-pref}}$ of w is called a *bad prefix* for Φ and the set of all bad prefixes for Φ is denoted by $\mathit{BadPref}(\Phi)$.

To see that invariants are safety properties, let Φ_{inv} be an invariant with invariant-condition φ. All finite words $A_0 \dots A_n \in \left(2^{AP}\right)^{+}$ with $A_{i-1} \models \varphi$ for

$i \in [1..n]$ and $A_n \not\models \varphi$ are bad prefixes of minimal length and $\mathit{BadPref}(\Phi) = \{A_0 \dots A_n \in (2^{AP})^+ \mid A_{i-1} \models \varphi \text{ for } i \in [1..n] \text{ and } A_n \not\models \varphi\}.(2^{AP})^*$.[4]

Let $\mathit{Pref}(\mathcal{S})$ define the *prefix closure* of a set $\mathcal{S} \subseteq (2^{AP})^\omega$, i.e.,

$$\mathit{Pref}(\mathcal{S}) := \{w \in (2^{AP})^* \mid w \text{ is the prefix of some word in } \mathcal{S}\}.$$

To conclude, we notice that safety properties are system requirements verifiable using finite prefixes of traces.

Corollary 1 ([BK08])**.** *Let TS be a transition system without terminal states and let* Φ *be a safety property. Then,*

$$\mathit{TS} \models \Phi \textit{ if and only if } \mathit{Pref}(\mathit{Traces}(\mathit{TS})) \cap \mathit{BadPref}(\Phi) = \emptyset.$$

Proof. To prove the left-to-right implication, let $\mathit{TS} \models \Phi$ and assume, to the contrary, that $w' \in \mathit{Pref}(\mathit{Traces}(\mathit{TS})) \cap \mathit{BadPref}(\Phi)$. By definition of $\mathit{Pref}(\mathit{Traces}(\mathit{TS}))$, w' is the prefix of some trace $w \in \mathit{Traces}(\mathit{TS})$ and, by definition of $\mathit{BadPref}(\Phi)$, $w \notin \Phi$. Hence, $\mathit{TS} \not\models \Phi$, which contradicts the initial assumption.

For the reverse direction, let $\mathit{Pref}(\mathit{Traces}(\mathit{TS})) \cap \mathit{BadPref}(\Phi) = \emptyset$ and assume, to the contrary, that $\mathit{TS} \not\models \Phi$. Then $\mathit{trace}(\rho) \notin \Phi$ for some execution ρ of TS. Therefore, $\mathit{trace}(\rho)$ starts with a bad prefix w' for Φ. But then $w' \in \mathit{Pref}(\mathit{Traces}(\mathit{TS})) \cap \mathit{BadPref}(\Phi)$ contradicts our previous assumption. □

Additional to safety, liveness completes the picture behind LT properties. Intuitively, a *liveness property* over AP is an LT property Φ such that any $(2^{AP})^*$ word can be extended to some $(2^{AP})^\omega$ word that satisfies Φ. One can, in fact, show that every LT property is equivalent to the intersection of a safety and a liveness property. Since our contributions in Chapters 3 and 4 target safety specifications, we do not further detail liveness specifications.

Linear Temporal Logic Alternatively to the set-description used so far, LT properties can be more concisely expressed using the LTL formalism.

Given some set of atomic propositions AP, LTL over AP is defined by

$$\varphi ::= \mathit{true} \mid a \in AP \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathcal{X}\,\varphi \mid \varphi\,\mathcal{U}\,\varphi,$$

i.e., LTL is PL enhanced with the step ($\mathcal{X}$) and until ($\mathcal{U}$) operators.

Using the until operator one can then derive the temporal modalities eventually ($\Diamond$) and always ($\Box$). Formally, $\Diamond\varphi := \mathit{true}\,\mathcal{U}\,\varphi$ and $\Box\varphi := \neg\Diamond\neg\varphi$.

To give some examples, the LTL specifications for Φ_{mutex} and $\Phi_{\neg\text{starving}}$ from page 22 are $\Box\neg(cs \wedge cs')$ and $\Box(f_0 \wedge cs \rightarrow \Diamond cs)$. Moreover, the earlier safety property stating that "the critical-section cs is entered only after

[4]We use $\big(\big)^+$ to denote the positive Kleene closure, i.e., $\big(\big)^*$ without ε.

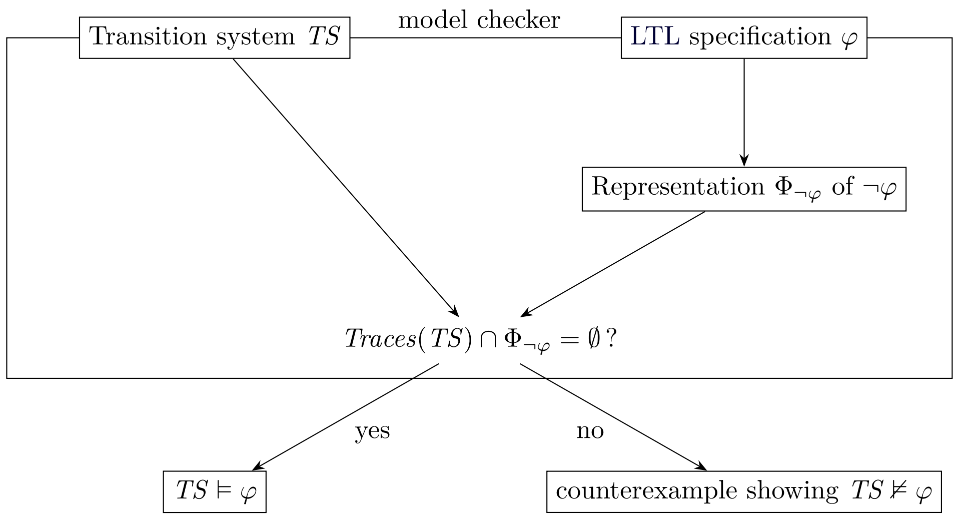

Transition system TS
model checker
LTL specification φ
Representation $\Phi_{\neg\varphi}$ of $\neg\varphi$
$\mathit{Traces}(TS) \cap \Phi_{\neg\varphi} = \emptyset$?
yes
no
$TS \vDash \varphi$
counterexample showing $TS \nvDash \varphi$

Figure 2.10: Typical model checker for $TS \vDash \varphi$. The LTL specification is used to construct a representation $\Phi_{\neg\varphi}$ for its complement. The model checker then checks whether $\mathit{Traces}(TS)$ and $\Phi_{\neg\varphi}$ are disjoint.

process P_0 signaled its contention through f_0" could be specified in LTL as $\square(f_0 \wedge \neg cs \rightarrow \mathcal{X}\, cs)$.

To recapitulate, a concurrent system can be modeled by an IMP program whose semantics is a transition system TS. Undesired system behavior can then be modeled by an LT property $\Phi_{\neg\varphi}$ (constructible from some desirable LTL specification φ). Finally, one can check if the system has the desired behavior by checking whether $TS \vDash \Phi_{\neg\varphi}$.

Figure 2.10 sketches the model checking approach for verifying $TS \vDash \varphi$. We refrain from discussing the model checking procedure in depth since its details are only loosely connected to our contributions.

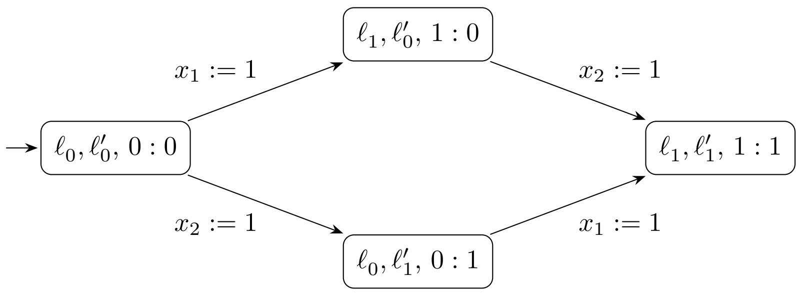
2.2.2 Partial Order Reduction for LTL$_{-\mathcal{X}}$

As is generally the case for concurrent systems, their semantic state-space may grow exponentially wrt the number of concurrent components.

Consider, for example, an IMP program that concurrently increments n variables $x_1, \ldots, x_n$ within n processes

$$\texttt{process}\ P_i\ \texttt{begin}\ x_i := 1\ \texttt{end.}$$

The transition semantics of this program for $n = 2$ is shown in Figure 2.11. In general, for $n \in \mathbb{N}$ processes, the transition semantics contains 2^n states and $n!$ different executions. This does not fare well with any exploration technique used while model checking. However, as long as the intermediary states reached after the execution of either $x_1 := 1, \ldots, x_n := 1$ are irrelevant



$$\begin{tikzcd}
 & \boxed{\ell_1, \ell'_0, 1:0} \arrow[dr, "x_2 := 1"] & \\
\arrow[r] & \boxed{\ell_0, \ell'_0, 0:0} \arrow[ur, "x_1 := 1"] \arrow[dr, "x_2 := 1"'] & & \boxed{\ell_1, \ell'_1, 1:1} \\
 & \boxed{\ell_0, \ell'_1, 0:1} \arrow[ur, "x_1 := 1"'] &
\end{tikzcd}$$

Figure 2.11: Transition semantics of an IMP program that does not scale. The state labeling L is left out and valuations for variables x_1 and x_2 are separated by a colon. We use ℓ_0 and ℓ_1 for the entry and exit locations of process P_1 and, respectively, ℓ'_0 and ℓ'_1 for the entry and exit locations of process P_2.

for the property checked, it suffices to consider an arbitrary interleaving of different process commands.

Briefly put, Partial Order Reduction (POR) aims to reduce the number of execution interleavings that need to be analyzed when model checking. To show that POR preserves correctness for model checking $\text{LTL}_{-\mathcal{X}}$ we introduce the notions of *independent actions* and *stuttering.* Afterward, we sketch the essential constraints behind the POR-underlying *ample sets.* A more detailed presentation can be found in, e.g., [CGP99; BK08].

Independent actions and stutter equivalence As before, we assume that $TS = (Act, S, \rightarrow, s_0, AP, L)$ is a transition system without terminal states. Furthermore, we assume TS is transition-deterministic, use $a(s)$ for the state reached from s by following the a-labeled transition, and use $enabled(s)$ to denote the set of actions $a \in Act$ for which $s \xrightarrow{a} a(s)$ in TS.

Intuitively, the two characteristics of the independence of two actions $a, b \in Act$, both enabled in some state $s \in S$, are (1) *enabledness*: executing a does not disable b and vice versa, and (2) *commutativity*: executing either “b after a” or “a after b” yields the same state.

Formally, actions $a \neq b \in Act$ are *independent* in TS if for any $s \in S$ with $a, b \in enabled(s)$:

$$b \in enabled(a(s)),\ \ a \in enabled(b(s)),\ \text{ and } a(b(s)) = b(a(s)).$$

We say a and b are *dependent* in TS if they are not independent.

A first observation about independent actions is that any $a \in enabled(s)$ can be permuted with the actions of an execution fragment ρ starting in s, provided that action a is independent from the action labels of ρ.

Figure 2.12 depicts the intuition behind Corollary 2 below.

$$\begin{tikzcd}
s_0 \arrow[r, "b_1"] \arrow[d, "a"] & s_1 \arrow[r, "b_2"] & s_2 & \cdots \\
u_0 & \text{can be extended to} & &
\end{tikzcd}
\qquad
\begin{tikzcd}
s_0 \arrow[r, "b_1"] \arrow[d, "a"] & s_1 \arrow[r, "b_2"] \arrow[d, "a"] & s_2 \arrow[d, "a"] & \cdots \\
u_0 \arrow[r, "b_1"'] & u_1 \arrow[r, "b_2"'] & u_2 & \cdots
\end{tikzcd}$$

Figure 2.12: Permuting a with independent actions $b_1, b_2, \ldots$

Corollary 2 ([PW97; BK08])**.** *Let s be a state of some action-deterministic transition system TS and assume that*

$$\rho := s_0 \xrightarrow{b_1} s_1 \xrightarrow{b_2} s_2 \ldots \ \textit{with } s_0 := s$$

is an execution fragment starting in s. Then, for any action $a \in \mathit{enabled}(s)$ that is independent from $b_1, b_2, \ldots$, it holds that $a \in \mathit{enabled}(s_i)$ and

$$\rho' := s_0 \xrightarrow{a} u_0 \xrightarrow{b_1} u_1 \xrightarrow{b_2} u_2 \ldots$$

is an execution fragment starting in s such that $u_i = a(s_i)$.

Proof (sketch). Using the definition of independent actions and execution fragments one can prove by induction over $i \geq 1$ that

- a and b_{i+1} are enabled in state $s_i = b_i(\ldots b_1(s))$,
- b_i is enabled in state $u_{i-1} = b_{i-1}(\ldots b_1(a(s)))$, and
- $a(s_i) = u_i = b_i(u_{i-1})$.

This implies that ρ' is indeed an execution fragment starting in s. $\square$

Partial Order Reduction's connection to LTL specifications relies on independent *stutter* actions. Intuitively, an action $a \in \mathit{Act}$ is a *stutter action*[5] if $L(s) = L(a(s))$ for all transitions $s \xrightarrow{a} a(s)$ in *TS*.

Two execution fragments $\rho := s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \ldots$ and $\rho' := u_0 \xrightarrow{b_1} u_1 \xrightarrow{b_2} u_2 \ldots$ are *stuttering equivalent* — denoted by $\rho \sim_{\mathrm{st}} \rho'$ — if there exist sequences $0 = i_0 < i_1 < \ldots$ and $0 = j_0 < j_1 < \ldots$ such that, for all $k \geq 0$,

$$L(s_{i_k}) = \ldots = L(s_{i_{k+1}-1}) = L(u_{j_k}) = \ldots = L(u_{j_{k+1}-1}).$$

We call a finite sequence of identically labeled execution fragment states a *block*. Intuitively, $\rho \sim_{\mathrm{st}} \rho'$ if $\mathit{trace}(\rho)$ and $\mathit{trace}(\rho')$ can be partitioned in infinitely many blocks such that equivalent blocks are labeled the same.

An LTL formula φ is *invariant under stuttering* iff, for all stuttering equivalent $\rho \sim_{\mathrm{st}} \rho'$, $\mathit{trace}(\rho) \vDash \varphi$ if and only if $\mathit{trace}(\rho') \vDash \varphi$.

Consider now LTL$_{-\mathcal{X}}$, the subset of LTL that excludes the next step operator. Lamport, in [Lam83], motivates excluding the $\mathcal{X}$ operator when

[5]Stutter actions are sometimes called invisible actions in the literature.

reasoning about programs since "increasing the expressiveness of temporal logic with a next operator would destroy the entire logical foundation for its use in hierarchical methods". Furthermore, as proved through Corollary 3 below, every $\text{LTL}_{-\mathcal{X}}$ formula is invariant under stuttering. For more details concerning stuttering principles, the reader can consult, e.g., [Str04].

Corollary 3 ([PW97]). *Any $\text{LTL}_{-\mathcal{X}}$ property is invariant under stuttering.*

Proof. Let TS be an arbitrary transition system over AP. By structural induction, every $\text{LTL}_{-\mathcal{X}}$ formula (over AP) is invariant under stuttering.

For the **base cases**, $\text{LTL}_{-\mathcal{X}}$ formulas *true* and a are both invariant under stuttering. To see this, let $\rho \sim_{\text{st}} \rho'$. By definition of $\vDash$ for LTL, both $\mathit{trace}(\rho) \vDash \mathit{true}$ and $\mathit{trace}(\rho') \vDash \mathit{true}$. On the other hand, for a, by definition of stutter invariance $L(s_0) = L(u_0)$. Hence, by definition of $\vDash$ for LTL, $\mathit{trace}(\rho) \vDash a$ iff $\mathit{trace}(\rho') \vDash a$.

For the induction **step case**, assume that φ and ψ are $\text{LTL}_{-\mathcal{X}}$ formulas that are invariant under stuttering and, as before, let $\rho \sim_{\text{st}} \rho'$. We make the following case distinction:

$\varphi' = \neg\varphi$ Since $\mathit{trace}(\rho) \vDash \varphi$ iff $\mathit{trace}(\rho') \vDash \varphi$, by definition of $\vDash$, also $\mathit{trace}(\rho) \vDash \varphi'$ iff $\mathit{trace}(\rho') \vDash \varphi'$.

$\varphi' = \varphi \wedge \psi$ Since $\mathit{trace}(\rho) \vDash \varphi$ iff $\mathit{trace}(\rho') \vDash \varphi$ and $\mathit{trace}(\rho) \vDash \psi$ iff $\mathit{trace}(\rho') \vDash \psi$, by definition of $\vDash$, also $\mathit{trace}(\rho) \vDash \varphi'$ iff $\mathit{trace}(\rho') \vDash \varphi'$.

$\varphi' = \varphi\, \mathcal{U}\, \psi$ On the one hand, since $\rho \sim_{\text{st}} \rho'$, if φ fails to hold for some block starting with k then $L(s_{i_k}) = L(u_{j_k})$ and

$$L(s_{i_k})L(s_{i_k+1})\ldots \text{ is stuttering-equivalent to } L(u_{j_k})L(u_{j_k+1})\ldots.$$

Then, since ψ is invariant under stuttering, $L(s_{i_k})L(s_{i_k+1})\ldots \vDash \psi$ iff $L(u_{j_k})L(u_{j_k+1})\ldots \vDash \psi$. Hence, $\mathit{trace}(\rho) \vDash \varphi'$ iff $\mathit{trace}(\rho') \vDash \varphi'$. On the other hand, if φ always holds for $\mathit{trace}(\rho)$ and $\mathit{trace}(\rho')$ then $\mathit{trace}(\rho) \nvDash \varphi'$ and $\mathit{trace}(\rho') \nvDash \varphi'$.

By structural induction we conclude that every $\text{LTL}_{-\mathcal{X}}$ formula is invariant under stuttering. □

One could, therefore, use the *stuttering-reduced* transition system TS' instead of the original system TS for model checking $\text{LTL}_{-\mathcal{X}}$ specifications. Intuitively, since a $\sim_{\text{st}}$-class of TS executions is represented by at least one execution in TS', model checking correctness is guaranteed for $\text{LTL}_{-\mathcal{X}}$.

Algorithm 2.2 includes POR in the earlier invariant checker given in Algorithm 2.1. When TS and its sub-system explored by the algorithm are equivalent wrt stuttering, we know that using POR preserves correctness of $\text{LTL}_{-\mathcal{X}}$ and, in particular, of invariant checking.

The main difference between the two invariant checking algorithms is highlighted at Line 7. Namely, while Algorithm 2.1 explores all transitions enabled in a state, the POR-enhanced algorithm only considers a subset $\mathit{ample}(s) \subseteq \mathit{enabled}(s)$ when constructing the reduced system.

Algorithm 2.2 Naive DFS invariant checker with POR

Input: Finite transition system TS and invariant-condition φ
Output: *true* if TS satisfies "always φ" and, otherwise, *false*
Global Variable: $\mathit{visited} \subseteq S$, initially $\mathit{visited} = \emptyset$

```
 1: procedure POR-Explorer($TS, s$)
 2:     if $s \notin visited$ then                  // check that a new $S$ state is explored
 3:         if $L(s) \nvDash \varphi$ then
 4:             return $false$;
 5:         end if
 6:         add $s$ to $visited$;
 7:         for all $a \in ample(s)$ do                    // $ample(s) \subseteq enabled(s)$
 8:             if $\neg$ POR-Explorer($TS, a(s)$) then
 9:                 return $false$;
10:             end if
11:         end for
12:     end if
13:     return $true$;
14: end procedure
```

Ample set constraints What makes POR work correctly in combination with checking LTL$_{-\mathcal{X}}$ specifications is an appropriate choice for the *ample* transitions out of each state.

Assume we want to check $\mathit{TS} \vDash \varphi$ for some LTL$_{-\mathcal{X}}$ specification φ and let TS' be the POR-reduced version of TS. Intuitively,

(1) TS and TS' should be equivalent wrt LTL$_{-\mathcal{X}}$ specifications,
(2) TS' should be smaller (and thus easier to analyze) than TS, and
(3) the effort to compute TS' should be less than to check $\mathit{TS} \vDash \varphi$.

Figure 2.13 outlines the standard constraints on ample sets that ensure point (1) above [CGP99; BK08]. An in-depth analysis of finer constraints can be found, e.g., in [God96].

Intuitively, the ample set constraints ensure that for any TS execution

$$\rho_0 := \underbrace{s_0 \xrightarrow{a_1} \dots \xrightarrow{a_m}}_{\text{prefix in } \mathit{TS}'} s \xrightarrow{b_1} s_1 \xrightarrow{b_2} s_2 \dots$$

there exists some TS' execution ρ' such that $\rho_0 \sim_{\mathrm{st}} \rho'$.

(C0) $\mathit{ample}(s) = \emptyset$ iff $\mathit{enabled}(s) = \emptyset$.
(C1) Let $\rho = s \xrightarrow{a_1} s_1 \ldots \xrightarrow{a_n} s_n \xrightarrow{a} u$ be a finite execution fragment in TS that starts with s. If $a \notin \mathit{ample}(s)$ and some action $b \in \mathit{ample}(s)$ are dependent then $a_k \in \mathit{ample}(s)$ for some $k \in [1..n]$.
(C2) If $\mathit{ample}(s) \neq \mathit{enabled}(s)$ then every $a \in \mathit{ample}(s)$ is a stutter action.
(C3) For all cycles $s_0, s_1, \ldots, s_n$ in TS', if (in TS) $a \in \bigcup_{k\in[1..n]} \mathit{enabled}(s_k)$ then also $a \in \bigcup_{k\in[1..n]} \mathit{ample}(s_k)$.

Figure 2.13: Ample set constraints. (C0) ensures that if s has a successor in TS then it has a successor in TS' too. (C1) implies that any $a \in \mathit{ample}(s)$ and $b \in \mathit{enabled}(s) \setminus \mathit{ample}(s)$ are independent. (C2) guarantees that any $a \in \mathit{ample}(s)$ can be performed earlier in a stutter-equivalent execution. (C3) makes certain that no non-stutter action may be postponed forever.

Technically, ρ' can be constructed as the limit of the sequence $(\rho_i)_{i\geq 0}$ by iteratively using the following transformations:

- if $b = b_{n+1}$ is the earliest $b \in \mathit{ample}(s)$ action occurring in the sequence $(b_i)_{i\geq 1}$ then, by constraints (C0)–(C3), b is a stutter action that is independent of $b_1, \ldots, b_n$. Then

$$\rho_1 := \underbrace{s_0 \xrightarrow{a_1} \ldots \xrightarrow{a_m} s}_{\text{common } \rho_0 \text{ prefix}} \underbrace{\xrightarrow{b} u_0 \xrightarrow{b_1} u_1 \ldots \xrightarrow{b_n} u_n}_{\text{stutter equivalent}} \underbrace{\xrightarrow{b_{n+1}} s_{n+2} \ldots}_{\text{common suffix}}$$

- if $b_i \notin \mathit{ample}(s)$ for all $i \geq 1$ then, by constraints (C0)–(C3), any arbitrary $a \in \mathit{ample}(s)$ is independent of $b_1, b_2, \ldots$ and

$$\rho_1 := \underbrace{s_0 \xrightarrow{a_1} \ldots \xrightarrow{a_m} s}_{\text{common } \rho_0 \text{ prefix}} \underbrace{\xrightarrow{a} u_0 \xrightarrow{b_1} u_1 \xrightarrow{b_2} u_2 \ldots}_{\text{stutter equivalent suffix}}$$

For constructing the entire sequence $(\rho_i)_{i\geq 0}$ one has to iteratively substitute ρ_i for ρ_0 and ρ_{i+1} for ρ_1 in the above description.

To conclude, consider the IMP implementation of the turn-based mutual exclusion protocol shown in Figure 2.14. Similarly to the earlier simplified Dekker algorithm (Figure 2.2 on page 13), this algorithm underpins the Dekker algorithm on page 14.

The following Figure 2.15 depicts the process graphs of Algorithm 2.14 while Figure 2.16 shows its transition semantics TS(Turn). As one can notice, by using POR almost half of the TS(Turn) states are reduced — 5 out of 12 to be precise.

Actually, any TS(Turn) action

$$a \in \{\mathit{turn} = 0, \mathit{turn} \neq 0, \mathit{turn} = 1, \mathit{turn} \neq 1\}$$

is independent from the stutter actions skip_0 and skip_1, the two actions denoting x := x in P_0 and, respectively, y := y in P_1.

```
    program Turn variables $turn = x = y = 0$
    proc $P_0$ begin
1:    while $true$ do
        $x := x$;                               // non-critical section
2:      while ($turn \neq 0$) do                  // busy wait for turn
3:        skip od;
4:      skip;                                       // critical section
5:      $turn := 1$;                              // hand turn to $P_1$
      od
    end
    proc $P_0$ begin
6:    while $true$ do
        $y := y$;                               // non-critical section
7:      while ($turn \neq 1$) do                  // busy wait for turn
8:        skip od;
9:      skip;                                       // critical section
10:     $turn := 0$;                              // hand turn to $P_0$
      od
    end
```

Figure 2.14: IMP implementation of a turn-based mutex. In the entry sections critical-section-contention is always resolved in favor of the process whose turn it is. In the exit sections the turn is flipped.

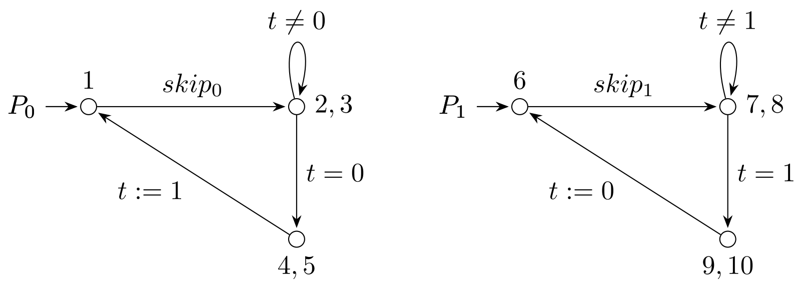


Figure 2.15: Graph processes for the turn-based mutex. We use t as short notation for $turn$, $skip_0$ for $x := x$ from P_0, and $skip_1$ for $y := y$ from P_1.

This permits selecting

$$\{skip_0\} = ample(((\ell_0, \ell_0'), 0)) = ample(((\ell_0, \ell_1'), 1))$$

and

$$\{skip_1\} = ample(((\ell_1, \ell_0'), 0))$$

as ample sets.

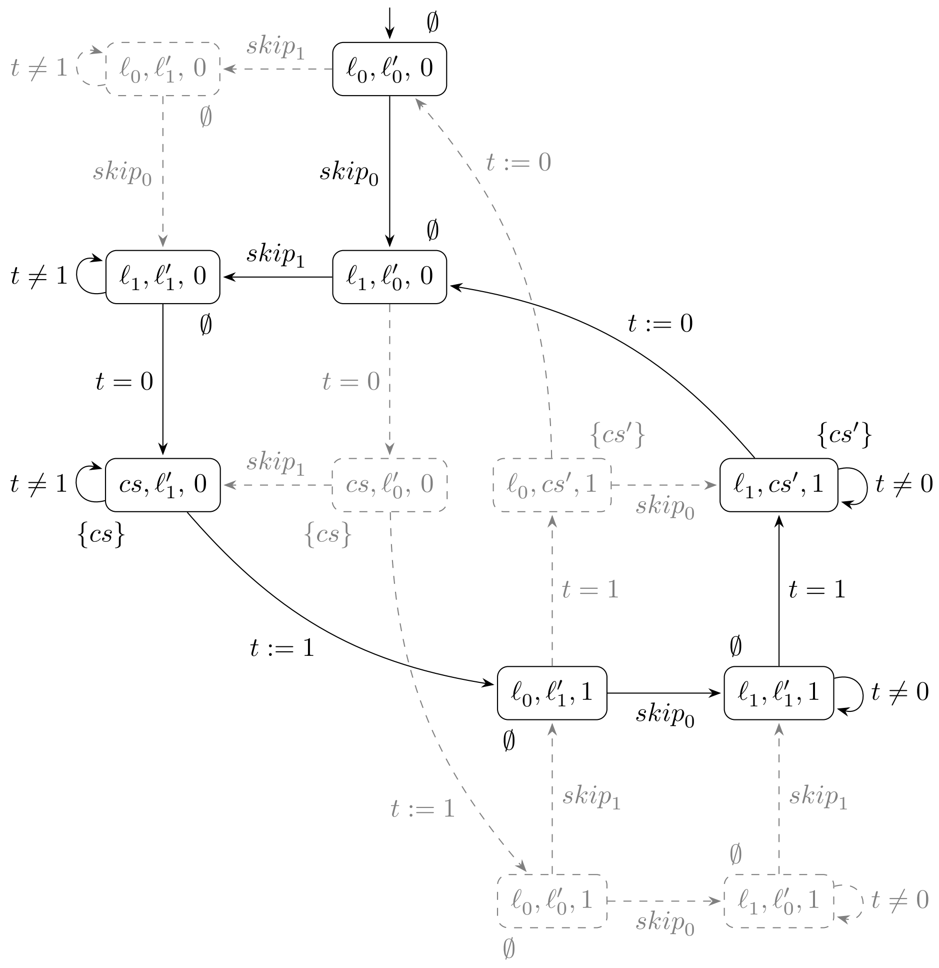


Figure 2.16: Transition semantics for the turn-based mutex assuming atomic propositions set $AP = \{cs, cs'\}$. Valuations show the entry for *turn* and the labeling function L is indicated by annotated sets. Since the assignments x := x and y := y do not change anything we do not depict the values of x and y in the transition system states. We use $\ell_0 := 1$, $\ell_1 := 2, 3$, and $cs := 4, 5$ for P_0 locations and $\ell'_0 := 6$, $\ell'_1 := 7, 8$, and $cs' := 9, 10$ for P_1 locations. POR-reduced transitions and states are indicated by dashed lines.

2.3 Relaxed Memory Models

Model checking concurrent IMP programs as in Section 2.2 relies on two key assumptions: action atomicity and sequential consistency. We already sketched how the atomicity assumption may impact a system's semantics in the example at the end of Section 2.1. The restrictive behavior of sequential consistency is a finer assumption (implicit to the SC memory model) that is best understood by comparison to more relaxed memory models.

Total Store Order (TSO) is one such relaxed memory model that is more relaxed than SC. Additional to its relative simplicity its fame is due to its hardware implementation in both SPARC and Intel-x86 multiprocessors. In the spirit of hardware circuit implementations, in the remainder of the manuscript (Chapters 3—5) we use an idealized Assembly representation of programs. While the principles behind TSO-relaxed programs could be presented as a concretization of IMP programs and transition systems, we choose to use a separate automata-based description for clarity.

A (non-deterministic) *automaton* over a (not necessarily finite) alphabet Σ is a tuple $A = (\Sigma, S, \rightarrow, s_0)$, where S is a set of states, $\rightarrow \subseteq S \times \Sigma \times S$ is a set of transitions, and $s_0 \in S$ is an initial state. The automaton is *finite* if Σ, S and $\rightarrow$ are finite. We write $s \xrightarrow{a} s'$ if $(s, a, s') \in \rightarrow$ and we extend the transition relation to sequences $w \in \Sigma^*$ as expected:

$$s \xrightarrow{w} s' \text{ iff } s \xrightarrow{a_1} \dots \xrightarrow{a_n} s' \text{ for } w = a_1 \dots a_n \in \Sigma^* \text{ and } n \in \mathbb{N}.$$

We say that state $s \in S$ is *reachable* if $s_0 \xrightarrow{w} s$ for some $w \in \Sigma^*$ and that letter *a precedes b in w*, denoted by $a <_w b$, if $w = w_1 \cdot a \cdot w_2 \cdot b \cdot w_3$ for some $w_1, w_2, w_3 \in \Sigma^*$. Moreover, we say the set $\mathcal{L}_F(A) := \{w \in \Sigma^* \mid s_0 \xrightarrow{w} s \in F\}$ is the *language of A with final states $F \subseteq S$.*

As mentioned above, we use automata to describe Assembly programs and their transition semantics in the context of relaxed memory models.

A (concurrent) Assembly program $\mathcal{P}$ is a finite sequence of threads identified by indices t from TID. For brevity we overload a thread's name and its index. Each thread $t := (Com_t, Q_t, I_t, q_{0,t})$ is a finite automaton with transitions I_t that we call *instructions.* Each thread's instructions I_t are labeled by *commands* from the set Com_t which we define in the next paragraph. We assume, wlog, that states of different threads are disjoint. This implies that instructions of different threads are distinct. Furthermore, we use $I := \biguplus_{t \in \mathsf{TID}} I_t$ for the disjoint union of instructions and $Com := \bigcup_{t \in \mathsf{TID}} Com_t$ for all commands. For an instruction $inst := (s, cmd, s') \in I$ we define $cmd(inst) := cmd$, $src(inst) := s$, and $dst(inst) := s'$ to be the command, source state and, respectively, destination state of *inst*.

To define the set of commands, let DOM be a finite domain of values that we also use as addresses ADR. We assume that values $[0..|\mathsf{TID}|]$ are in DOM. For each thread t, let REG_t be a finite set of registers that take their values from DOM. We assume per-thread disjoint sets of registers.

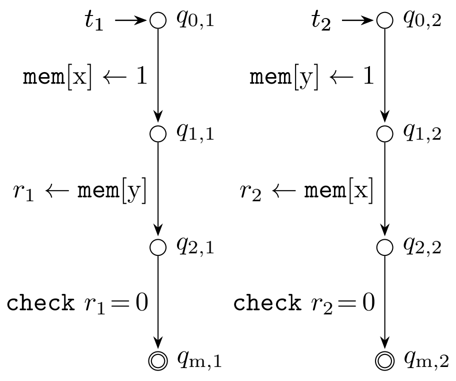


Figure 2.17: ASSEMBLY simplified Dekker algorithm.

The set of expressions of thread t, denoted by EXP_t, is defined over registers from REG_t, constants from DOM, and (unspecified) operators over DOM. If $r \in \mathsf{REG}_t$ and $e, e' \in \mathsf{EXP}_t$, the set of commands Com_t consists of loads from memory $r \leftarrow \mathtt{mem}[e]$, stores to memory $\mathtt{mem}[e] \leftarrow e'$, memory fences $\mathtt{mf}$, assignments $r \leftarrow e$, and conditionals $\mathtt{check}\ e$. We write $\mathsf{REG} := \biguplus_{t \in \mathsf{TID}} \mathsf{REG}_t$ for all registers and $\mathsf{EXP} := \bigcup_{t \in \mathsf{TID}} \mathsf{EXP}_t$ for all expressions.

The ASSEMBLY program from Figure 2.17 is a further simplified version of the Dekker algorithm. It consists of two threads t_1 and t_2 implementing critical-section-exclusion. Initially, the addresses x and y contain 0. The first thread signals its intent to enter the critical section by setting the content of address x to 1. Next, the thread checks whether the second thread wants to enter the critical section. It loads the content of address y and, if it is 0, the first thread enters its critical section. The critical section is indicated by the control state $q_{\mathrm{m},1}$. The second thread behaves symmetrically.

One can best notice the similarity between the simplified ASSEMBLY program in Figure 2.17 and the simplified Dekker algorithm (Algorithm 2.2) by looking at the process graphs in Figure 2.5. Intuitively, the simplified ASSEMBLY version synthesizes the fastest control flow to the critical section in the IMP processes (modulo the use of addresses and registers instead of variables and of ASSEMBLY commands instead of process graph actions).

The more realistic example depicted in Figure 2.18 shows the ASSEMBLY thread representation of the Dekker process graph in Figure 2.6. For clarity, the commands corresponding to the actions highlighted in Figure 2.6 are colored in red in Figure 2.18 as well. Similarly to the ASSEMBLY simplified Dekker algorithm in Figure 2.17, addresses x and y correspond to the two original flags $flag_0$ and $flag_1$ from Algorithm 2.3. Furthermore, address a in Figure 2.18 corresponds to the original $turn$ variable, registers r_1 and r_a are used to implement the original program graph conditions and, like in Figure 2.17, the thread's critical section is indicated by control state $q_{\mathrm{m},1}$.

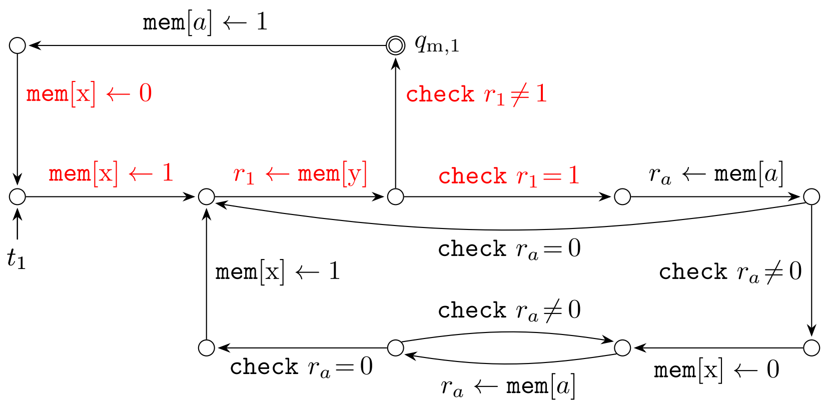


Figure 2.18: ASSEMBLY thread for the first Dekker algorithm process. The highlighted commands correspond to the colored transitions from Figure 2.6.

2.3.1 SC and TSO Semantics

The semantics of a concurrent ASSEMBLY program $\mathcal{P}$ under memory model $\mathrm{M} \in \{\mathrm{TSO}, \mathrm{SC}\}$ follows [OSS09b]. We define it as the *state-space automaton* $\mathrm{X_M}(\mathcal{P}) := (E, S_\mathrm{M}, \Delta_{\mathrm{X,M}}, s_0)$. Each state $s = (\mathsf{pc}, \mathsf{val}, \mathsf{buf}) \in S_\mathrm{M}$ is a tuple where the program counter $\mathsf{pc}\colon \mathsf{TID} \to Q$ holds the current control state of each thread, the valuation $\mathsf{val}\colon \mathsf{REG} \cup \mathsf{ADR} \to \mathsf{DOM}$ holds the values stored in registers and at memory addresses, and the buffer configuration $\mathsf{buf}\colon \mathsf{TID} \to (\mathsf{ADR} \times \mathsf{DOM})^*$ holds a sequence of address-value pairs.

In the *initial state* $s_0 := (\mathsf{pc}_0, \mathsf{val}_0, \mathsf{buf}_0)$, the program counter holds the initial control states, $\mathsf{pc}_0(t) := q_{0,t}$ for all $t \in \mathsf{TID}$, all registers and addresses contain value 0, and all buffers are empty, $\mathsf{buf}_0(t) := \varepsilon$ for all $t \in \mathsf{TID}$.

The TSO transition relation $\Delta_{\mathrm{X,TSO}}$ satisfies the rules in Figure 2.19. A more concrete semantics that makes explicit the partial order of events can be consulted in Appendix C. TSO architectures implement (FIFO) store buffering, which means stores are buffered and their effects become visible only later in the shared memory. An intuitive view of how buffers are used to access shared memory under TSO is depicted in Figure 2.20.

Formally, a load from an address a takes its value from the most recent store to address a that is buffered. If there is no such buffered store, the load takes its value from the shared memory. This is modeled by the two rules (RB) and (RM). Through rule (LS) store operations are enqueued as address-value pairs to the buffer. Rule (WM) non-deterministically dequeues store operations and executes them in the shared memory. Rule (LF) states that a thread can execute a fence only if its buffer is empty. As can be understood from Figure 2.19, events labeling TSO transitions take the form $E \subseteq \mathsf{TID} \times (I \cup \{\mathrm{flush}\}) \times (\mathsf{ADR} \cup \{\bot\})$. Furthermore, this minimal semantics can be easily extended to include locks and atomically executing command sequences — more details can be found in Appendix C.

$$\frac{\mathit{cmd} = r \leftarrow \mathtt{mem}[e_a], \quad a = \widehat{e_a}, \quad \mathsf{buf}(t){\downarrow}(\{a\} \times \mathsf{DOM}) = (a, v) \cdot \beta}{s \xrightarrow{(t, \mathit{inst}, a)} (\mathsf{pc}', \mathsf{val}[r := v], \mathsf{buf})} \text{(RB)}$$

$$\frac{\mathit{cmd} = r \leftarrow \mathtt{mem}[e_a], \quad a = \widehat{e_a}, \quad \mathsf{buf}(t){\downarrow}(\{a\} \times \mathsf{DOM}) = \varepsilon}{s \xrightarrow{(t, \mathit{inst}, a)} (\mathsf{pc}', \mathsf{val}[r := \mathsf{val}(a)], \mathsf{buf})} \text{(RM)}$$

$$\frac{\mathit{cmd} = \mathtt{mem}[e_a] \leftarrow e_v, \quad a = \widehat{e_a}, \quad v = \widehat{e_v},}{s \xrightarrow{(t, \mathit{inst}, a)} (\mathsf{pc}', \mathsf{val}, \mathsf{buf}[t := (a, v) \cdot \mathsf{buf}(t)])} \text{(LS)}$$

$$\frac{\mathsf{buf}(t) = \beta \cdot (a, v)}{s \xrightarrow{(t, \mathrm{flush}, a)} (\mathsf{pc}, \mathsf{val}[a := v], \mathsf{buf}[t := \beta])} \text{(WM)}$$

$$\frac{\mathit{cmd} = \mathtt{mf}, \quad \mathsf{buf}(t) = \varepsilon}{s \xrightarrow{(t, \mathit{inst}, \bot)} (\mathsf{pc}', \mathsf{val}, \mathsf{buf})} \text{(LF)}$$

$$\frac{\mathit{cmd} = r \leftarrow e, \quad v = \widehat{e}}{s \xrightarrow{(t, \mathit{inst}, \bot)} (\mathsf{pc}', \mathsf{val}[r := v], \mathsf{buf})} \text{(LA)}$$

$$\frac{\mathit{cmd} = \mathtt{check}\ e, \quad \widehat{e} \neq 0}{s \xrightarrow{(t, \mathit{inst}, \bot)} (\mathsf{pc}', \mathsf{val}, \mathsf{buf})} \text{(LC)}$$

Figure 2.19: Transition semantics rules for $\mathrm{X}_{\mathrm{TSO}}(\mathcal{P})$ assuming $s = (\mathsf{pc}, \mathsf{val}, \mathsf{buf})$ with $\mathsf{pc}(t) = q$ and $\mathit{inst} = q \xrightarrow{\mathit{cmd}} q'$ in thread t. With the exception of rule (WM), the program counter is always updated by $\mathsf{pc}' = \mathsf{pc}[t := q']$. We use $\widehat{e}$ for the result of atomically evaluating expression e under valuation val and $\mathsf{buf}(t){\downarrow}(\{a\} \times \mathsf{DOM})$ for the projection of $\mathsf{buf}(t)$ to store operations that access address a.

The simpler SC semantics executes stores atomically instead of buffering them [Lam79]. Technically, the set of state-space automaton states stays unchanged and rules (LS) and (WM) of $\Delta_{\mathrm{X,TSO}}$ in $\mathrm{X}_{\mathrm{TSO}}(\mathcal{P})$ by the immediately-flushing rule (LSWM) of $\Delta_{\mathrm{X,SC}}$ in $\mathrm{X}_{\mathrm{SC}}(\mathcal{P})$.

$$\frac{\mathit{cmd} = \mathtt{mem}[e_a] \leftarrow e_v, \quad a = \widehat{e_a}, \quad v = \widehat{e_v}}{s \xrightarrow{(t, \mathit{inst}, a)(t, \mathrm{flush}, a)} (\mathsf{pc}', \mathsf{val}[a := v], \mathsf{buf})} \text{(LSWM)}$$

As mentioned earlier, the state-space automaton $\mathrm{X}_{\mathrm{TSO}}(\mathcal{P})$ that describes the TSO semantics is a (potentially infinite-state) transition system. For the SC semantics, since stores are not buffered we rediscover a concrete version of the model checking framework where neither conditions nor assignments (in the IMP sense) are atomic. In other words, by adopting a load-and-check implementation of IMP conditions and a load-and-store implementation of IMP assignments one would get an interleaving transition system semantics similar to the one in Section 2.1.

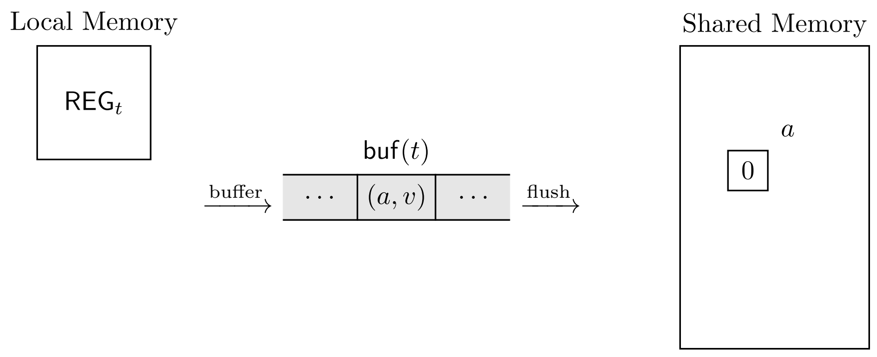


Figure 2.20: A thread's view of TSO memory. Evaluating EXP expressions during the execution determines the address-value pairs (a, v) that stores enqueue in the buffer and then flush into the shared memory. Loads from address a take their value from the most recent buffered (a, v) pair or from memory if no such pair exists.

Since we target safety specifications, we are interested in a program's *finite computations* under $\mathrm{M} \in \{\mathrm{TSO}, \mathrm{SC}\}$. For a program $\mathcal{P}$ they are given by $\mathcal{C}_{\mathrm{M}}(\mathcal{P}) := \mathcal{L}_F(\mathrm{X}_{\mathrm{M}}(\mathcal{P}))$, where F is the set of states with empty buffers. With this choice of final states, we avoid incomplete computations that have pending stores. Note that, since all SC states have empty buffers, $\mathcal{P}$'s SC computations form a subset of its TSO computations: $\mathcal{C}_{\mathrm{SC}}(\mathcal{P}) \subseteq \mathcal{C}_{\mathrm{TSO}}(\mathcal{P})$. We will use $\mathit{Reach}_{\mathrm{M}}(\mathcal{P})$ to denote the set of all states $s \in F$ that are reachable by some computation in $\mathcal{C}_{\mathrm{M}}(\mathcal{P})$.

To give an example, the ASSEMBLY program in Figure 2.18 admits the TSO computation τ_{wit} below whose first thread store is flushed at the end:

$$\tau_{\mathrm{wit}} \;=\; \mathtt{store}_1 \cdot \mathtt{load}_1 \cdot \mathtt{store}_2 \cdot \mathtt{flush}_2 \cdot \mathtt{load}_2 \cdot \mathtt{flush}_1. \qquad (2.1)$$

Consider an event $\mathtt{e} = (t, \mathit{inst}, a)$. Function $\mathit{thread}(\mathtt{e}) := t$ identifies the thread that produced the event. Using $\mathit{inst}(\mathtt{e}) := \mathit{inst}$ we refer to the instruction of the event. For flush events, $\mathit{inst}(\mathtt{e})$ gives the instruction of the matching store event. Lastly, by $\mathit{addr}(\mathtt{e}) := a$ we denote the address that is accessed (if any). In the given example

$$\begin{aligned}
&\mathit{thread}(\mathtt{store}_1) = \mathit{thread}(\mathtt{flush}_1) = \mathit{thread}(\mathtt{load}_1) = t_1,\\
&\mathit{thread}(\mathtt{store}_2) = \mathit{thread}(\mathtt{flush}_2) = \mathit{thread}(\mathtt{load}_2) = t_2,\\
&\mathit{inst}(\mathtt{store}_1) = \mathit{inst}(\mathtt{flush}_1) = q_{0,1} \xrightarrow{\mathtt{mem}[\mathrm{x}] \leftarrow 1} q_{1,1},\\
&\mathit{inst}(\mathtt{load}_1) = q_{1,1} \xrightarrow{r_1 \leftarrow \mathtt{mem}[\mathrm{y}]} q_{2,1},\\
&\mathit{inst}(\mathtt{store}_2) = \mathit{inst}(\mathtt{flush}_2) = q_{0,2} \xrightarrow{\mathtt{mem}[\mathrm{y}] \leftarrow 1} q_{1,2},\\
&\mathit{inst}(\mathtt{load}_2) = q_{1,2} \xrightarrow{r_2 \leftarrow \mathtt{mem}[\mathrm{x}]} q_{2,2},\\
&\mathit{addr}(\mathtt{store}_1) = \mathit{addr}(\mathtt{flush}_1) = \mathit{addr}(\mathtt{load}_2) = \mathrm{x}, \text{ and}\\
&\mathit{addr}(\mathtt{store}_2) = \mathit{addr}(\mathtt{flush}_2) = \mathit{addr}(\mathtt{load}_1) = \mathrm{y}.
\end{aligned}$$

2.3.2 Unreachability as Safety Specification

As described in the context of model checking, certain safety properties — invariants to be precise — can be verified by considering a system's reachable states. Such safety specifications can, hence, be easily encoded as unreachability queries. From a practical point of view, typical sanity checks and assertions available in many programming languages can be encoded as unreachability queries. State reachability can, therefore, be seen as a desirable analysis for ASSEMBLY programs under relaxed memory.

Given a memory model $\mathrm{M} \in \{\mathrm{SC}, \mathrm{TSO}\}$, the M reachability problem expects as input a program P and a set of *goal states* $G \subseteq S_{\mathrm{M}}$. Wlog, we assume that goal states $(\mathsf{pc}, \mathsf{val}, \mathsf{buf})$ specify a program counter pc (through per-thread marked control states) while leaving the memory valuation and buffers unconstrained. Formally, the *M reachability problem* asks if some state in G is reachable in the automaton $\mathrm{X}_{\mathrm{M}}(\mathcal{P})$.

Given: A parallel program $\mathcal{P}$ and goal states G.
Problem: Decide $\mathcal{L}_{F\cap G}(\mathrm{X}_{\mathrm{M}}(\mathcal{P})) \neq \emptyset$.

We will use notation $\mathit{Reach}_{\mathrm{M}}(\mathcal{P}) \cap G$ for the set of goal states that are reachable by some computation in $\mathcal{C}_{\mathrm{M}}(\mathcal{P})$.

A naive explicit-state Depth-First Search (DFS) implementation able to check M reachability is shown in Algorithm 2.3.

Algorithm 2.3 Explicit-state (DFS) M reachability checker

Input: Memory model M, marked program $\mathcal{P}$ and state $s \in S_{\mathrm{M}}$
Output: *true* if some goal state is M-reachable from s in $\mathcal{P}$
false if no goal state is M-reachable from s in $\mathcal{P}$
Global Variable: $\mathit{visited} \subseteq S_{\mathrm{M}}$, initially $\mathit{visited} = \emptyset$

```
1:  procedure EXPLICITDFS($\mathrm{M}, \mathcal{P}, s$)
2:      if $s \notin visited$ then            // check that a new $S_{\mathrm{M}}$ state is explored
3:          if $s \in G$ then
4:              return true;
5:          end if
6:          add $s$ to visited;
7:          for all $\mathsf{e} \in E$ such that $s \xrightarrow{\mathsf{e}} s' \in \Delta_{\mathrm{X,M}}$ do
8:              if EXPLICITDFS($\mathrm{M}, \mathcal{P}, s'$) then
9:                  return true;
10:             end if
11:         end for
12:     end if
13:     return false;
14: end procedure
```

Algorithm 2.3 is a decision procedure for Assembly programs under SC. Intuitively, on one hand, ExplicitDFS(SC, $\mathcal{P}, s_0$) decides positive SC reachability instances since *true* is returned if and only if the depth-first recursion can construct a computation $\alpha \in \mathcal{C}_{\text{SC}}(\mathcal{P})$ such that $s_0 \xrightarrow{\alpha} s$ and $s \in G$. On the other hand, if ExplicitDFS(SC, $\mathcal{P}, s_0$) does not return *true* then the global variable *visited* eventually includes all the finitely-many states S_{SC} of $\text{X}_{\text{SC}}(\mathcal{P})$ and the procedure returns *false*.

Under TSO however, Algorithm 2.3 is not guaranteed to terminate even for positive instances when a goal state is reachable. Indeed, if the depth-first recursion explores a loop in an input Assembly program for which a state's buffer content grows then ExplicitDFS never terminates. A possible solution to this problem is to use a Breadth-First Search (BFS) implementation for M reachability, like the one in Algorithm 2.4.

Algorithm 2.4 Explicit-state (BFS) M reachability checker

```
Input: Memory model M, marked program $\mathcal{P}$ and states $frontier \subseteq S_{\text{M}}$
Output: $true$ if some goal state is M-reachable from $s \in frontier$ in $\mathcal{P}$
        $false$ if no goal state is M-reachable from any $s \in frontier$ in $\mathcal{P}$
Global Variable: $visited \subseteq S_{\text{M}}$, initially $visited = \emptyset$

procedure ExplicitBFS(M, $\mathcal{P}$, $frontier$)
    $frontier' := \emptyset$;            // will contain the next depth-level frontier
    for all $s \in frontier$ do
        if $s \notin visited$ then                  // only check new $S_{\text{M}}$ states
            if $s \in G$ then
                return $true$;
            end if
            add $s$ to $visited$;
            for all $\mathsf{e} \in E$ such that $s \xrightarrow{\mathsf{e}} s' \in \Delta_{\text{X,M}}$ do
                add $s'$ to $frontier'$;
            end for
        end if
    end for
    if $frontier' \neq \emptyset$ then
        return ExplicitBFS(M, $\mathcal{P}$, $frontier'$);
    else
        return $false$;
    end if
end procedure
```

Like the preceding depth-first algorithm, Algorithm 2.4 is a decision procedure under SC. Furthermore, owing to its breadth-first approach, this algorithm is also guaranteed to terminate correctly for positive instances of

TSO reachability. However, neither of the two presented algorithms is a decision procedure for programs where no goal state is TSO-reachable.

It should be clear by now that TSO reachability is a hard problem. To be precise, it was only recently proved that TSO reachability is non-primitive-recursive-complete decidable [Ati+10] — MEMORAX [Abd+13] provides a sound and complete implementation that follows the decidability proof's approach. As further witness to the problem's intractability, approximative heuristics for TSO reachability (and for the afferent fence synthesis problem) abound [KVY10; ABP11; KVY11; LW11; Liu+12; Alg+13; Bou+15].

2.3.3 Robustness as Safety Specification

Robustness [SS88; AM11; BDM13] is a (non-invariant-based) complexity-wise simpler correctness criterion. Two of its aspects make it appealing in comparison to TSO reachability checking:

(1) checking robustness is only PSPACE-complete [BMM11] and

(2) if robustness holds for some program $\mathcal{P}$ then its SC- and TSO-reachable states are the same, i.e., $\mathit{Reach}_{\mathrm{SC}}(\mathcal{P}) = \mathit{Reach}_{\mathrm{TSO}}(\mathcal{P})$.

Moreover, since checking SC reachability is PSPACE-complete [Koz77], (2) actually implies that TSO reachability can be checked using two PSPACE procedures for any program for which robustness holds.

Intuitively, robustness requires that for each TSO computation of an ASSEMBLY program there is an SC computation that has the same data and control dependencies. Delays due to store buffering are still allowed, as long as they do not produce dependencies between instructions that SC computations forbid.

Formally, dependencies between computation events are described in terms of the *happens-before* relation. More precisely, given a computation $\tau \in \mathcal{C}_{\mathrm{TSO}}(\mathcal{P})$, the happens-before relation $\rightarrow_{hb}(\tau)$ is a union of the three relations that we define below: $\rightarrow_{hb}(\tau) := \rightarrow_{po} \cup \leftrightarrow \cup \rightarrow_{cf}$.

The *program order relation* $\rightarrow_{po}$ represents the order in which threads issue their commands: $\rightarrow_{po} := \bigcup_{t \in \mathsf{TID}} \rightarrow_{po}^{t}$. Each $\rightarrow_{po}^{t}$ relation gives the order of non-flush events in thread t: if τ' is the subsequence of all non-flush events of thread t in τ then $\rightarrow_{po}^{t} := <_{\tau'}$.

The *equivalence relation* $\leftrightarrow$ links, in each thread, flush events and their matching store events: $(t, \mathit{inst}, a) \leftrightarrow (t, \mathrm{flush}, a)$.

The *conflict relation* $\rightarrow_{cf}$ orders accesses to the same address. Assume, on the one hand, that $\tau = \tau_1 \cdot \texttt{store} \cdot \tau_2 \cdot \texttt{load} \cdot \tau_3 \cdot \texttt{flush} \cdot \tau_4$ such that $\texttt{store} \leftrightarrow \texttt{flush}$, events `store` and `load` access the same address a and come from thread t, and there is no other store event $\texttt{store}' \in \tau_2$ such that $\mathit{thread}(\texttt{store}') = t$ and $\mathit{addr}(\texttt{store}') = a$. Then the load event `load` is an *early read* of the value buffered by the event `store` and $\texttt{store} \rightarrow_{cf} \texttt{load}$.

$$\begin{tikzcd}
\texttt{store}_1 \arrow[d, leftrightarrow] \arrow[dd, bend right, "po"'] & & \texttt{store}_2 \arrow[d, leftrightarrow] \arrow[dd, bend left, "po"] \\
\texttt{flush}_1 & & \texttt{flush}_2 \\
\texttt{load}_1 \arrow[urr, "cf"'] & & \texttt{load}_2 \arrow[ull, "cf"]
\end{tikzcd}$$

Figure 2.21: The happens-before relation $\rightarrow_{hb}$ (τ_{wit}).

On the other hand, assume $\tau = \tau_1 \cdot \texttt{e} \cdot \tau_2 \cdot \texttt{e}' \cdot \tau_3$ such that $\texttt{e}$ and $\texttt{e}'$ are either load or flush events that access the same address a, neither $\texttt{e}$ nor $\texttt{e}'$ is an early read, and at least one of $\texttt{e}$ or $\texttt{e}'$ is a flush to a. If there is no other flush event $\texttt{flush} \in \tau_2$ with $addr(\texttt{flush}) = a$ then $\texttt{e} \rightarrow_{cf} \texttt{e}'$.

As a first relevant observation, one can notice that any two computations with the same happens-before relation will reach the same state.

Lemma 4. *If $\alpha, \beta \in \mathcal{C}_{TSO}(\mathcal{P})$, $s_0 \xrightarrow{\alpha} s$, and $\rightarrow_{hb} (\alpha) =\rightarrow_{hb} (\beta)$ then $s_0 \xrightarrow{\beta} s$.*

Proof. Assume $s_0 \xrightarrow{\beta} s'$. Since α and β have the same program order $\rightarrow_{po}$, it means s and s' have the same program counter pc. Moreover, since α and β have the same conflict order $\rightarrow_{cf}$, s and s' have the same memory valuation val. Finally, since computations α and β empty the buffers, s and s' have empty buffers. In conclusion, $s = s'$. □

To give an example, Figure 2.21 depicts the happens-before relation of the computation τ_{wit} introduced earlier, on page 37.

The robustness correctness criterion is defined as follows: a program $\mathcal{P}$ is said to be *robust* against TSO iff for each computation $\tau \in \mathcal{C}_{\text{TSO}}(\mathcal{P})$ there exists a computation $\sigma \in \mathcal{C}_{\text{SC}}(\mathcal{P})$ such that $\rightarrow_{hb} (\tau) = \rightarrow_{hb} (\sigma)$.

As already mentioned, a convenient benefit of robustness is that, if a program $\mathcal{P}$ is robust then the same set of states are reachable in $\mathcal{P}$ both under SC as well as under TSO:

Theorem 5 ([Bou+15])**.** *If a program $\mathcal{P}$ is robust against TSO then its SC- and TSO-reachable states are the same: $Reach_{SC}(\mathcal{P}) = Reach_{TSO}(\mathcal{P})$.*

Proof. The $\subseteq$ inclusion holds by $\mathcal{C}_{\text{SC}}(\mathcal{P}) \subseteq \mathcal{C}_{\text{TSO}}(\mathcal{P})$. For the reverse, assume that there is a TSO computation $\tau \in \mathcal{C}_{\text{TSO}}(\mathcal{P})$ such that $s_0 \xrightarrow{\tau} s$. Since $\mathcal{P}$ is robust, there is an SC computation $\sigma \in \mathcal{C}_{\text{SC}}(\mathcal{P})$ such that $\rightarrow_{hb} (\tau) =\rightarrow_{hb} (\sigma)$. Then $\sigma \in \mathcal{C}_{\text{TSO}}(\mathcal{P})$ and, by Lemma 4, $s_0 \xrightarrow{\sigma} s$ so s is SC-reachable. □

We will use Theorem 5 in Section 3.1.2 to justify that robustness can be used to implement an oracle for lazily checking TSO reachability.

Chapter 3

Heuristics for TSO Reachability

Contents

Undeterred by the difficulty of the TSO reachability problem, we propose two verification approaches: lazy TSO reachability for under-approximating and using set-based abstractions for over-approximating.

A sketch of the idea behind using approximations for TSO reachability is depicted in Figure 3.1. Approximations provide a set of advantages over the complete method [Abd+13] for TSO reachability. The common target of approximations is that they provide methods meant to be generally faster. Indeed, both finite- and infinite-state approximations may yield faster on-the-fly reachability checking. This can, e.g., be achieved by (1) looking for bad behaviors within iterative refinement steps, as lazy TSO reachability does, or by (2) approximating the system's semantics with a simpler (and algorithmically easier to encode) semantics, as set-based abstractions do. Furthermore, approximations usually facilitate re-using existing program verification techniques, thus sharing the analysis burden when necessary.

We present the iterative approach to lazy TSO reachability in Section 3.1. Lazy TSO reachability uses queries to an oracle to identify sequences of instructions that lead to states reachable under TSO and not reachable under SC. In Section 3.1.1, the most technical part of the manuscript, we explain

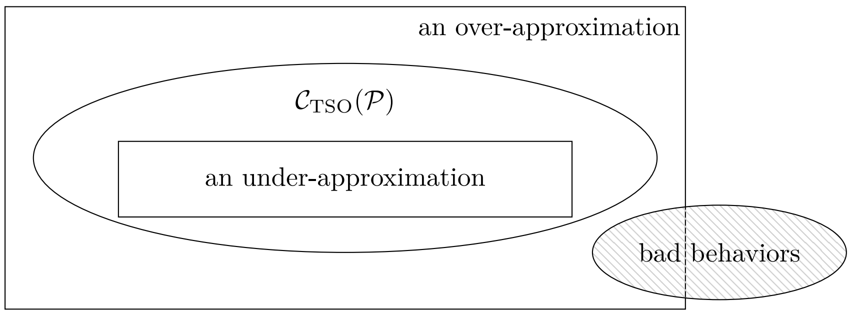


Figure 3.1: Approximations wrt TSO reachability. TSO reachability checks if some TSO computation of $\mathcal{P}$ represents a bad behavior, i.e., ends in a goal state. This could be done either using an under-approximation or an over-approximation. If a bad behavior belongs to an under-approximation then this is a bad behavior of the program, while if no bad behavior belongs to an over-approximation then the program has no bad behaviors. In the picture, it cannot be concluded that the program is safe (or unsafe) wrt TSO reachability using the depicted approximations.

how the lazy TSO reachability algorithm yields a sound and complete semi-decision procedure. Afterward, in Section 3.1.2, we show how robustness — the inspiration for lazy TSO reachability — can be used to implement a robustness-based oracle.

In Section 3.2 we present several abstractions that can be used to prove safety of programs under TSO. First, we outline how to use a set abstraction of buffers to account for TSO relaxation. Subsequently, we generalize this abstraction and come up with an abstraction refinement algorithm for checking safety under TSO. In the more technical side of this contribution we prove that reachability is decidable for the multiset buffer abstraction with per-address last-added-value information: the multiset-abstract semantics is a well-structured transition system with computable minimal predecessors and decidable well-quasi order.

3.1 Lazy TSO Reachability

Instead of solving reachability under TSO directly, the algorithm we propose solves SC reachability and, if no goal state is reachable, tries to lazily introduce store buffering on a certain control path of the program. The algorithm delegates choosing the control path to an *oracle function* O. Given an input program $\mathcal{R}$, the oracle returns a sequence of instructions I^* in that program. Formally, the oracle satisfies the following requirements:

- if $O(\mathcal{R}) = \varepsilon$ then $\mathit{Reach}_{\mathrm{SC}}(\mathcal{R}) = \mathit{Reach}_{\mathrm{TSO}}(\mathcal{R})$;
- otherwise, $O(\mathcal{R}) = \mathit{inst}_1 \mathit{inst}_2 \ldots \mathit{inst}_n$ such that $\mathit{cmd}(\mathit{inst}_1)$ is a store, $\mathit{cmd}(\mathit{inst}_n)$ is a load, and — for all $i \in [1..n-1]$ — $\mathit{cmd}(\mathit{inst}_i) \neq \texttt{mf}$ and $\mathit{dst}(\mathit{inst}_i) = \mathit{src}(\mathit{inst}_{i+1})$.

Intuitively, whenever $O(\mathcal{R})$ returns the empty sequence then the SC- and TSO-reachable states of $\mathcal{R}$ coincide. Otherwise, $O(\mathcal{R})$ returns an instruction sequence in one of $\mathcal{R}$'s threads. This instruction sequence starts with a store, ends with a load, and contains no memory fence in-between.

The lazy TSO reachability checker is outlined in Algorithm 3.1. As input, it takes a program $\mathcal{P}$ and an oracle O. We assume some control states in each thread to be marked to define a set of goal states. The algorithm returns *true* iff the program can reach a goal state under TSO.

Algorithm 3.1 Lazy TSO reachability checker.

```
Input: Marked program $\mathcal{P}$ and oracle $O$
Output: $true$ if some goal state is TSO-reachable in $\mathcal{P}$
        $false$ if no goal state is TSO-reachable in $\mathcal{P}$

 1: $\mathcal{R} := \mathcal{P}$;
 2: while $true$ do
 3:     if $Reach_{SC}(\mathcal{R}) \cap G \neq \emptyset$ then   // check if $G$ states are SC-reachable
 4:         return $true$;
 5:     else
 6:         $\iota := O(\mathcal{R})$;            // ask the oracle where to use store buffering
 7:         if $\iota \neq \varepsilon$ then
 8:             $\mathcal{R} := \mathcal{R} \oplus \iota$;
 9:         else
10:             return $false$;
11:         end if
12:     end if
13: end while
```

Algorithm 3.1 works as follows. First, it creates a copy $\mathcal{R}$ of the program $\mathcal{P}$. Next, it checks if a goal state is SC-reachable in $\mathcal{R}$ (Line 3). If that is the case, the algorithm returns *true*. Otherwise, it asks the oracle O where in the program to introduce store buffering. If $O(\mathcal{R}) \neq \varepsilon$, the algorithm extends $\mathcal{R}$ to emulate store buffering on the path $O(\mathcal{R})$ under SC (Line 8) and it goes back to the beginning of the loop. If $O(\mathcal{R}) = \varepsilon$, by the first property of oracles, $\mathcal{R}$'s reachable states under SC and under TSO are the same. This means the algorithm can safely return *false* (Line 10). Since $\mathcal{R}$ emulates TSO behavior of $\mathcal{P}$, the algorithm solves TSO reachability for $\mathcal{P}$.

Let $\iota := O(\mathcal{R}) = inst_1 inst_2 \dots inst_n$ and let $t := (Com_t, Q_t, I_t, q_{0,t})$ be the thread of the instructions in ι. The modified program $\mathcal{R} \oplus \iota$ replaces thread t by a new thread $t \oplus \iota$. The new thread emulates under SC the TSO semantics of ι.

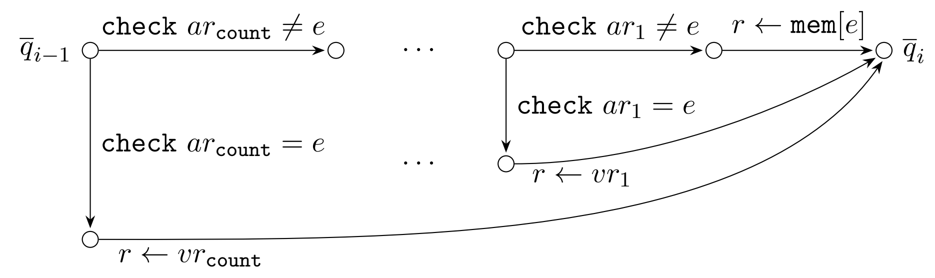
Formally, the *extension of* t *by* ι is $t \oplus \iota := (Com'_t, Q'_t, I'_t, q_{0,t})$. The thread $t \oplus \iota$ is obtained from t by adding sequences of instructions starting from $\overline{q}_0 :=$

$src(inst_1)$. To remember the addresses and values of the buffered stores, we use auxiliary registers $ar_1, \ldots, ar_{\mathtt{max}}$ and $vr_1, \ldots, vr_{\mathtt{max}}$, where $\mathtt{max} \leq n-1$ is the total number of store instructions in ι. The sets $Com'_t \supseteq Com_t$ and $Q'_t \supseteq Q_t$ are extended as necessary.

We define the extension by describing the new transitions that are added to I'_t for each instruction $inst_i$. In our construction, we use a variable `count` to keep track of the number of store instructions already processed. Initially, $Q'_t := Q_t$ and $\mathtt{count} := 0$. Based on the type of instructions, we distinguish the following cases.

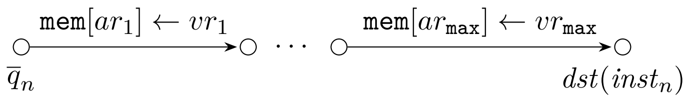
If $cmd(inst_i) = \mathtt{mem}[e] \leftarrow e'$, then we increment `count` by 1 and add instructions that remember the address and the value that are being written in the auxiliary registers $ar_{\mathtt{count}}$ and $vr_{\mathtt{count}}$.

If $cmd(inst_i) = r \leftarrow \mathtt{mem}[e]$, we add instructions to I'_t that perform a load from memory only when a load from the simulated buffer is not possible. More precisely, if $j \in [1, \mathtt{count}]$ is found so that $ar_j = e$ then register r is assigned the value of vr_j. Otherwise, register r receives its value from the address indicated by e.


$\overline{q}_{i-1}$
check $ar_{\mathtt{count}} \neq e$
check $ar_{\mathtt{count}} = e$
$r \leftarrow vr_{\mathtt{count}}$
check $ar_1 \neq e$
$r \leftarrow \mathtt{mem}[e]$
$\overline{q}_i$
check $ar_1 = e$
$r \leftarrow vr_1$

If $cmd(inst_i)$ is an assignment or a conditional, we add the instruction $(\overline{q}_{i-1}, cmd(inst_i), \overline{q}_i)$ to I'_t. By the definition of an oracle, $cmd(inst_i)$ is never a fence command.

The above cases handle all instructions in ι. So far, the extension added new instructions to I'_t that lead through the fresh states $\overline{q}_1, \ldots, \overline{q}_n$. Out of control state $\overline{q}_n$ we then recreate the sequence of stores remembered by the auxiliary registers. Afterwards, we return to the control flow of the original thread t.


$\mathtt{mem}[ar_1] \leftarrow vr_1$
$\mathtt{mem}[ar_{\mathtt{max}}] \leftarrow vr_{\mathtt{max}}$
$\overline{q}_n$
$dst(inst_n)$

Next, we remove $inst_1$ from the program. This prevents the oracle from discovering in the future another instruction sequence that is essentially the same as ι. As we will show, this is key to guaranteeing termination of the algorithm for acyclic programs. However, the removal of $inst_1$ may reduce the set of TSO-reachable states. To overcome this problem, we insert

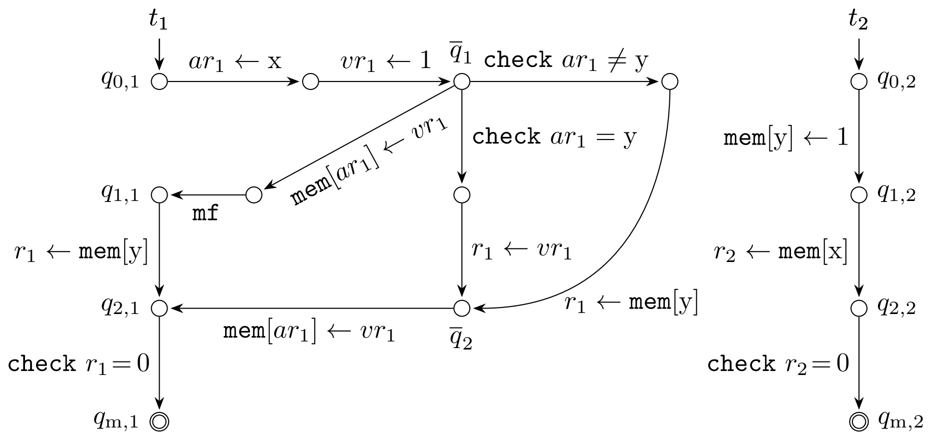


Figure 3.2: Extension by $q_{0,1} \xrightarrow{\texttt{mem}[\text{x}] \leftarrow 1} q_{1,1} \xrightarrow{r_1 \leftarrow \texttt{mem}[\text{y}]} q_{1,2}$ of the ASSEMBLY simplified Dekker algorithm in Figure 2.18. The goal state $(\mathsf{pc}, \mathsf{val}, \mathsf{buf})$ where $\mathsf{val}(\text{x}) = \mathsf{val}(\text{y}) = 1$ and $\mathsf{val}(r_1) = \mathsf{val}(r_2) = 0$ is now SC-reachable.

additional instructions. Consider an instruction $inst \in I_t$ with $src(inst) = src(inst_i)$ for some $i \in [1..n]$ and check that $inst \neq inst_i$. We add instructions that recreate the stores buffered in the auxiliary registers and return to $dst(inst)$.

$$\begin{tikzcd}
\overline{q}_i \arrow[r, "\texttt{mem}[ar_1] \leftarrow vr_1"] & \circ & \cdots & \circ \arrow[r, "\texttt{mem}[ar_{\texttt{count}}] \leftarrow vr_{\texttt{count}}"] & \circ \arrow[r, "cmd(inst)"] & dst(inst)
\end{tikzcd}$$

Similarly, for all load instructions $inst_i$ as well as out of $\overline{q}_1$ we add instructions that flush and fence the pair (ar_1, vr_1), make visible the remaining buffered stores, and return to state q in the original control flow. Below, $q := src(inst_i)$ if $inst_i$ is a load and $q := dst(inst_1)$, otherwise. Intuitively, this captures behaviors that delay $inst_1$ past loads earlier than $inst_n$, and that do not delay $inst_1$ past the first load in ι.

$$\begin{tikzcd}
\overline{q}_i \arrow[r, "\texttt{mem}[ar_1] \leftarrow vr_1"] & \circ \arrow[r, "\texttt{mf}"] & \circ & \cdots & \circ \arrow[r, "\texttt{mem}[ar_{\texttt{count}}] \leftarrow vr_{\texttt{count}}"] & q
\end{tikzcd}$$

Figure 3.2 shows the extension of the program in Figure 2.18 by the instruction sequence $q_{0,1} \xrightarrow{\texttt{mem}[\text{x}] \leftarrow 1} q_{1,1} \xrightarrow{r_1 \leftarrow \texttt{mem}[\text{y}]} q_{1,2}$.

3.1.1 Soundness and Completeness

We show that Algorithm 3.1 is a decision procedure for acyclic programs. From here until (inclusively) Theorem 8 we assume programs to be acyclic, i.e., their instructions and control states form directed acyclic graphs.

Theorem 9 then explains how Algorithm 3.1 yields a semi-decision procedure for all programs.

We first prove the extension sound and complete (Lemma 6): extending $\mathcal{R}$ by sequence $\iota := O(\mathcal{R})$ does neither add nor remove TSO-reachable states. Afterwards, Lemma 7 shows that if Algorithm 3.1 extends $\mathcal{R}$ by ι (Line 8) then, in subsequent iterations of the algorithm, no new sequence returned by the oracle is the same as ι (projected back to $\mathcal{P}$). Next, by the first condition of an oracle and using Lemma 7, we establish that Algorithm 3.1 is a decision procedure for acyclic programs (Theorem 8). Finally, we show that Algorithm 3.1 can be turned into a semi-decision procedure for all programs using a bounded model checking approach (Theorem 9).

Lemma 6. *Let $\mathsf{ADR} \cup \mathsf{REG}$ be the addresses and registers of program $\mathcal{R}$ and let $\iota := O(\mathcal{R})$. Then, $(\mathsf{pc}, \mathsf{val}', \mathsf{buf}) \in \mathit{Reach}_{TSO}(\mathcal{R} \oplus \iota)$ if and only if $(\mathsf{pc}, \mathsf{val}, \mathsf{buf}) \in \mathit{Reach}_{TSO}(\mathcal{R})$ and $\mathsf{val}(a) = \mathsf{val}'(a)$ for all $a \in \mathsf{ADR} \cup \mathsf{REG}$.*

Let t be the thread that is modified by $\mathcal{R} \oplus \iota$. To prove Lemma 6, one can show that for any prefix α' of $\alpha \in \mathcal{C}_{\mathrm{TSO}}(\mathcal{R})$ there is a prefix β' of $\beta \in \mathcal{C}_{\mathrm{TSO}}(\mathcal{R} \oplus \iota)$, and vice versa, that maintain the following invariants.

I-0 $s_0 \xrightarrow{\alpha'} (\mathsf{pc}, \mathsf{val}, \mathsf{buf})$ and $s_0 \xrightarrow{\beta'} (\mathsf{pc}', \mathsf{val}', \mathsf{buf}')$.

I-1 If pc and pc' differ, they only differ for thread t. If $\mathsf{pc}(t) \neq \mathsf{pc}'(t)$, then $\mathsf{pc}(t) = \mathit{dst}(\mathit{inst}_i)$ and $\mathsf{pc}'(t) = \overline{q}_i$ for some $i \in [1..n-1]$.

I-2 $\mathsf{val}'(a) = \mathsf{val}(a)$ for all $a \in \mathsf{ADR} \cup \mathsf{REG}$.

I-3 buf and buf' differ at most for t. If $\mathsf{buf}(t) \neq \mathsf{buf}'(t)$, then $\mathsf{pc}'(t) = \overline{q}_i$ for some $i \in [1..n-1]$ and $\mathsf{buf}(t) = (\widehat{ar_{\mathtt{count}}}, \widehat{vr_{\mathtt{count}}}) \cdots (\widehat{ar_1}, \widehat{vr_1}) \cdot \mathsf{buf}'(t)$ where $\mathtt{count}$ stores are seen along ι from $\mathit{src}(\mathit{inst}_1)$ to $\mathit{dst}(\mathit{inst}_i)$.

For clarity, the proof of Lemma 6 is presented in Appendix A.

We now show that the oracle never suggests the same sequence σ twice. Since in $\mathcal{R} \oplus \iota$ we introduce new instructions that correspond to instructions in $\mathcal{R}$, we have to map back sequences of $I_{\oplus}$ instructions from $\mathcal{R} \oplus \iota$ to sequences of I instructions from $\mathcal{R}$. Intuitively, the mapping gives the original instructions from which the sequence was produced.

Formally, we define a family of projection functions $h_\iota \colon I_{\oplus}^* \to I^*$ with $h_\iota(\varepsilon) := \varepsilon$ and $h_\iota(w \cdot \mathit{inst}) := h_\iota(w) \cdot h_\iota(\mathit{inst})$. For an instruction $\mathit{inst} \in I_{\oplus}$, we define $h_\iota(\mathit{inst}) := \mathit{inst}$ provided $\mathit{inst} \in I$. We set $h_\iota(\mathit{inst}) := \mathit{inst}_i$ if inst is a first instruction on the path between $\overline{q}_{i-1}$ and $\overline{q}_i$ for some $i \in [1..n]$. In all other cases, we skip the instruction, $h_\iota(\mathit{inst}) := \varepsilon$. Then, if $\mathcal{R}_0 := \mathcal{P}$ is the original program, ι_j is the sequence that the oracle returns in iteration $j \in \mathbb{N}$ of the while loop, and w is a sequence of instructions in $\mathcal{R}_{j+1}$, we define $h(w) := h_{\iota_0}(\ldots h_{\iota_j}(w))$. This latter function maps sequences of instructions in program $\mathcal{R}_{j+1}$ back to sequences of instructions in $\mathcal{P}$.

We are ready to state our key lemma. Intuitively, Lemma 7 states that the oracle does not repeat itself.

Lemma 7. *Let $\mathcal{R}_0 := \mathcal{P}$ and $\mathcal{R}_{i+1} := \mathcal{R}_i \oplus \iota_i$ for each $\iota_i := O(\mathcal{R}_i)$ as in Algorithm 3.1. If $\iota_{j+1} \neq \varepsilon$ then $h(\iota_{j+1}) \neq h(\iota_i)$ for all $i \leq j$.*

Proof. Assume, to the contrary, that $h(\iota_{j+1}) = h(\iota_i)$ for some $i \leq j$ where $\iota_{j+1} := O(\mathcal{R}_{j+1})$ and $\iota_i := O(\mathcal{R}_i)$. Furthermore, let $\mathit{inst}_{\text{first}}$ be the first (store) instruction and $\mathit{inst}_{\text{last}}$ be the last (load) instruction of the instruction sequence ι_{j+1}. Similarly, let $\mathit{inst}'_{\text{first}}$ and $\mathit{inst}'_{\text{last}}$ be the first and last instructions of the sequence ι_i. Since $h(\iota_{j+1}) = h(\iota_i)$ it means that $h(\mathit{inst}_{\text{first}}) = h(\mathit{inst}'_{\text{first}})$ and $h(\mathit{inst}_{\text{last}}) = h(\mathit{inst}'_{\text{last}})$.

However, since all control flows of $\mathcal{R}_{i+1} := \mathcal{R}_i \oplus \iota_i$ that recreate $h(\mathit{inst}'_{\text{first}})$ before $h(\mathit{inst}'_{\text{last}})$ also place a fence between the two, no other later sequences returned by the oracle have $h(\mathit{inst}'_{\text{first}})$ come before $h(\mathit{inst}'_{\text{last}})$. This in particular means that $\iota_{j+1} = O(\mathcal{R}_{j+1})$ where $h(\mathit{inst}_{\text{first}})$ comes before $h(\mathit{inst}_{\text{last}})$ does not exist. In conclusion, the initial assumption is false. □

We can now prove Algorithm 3.1 is sound and complete for acyclic programs (Theorem 8). Lemma 7 and the assumption that the input program is acyclic ensure that if no goal state is found SC-reachable (Line 4), then Algorithm 3.1 eventually runs out of sequences ι to return (Line 7). If that is the case, $O(\mathcal{R})$ returns ε in the last iteration of Algorithm 3.1. By the first oracle condition, we know that the SC- and TSO-reachable states of $\mathcal{R}$ are the same. Hence, no goal state is TSO-reachable in $\mathcal{R}$ and, by Lemma 6, no goal state is TSO-reachable in the input program $\mathcal{P}$ either. Otherwise, a goal state s is SC-reachable by some computation τ in $\mathcal{R}_j$ for some $j \in \mathbb{N}$ and, by Lemma 6, there is a TSO computation in $\mathcal{P}$ corresponding to τ that reaches s.

Theorem 8. *For acyclic programs, Algorithm 3.1 terminates. Moreover, it returns true on input $\mathcal{P}$ if and only if $\mathit{Reach}_{TSO}(\mathcal{P}) \cap G \neq \emptyset$.*

Proof. It is immediate that Algorithm 3.1 terminates for acyclic programs. On the one hand, the number of instruction sequences that start with a store and end with a load (as the second oracle condition requires) is finite in such a program $\mathcal{P}$. On the other hand, by Lemma 7, at each iteration the oracle returns a sequence that differs (in $\mathcal{P}$) from the previous ones. These two facts imply termination.

We now prove that $\mathit{Reach}_{\text{TSO}}(\mathcal{P}) \cap G \neq \emptyset$ iff Algorithm 3.1 returns *true* on input $\mathcal{P}$. For the easy direction ($\Leftarrow$), assume that Algorithm 3.1 returns *true* on input $\mathcal{P}$. This means that $\mathit{Reach}_{\text{SC}}(\mathcal{R}) \cap G \neq \emptyset$ in the last iteration of the algorithm's loop. Then, by $\mathit{Reach}_{\text{SC}}(\mathcal{R}) \subseteq \mathit{Reach}_{\text{TSO}}(\mathcal{R})$ and Lemma 6, we know that $\mathit{Reach}_{\text{SC}}(\mathcal{R}) \subseteq \mathit{Reach}_{\text{TSO}}(\mathcal{P})$. Hence, $\mathit{Reach}_{\text{TSO}}(\mathcal{P}) \cap G \neq \emptyset$.

For the reverse direction ($\Rightarrow$), assume that $\mathit{Reach}_{\text{TSO}}(\mathcal{P}) \cap G \neq \emptyset$. Furthermore, let $\mathcal{R}_0 := \mathcal{P}$ and $\mathcal{R}_{i+1} := \mathcal{R}_i \oplus \iota_i$ for $\iota_i := O(\mathcal{R}_i)$. By the initial termination argument we know there exists $j \in \mathbb{N}$ such that the algorithm terminates with $\mathcal{R} = \mathcal{R}_j$ in its last loop iteration. That means that either the check in Line 3 of the algorithm succeeds, in which case Algorithm 3.1 returns *true*, or the check in Line 7 of the algorithm fails, i.e. $O(\mathcal{R}_j) = \epsilon$ and $\mathit{Reach}_{\mathrm{SC}}(\mathcal{R}_j) \cap G = \emptyset$. In the latter case, by the first oracle condition we know that $\mathit{Reach}_{\mathrm{TSO}}(\mathcal{R}_j) \cap G = \emptyset$ and, by Lemma 6, we get $\mathit{Reach}_{\mathrm{TSO}}(\mathcal{R}_j) \subseteq \mathit{Reach}_{\mathrm{TSO}}(\mathcal{R}_0)$. Then, $\mathit{Reach}_{\mathrm{TSO}}(\mathcal{P}) \cap G = \emptyset$ contradicts the above assumption and concludes the proof. □

To establish that Algorithm 3.1 yields a semi-decision procedure for all programs, one can use an iterative bounded model checking approach. Bounded model checking unrolls the input program $\mathcal{P}$ up to a bound $k \in \mathbb{N}$ on the length of computations. Then Algorithm 3.1 is applied to the resulting programs $\mathcal{P}_k$. If it finds a goal state TSO-reachable in $\mathcal{P}_k$, by Lemma 6, this state corresponds to a TSO-reachable goal state in $\mathcal{P}$. Otherwise, we increase k and try again. By Theorem 8, we know that Algorithm 3.1 is a decision procedure for each $\mathcal{P}_k$. This implies that Algorithm 3.1 together with iterative bounded model checking yields a semi-decision procedure that terminates for all positive instances of TSO reachability. For negative instances of TSO reachability, however, the procedure is guaranteed to terminate only if the input program $\mathcal{P}$ is acyclic.

Theorem 9. *We have $\mathit{Reach}_{TSO}(\mathcal{P}) \cap G \neq \emptyset$ if and only if, for large enough $k \in \mathbb{N}$, Algorithm 3.1 returns true on input $\mathcal{P}_k$.*

Proof. Assume that $\mathit{Reach}_{\mathrm{TSO}}(\mathcal{P}) \cap G \neq \emptyset$. Then there exist some state $s \in G$ and $\alpha \in \mathcal{C}_{\mathrm{TSO}}(\mathcal{P})$ such that $s_0 \xrightarrow{\alpha} s$. Let k be the length of α and G' be the goal states of $\mathrm{X}_{\mathrm{TSO}}(\mathcal{P}_k)$. There exists a computation $\beta \in \mathcal{C}_{\mathrm{TSO}}(\mathcal{P}_k)$ that mimics α and reaches $s' \in G'$. Hence, $G' \cap \mathit{Reach}_{\mathrm{TSO}}(\mathcal{P}_k) \neq \emptyset$ and, by Theorem 8, Algorithm 3.1 returns *true* on input $\mathcal{P}_k$.

For the reverse direction, assume that Algorithm 3.1 returns *true* on input $\mathcal{P}_k$ for some $k \in \mathbb{N}$. Let s'_0 be the initial state of $\mathrm{X}_{\mathrm{TSO}}(\mathcal{P}_k)$ and, as before, G' be the goal states of $\mathrm{X}_{\mathrm{TSO}}(\mathcal{P}_k)$. By Theorem 8, there exists $s' \in G' \cap \mathit{Reach}_{\mathrm{TSO}}(\mathcal{P}_k)$ and $\beta \in \mathcal{C}_{\mathrm{TSO}}(\mathcal{P}_k)$ such that $s'_0 \xrightarrow{\beta} s'$. Since $\mathcal{P}_k$ unrolls $\mathcal{P}$ up to bound k, there exists a computation $\alpha \in \mathcal{C}_{\mathrm{TSO}}(\mathcal{P})$ that mimics β and reaches $s \in G$. Therefore, $G \cap \mathit{Reach}_{\mathrm{TSO}}(\mathcal{P}) \neq \emptyset$. □

An example of a safe program — wrt TSO unreachability[1] — for which the algorithm described in Theorem 9 does not terminate is depicted in Figure 3.3. The program is safe since none of its goal states is TSO-reachable: the initial control states will never be left since the conditionals will never succeed. However, although every $\mathcal{P}_k$ that unrolls the program in Figure 3.3 up to $k \in \mathbb{N}$ is found safe, the algorithm only stops if a TSO-reachable state is found or if $O(\mathcal{R}) = \epsilon$, which is never the case.

[1]Although this program is safe wrt TSO unreachability it is not safe wrt robustness.

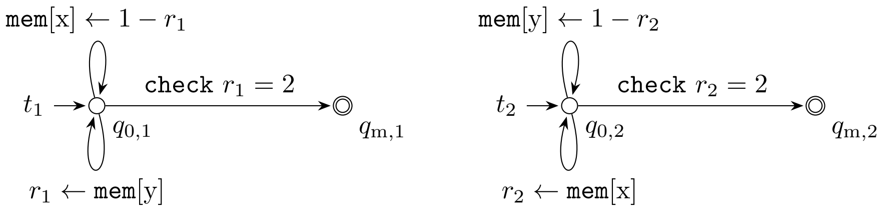


Figure 3.3: A safe program for which Algorithm 3.1 never terminates.

The underlying reason why always $O(\mathcal{R}) \neq \epsilon$ is that there are infinitely many sequences $inst_{\text{store}}^{m} \cdot inst_{\text{load}}$ where $m \in \mathbb{N}$ and

$$\text{either } inst_{\text{store}} = q_{0,1} \xrightarrow{\texttt{mem}[x]\leftarrow 1 - r_1} q_{0,1} \text{ and } inst_{\text{load}} = q_{0,1} \xrightarrow{r_1 \leftarrow \texttt{mem}[y]} q_{0,1}$$
$$\text{or } inst_{\text{store}} = q_{0,2} \xrightarrow{\texttt{mem}[y]\leftarrow 1 - r_2} q_{0,2} \text{ and } inst_{\text{load}} = q_{0,2} \xrightarrow{r_2 \leftarrow \texttt{mem}[x]} q_{0,2}.$$

3.1.2 A Robustness-based Oracle

We conclude this section by describing how robustness — introduced in Section 2.3.3 — can be used as an oracle.

Our robustness-based oracle is best described in terms of the following characterization of robustness from earlier work [BDM13]: a program $\mathcal{P}$ is not robust against TSO iff $\mathcal{C}_{\text{TSO}}(\mathcal{P})$ contains a computation, called *witness*, as in Figure 3.4. In contraposition this can be stated as follows.

Theorem 10 ([BDM13])**.** *A program $\mathcal{P}$ is robust against TSO if and only if the set of TSO computations $\mathcal{C}_{TSO}(\mathcal{P})$ contains no witness.*

Intuitively, a witness τ delays stores of only one thread in $\mathcal{P}$. The other threads adhere to the SC semantics. Conditions (W1) – (W4) in Figure 3.4 describe formally this restrictive behavior. Furthermore, condition (W5) implies that no computation $\sigma \in \mathcal{C}_{\text{SC}}(\mathcal{P})$ can satisfy $\rightarrow_{hb} (\tau) = \rightarrow_{hb} (\sigma)$.

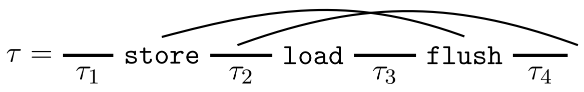


Figure 3.4: A witness τ with `store` $\leftrightarrow$ `flush` and store-delaying thread $t := thread(\texttt{store}) = thread(\texttt{load})$ satisfies the following constraints: (W1) Only thread t delays stores. (W2) Event `flush` represents the first delayed store of t and `load` is the last event of t past which `flush` is delayed. So τ_2 contains neither flush events nor fences of t. (W3) Sequence τ_3 contains no events of thread t. (W4) Sequence τ_4 consists only of flush events `e` of thread t. All these events `e` satisfy $addr(\texttt{e}) \neq addr(\texttt{load})$. (W5) We require `load` $\rightarrow_{hb}^{+}$ `e` for all events `e` in $\tau_3 \cdot$ `flush`.

To give an example, computation τ_{wit} from page 37 is a witness for the Assembly Dekker algorithm in Figure 2.18. Indeed, in no SC computation of this program can both loads read the initial values of x and y. Relative to Figure 3.4, we have $\texttt{store} = \texttt{store}_1$, $\texttt{load} = \texttt{load}_1$, $\texttt{flush} = \texttt{flush}_1$, $\tau_3 = \texttt{store}_2 \cdot \texttt{flush}_2 \cdot \texttt{load}_2$, and $\tau_1 = \tau_2 = \tau_4 = \varepsilon$.

The *robustness-based oracle*, given input $\mathcal{P}$, finds a witness τ as described in Figure 3.4 and returns the sequence of instructions for the events in $\texttt{store} \cdot \tau_2 \cdot \texttt{load}$ that belong to thread t. If no witness exists, it returns ε.

By Theorems 5 and 10, we find that the robustness-based oracle indeed satisfies the oracle conditions from Section 3.1. Furthermore, given a robust program and the robustness-based oracle as input, Algorithm 3.1 returns within the first iteration of its while loop.

3.2 Over-approximating Buffer Abstractions

In this section we describe several over-approximative abstractions for the earlier (Figure 2.19) TSO semantics. Throughout this section we use ${}^{k}\texttt{buf}$ to denote, for $k \in \{s, m\} \cup \mathbb{N}$, the various buffer abstractions. More precisely, we use s to denote the set abstraction of TSO buffers, m the multiset buffer abstraction, and $k \in \mathbb{N}$ the partial coherent abstraction [KVY11] that uses a k-bounded queue.

The first natural abstraction that we describe approximates TSO buffers by sets, as first introduced in [KVY11]. After we prove that this first abstraction preserves TSO reachability properties we generalize it to multisets and, as introduced in [KVY11], enhance it by bounded queues. Using similar invariant arguments we are able to show all the buffer abstractions preserve TSO reachability properties.

We acknowledge that the abstractions we present may arguably be either imprecise or too expensive for many safe programs. However, additionally to being useful for a large class of programs, they are a good alternative to the non-primitive-recursive-complete TSO reachability approach [Ati+10; Abd+13]. Furthermore, these set-based abstractions may very well serve as the basis for better over-approximations.

A novel aspect in our study consists in identifying that, when dealing with spurious set-buffer abstraction counterexamples, the multiset-buffer abstraction is naturally complementary to partial coherence abstractions. This makes it possible to come up with a refinement algorithm for checking safety wrt TSO reachability using partial coherence abstractions.

Furthermore, we show that reachability is decidable for the multiset buffer abstraction with per-address last-added-value information. This is the case since the multiset-abstract semantics is provably a well-structured transition system with computable minimal predecessors and decidable well-quasi order. We present the details to the decidability proof in Appendix B.

3.2.1 Set Buffer Abstractions

The most natural way to approximate TSO buffers is by using sets instead of queues. For a little extra precision we also assume that the structure of a thread's set-buffer approximation tracks the last buffered values per address. Such a set-based abstraction corresponds to a partial coherence abstraction with $k = 0$ bounded-buffers [KVY11]. Figure 3.5 depicts the intuitive view that a thread's buffer is an $\mathsf{ADR} \times \mathsf{DOM}$ subset with protuberances for per-address last added values.

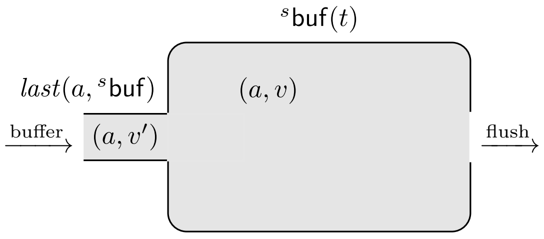


Figure 3.5: Shape of a set-approximating store buffer. Buffering a new pair (a, v'') replaces (a, v') as a's last added value. Flushing (a, v') from a's last added value position is possible iff no other (a, v) pair — for the same address a — is in the set.

Figure 3.6 describes formally the set-based abstract semantics $A_{set}(\mathcal{P})$. In contrast to the concrete TSO semantics on page 36 (Figure 2.19) store flushes are now (non-deterministically) either destructive or non-destructive: a pair (a, v) in the set buffer can be flushed either by rule (WM-D), and thus be removed from ${}^{s}\mathsf{buf}$, or by rule (WM-ND), and thus the flush would leave ${}^{s}\mathsf{buf}$ unaltered. The other rules of $A_{set}(\mathcal{P})$ stay the same — up to using sets (as described in Figure 3.5) instead of queues for buffers.

Similarly to the concrete TSO semantics, abstract TSO computations in the semantics $A_{set}(\mathcal{P})$ are defined by the automaton language $\mathcal{L}_F(A_{set}(\mathcal{P}))$ where F is the set of states with empty (set) buffers. Lemma 11 below shows that abstract TSO computations in $A_{set}(\mathcal{P})$ are also a superset of $\mathcal{P}$'s concrete TSO computations: $\mathcal{C}_{\mathrm{TSO}}(\mathcal{P}) = \mathcal{L}_F(\mathrm{X}_{\mathrm{TSO}}(\mathcal{P})) \subseteq \mathcal{L}_F(A_{set}(\mathcal{P}))$. The detailed proof of Lemma 11 is presented in Appendix A.

Lemma 11. *For any program $\mathcal{P}$, $\mathcal{C}_{TSO}(\mathcal{P}) \subseteq \mathcal{L}_F(A_{set}(\mathcal{P}))$.*

To prove Lemma 11, one can show that for any prefix α' of $\alpha \in \mathcal{C}_{\mathrm{TSO}}(\mathcal{P})$ such that $s_0 \xrightarrow{\alpha'} s := (\mathsf{pc}, \mathsf{val}, \mathsf{buf})$ the following invariants are maintained:

I-7 $s_0 \xrightarrow{\alpha'} s' := (\mathsf{pc}', \mathsf{val}', {}^{s}\mathsf{buf})$ is a valid computation prefix in $A_{set}(\mathcal{P})$.

I-8 $\mathsf{pc} = \mathsf{pc}'$ and $\mathsf{val} = \mathsf{val}'$.

I-9 for all threads t and addresses a, $last(a, \mathsf{buf}(t)) = last(a, {}^{s}\mathsf{buf}(t))$ and $(a, v) \in \mathsf{buf}(t) \cap (\mathsf{ADR} \times \mathsf{DOM})$ iff $(a, v) \in {}^{s}\mathsf{buf}(t)$.

$$\frac{cmd = r \leftarrow \texttt{mem}[e_a], \quad a = \widehat{e_a}, \quad \exists\, v = last(a, {}^s\mathsf{buf}(t))}{s \xrightarrow{(t,inst,a)} (\mathsf{pc}', \mathsf{val}[r := v], {}^s\mathsf{buf})} \text{(RB)}$$

$$\frac{cmd = r \leftarrow \texttt{mem}[e_a], \quad a = \widehat{e_a}, \quad {}^s\mathsf{buf}(t) \cap (\{a\} \times \mathsf{DOM}) = \emptyset}{s \xrightarrow{(t,inst,a)} (\mathsf{pc}', \mathsf{val}[r := \mathsf{val}(a)], {}^s\mathsf{buf})} \text{(RM)}$$

$$\frac{cmd = \texttt{mem}[e_a] \leftarrow e_v, \quad a = \widehat{e_a}, \quad v = \widehat{e_v}}{s \xrightarrow{(t,inst,a)} (\mathsf{pc}', \mathsf{val}, {}^s\mathsf{buf}[t := {}^s\mathsf{buf}(t) \cup (a, v)])} \text{(LS)}$$

$$\frac{{}^s\mathsf{buf}(t) = \mathcal{S} \uplus \{(a,v)\}, \quad v \neq last(a, {}^s\mathsf{buf}(t)) \vee \mathcal{S} \cap (\{a\} \times \mathsf{DOM}) = \emptyset}{s \xrightarrow{(t,\text{flush},a)} (\mathsf{pc}, \mathsf{val}[a := v], {}^s\mathsf{buf}[t := \mathcal{S}])} \text{(WM-D)}$$

$$\frac{{}^s\mathsf{buf}(t) = \mathcal{S} \uplus \{(a,v)\}}{s \xrightarrow{(t,\text{flush},a)} (\mathsf{pc}, \mathsf{val}[a := v], {}^s\mathsf{buf})} \text{(WM-ND)}$$

$$\frac{cmd = \texttt{mf}, \quad {}^s\mathsf{buf}(t) = \emptyset}{s \xrightarrow{(t,inst,\perp)} (\mathsf{pc}', \mathsf{val}, {}^s\mathsf{buf})} \text{(LF)}$$

$$\frac{cmd = r \leftarrow e, \quad v = \widehat{e}}{s \xrightarrow{(t,inst,\perp)} (\mathsf{pc}', \mathsf{val}[r := v], {}^s\mathsf{buf})} \text{(LA)}$$

$$\frac{cmd = \texttt{check}\ e, \quad \widehat{e} \neq 0}{s \xrightarrow{(t,inst,\perp)} (\mathsf{pc}', \mathsf{val}, {}^s\mathsf{buf})} \text{(LC)}$$

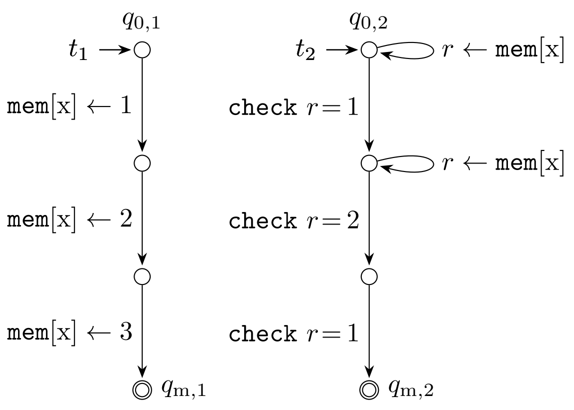
Figure 3.6: Transition semantics rules for $A_{set}(\mathcal{P})$ assuming $s = (\mathsf{pc}, \mathsf{val}, {}^s\mathsf{buf})$ with $\mathsf{pc}(t) = q$ and $inst = q \xrightarrow{cmd} q'$ in thread t. Except for rules (WM-D) and (WM-ND), the program counter is always updated by $\mathsf{pc}' = \mathsf{pc}[t := q']$. We use (as before) $\widehat{e}$ for the result of evaluating expression e under val and $last(a, {}^s\mathsf{buf}(t))$ for the last value to address a buffered by thread t stores. Operators $\cup$ and $\uplus$ denote set union and, respectively, disjoint set union.

With the same meaning of goal state as described for TSO reachability in section 2.3.2, we use notation $Reach_{set}(\mathcal{P}) \cap G$ for the set of goal states that are reachable by some abstract TSO computation in $A_{set}(\mathcal{P})$. Using Lemma 11 one can conclude that the set buffer abstraction is a *safe* over-approximation for TSO reachability.

Theorem 12. *For any program $\mathcal{P}$, $Reach_{TSO}(\mathcal{P}) \cap G \subseteq Reach_{set}(\mathcal{P}) \cap G$.*

Proof. Let $s \in Reach_{\text{TSO}}(\mathcal{P}) \cap G$ and assume τ is a TSO computation that ends in this state s. By Lemma 11, computation τ belongs to the language $\mathcal{L}_F(A_{set}(\mathcal{P}))$ of the abstract semantics $A_{set}(\mathcal{P})$. Hence, $s \in Reach_{set}(\mathcal{P})$ and, since $s \in G$ as well, we conclude that $s \in Reach_{set}(\mathcal{P}) \cap G$. □

Theorem 12 guarantees that if no TSO goal state is reachable in $A_{set}(\mathcal{P})$



$q_{0,1}$
t_1
$\texttt{mem}[x] \leftarrow 1$
$\texttt{mem}[x] \leftarrow 2$
$\texttt{mem}[x] \leftarrow 3$
$q_{m,1}$
$q_{0,2}$
t_2
$r \leftarrow \texttt{mem}[x]$
$\texttt{check } r=1$
$r \leftarrow \texttt{mem}[x]$
$\texttt{check } r=2$
$\texttt{check } r=1$
$q_{m,2}$

Figure 3.7: A program that the set abstraction cannot prove safe. This program can be proved safe using the *multiset buffer abstraction* as well as using a k-bounded *partial coherence abstraction* where $k \geq 3$.

then the program $\mathcal{P}$ is safe under TSO reachability. Concretely, this means that $\mathit{Reach}_{set}(\mathcal{P}) \cap G = \emptyset$ implies $\mathit{Reach}_{\mathrm{TSO}}(\mathcal{P}) \cap G = \emptyset$.

For example, for the Figure 3.3 program on page 51 we have[2]

$$\mathit{Reach}_{set}(\mathcal{P}) = \{(\mathsf{pc}, \mathsf{val}, {}^{s}\mathsf{buf}) \mid \mathsf{pc} = (q_{0,1}, q_{0,2}),\ \mathsf{val} \in \{\mathrm{x}, \mathrm{y}, r_1, r_2\} \rightarrow \{0,1\},$$
$$\text{and } {}^{s}\mathsf{buf} \subseteq \{(\mathrm{x},0),(\mathrm{x},1)\} \times \{(\mathrm{y},0),(\mathrm{y},1)\}\}.$$

To see that these are indeed the reachable states, notice that the state $(\mathsf{pc}_0, \mathsf{val}_0, {}^{s}\mathsf{buf})$ with ${}^{s}\mathsf{buf}(t_1) = \{(\mathrm{x},0),(\mathrm{x},1)\}$ and ${}^{s}\mathsf{buf}(t_2) = \{(\mathrm{y},0),(\mathrm{y},1)\}$ is reachable in $A_{set}(\mathcal{P})$. From this state one can reach any possible $\{0,1\}$-valued configuration of $\mathsf{ADR} \cup \mathsf{REG}$ by appropriately interleaving (WM-ND) flushes and (RM) memory loads. The states with smaller buffer contents can then be reached using (WM-D) flushes.

Since no state $((q_{m,1}, q_{m,2}), \mathsf{val}, \mathsf{buf})$ belongs to the above set $\mathit{Reach}_{set}(\mathcal{P})$ the Figure 3.3 program is safe wrt TSO reachability.

As one may think, the set buffer abstraction is not sufficiently precise for some safe programs. Figure 3.7 depicts an Assembly program that cannot be proved safe wrt TSO reachability using the set-buffer abstraction. This prompts looking for finer abstractions like the ones we are about to present.

Let's assume that some program $\mathcal{P}$ is unsafe wrt reachability under the set buffer abstraction. This will be witnessed by some counterexample computation in $A_{set}(\mathcal{P})$. If the counterexample is a valid TSO computation we would know, according to Theorem 12, that $\mathcal{P}$ is indeed not safe wrt TSO reachability. However, if the counterexample is spurious, i.e., it is not a concrete TSO computation, one must decide what measures can be taken

[2]I.e., for the Figure 3.3 program, $\mathit{Reach}_{set}(\mathcal{P})$ contains $\underbrace{2 \times 2 \times 2 \times 2}_{\text{valuation}} \times \underbrace{4 \times 4}_{\text{buffers}}$ states.

towards deciding TSO reachability. An informed decision should rely on the two reasons why an $A_{set}(\mathcal{P})$ counterexample may be spurious:

(i) the number of distinct address-value pairs buffered and flushed differs;

(ii) the set abstraction loses the order of buffered address-value pairs.

To address point (i) above we propose the *multiset buffer abstraction* $A_{mset}(\mathcal{P})$, a natural extension to the presented set buffer abstraction. Using the theory of Well-structured Transition System we furthermore conclude, in Section 3.2.2, that reachability in this finer abstraction is still decidable.

To address the second reason (ii) why an $A_{set}(\mathcal{P})$ counterexample may be spurious we present *partial coherence abstractions* in Section 3.2.3. Intuitively, given some $k \in \mathbb{N}$, a k-bounded partial coherence abstraction $A_k(\mathcal{P})$ refines the set buffers by combining them with (per-thread) k-bounded queues — setting k to 0 defaults to a set abstraction. These abstractions were introduced by Kuperstein et al. [KVY11] and represent one way for regaining potentially relevant orderings of buffered stores.

Algorithm 3.2 Combining abstractions for unreachability checking.

```
Input: Marked program $\mathcal{P}$
Output: true if $\mathcal{P}$ is safe (no goal state is TSO-reachable in $\mathcal{P}$)
        false if $\mathcal{P}$ is not safe (some goal state is TSO-reachable in $\mathcal{P}$)
 1: if $Reach_{mset}(\mathcal{P}) \cap G = \emptyset$ then          // check reachability in $A_{mset}(\mathcal{P})$
 2:     return true;
 3: else
 4:     while true do
 5:         if previous check finds a concrete $\mathcal{C}_{\mathrm{TSO}}(\mathcal{P})$ counterexample then
 6:             return false;
 7:         else
 8:                         // use previous counterexamples to find a finer $k \in \mathbb{N}$
 9:             if $Reach_k(\mathcal{P}) \cap G = \emptyset$ then    // check reachability in $A_k(\mathcal{P})$
10:                 return true;
11:             end if
12:         end if
13:     end while
14: end if
```

Algorithm 3.2 depicts a possible way to combine the partial coherence and multiset abstractions that we will present next. The algorithm should be interpreted as a refinement scheme that uses partial coherence as a failsafe when reachability is unsuccessful under the multiset abstraction.

If, for example, finding a finer $k \in \mathbb{N}$ at line 8 of Algorithm 3.2 would simply increment the value of k then the algorithm would eventually (and correctly) terminate for programs whose goal states are TSO reachable. As is, termination in other cases is generally not guaranteed.

3.2.2 Multiset Buffer Abstractions

A simple extension to the previous set buffer abstraction sees multisets replace the buffer sets. Stated differently, starting from the concrete TSO semantics, the multiset buffer abstraction tracks per-thread-and-address last-added values while replacing the queue structure of the TSO buffers by multisets.

We use a numeric-valued function notation for multisets. Formally, given some finite ground set $\mathcal{S}$, any mapping $f\colon \mathcal{S} \to \mathbb{N}$ defines a multiset. We will also use $[a_1, \ldots, a_n]_{i_1,\ldots,i_n}$ as notation for the multiset containing distinct elements $a_1, \ldots, a_n \in \mathcal{S}$ with multiplicities $i_1, \ldots, i_n \in \mathbb{N}$. Intuitively, a multiset f over ground set $\mathcal{S}$ is the same as $[a_1, \ldots, a_n]_{i_1,\ldots,i_n}$ iff

$$f(a) = \begin{cases} i_j & \text{if } a = a_j \text{ for some } j \in [1..n], \\ 0 & \text{otherwise, i.e., if } a \in \mathcal{S} \setminus \{a_1, \ldots, a_n\}. \end{cases}$$

To give some concrete examples, $[\,]$ always denotes the empty multiset while $[(0,1)]_2$ denotes the multiset containing twice the pair $(0,1)$.

Figure 3.8 describes the multiset-based abstract semantics $A_{mset}(\mathcal{P})$. In contrast to the concrete TSO semantics on page 36 (Figure 2.19) stores add and flush address-value pairs to/from a multiset instead of to/from a queue. Similarly to the Figure 3.5 set buffers (used by the Figure 3.6 semantics), the last-added address-value pairs are per-address tracked. This reflects in the way address-value pairs are flushed from the multiset buffer ${}^{m}\mathsf{buf}$. Namely, an address-value pair (a, v) is flushed from ${}^{m}\mathsf{buf}$ either

(1) if it's value v is different from the last-added one for address a, or
(2) if its multiplicity ${}^{m}\mathsf{buf}((a, v))$ is higher than 1, or
(3) if ${}^{m}\mathsf{buf}{\downarrow}(\{a\} \times \mathsf{DOM})$ is precisely the multiset $[(a, v)]_1$.

The latter case takes care that it never occurs for neither (1) nor (2) to hold while a last-added address-value pair is flushed before another pair having the same address but a different value.

As before, abstract TSO computations in $A_{mset}(\mathcal{P})$ are defined by the automaton language $\mathcal{L}_F(A_{mset}(\mathcal{P}))$ where F is the set of states with empty buffers. Moreover, Lemma 13 below posits, unsurprisingly, that $A_{mset}(\mathcal{P})$ computations are a finer superset of $\mathcal{P}$'s concrete TSO computations, i.e., $\mathcal{C}_{\text{TSO}}(\mathcal{P}) = \mathcal{L}_F(\mathrm{X}_{\text{TSO}}(\mathcal{P})) \subseteq \mathcal{L}_F(A_{mset}(\mathcal{P})) \subseteq \mathcal{L}_F(A_{set}(\mathcal{P}))$.

Lemma 13. *For any program $\mathcal{P}$, $\mathcal{C}_{TSO}(\mathcal{P}) \subseteq \mathcal{L}_F(A_{mset}(\mathcal{P})) \subseteq \mathcal{L}_F(A_{set}(\mathcal{P}))$.*

Proof (sketch). Proving the left-hand-side inclusion follows similarly to the proof of Lemma 11. More precisely, one can show that for any prefix α' of $\alpha \in \mathcal{C}_{\text{TSO}}(\mathcal{P})$ such that $s_0 \xrightarrow{\alpha'} s := (\mathsf{pc}, \mathsf{val}, \mathsf{buf})$ the following — slightly strengthened — invariants are maintained:

$$\frac{cmd = r \leftarrow \texttt{mem}[e_a], \quad a = \widehat{e_a}, \quad \exists\, v = last(a, {}^{m}\mathsf{buf}(t))}{s \xrightarrow{(t,inst,a)} (\mathsf{pc}', \mathsf{val}[r := v], {}^{m}\mathsf{buf})} \text{(RB)}$$

$$\frac{cmd = r \leftarrow \texttt{mem}[e_a], \quad a = \widehat{e_a}, \quad {}^{m}\mathsf{buf}(t){\downarrow}(\{a\} \times \mathsf{DOM}) = [\,]}{s \xrightarrow{(t,inst,a)} (\mathsf{pc}', \mathsf{val}[r := \mathsf{val}(a)], {}^{m}\mathsf{buf})} \text{(RM)}$$

$$\frac{cmd = \texttt{mem}[e_a] \leftarrow e_v, \quad a = \widehat{e_a}, \quad v = \widehat{e_v}}{s \xrightarrow{(t,inst,a)} (\mathsf{pc}', \mathsf{val}, {}^{m}\mathsf{buf}[t := {}^{m}\mathsf{buf}(t) \uplus [(a,v)]_1])} \text{(LS)}$$

$$\frac{{}^{m}\mathsf{buf}(t) = f \uplus [(a,v)]_1, \quad v \neq last(a, {}^{m}\mathsf{buf}(t)) \lor f((a,v)) \neq 0 \lor f{\downarrow}(\{a\} \times \mathsf{DOM}) = [\,]}{s \xrightarrow{(t,\text{flush},a)} (\mathsf{pc}, \mathsf{val}[a := v], {}^{m}\mathsf{buf}[t := f])} \text{(WM)}$$

$$\frac{cmd = \texttt{mf}, \quad {}^{m}\mathsf{buf}(t) = [\,]}{s \xrightarrow{(t,inst,\perp)} (\mathsf{pc}', \mathsf{val}, {}^{m}\mathsf{buf})} \text{(LF)}$$

$$\frac{cmd = r \leftarrow e, \quad v = \widehat{e}}{s \xrightarrow{(t,inst,\perp)} (\mathsf{pc}', \mathsf{val}[r := v], {}^{m}\mathsf{buf})} \text{(LA)}$$

$$\frac{cmd = \texttt{check}\ e, \quad \widehat{e} \neq 0}{s \xrightarrow{(t,inst,\perp)} (\mathsf{pc}', \mathsf{val}, {}^{m}\mathsf{buf})} \text{(LC)}$$

Figure 3.8: Transition semantics rules for $A_{mset}(\mathcal{P})$ assuming $s = (\mathsf{pc}, \mathsf{val}, {}^{m}\mathsf{buf})$ with $\mathsf{pc}(t) = q$ and $inst = q \xrightarrow{cmd} q'$ in thread t. Except for rule (WM), the program counter is always updated by $\mathsf{pc}' = \mathsf{pc}[t := q']$. We use $\widehat{e}$ for the result of evaluating expression e under val, $last(a, {}^{m}\mathsf{buf}(t))$ for the last value to address a buffered by thread t stores, and $f{\downarrow}(\{a\} \times \mathsf{DOM})$ for the projection of f to store operations that access address a. Furthermore, unlike in Figure 3.6, here the operator $\uplus$ denotes multiset merge/addition.

I-10 $s_0 \xrightarrow{\alpha'} s' := (\mathsf{pc}', \mathsf{val}', {}^{m}\mathsf{buf})$ is a valid computation prefix in $A_{mset}(\mathcal{P})$.

I-11 $\mathsf{pc} = \mathsf{pc}'$ and $\mathsf{val} = \mathsf{val}'$.

I-12 for all threads t and addresses a, $last(a, \mathsf{buf}(t)) = last(a, {}^{m}\mathsf{buf}(t))$ and, for all values $v \in \mathsf{DOM}$, $|\mathsf{buf}(t){\downarrow}(\{(a,v)\})| = {}^{m}\mathsf{buf}(t)((a,v))$.

By complete induction one can then infer that $\alpha \in \mathcal{L}_F(A_{mset}(\mathcal{P}))$.

To prove the right-hand-side inclusion it remains to show, by complete induction over the length of some arbitrary $A_{mset}(\mathcal{P})$ computation, that all $A_{mset}(\mathcal{P})$ computations are also $A_{set}(\mathcal{P})$ computations. □

As before, we use notation $Reach_{mset}(\mathcal{P}) \cap G$ for the set of goal states that are reachable by some abstract TSO computation in $A_{mset}(\mathcal{P})$. Using Lemma 13 one can conclude that the multiset buffer abstraction is a safe over-approximation for TSO reachability.

Theorem 14. *For any program $\mathcal{P}$, $Reach_{TSO}(\mathcal{P}) \cap G \subseteq Reach_{mset}(\mathcal{P}) \cap G$.*

Proof. Let $s \in Reach_{\mathrm{TSO}}(\mathcal{P}) \cap G$ and assume τ is a TSO computation that ends in this state s. By Lemma 13, computation τ belongs to the language $\mathcal{L}_F(A_{mset}(\mathcal{P}))$ of the abstract semantics $A_{mset}(\mathcal{P})$. Hence, $s \in Reach_{mset}(\mathcal{P})$ and, since $s \in G$ as well, we conclude that $s \in Reach_{mset}(\mathcal{P}) \cap G$. □

Theorem 14 guarantees that if no TSO goal state is reachable in $A_{mset}(\mathcal{P})$ then the program $\mathcal{P}$ is safe under TSO reachability. Concretely, this means that $Reach_{mset}(\mathcal{P}) \cap G = \emptyset$ implies $Reach_{\mathrm{TSO}}(\mathcal{P}) \cap G = \emptyset$.

However, in order for the multiset abstraction to be useful, checking reachability in $A_{mset}(\mathcal{P})$ should, at the very least, be decidable. This is not immediate. First off, unlike for the always-finite $A_{set}(\mathcal{P})$, the finer multiset-abstracted semantics $A_{mset}(\mathcal{P})$ is, typically, infinite-state. In fact, to prove that reachability in $A_{mset}(\mathcal{P})$ is decidable we show that, for an appropriately chosen (and decidable) Well-quasi-ordering (WQO) over its states, $A_{mset}(\mathcal{P})$ is a Well-structured Transition System (WSTS) with effectively computable minimal predecessors.

Theorem 15. *For any program $\mathcal{P}$, $A_{mset}(\mathcal{P})$ reachability is decidable.*

The proof of Theorem 15 is presented in detail in Appendix B.

Possible algorithms for reachability in $A_{mset}(\mathcal{P})$ may, therefore, either chose to implement a typical backward reachability arising from computing minimal predecessors or, perhaps more efficiently, rely on a state-of-the-art forward EEC algorithm [GRB06].

3.2.3 Partial Coherence Abstractions

If the multiset abstraction fails to prove a program safe, partial coherence abstractions can be used as a last resort.

Intuitively, a k-bounded partial coherence abstraction $A_k(\mathcal{P})$ enhances set buffers by per-thread k-bounded queues. As depicted in Figure 3.9, for each thread, a k-bounded queue together with an $\mathsf{ADR} \times \mathsf{DOM}$ set with protuberances for per-address last added values are used to approximate store buffering.

Figure 3.10 describes the k-bounded partial coherence semantics $A_k(\mathcal{P})$.

As before, abstract TSO computations in $A_k(\mathcal{P})$ are defined by the automaton language $\mathcal{L}_F(A_k(\mathcal{P}))$ where F is the set of states with empty buffers. Moreover, Lemma 16 below posits, unsurprisingly, that $A_k(\mathcal{P})$ computations are also a finer superset of $\mathcal{P}$'s concrete TSO computations, i.e., $\mathcal{C}_{\mathrm{TSO}}(\mathcal{P}) = \mathcal{L}_F(\mathrm{X}_{\mathrm{TSO}}(\mathcal{P})) \subseteq \mathcal{L}_F(A_k(\mathcal{P})) \subseteq \mathcal{L}_F(A_{set}(\mathcal{P}))$.

Lemma 16. *For any program $\mathcal{P}$, $\mathcal{C}_{TSO}(\mathcal{P}) \subseteq \mathcal{L}_F(A_k(\mathcal{P})) \subseteq \mathcal{L}_F(A_{set}(\mathcal{P}))$.*

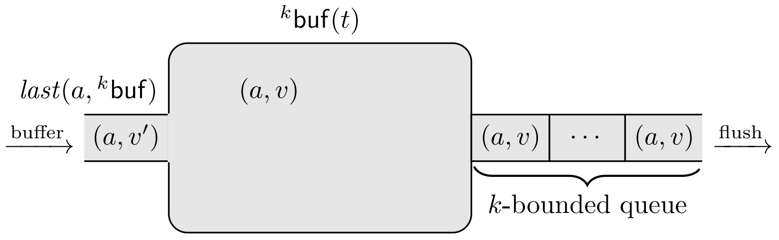


Figure 3.9: Shape of a k-bounded set-approximating store buffer. The set component from Figure 3.5 serves as failsafe when the k-bounded queue is full. Reusing the k-bounded queue is possible only if both the set and queue components are empty. E.g., before reusing the k-bounded queue out of the depicted configuration, all $\mathsf{ADR} \times \mathsf{DOM}$ pairs in $^k\mathsf{buf}(t)$ must be flushed (the items in the queue are flushed first, (a, v) second, and (a, v') last).

$$\frac{cmd = \mathtt{mem}[e_a] \leftarrow e_v, \quad a = \widehat{e_a}, \quad v = \widehat{e_v}, \quad |Q| < k, \quad R = \emptyset}{s \xrightarrow{(t, inst, a)} (\mathsf{pc}', \mathsf{val}, {}^k\mathsf{buf}[t := (a, v) \cdot Q])} \text{(LS-Q)}$$

$$\frac{cmd = \mathtt{mem}[e_a] \leftarrow e_v, \quad a = \widehat{e_a}, \quad v = \widehat{e_v} \quad R = \emptyset \vee |Q| = k}{s \xrightarrow{(t, inst, a)} (\mathsf{pc}', \mathsf{val}, {}^k\mathsf{buf}[t := (Q, R \cup (a, v))])} \text{(LS-S)}$$

$$\frac{Q = Q' \cdot (a, v),}{s \xrightarrow{(t, \text{flush}, a)} (\mathsf{pc}, \mathsf{val}[a := v], {}^s\mathsf{buf}[t := (Q', R)])} \text{(WM-Q)}$$

$$\frac{|Q| = 0, \quad R = \mathcal{S} \uplus \{(a, v)\}, \quad v \neq last(a, {}^k\mathsf{buf}(t)) \vee \mathcal{S} \cap (\{a\} \times \mathsf{DOM}) = \emptyset}{s \xrightarrow{(t, \text{flush}, a)} (\mathsf{pc}, \mathsf{val}[a := v], {}^s\mathsf{buf}[t := (Q, \mathcal{S})])} \text{(WM-SD)}$$

$$\frac{R = \mathcal{S} \uplus \{(a, v)\}}{s \xrightarrow{(t, \text{flush}, a)} (\mathsf{pc}, \mathsf{val}[a := v], {}^s\mathsf{buf})} \text{(WM-SND)}$$

Figure 3.10: Transition transition semantics rules for $A_k(\mathcal{P})$ assuming $s = (\mathsf{pc}, \mathsf{val}, {}^k\mathsf{buf})$ with $\mathsf{pc}(t) = q$ and $inst = q \xrightarrow{cmd} q'$ in thread t. Except in (WM-Q), (WM-SD) and (WM-SND), the program counter is always updated by $\mathsf{pc}' = \mathsf{pc}[t := q']$. As before $\widehat{e}$ evaluates e under val, $last(a, {}^k\mathsf{buf}(t))$ tracks the last value to address a buffered by thread t stores, and $\cup$ and $\uplus$ denote union and disjoint set union. Moreover, we use (Q, R) to denote the k-bounded queue and set components making up $^k\mathsf{buf}(t)$. Being the same as in the — Figure 3.6 — $A_s(\mathcal{P})$ semantics, rules (RB), (RM), (LF), (LA), (LC) are omitted.

Proof (sketch). Proving the left-hand-side inclusion follows similarly to the Lemma 11 proof. Namely, one can show that for any prefix α' of $\alpha \in \mathcal{C}_{\text{TSO}}(\mathcal{P})$ such that $s_0 \xrightarrow{\alpha'} s := (\mathsf{pc}, \mathsf{val}, \mathsf{buf})$ the following invariants hold:

I-13 $s_0 \xrightarrow{\alpha'} s' := (\mathsf{pc}', \mathsf{val}', {}^m\mathsf{buf})$ is a valid computation prefix in $A_k(\mathcal{P})$.

I-14 $\mathsf{pc} = \mathsf{pc}'$ and $\mathsf{val} = \mathsf{val}'$.

I-15 for all threads t and addresses a, $\mathit{last}(a, \mathsf{buf}(t)) = \mathit{last}(a, {}^{k}\mathsf{buf}(t))$ and, if ${}^{k}\mathsf{buf}(t) = (Q, R)$ such that Q is the queue and R the set component then $\mathsf{buf}(t) = Q' \cdot Q$ such that $(a, v) \in Q' \cap (\mathsf{ADR} \times \mathsf{DOM})$ iff $(a, v) \in R$.

By complete induction one can then infer that $\alpha \in \mathcal{L}_F(A_k(\mathcal{P}))$.

To prove the right-hand-side inclusion it remains to show, by complete induction over the length of some arbitrary $A_k(\mathcal{P})$ computation, that all $A_k(\mathcal{P})$ computations are also $A_{set}(\mathcal{P})$ computations. $\square$

As before, we use notation $\mathit{Reach}_k(\mathcal{P}) \cap G$ for the set of goal states that are reachable by some abstract TSO computation in $A_k(\mathcal{P})$. Using Lemma 16 one can then conclude that each k-bounded partial coherence abstraction is a safe over-approximation for TSO reachability.

Theorem 17. *For any program $\mathcal{P}$, $\mathit{Reach}_{TSO}(\mathcal{P}) \cap G \subseteq \mathit{Reach}_k(\mathcal{P}) \cap G$.*

Proof. Let $s \in \mathit{Reach}_{\mathrm{TSO}}(\mathcal{P}) \cap G$ and assume τ is a TSO computation that ends in this state s. By Lemma 16, computation τ belongs to the language $\mathcal{L}_F(A_k(\mathcal{P}))$ of the abstract semantics $A_k(\mathcal{P})$. Hence, $s \in \mathit{Reach}_k(\mathcal{P})$ and, since $s \in G$ as well, we conclude that $s \in \mathit{Reach}_k(\mathcal{P}) \cap G$. $\square$

Overall, we showed that each component of the hierarchy of approximations depicted below preserves TSO reachability properties. The partial coherence approximations are naturally ordered using their precision and, correspondingly, verification complexity. This allows for a gradual approximation scheme, as depicted in Algorithm 3.2.

$$\begin{array}{ccccccc}
\text{multiset} & & & & & & \\
\rotatebox{90}{$\subseteq$} & & & & & & \\
\text{set} & = & \begin{array}{c}\text{0-bounded}\\ \text{partial coherence}\end{array} & \subseteq & \begin{array}{c}\text{1-bounded}\\ \text{partial coherence}\end{array} & \subseteq & \cdots
\end{array}$$

Furthermore, using the theory of well-structured transition systems we could show that reachability in the multiset abstraction with per address last-added-values is decidable. Appendix B contains the details of our proof. Tight complexity bounds for this problem are not yet known. Such bounds might be found through a reduction from/to the ExpSpace-complete Petri net coverability problem [Lip76; Rac78]. This, however, seems to be nontrivial. Indeed, our attempts to find a reduction from Petri net coverability led to the conclusion that the nets require zero-test arcs.

Chapter 4

Partial Order Reduction

Contents

Explicit-state model checking is a state-of-the-art method for verifying concurrent programs. However, it typically suffers from exponential state-space explosion due to thread interleaving. Partial Order Reduction (POR) is one way to address this problem and it relies on building a subset of all program executions that is sufficient to explore all relevant states.

We described POR for $\mathrm{LTL}_{-\mathcal{X}}$ in the context of finite transition systems in Chapter 2. Now we introduce and discuss POR approaches applicable to the (potentially infinite) state-spaces of Assembly programs. Concretely, we show that — with a small change — the happens-before trace notion due to Shasha and Snir [SS88] is a good candidate to represent equivalent interleaved executions.[1] Technically, we present a state-space program semantics that includes trace information and can be used to check program safety. Moreover, in the context of reachability, we describe necessary conditions for an exploration method to be a POR technique as well as sufficient conditions for it to be optimal in the sense that no two executions with the same trace are explored. Finally, we show that the given state-space semantics is the backbone that can explain both *dynamic partial order reduction* [FG05] and

[1]The (Mazurkiewicz) traces used in this chapter resemble the happens-before relation in Section 2.3.3 and differ from the Section 2.2.1 traces used for model checking.

cartesian partial order reduction [Gue+07], two POR techniques introduced first in the context of finite transition systems under SC.

We describe POR for Assembly programs using persistent sets [God96] in Section 4.1. Apart from using the richer Assembly modeling language, the details of the persistent set perspective are very similar to the POR description in [Lin14]. Section 4.2 contains the chapter's main theoretical contribution: a trace-based description of Partial Order Reduction and an analysis of the reduction achievable in Assembly programs. To conclude, in Section 4.3 we recall two well known POR techniques in the context of Assembly programs under TSO.

Related work Various approaches to Partial Order Reduction exist in the literature, the majority of which target systems with finite state-spaces. The POR concept was independently studied by Valmari [Val90], Peled [Pel93], and Godefroid [God96]. Each of them introduced their own, albeit similar, variant of ample sets — called stubborn, ample and, respectively, persistent.

On the theoretical side, Mazurkiewicz traces [Maz86] are the algebraic concept to underpin both POR as well as Shasha and Snir traces [SS88]. On the practical side, Peled showed that POR can be combined with model checking $\mathrm{LTL}_{-\mathcal{X}}$ on-the-fly [Pel96] while Flanagan and Godefroid came up with a way to statelessly explore a concurrent program's reduced state-space dynamically [FG05]. More recently, dynamic POR has been enhanced to explore exactly one interleaving trace and never initiate sleep-set blocked exploration in [Abd+14] and, in [Abd+15], it has been generalized to TSO. Concretely, [Abd+15] implements dynamic POR using a scheduler for the depth-first traversal of the finite state-space of bounded programs specified through LLVM IR [LA04]. In other work, systems with dynamically allocated resources [KR08], bounded model checking [MP09; KWG09; CMM13], as well as concolic testing [SKH12], are combined with dynamic POR.

In other POR-related research, stateful POR exploration is claimed to be comparably as effective as dynamic POR [YWY06; Yan+08], POR is developed through compositional confluence detection (à la Milner [Mil89]), and empirical bounds for static and dynamic POR using different dependency approximations are found [GHV09]. Yet more recent works discuss minimality of stubborn sets when checking deadlocks [VH10] as well as statically finding stubborn sets heuristically (using guard-based necessary enabling/disabling sets of transitions) [Laa+13]. Further results target the efficient implementation of condition (C3) in Figure 2.13 (page 30) [BLL06; BBR10] — the so called *cycle proviso* — while other results [Sie12] seek to relax the stuttering restriction (C2). Finally, in [Gue+07] Gueta et al. propose an alternative dynamic POR method which they call cartesian POR. Their approach uses *cartesian vectors* for states in order to determine step-wise longest transition sequences that threads can perform without context switches. Intuitively, this yields an iterative batch state-space exploration where only the last per-thread transitions may be dependent.

4.1 The Persistent Set Perspective

As described in Section 2.2.2, Partial Order Reduction exploits action independence. Similarly to the model checking setting, events $\mathtt{e}_1$ and $\mathtt{e}_2$ that are enabled in some state $s \in S_\mathrm{M}$ are *independent* if both enabledness and commutativity hold. Formally, $\mathtt{e}_1$ and $\mathtt{e}_2$ such that $s \xrightarrow{\mathtt{e}_1} s_1$ and $s \xrightarrow{\mathtt{e}_2} s_2$ are independent iff $\mathtt{e}_2 \in \mathit{enabled}(s_1)$, $\mathtt{e}_1 \in \mathit{enabled}(s_2)$, and there exists $s' \in S_\mathrm{M}$ such that $s_1 \xrightarrow{\mathtt{e}_2} s'$ and $s_2 \xrightarrow{\mathtt{e}_1} s'$. By the symmetric anti-reflexive relation $\mathcal{I} \subseteq E \times E$ consisting of independent events in a program's semantics we denote the program's independence relation while by its complement set $\mathcal{D} := (E \times E) \setminus \mathcal{I}$ we denote the program's dependence relation.

Due to the systems' complexity, checking (in)dependence is difficult. For example, a sufficient syntactic condition for events $\mathtt{e}_1, \mathtt{e}_2 \in \mathit{enabled}(s)$ to be independent is that $\mathit{thread}(\mathtt{e}_1) \neq \mathit{thread}(\mathtt{e}_2)$ and $\mathit{addr}(\mathtt{e}_1) \neq \mathit{addr}(\mathtt{e}_2)$. This condition corresponds to the one in [God96] that requires disjoint active processes and disjoint accessed objects. However, it does not yield a criteria to effectively check (in)dependence for *any* two events $\mathtt{e}_1, \mathtt{e}_2 \in \mathit{enabled}(s)$.

Turning to persistence, a set $\mathcal{S}$ of events — all of which are enabled in state s — is *persistent in* s iff, for all execution fragments

$$s_1 \xrightarrow{\mathtt{e}_1} \dots \xrightarrow{\mathtt{e}_{n-1}} s_n \xrightarrow{\mathtt{e}_n} s'$$

starting from $s_1 := s$ and staying outside of $\mathcal{S}$ up to s_n — i.e., $\mathtt{e}_i \notin \mathcal{S}$ for all $i \in [1..n-1]$ — it holds that $\mathtt{e}_n$ is independent from all events $\mathtt{e} \in \mathcal{S}$.

Note that the above definition of persistent sets implies that if some set $\mathcal{S}$ is persistent in state s then all execution fragments starting in s and using non-$\mathcal{S}$ events contain only events independent from any $\mathtt{e} \in \mathcal{S}$.

Algorithmically computing persistent sets is not an easy task. More precisely, there is a tradeoff between computing small persistence sets (and thus getting more reduction) and the time required for the finer dependence analysis. For example, the simplest algorithm to compute a persistent set (in s) from [God96] has worst-case time complexity $\mathcal{O}(|\mathit{enabled}(s)|^2)$ and performs the following steps:

1. Take an event $\mathtt{e} \in \mathit{enabled}(\mathtt{e})$ and let $\mathcal{S} := \{\mathtt{e}\}$.
2. For all events $\mathtt{e} \in \mathcal{S}$ and $\mathtt{e}' \in \mathit{enabled}(s)$, add $\mathtt{e}'$ to $\mathcal{S}$ if
   (a) $\mathit{thread}(\mathtt{e}) = \mathit{thread}(\mathtt{e}')$ or
   (b) $\mathit{thread}(\mathtt{e}) \neq \mathit{thread}(\mathtt{e}')$ and there exists some state $s' \in S_\mathrm{M}$ such that $\mathtt{e}$ and $\mathtt{e}'$ are dependent in s'.

3. Repeat step 2 until an initially-disabled event is introduced in $\mathcal{S}$ or until fixpoint. If an initially-disabled event is added to $\mathcal{S}$ then the set $\mathit{enabled}(s)$ is returned.

Applied to the relaxed memory semantics, the above algorithm computes a subset $\mathcal{S} \subseteq \mathit{enabled}(s)$ such that no event $\mathtt{e} \in \mathit{enabled}(s) \setminus \mathcal{S}$ belongs to the same thread as any of the $\mathcal{S}$ events and no event $\mathtt{e} \in \mathit{enabled}(s) \setminus \mathcal{S}$ accesses the same address as any of the $\mathcal{S}$ events.

In fact, due to the relaxed memory semantics, $\mathtt{e}, \mathtt{e}' \in \mathit{enabled}(s)$ are either dependent or independent for all S_{M} states and it is not possible that an initially-disabled event is introduced in $\mathcal{S}$ by the algorithm's step 2. This simplifies steps 2.b and 3 of the previous algorithm from [God96].

To summarize sufficient conditions describing the independence relation $\mathcal{I}$ for $\mathrm{X}_{\mathrm{M}}(\mathcal{P})$, let events $\mathtt{e}, \mathtt{e}' \in \mathit{enabled}(s)$ such that $\mathit{cmd} := \mathit{cmd}(\mathit{inst}(\mathtt{e}))$ and $\mathit{cmd}' := \mathit{cmd}(\mathit{inst}(\mathtt{e}'))$. The events $\mathtt{e}$ and $\mathtt{e}'$ are independent if

(1) both *cmd* and *cmd'* are either a store corresponding to a buffering event, a memory fence, a conditional, or an assignment — while not concomitantly being stores of buffering events in the same thread;

(2) one of *cmd* and *cmd'* is a load $r \leftarrow \mathtt{mem}[e]$ and the other is either

- a store corresponding to a buffering event such that if $\mathit{thread}(\mathtt{e}) = \mathit{thread}(\mathtt{e}')$ then also $\mathit{addr}(\mathtt{e}) \neq \mathit{addr}(\mathtt{e}')$,
- a memory fence,
- a conditional $\mathtt{check}\ e'$ such that e' does not depend on r,
- an assignment $r' \leftarrow e'$ such that e does not depend on r' and also e' does not depend on r,
- a load $r' \leftarrow \mathtt{mem}[e']$ such that e does not depend on r' and also e' does not depend on r,
- a store corresponding to a flush event with $\mathit{addr}(\mathtt{e}) \neq \mathit{addr}(\mathtt{e}')$;

(3) one of *cmd* and *cmd'* is a store corresponding to a flush event and the other one is either

- a store corresponding to a buffering event,
- a memory fence (in some other thread),
- a conditional check,
- an assignment,
- a load such that $\mathit{addr}(\mathtt{e}) \neq \mathit{addr}(\mathtt{e}')$,
- a store corresponding to a flush event with $\mathit{addr}(\mathtt{e}) \neq \mathit{addr}(\mathtt{e}')$.

The above conditions are not precise because they do not account for identically overwritten/re-read values. E.g., two buffer (respectively flush) events to the same address are independent if they buffer (respectively flush) identical values. Similarly, two load events with, e.g., $\mathit{cmd} = r \leftarrow \mathtt{mem}[r + r']$ and $\mathit{cmd}' = r' \leftarrow \mathtt{mem}[r + r']$ are independent if $r = r' = 0$ in state s.

In the context of TSO, we call *local* all the events that are executing a memory fence, a conditional, an assignment, or that are buffering a store. These events are local in the sense that their execution does not imply a direct flow of address valuations. More precisely, while memory fences, conditionals and assignments do not involve addresses at all, buffering stores represent only an intermediary step when updating some addresses' value. In the context of SC, only memory fences, conditionals and assignments are *local* since stores update addresses atomically.

A straightforward algorithm[2] that computes persistent sets can, hence, perform the following steps:

1. search for a thread t such that $\mathit{enabled}(s, t)$ — i.e., the set of events $\mathtt{e} \in \mathit{enabled}(s)$ with $t = \mathit{thread}(\mathtt{e})$ — consists of only local events;
2. if such a thread exists return $\mathit{enabled}(s, t)$; otherwise return $\mathit{enabled}(s)$.

The worst-case time complexity of the above algorithm is $\mathcal{O}(|\mathit{enabled}(s)|)$. Its correctness is immediate from the earlier conditions (1)–(3). Indeed, if the algorithm returns $\mathcal{S} = \mathit{enabled}(s, t)$ for some thread t then all events in $\mathcal{S}$ are independent from all other threads' events in all execution fragments

$$s_1 \xrightarrow{\mathtt{e}_1} \dots \xrightarrow{\mathtt{e}_{n-1}} s_n \xrightarrow{\mathtt{e}_n} s'$$

starting from $s_1 := s$ and staying outside of $\mathcal{S}$. Intuitively, this holds since $\mathcal{S}$ contains only local events of thread t. Formally, one can prove that all non-$\mathcal{S}$ events $\mathtt{e}_1, \dots, \mathtt{e}_{n-1}, \mathtt{e}_n$ are independent from any $\mathtt{e} \in \mathcal{S}$ by induction over the length of the execution fragment. A different proof approach involving stubborn sets can be consulted in [Lin14].

However, it is actually local events and not so much persistent sets that drive a local-events-based partial order reduction. Consider, for example, the local-events-first reduction implemented under SC in TRENCHER [Der15] (through which the DFS exploration of the state-space follows local events of a thread for as long as possible). Both such a local-events-first reduced exploration as well as an exploration that uses persistent sets computed by the above algorithm have one thing in common. Namely, they aim to explore a smaller number of interleavings of local events from different threads.

In the following, we will describe a systematic way to achieve state-space reduction for ASSEMBLY programs that generalizes local-events-based POR (as used by the methods mentioned above) as well as other known approaches such as dynamic and, respectively, cartesian partial order reduction.

[2]This algorithm restates Algorithm 12 from Linden's PhD thesis [Lin14].

$$\begin{tikzcd}
\texttt{store}_1 \arrow[d, "fo"] \arrow[dd, bend right, "po"'] & & \texttt{store}_2 \arrow[d, "fo"'] \arrow[dd, bend left, "po"] \\
\texttt{flush}_1 & & \texttt{flush}_2 \\
\texttt{load}_1 \arrow[urr, "cf"] & & \texttt{load}_2 \arrow[ull, "cf"']
\end{tikzcd}$$

Figure 4.1: Trace $Tr(\sigma)$ for the longest strict prefix of (page 37) computation τ_{wit}. Trace $Tr(\tau_{\text{wit}})$ is obtained by including the light-colored event and relations.

4.2 Traces for Partial Order Reduction

As seen in Section 2.2.2 and in Section 4.1, existing POR techniques rely on an a priori defined valid dependency relation between computation events. Here, we formalize these dependencies for ASSEMBLY programs under TSO using a *trace* relation [Maz86; SS88]. More precisely, for any prefix σ of some computation $\tau \in \mathcal{C}_{\text{TSO}}(\mathcal{P})$, we define the *trace of* σ as the union of the three relations that we describe below, $Tr(\sigma) := \rightarrow_{po} \cup \rightarrow_{fo} \cup \rightarrow_{cf}$.

The *program order relation* $\rightarrow_{po}$ is the union of the per-thread program order relations: $\rightarrow_{po} := \bigcup_{t \in \mathsf{TID}} \rightarrow_{po}(t, \sigma)$. If σ' is the subsequence of all non-flush events of thread t in the prefix σ then $\rightarrow_{po}(t, \sigma) := <_{\sigma'}$.

The *follow order relation* $\rightarrow_{fo}$ links, per-thread, matching store and flush events: $(t, inst, a) \rightarrow_{fo} (t, \text{flush}, a)$. How this can be technically achieved is described in Appendix C.

The *conflict relation* $\rightarrow_{cf}$ orders accesses to the same address. On the one hand, assume $\sigma = \tau_1 \cdot \texttt{store} \cdot \tau_2 \cdot \texttt{load} \cdot \tau_3 \cdot \texttt{flush} \cdot \tau_4$ such that $\texttt{store} \rightarrow_{fo} \texttt{flush}$, events $\texttt{store}$ and $\texttt{load}$ access the same address a and come from the same thread t, and there is no other store event $\texttt{store}' \in \tau_2$ such that $thread(\texttt{store}') = t$ and $addr(\texttt{store}') = a$. We then say that the event $\texttt{load}$ is an *early read* of the value buffered by $\texttt{store}$ and $\texttt{store} \rightarrow_{cf} \texttt{load}$.

On the other hand, assume $\sigma = \tau_1 \cdot \texttt{e} \cdot \tau_2 \cdot \texttt{e}' \cdot \tau_3$ such that $\texttt{e}$ and $\texttt{e}'$ are either load or flush events that access address a, neither $\texttt{e}$ nor $\texttt{e}'$ is an early read, and at least one of $\texttt{e}$ or $\texttt{e}'$ is a flush to a. If there is no flush event $\texttt{flush} \in \tau_2$ such that $addr(\texttt{flush}) = a$ then $\texttt{e} \rightarrow_{cf} \texttt{e}'$.

To give an example, Figure 4.1 depicts the trace of the longest (strict) prefix of computation τ_{wit} introduced on page 37.

Compared to the happens-before relation on page 40 — which in turn adapts Shasha and Snir traces [SS88] — the follow order relation $\rightarrow_{fo}$ is unidirectional instead of bidirectional like the equivalence $\leftrightarrow$. Furthermore, compared with both the happens-before relation on page 40 as well as with Alglave's definition of global happens-before [Alg10], we define traces for computation prefixes instead of for computations. Finally, the chronological traces in [Abd+15] restrict our trace definition such that every subgraph

$$\begin{tikzcd}
\texttt{store} \arrow[rr, "fo"] & & \texttt{flush} \arrow[rr, "cf"] & & \texttt{load} \arrow[dl, "cf"'] \arrow[dr, "po"] & \\
& \mathsf{e}_1 \arrow[ur, "cf"] & & \mathsf{e}_2 & & \mathsf{e}_3
\end{tikzcd}$$

with same-thread events **`store`**, **`flush`**, and **`load`**, is replaced by

$$\begin{tikzcd}
\texttt{store} \arrow[rr, "fo"] \arrow[rrrr, bend left, "cf"] & & \texttt{flush} \arrow[d, "cf"] & & \texttt{load} \arrow[d, "po"] \\
& \mathsf{e}_1 \arrow[ur, "cf"] & \mathsf{e}_2 & & \mathsf{e}_3
\end{tikzcd}$$

In the following, we will use ε to denote the trace of the computation prefix containing no events and $\mathit{Traces}_{\mathrm{M}}(\mathcal{P})$ to denote the set of traces of prefixes of computations in $\mathcal{C}_{\mathrm{M}}(\mathcal{P})$.

Similarly to Lemma 4 in Section 2.3.3 for the happens-before relation, we prove in Lemma 18 that any two computation prefixes with the same trace reach the same state. Using Lemma 18 one could then prove that every linearization of a trace in $\mathit{Traces}_{\mathrm{M}}(\mathcal{P})$ is a $\mathcal{C}_{\mathrm{M}}(\mathcal{P})$ computation prefix that reaches the same state in S_{M}.

Lemma 18. *Let $Tr(\sigma), Tr(\sigma') \in \mathit{Traces}_M(\mathcal{P})$ such that $Tr(\sigma) = Tr(\sigma')$. If $s_0 \xrightarrow{\sigma} s \in \Delta^*_{X,M}$ then also $s_0 \xrightarrow{\sigma'} s \in \Delta^*_{X,M}$.*

Proof. Assume $s_0 \xrightarrow{\sigma} s \in \Delta^*_{\mathrm{X,M}}$ and $s_0 \xrightarrow{\sigma'} s' \in \Delta^*_{\mathrm{X,M}}$. Since σ and σ' have the same program order $\rightarrow_{po}$, s and s' have the same program counter pc. Moreover, since σ and σ' have the same conflict order $\rightarrow_{cf}$, s and s' have the same memory valuation val. Finally, since computation prefixes σ and σ' have the same $\rightarrow_{fo}$ unmatched store events and $\rightarrow_{po}$ orders these store events the same, s and s' have the same buffer contents. Hence, $s = s'$. $\square$

Given a program $\mathcal{P}$ and a memory model $\mathrm{M} \in \{\mathrm{TSO}, \mathrm{SC}\}$ we define the transition semantics that includes trace information as the *state-space automaton* $\mathrm{Y}_{\mathrm{M}}(\mathcal{P}) := (E, S_{\mathrm{M}} \times \mathit{Traces}_{\mathrm{M}}(\mathcal{P}), \Delta_{\mathrm{Y,M}}, (s_0, \varepsilon))$. The transition relation $\Delta_{\mathrm{Y,M}}$ relies on $\Delta_{\mathrm{X,M}}$ described on page 35 of Section 2.3.1 and is defined by the following constraint:

$$(s, tr) \xrightarrow{\mathsf{e}} (s', tr') \in \Delta_{\mathrm{Y,M}} \text{ iff there exists a computation prefix } \sigma \text{ such that}$$
$$Tr(\sigma) = tr,\ \ Tr(\sigma \cdot \mathsf{e}) = tr', \text{ and } s_0 \xrightarrow{\sigma} s \xrightarrow{\mathsf{e}} s' \in \Delta^*_{\mathrm{X,M}}.$$

Notice that, for general programs, $\mathrm{Y}_{\mathrm{M}}(\mathcal{P})$ may be infinite already if $\mathrm{M} = \mathrm{SC}$. This, indeed, depends on whether $\mathit{Traces}_{\mathrm{M}}(\mathcal{P})$ is infinite or not.

Algorithm 4.1 depicts a state exploration algorithm that uses $\mathrm{Y_M}(\mathcal{P})$ to solve M reachability. It is a decision procedure iff a fixpoint set *visited* $\subseteq S_{\mathrm{M}}$ is always reached. This is the case if the state-space S_{M} of $\mathrm{X_M}(\mathcal{P})$ is finite. Such a situation arises, e.g., if M = TSO and the input program is acyclic or if M = SC. On the other hand, if S_{M} is infinite, like for some cyclic programs under M = TSO, the algorithm can be a decision procedure for positive cases of M reachability. However, as we will explain later in more detail, whether Algorithm 4.1 is a decision procedure for positive M reachability cases depends on choosing a good exploration heuristics.

Algorithm 4.1 Explicit-state trace-based M reachability checker

```
Input: Memory model M, marked program $\mathcal{P}$ and state $(s, tr)$ of $\mathrm{Y_M}(\mathcal{P})$
Output: $true$ if some goal state is reachable from $(s, tr)$ in $\mathcal{P}$
        $false$ if no goal state is reachable from $(s, tr)$ in $\mathcal{P}$
Global Variable: $visited \subseteq S_{\mathrm{M}}$, initially $visited = \emptyset$

 1: procedure ExplicitReach(M, $\mathcal{P}$, $s$, $tr$)
 2:     if $s \notin visited$ then    // check that we explore a new state in $\mathrm{X_M}(\mathcal{P})$
 3:         if $s \in G$ then
 4:             return $true$;
 5:         end if
 6:         add $s$ to $visited$;
 7:         for all $\mathsf{e} \in$ NextEvents(M, $\mathcal{P}$, $s$, $tr$) do
 8:             let $(s', tr')$ such that $(s, tr) \xrightarrow{\mathsf{e}} (s', tr') \in \Delta_{\mathrm{Y,M}}$ in
 9:             if ExplicitReach(M, $\mathcal{P}$, $s'$, $tr'$) then
10:                 return $true$;
11:             end if
12:         end for
13:     end if
14:     return $false$;
15: end procedure
```

Different exploration heuristics for Algorithm 4.1 depend on the choice of the NextEvents procedure used in Line 7. Furthermore, different strategies for the Line 7 loop (e.g. depth- or breadth-first) give way to different implementations of the algorithm. Note that the *most general heuristics* for NextEvents(s, tr) returns all events e such that $s \xrightarrow{\mathsf{e}} s' \in \Delta_{\mathrm{X,M}}$. This corresponds to a full state-space exploration (similar to the exploration in Algorithms 2.3 and 2.4 from Section 2.3.2).

We will say that Algorithm 4.1 *explores* some computation prefix σ if the stack of events during recursive calls ever amounts to σ. Furthermore, we say that Algorithm 4.1 *uses* Partial Order Reduction for $\mathcal{P}$ iff at least sometime during the recursive calls within ExplicitReach$(\mathrm{M}, \mathcal{P}, s_0, \varepsilon)$ the procedure NextEvents$(\mathrm{M}, \mathcal{P}, s, tr)$ returns a strict subset of all events e

such that $s \xrightarrow{\mathsf{e}} s' \in \Delta_{\mathrm{X,M}}$. Furthermore, we say that Algorithm 4.1 uses *state-reducing* POR for $\mathcal{P}$ if, upon calling ExplicitReach$(\mathrm{M}, \mathcal{P}, s_0, \varepsilon)$, for all $(s, tr) \in S_{\mathrm{M}} \times \mathit{Traces}_{\mathrm{M}}(\mathcal{P})$ and for at least some state $s' \in S_{\mathrm{M}}$ such that $s \xrightarrow{\mathsf{e}} s' \in \Delta_{\mathrm{X,M}}$, NextEvents$(\mathrm{M}, \mathcal{P}, s, tr)$ never returns the event $\mathsf{e} \in E$ leading to s'. Intuitively, the above conditions are necessary for NextEvents to implement POR exploration.

Sufficient conditions for NextEvents to implement *trace-optimal* POR exploration (in the sense that no two computation prefixes with the same trace are explored) can be explained as follows. First, NextEvents should implement state-space exploration, meaning, NextEvents$(\mathrm{M}, \mathcal{P}, s, tr) \subseteq \mathit{enabled}(s)$ for all states s — whether Algorithm 4.1 uses POR or state-reducing POR is not relevant. Second, and more importantly, the procedure NextEvents must constrain ExplicitReach$(\mathrm{M}, \mathcal{P}, s_0, \varepsilon)$ to not (recursively) call ExplicitReach twice with the same input. The latter condition implies that Algorithm 4.1 never explores prefixes $\sigma \neq \sigma'$ with $\mathit{Tr}(\sigma) = \mathit{Tr}(\sigma')$. We prove this claim in Theorem 19 below.

Theorem 19. *If* ExplicitReach$(M, \mathcal{P}, s_0, \varepsilon)$ *does not recursively call itself with the same input twice then* $\mathit{Tr}(\sigma) \neq \mathit{Tr}(\sigma')$ *for all explored* $\sigma \neq \sigma'$.

Proof. Assume that ExplicitReach$(\mathrm{M}, \mathcal{P}, s_0, \varepsilon)$ explores two computation prefixes $\sigma \neq \sigma'$ such that $tr := \mathit{Tr}(\sigma) = \mathit{Tr}(\sigma')$ even tough ExplicitReach is not called twice with the same input. Wlog, assume that the prefix σ is explored first, i.e., the last event of σ is considered in Line 7 and ExplicitReach$(\mathrm{M}, \mathcal{P}, s, tr)$ is called in Line 9 for some state $s \in S_{\mathrm{M}}$ such that $s_0 \xrightarrow{\sigma} s \in \Delta^*_{\mathrm{X,M}}$.

If the prefix σ' is explored as well then its last event is also considered in Line 7 of some recursive call of ExplicitReach$(\mathrm{M}, \mathcal{P}, s_0, \varepsilon)$. As before, ExplicitReach$(\mathrm{M}, \mathcal{P}, s', tr)$ is then called in Line 9 for some state $s' \in S_{\mathrm{M}}$ such that $s_0 \xrightarrow{\sigma'} s' \in \Delta^*_{\mathrm{X,M}}$.

However, by Lemma 18, $s = s'$ yields a contradiction to ExplicitReach not having been called twice with the same input. □

While Theorem 19 describes a general measure for Algorithm 4.1 to be trace-optimal, it addresses neither its correctness nor how to implement it.

In the following subsection we explain why Algorithm 4.1 is sound for positive instances of reachability — i.e., if the algorithm returns *true* then some goal state is M-reachable — as well as describe sufficient conditions on NextEvents such that the algorithm is complete for positive instances of reachability — i.e., if some goal state is M-reachable then the algorithm does not return *false*.

4.2.1 Soundness and Completeness

Regardless of the heuristics that one uses for NextEvents, the procedure ExplicitReach in Algorithm 4.1 is a sound under-approximation for state-based reachability under $\mathrm{M} \in \{\mathrm{SC}, \mathrm{TSO}\}$. Furthermore, if a heuristics for NextEvents is *safe* then ExplicitReach is precise enough to cover the program's reachable states. Turning back to soundness, we show that some goal state is M-reachable in $\mathcal{P}$ when ExplicitReach returns *true*.

Lemma 20. *If* ExplicitReach$(M, \mathcal{P}, s_0, \varepsilon)$ *returns true then some goal state is M-reachable in* $\mathcal{P}$*.*

Proof. Assume that Algorithm 4.1 returns true. Let (s, tr) be the parameters of ExplicitReach when the Line 3 check first succeeds, i.e. $s \in G$.

When this happens, the recursive calls of ExplicitReach explored a computation prefix σ such that $(s_0, \varepsilon) \xrightarrow{\sigma} (s, tr) \in \Delta^*_{\mathrm{Y,M}}$. This additionally means that $s_0 \xrightarrow{\sigma} s \in \Delta^*_{\mathrm{X,M}}$.

Now, let the event sequence $\tau \in E^*$ flush the buffer contents of state s (in some arbitrary thread-interleaving order). Then $s_0 \xrightarrow{\sigma \cdot \tau} s' \in \Delta^*_{\mathrm{X,M}}$ and $s' \in \mathit{Reach}_{\mathrm{M}}(\mathcal{P}) \cap G$. ☐

Furthermore (and similarly to Algorithms 2.3 and 2.4), if the state-space S_{M} of program $\mathcal{P}$ is finite then Algorithm 4.1 is a decision procedure. Note that this holds even if $\mathit{Traces}_{\mathrm{M}}(\mathcal{P})$ is infinite.

Lemma 21. *If the state-space S_M of $X_M(\mathcal{P})$ is finite then Algorithm 4.1 always terminates with the correct answer.*

Proof. By Lemma 20 we know that if Algorithm 4.1 returns *true* then there exists some goal state that is M-reachable in $\mathcal{P}$. Therefore, assume that ExplicitReach$(\mathrm{M}, \mathcal{P}, s_0, \varepsilon)$ does not return *true*. Note that this implies none of the ExplicitReach recursive calls returns *true* either.

Because of the Line 2 check, we know that every successful recursion of the algorithm adds a state to the set *visited* at Line 6. Therefore, since $\mathit{visited} \subseteq S_{\mathrm{M}}$ and S_{M} is finite, if no ExplicitReach recursive call returns *true* (according to our assumption) then *visited* will reach a fixpoint. Hence, ExplicitReach$(\mathrm{M}, \mathcal{P}, s_0, \varepsilon)$ terminates and returns *false*. ☐

Correctness of Algorithm 4.1 depends on which NextEvents heuristics is chosen in the implementation. Lemma 20 shows that if Algorithm 4.1 returns *true* then it soundly does so. However, in order for the algorithm to always return correct answers the NextEvents calls must not throw away all events that might be relevant to reaching goal states. In order to formalize the latter, we have to describe what it means for a certain heuristics to be "safe".

Recall that given memory model M, the M reachability problem expects as input a program P and a set of goal states $G \subseteq S_{\mathrm{M}}$. For simplicity, we previously assumed that goal states are signaled by marked states in each of the threads. Technically however, when checking M reachability we would use two dedicated addresses $a_{\mathrm{goal}}, a_{\mathrm{success}} \in \mathsf{ADR}$ such that, out of every marked control state $q_{\mathrm{m},t}$ of thread t, transitions are added that set a_{success} to 1 if the marked states are simultaneously reachable in all threads.

Concretely, by adding the Assembly instructions below to each thread the address a_{goal} would be used to count how many threads reached their marked control states and when all threads successfully reach their marked control states, i.e., when $\texttt{mem}[a_{\mathrm{goal}}] = |\mathsf{TID}|$, a_{success} would be set to 1. The atomic instructions used here are a natural extension of the $\mathrm{X}_{\mathrm{M}}(\mathcal{P})$ semantics and guarantee that their internal commands are executed as if running on SC and in one shot, without any intermediary interleaving. More details concerning atomic instruction can be consulted in Appendix C.

$$\begin{tikzcd} q_{\mathrm{m},t} \arrow[d, "\texttt{atomic}\,\{\, r \leftarrow \texttt{mem}[a_{\mathrm{goal}}],\ \texttt{mem}[a_{\mathrm{goal}}] \leftarrow r+1 \,\}"] \\ q_{\mathrm{sink},t} \arrow[loop below, "\texttt{atomic}\,\{\, r \leftarrow \texttt{mem}[a_{\mathrm{goal}}],\ \texttt{check}\ r = |\mathsf{TID}|,\ \texttt{mem}[a_{\mathrm{success}}] \leftarrow 1 \,\}"] \end{tikzcd}$$

Using the above modeling artifact, we can consider goal states to be those states $(\mathsf{pc}, \mathsf{val}, \mathsf{buf})$ in S_{M} for which $\mathsf{val}(a_{\mathrm{success}}) = 1$. As before, the M reachability problem decides whether $\mathcal{L}_{F\cap G}(\mathrm{X}_{\mathrm{M}}(\mathcal{P})) \neq \emptyset$.

To give an example extending τ_{wit} on page 37, the computation

$$\tau_{\mathrm{reach}} := \tau_{\mathrm{wit}} \cdot \texttt{atomic}_1 \cdot \texttt{atomic}_2 \cdot \texttt{atomic}, \tag{4.1}$$

reaches under TSO the goal state $(\mathsf{pc}, \mathsf{val}, \mathsf{buf})$ where $\mathsf{pc} = (q_{\mathrm{sink},1}, q_{\mathrm{sink},2})$ and $\mathsf{val}(a_{\mathrm{success}}) = 1$. Here, by $\texttt{atomic}$ we mean the events produced by one of the two instructions

$$q_{\mathrm{sink},t} \xrightarrow{\texttt{atomic}\,\{\, r \leftarrow \texttt{mem}[a_{\mathrm{goal}}],\, \texttt{check}\ r = |\mathsf{TID}|,\, \texttt{mem}[a_{\mathrm{success}}] \leftarrow 1 \,\}} q_{\mathrm{sink},t}$$

that sets the value of address a_{success} to 1, by $\texttt{atomic}_1$ the events produced by instruction $(q_{\mathrm{m},1},\ \texttt{atomic}\,\{\, r \leftarrow \texttt{mem}[a_{\mathrm{goal}}],\ \texttt{mem}[a_{\mathrm{goal}}] \leftarrow r+1 \,\},\ q_{\mathrm{sink},1})$ and, by $\texttt{atomic}_2$ the (thread t_2) symmetric events.

Now, we say that the procedure NextEvents provides a *safe heuristics* for Algorithm 4.1 if, whenever ExplicitReach$(\mathrm{M}, \mathcal{P}, s_0, \varepsilon)$ returns *false*, ExplicitReach$(\mathrm{M}, \mathcal{P}, s_0, \varepsilon)$ explores at least one computation prefix σ such that $s_0 \xrightarrow{\sigma} s \in \Delta^*_{\mathrm{X},\mathrm{M}}$ for all possible memory valuations $\mathsf{val}{\downarrow}(\mathsf{ADR} \to \mathsf{DOM})$ of states $s = (\mathsf{pc}, \mathsf{val}, \mathsf{buf})$ reachable in $\mathrm{X}_{\mathrm{M}}(\mathcal{P})$. Intuitively, since checking reachability is encoded by setting a_{success} to 1, to guarantee correctness it suffices to explore computation prefixes leading to states in $\mathrm{X}_{\mathrm{M}}(\mathcal{P})$ that have different memory valuations $\mathsf{val}{\downarrow}(\mathsf{ADR} \to \mathsf{DOM})$.

To sum up, we prove that if NextEvents implements a safe heuristics then ExplicitReach$(\mathrm{M}, \mathcal{P}, s_0, \varepsilon)$ always returns correct results.

Theorem 22. *If the procedure* NextEvents *implements a safe heuristics and* ExplicitReach *terminates on input* $(M, \mathcal{P}, s_0, \varepsilon)$ *then some goal state of* $\mathcal{P}$ *is M-reachable iff* ExplicitReach$(M, \mathcal{P}, s_0, \varepsilon)$ *returns true.*

Proof. The easy implication holds by Lemma 20. For the reverse direction we do a proof by contradiction.

Assume that $s \in \mathit{Reach}_{\mathrm{M}}(\mathcal{P}) \cap G$ and that ExplicitReach$(\mathrm{M}, \mathcal{P}, s_0, \varepsilon)$ returns *false*. Since `NextEvents` implements a safe heuristics it means the algorithm explored some computation $\tau \in \mathcal{C}_{\mathrm{M}}(\mathcal{P})$ that reaches some state $s = (\mathsf{pc}, \mathsf{val}, \mathsf{buf})$ with $\mathsf{val}(a_{\text{success}}) = 1$ in $\mathrm{X}_{\mathrm{M}}(\mathcal{P})$.

Hence, the algorithm recursively called ExplicitReach$(\mathrm{M}, \mathcal{P}, s, \mathit{Tr}(\tau))$ in Line 9. Furthermore, the (first) recursive call ExplicitReach$(\mathrm{M}, \mathcal{P}, s, *)$ returns *true* since the Line 3 check succeeds and this gets propagated back to the initial recursive call.

Thus, the assumption that ExplicitReach$(\mathrm{M}, \mathcal{P}, s_0, \varepsilon)$ returns *false* was incorrect. □

Note that, as it stands, Theorem 22 has to assume termination for the TSO memory model (Lemma 21 guarantees termination for SC). Indeed, when the state-space S_{TSO} of $\mathrm{X}_{\mathrm{TSO}}(\mathcal{P})$ is not finite, both a depth-first and a breadth-first implementation of the Line 7 loop in Algorithm 4.1 may not suffice. Nevertheless, safe heuristics are an essential building block to guarantee correctness of ExplicitReach$(\mathrm{M}, \mathcal{P}, s_0, \varepsilon)$ when termination is assumed/given. As we will see in Section 4.3, there exist various implementations of safe POR exploration heuristics.

4.3 POR Techniques explained by Traces

Prior to describing dynamic and cartesian POR, we explain the ideas behind the local-events-first heuristics. This lightweight POR-inducing heuristics was implemented — for the SC memory model — in Trencher and adheres to the following schema:

(1) if some local event is explored out of some state $s \in S_{\mathrm{M}}$ then the thread of that event becomes the *favorite* thread;

(2) as long as the favorite thread for the exploration is set, only events of the favorite thread can be explored;

(3) if the favorite thread is set and a non-local event of this thread is explored then the favorite thread is reset to $\perp$.

Recall that, under TSO, local events are those events that are neither load nor flush events. Under SC, since stores are immediately followed by the matching flush events, only events for assignments, conditionals, and memory fences are local.

Algorithm 4.2 implements the local-events-first heuristics by refining the ExplicitReach procedure in Algorithm 4.1. Differences from the generic Algorithm 4.1 on page 70 are highlighted.

Algorithm 4.2 Refined explicit-state trace-based M reachability checker

```
**Input**: Memory model M, marked program $\mathcal{P}$ and state $(s, tr)$ of $\mathrm{Y_M}(\mathcal{P})$
**Output**: $true$ if some goal state is reachable from $(s, tr)$ in $\mathcal{P}$
            $false$ if no goal state is reachable from $(s, tr)$ in $\mathcal{P}$
**Global Variable**: $visited \subseteq S_\mathrm{M}$, initially $visited = \emptyset$

 1: procedure ExplicitReach($\mathrm{M}, \mathcal{P}, s, tr, favorite$)
 2:     if $s \notin visited$ then    // check that we explore a new state in $\mathrm{X_M}(\mathcal{P})$
 3:         if $s \in G$ then
 4:             return $true$;
 5:         end if
 6:         add $s$ to $visited$;
 7:         for all $\mathsf{e} \in \textsc{NextEvents}_{\textsc{local-first}}(\mathrm{M}, \mathcal{P}, s, tr, favorite)$ do
 8:             let $(s', tr')$ such that $(s, tr) \xrightarrow{\mathsf{e}} (s', tr') \in \Delta_{\mathrm{Y,M}}$ in
 9:             if $\mathsf{e}$ is a local event then
10:                 $favorite' := thread(\mathsf{e})$;
11:             else
12:                 $favorite' := \perp$;
13:             end if
14:             if ExplicitReach($\mathrm{M}, \mathcal{P}, s', tr', favorite'$) then
15:                 return $true$;
16:             end if
17:         end for
18:     end if
19:     return $false$;
20: end procedure
```

Algorithm 4.3 shows the procedure for computing next events using the local-events-first heuristics. Concretely, $\textsc{NextEvents}_{\textsc{local-first}}$ returns all events enabled in state s if $favorite = \perp$. If an event e out of these is local then the highlighted Line 9–13 instructions in Algorithm 4.2 restrict computation continuations following event e to explore within thread $thread(\mathsf{e})$. On the other hand, if $favorite \neq \perp$ then $\textsc{NextEvents}_{\textsc{local-first}}$ returns all events of thread $favorite$ enabled in state s. If an event e out of these is non-local then the highlighted Line 9–13 instructions in Algorithm 4.2 reset $favorite$ to $\perp$ and, hence, allow further continuations of the exploration by other threads. These principles behind local-events-first heuristics guarantee that $\textsc{NextEvents}_{\textsc{local-first}}$ provides a safe heuristics for Algorithm 4.2. Then, since Algorithm 4.2 is essentially Algorithm 4.1 up to using *favorite*, we know that Theorem 22 applies.

Corollary 23. *The function* $\textsc{NextEvents}_{\textit{LOCAL-FIRST}}$ *implements a safe exploration heuristics under memory model M.*

Proof. Assume that $\textsc{NextEvents}_{\textsc{local-first}}$ does not implement a safe heuristics under M. Then there exists a state $s = (\mathsf{pc}, \mathsf{val}, \mathsf{buf})$ reachable in $\mathrm{X}_{\mathrm{M}}(\mathcal{P})$ such that $\mathsf{val}\!\downarrow\!(\mathsf{ADR} \to \mathsf{DOM}) \neq \mathsf{val}'\!\downarrow\!(\mathsf{ADR} \to \mathsf{DOM})$ for all states $s' = (\mathsf{pc}', \mathsf{val}', \mathsf{buf}')$ reachable using $\textsc{NextEvents}_{\textsc{local-first}}$.

Let σ be the computation prefix that reaches s under memory model M, i.e., $s_0 \xrightarrow{\sigma} s \in \Delta^*_{\mathrm{X,M}}$. We will show that there exists another prefix σ' explored by Algorithm 4.2 such that $\mathit{Tr}(\sigma) = \mathit{Tr}(\sigma')$. Using Lemma 18, this will then yield $s = s'$, which will then contradict our initial assumption.

Let $\sigma\!\downarrow\! t := \alpha^t_0 \cdot \mathsf{e}^t_1 \cdot \alpha^t_1 \dots \cdot \mathsf{e}^t_{n(t)} \cdot \alpha^t_{n(t)}$ where $n(t) \in \mathbb{N}$ be the projection of σ to thread t such that all $\mathsf{e}^t_1, \dots, \mathsf{e}^t_{n(t)}$ are non-local events and $\alpha^t_0, \dots, \alpha^t_{n(t)}$ consist only of local events. The prefix σ' interleaves non-local events the same as in σ. However, whenever it executes some non-local e^t_i, for some $i \in [1..n(t)]$, it executes the entire α^t_i immediately after. Similarly before executing any non-local event, σ' interleaves block-wise entire sequences α^t_0.

Since the projections $\sigma\!\downarrow\! t$ and $\sigma'\!\downarrow\! t$ are the same for all threads $t \in \mathsf{TID}$, the program order $\to_{po}$ is the same for both σ and σ'. Moreover, $\to_{fo}$ is the same in σ and σ' since flush events still occur after their corresponding store events. Finally, since the interleaving of non-local events in both σ and σ' is the same, the conflict order $\to_{cf}$ is the same for σ and σ'. In conclusion $\mathit{Tr}(\sigma) = \mathit{Tr}(\sigma')$ with σ' as described above. $\square$

Algorithm 4.3 $\textsc{NextEvents}$ using the local-events-first heuristics

```
1: procedure $\textsc{NextEvents}_{\textsc{local-first}}(\mathrm{M}, \mathcal{P}, s, tr, \mathit{favorite})$
2:     $\mathit{result} := \emptyset$
3:     for all $\mathsf{e} \in \mathit{enabled}(s)$ do
4:         if $\mathit{favorite} = \bot$ or $\mathit{favorite} = \mathit{thread}(\mathsf{e})$ then
5:             append $\mathsf{e}$ to $\mathit{result}$;
6:         end if
7:     end for
8:     return $\mathit{result}$
9: end procedure
```

Figure 4.3 shows the local-events-first heuristics in action: it depicts the state-space explored by Algorithm 4.2 for the Figure 4.2 program below.

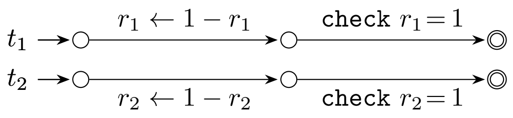


Figure 4.2: An ASSEMBLY program to showcase NEXTEVENTS$_{\text{LOCAL-FIRST}}$.

The state-space in Figure 4.3 is reduced since per-thread adjacent blocks of local events are considered in a shot. Figure 4.4 explains this by depicting the events in all Figure 4.2 program executions when checking reachability.

4.3.1 Dynamic Partial Order Reduction

Algorithm 4.4 describes the reimplementation of the original dynamic POR exploration for ASSEMBLY programs and SC reachability. However, unlike in [FG05], ASSEMBLY programs do not require per-thread-unique transitions out of each global state. In the algorithm, we use notation $last(\rightarrow_{cf}(\tau, a))$ for the last event e in computation τ that accessed a and $src(last(\rightarrow_{cf}(\tau, a)))$ for the last (source) SC state prior to the event e in $s_0 \xrightarrow{\tau} s$. The algorithm is *stateless* in the sense that the explored states are not tracked using some kind of *visited* set. Instead, the input computation τ to STATELESSDPOR and per-state-and-thread sets of backtracking events are used to guide the exploration of the state-space.

The section of the algorithm that updates *backtrack* states (Lines 6–19) is essential for the Line 22 loop: *backtrack*(s) used in the loop will likely be enlarged by subsequent recursive calls to STATELESSDPOR. Furthermore, since *backtrack* is updated whenever the depth-first exploration encounters same-address accessing events, dynamic POR will explore (as intended) all relevant interleavings of a finite, loop-free state-space.

In its simplest form, dynamic POR is essentially a DFS exploration of all traces of an input program with a finite, loop-free state-space (i.e., conflict resolution as in Figure 4.4 is done using DFS). This is what, e.g., the tool accompanying [Abd+15] does for programs described as LLVM IR [LA04].

4.3.2 Cartesian Partial Order Reduction

Cartesian POR relies on computing so called *cartesian vectors*: vectors of thread-local computation prefixes so that at most pairwise-last transitions of distinct prefixes are dependent. In the words of Gueta et. al [Gue+07], "a cartesian vector for a state describes a sequence of transitions that each thread can perform without context switches". In other words, cartesian POR exploration interleaves cartesian vector prefixes and performs context switches between these prefixes only upon each prefix's full processing.

The cartesian POR idea can be rephrased in terms of traces as: given some state s reachable in program $\mathcal{P}$ find traces $tr \in Traces_{\mathrm{M}}(\mathcal{P})$ starting

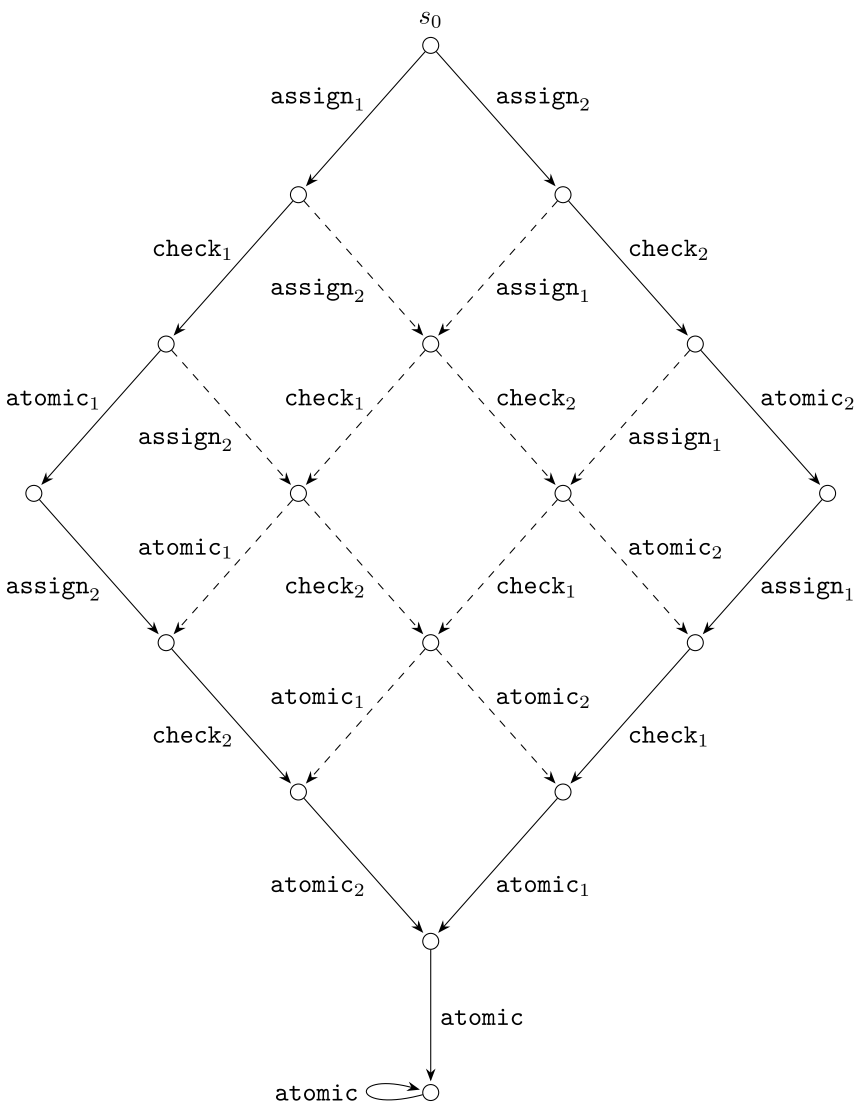


Figure 4.3: State-space of the Figure 4.2 program under $\mathrm{M} \in \{\mathrm{SC}, \mathrm{TSO}\}$. Labels of transitions correspond to events in $\mathrm{X}_\mathrm{M}(\mathcal{P})$. Their indexes indicate the event's thread. Non-local events `atomic`$_1$, `atomic`$_2$, and `atomic` are, as in τ_reach on page 73, used to determine when both threads reach their marked control states — the bottom state in the picture is the only one with $\mathsf{val}(a_\text{success}) = 1$. Local events `assign`$_i$ and `check`$_i$ (for $i \in \{1, 2\}$) are determined by instructions of thread t_i. The dotted transitions as well as the inner 4 states with only incoming dotted transitions are reduced by the local-events-first heuristics. Practically, a DFS implementation of Algorithm 4.2 finds the bottom state within only 7 steps in the reduced state-space while a BFS implementation will explore the 13 non-reduced transitions.

Algorithm 4.4 Stateless dynamic POR under SC

```
Input: Marked program $\mathcal{P}$ and computation $\tau$
Output: $true$ if some goal state is SC reachable in $\mathcal{P}$
        $false$ if no goal state is SC reachable in $\mathcal{P}$
Global Variables: $enabled, backtrack \in S_{SC} \times \mathsf{TID} \mapsto 2^{\Delta_{X,SC}}$,
        initially $backtrack(s,t) = \emptyset$ for all $s \in S_{SC}$ and $t \in \mathsf{TID}$

procedure StatelessDPOR($\mathcal{P}, \tau$)
    let $s \in S_{SC}$ such that $s_0 \xrightarrow{\tau} s$;
    if $s \in G$ then
        return $true$;
    end if
    for all $t \in \mathsf{TID}$ do                              // update $backtrack$ states
        for all $\mathtt{e} \in enabled(s,t)$ do
            if $addr(\mathtt{e}) = a$ and $last(\rightarrow_{cf}(\tau, a)) \neq \varepsilon$ then
                let $s_{\text{last}} = src(last(\rightarrow_{cf}(\tau, a)))$ and $t = thread(\mathtt{e})$;
                if $enabled(s_{\text{last}}, t) \neq \emptyset$ then
                    add $enabled(s_{\text{last}}, t)$ to $backtrack(s_{\text{last}}, t)$;
                else
                    for all $t' \in \mathsf{TID}$ do
                        add $enabled(s_{\text{last}}, t')$ to $backtrack(s_{\text{last}}, t)$;
                    end for
                end if
            end if
        end for
    end for
    if $\exists\, t \in \mathsf{TID}.enabled(s,t) \neq \emptyset$ then              // depth-first search
        let $backtrack(s,t) = enabled(s,t)$ and $done = \emptyset$;
        while $\exists\, (t', \mathtt{e}) \in backtrack(s) \setminus done$ do
            add $(t', \mathtt{e})$ to $done$;
            if $inst(\mathtt{e})$ is a store then
                let $\tau' = \tau \cdot \mathtt{e} \cdot \mathtt{flush}$ with $\mathtt{e} \rightarrow_{fo} \mathtt{flush}$;
            else
                let $\tau' = \tau \cdot \mathtt{e}$;
            end if
            if StatelessDPOR($\mathcal{P}, \tau'$) then
                return $true$;
            end if
        end while
    end if
    return $false$;
end procedure
```

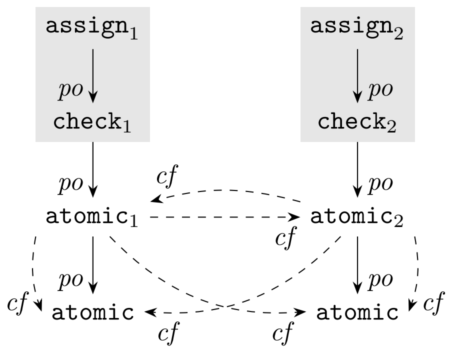


Figure 4.4: Representation of Figure 4.2 program traces: any prefix trace containing at most 7 events identifies a set of possible computations. By abuse of notation the last event in each thread is denoted by `atomic` as on page 73. The dotted conflict relations depict events accessing address a_{goal} — conflict resolution determines different traces. The Figure 4.3 state-space reduction is the result of considering the two blocks of local events in a shot.

from s such that at most the last per-thread events of tr are dependent.

Taking the Figure 4.2 program as example, the first cartesian vectors that one would determine contain three events — corresponding to the $\texttt{assign}_i \rightarrow_{po} \texttt{check}_i \rightarrow_{po} \texttt{atomic}_i$ order (for $i \in \{1, 2\}$) in Figure 4.4. When using cartesian POR exploration, these cartesian vectors are interleaved and the procedure is then reiterated starting from the newly reached states.

For an informative example, consider the Figure 4.5 program and its traces shown in Figure 4.6. This program depicts (as Assembly) the Figure 5(a) concurrent program of Gueta et al. [Gue+07] limited to two increments.

Since the only conflicting accesses of the Figure 4.5 program are implied by their program-order-last loads, starting from the initial SC state s_0 one first finds either of the following two cartesian vector pairs:

$$(\texttt{load}_1; \texttt{store}_1,\ \texttt{load}_2; \texttt{store}_2 + \texttt{flush}_2; \texttt{load}'_2; \texttt{store}'_2 + \texttt{flush}'_2; \texttt{load}''_2),$$
$$(\texttt{load}_1; \texttt{store}_1 + \texttt{flush}_1; \texttt{load}'_1; \texttt{store}'_1 + \texttt{flush}'_1; \texttt{load}''_1,\ \texttt{load}_2; \texttt{store}_2).$$

Continuing the cartesian POR exploration one will then extend the above prefixes to include the second increment in each of the threads, and then to verify the checks by ordering the conflicting atomic events.

The symmetric handling of finding new cartesian vectors (recall that in our context these are per-thread sequences of events) prompts for a parallel (and perhaps BFS) implementation of this exploration method. This could be achieved by adapting the saturation-based implementation proposed by Gueta et. al [Gue+07].

As a concluding remark, using traces to represent the partially-ordered event structure of a program is the way to go. The clarity they provide helps to easily grasp complex methods such as cartesian and dynamic POR.

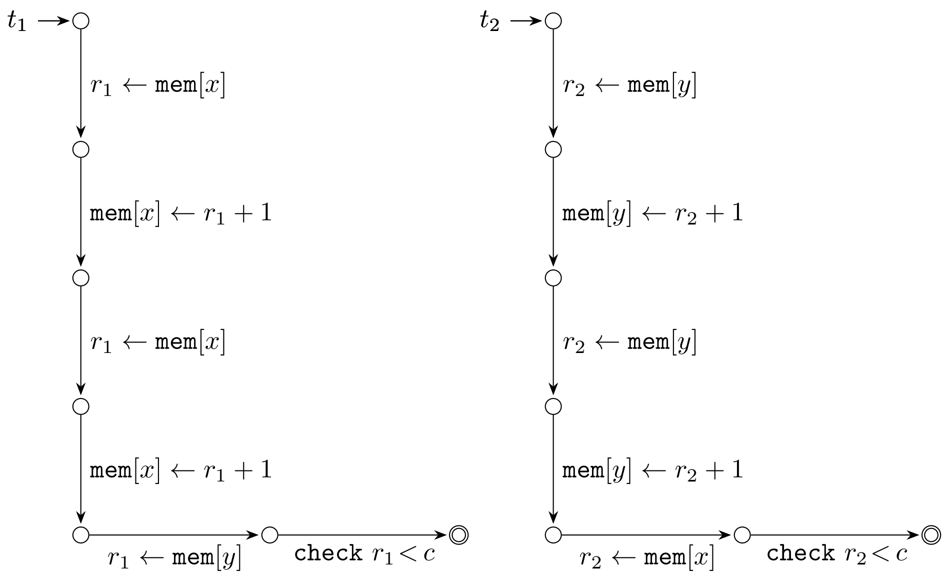

t_1
$r_1 \leftarrow$ mem$[x]$
mem$[x] \leftarrow r_1 + 1$
$r_1 \leftarrow$ mem$[x]$
mem$[x] \leftarrow r_1 + 1$
$r_1 \leftarrow$ mem$[y]$
check $r_1 < c$
t_2
$r_2 \leftarrow$ mem$[y]$
mem$[y] \leftarrow r_2 + 1$
$r_2 \leftarrow$ mem$[y]$
mem$[y] \leftarrow r_2 + 1$
$r_2 \leftarrow$ mem$[x]$
check $r_2 < c$

Figure 4.5: An Assembly program to showcase cartesian POR.

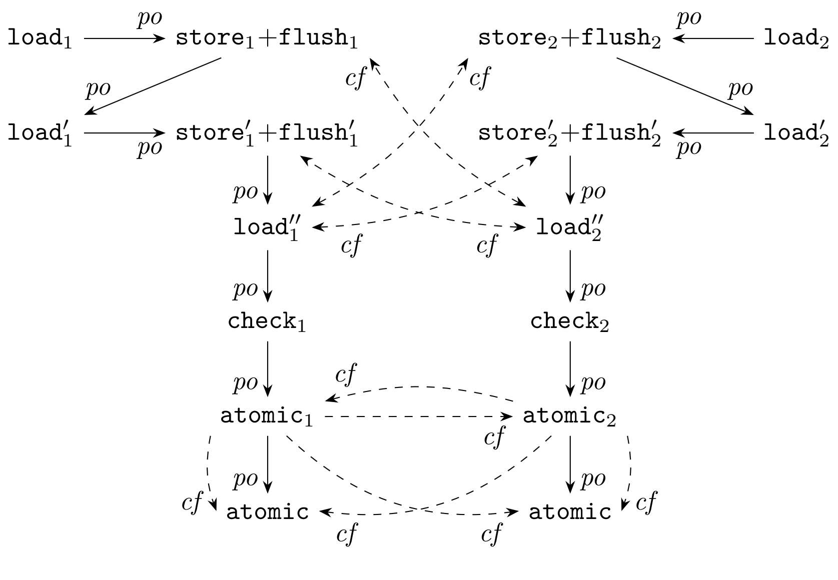


Figure 4.6: Representation of Figure 4.5 program traces: any prefix trace with at most 15 events identifies a set of possible computations. The `load`$_i$ and `store`$_i$+`flush`$_i$ events identify the first value-incrementing load and subsequent store+flush in t_i and their primed version the second value-incrementing load and store+flush in the corresponding thread. The `load`$''_i$ event represents the load preceding the final bound check in thread t_i. By abuse of notation the last event in each thread is denoted by `atomic` as on page 73. The dotted conflict relations depict (1) events of the threads accessing addresses x and y and (2) events accessing address a_{goal}. As in the earlier Figure 4.4 depiction, conflict resolution determines different traces.

Chapter 5

Experimental Evaluation

It is often said that experiments should be made without preconceived ideas. That is impossible. Not only would it make every experiment fruitless, but even if we wished to do so, it could not be done. Every man has his own conception of the world, and this he cannot so easily lay aside.

Henri Poincaré
Science and hypothesis

To asses the efficiency of the introduced verification techniques we implemented several algorithms on top of the tool Trencher [Der15]. Trencher was initially developed to check robustness and implements the algorithms in [BDM13] for finding witness computations and for deriving fences. Our subsequent evaluation is two-fold.

In Section 5.1 we provide a comparison between our implementation of lazy TSO reachability and two other tools that can be used to check TSO reachability, Memorax [Abd+12b] (revision 4f94ab6) and CBMC [CKL04] (version 4.7). Additionally to typical test cases we highlight two families of programs for which our implementation outperforms Memorax and CBMC, respectively. We then conclude that the three different approaches to check TSO reachability are good for orthogonal input classes.

In Section 5.2 we compare some of the POR techniques introduced in the previous chapter. To be precise, we first compare Trencher's standard depth-first [Der15] and a breadth-first implementation of SC reachability. Second, we compare a dynamic POR implementation for SC reachability with Trencher's standard [Der15] local-events-first reduction under SC. Our findings in both cases advocate that Trencher's standard implementation (DFS and local-events-first reduction) is faster for typical benchmark examples. However, we also confirm the existence of test inputs for which both (1) a BFS implementation of SC reachability, and (2) dynamic POR in spite of local-events-first reduction, are more advantageous.

5.1 Evaluation for Lazy TSO Reachability

We implemented lazy TSO reachability on top of TRENCHER [Der15] by reusing the algorithm for finding witness computations described in [BDM13] to derive a robustness-based oracle. TRENCHER originally used the SPIN model checker [Hol97] as a back-end for carrying SC reachability checking. The current implementation uses a simpler SC model checker equipped with the local-events-first reduction technique described in Section 4.3. By using the simpler model checker our implementation does not need to compile verifier executables (pan) as is the case for SPIN.

We have implemented Algorithm 3.1 with the following amendments. First, the extension does not delete the store instruction $inst_1$. This first simplification ensures the extended program has a (sound) superset of the TSO behaviors of the original program. Second, the extension only adds instructions along $\overline{q}_1, \ldots, \overline{q}_n$. The remaining instructions were added to ensure all behaviors of the original program exist in the extended program, once $inst_1$ is removed — as just noted, our implementation does not remove $inst_1$. The resulting algorithm is guaranteed to yield correct results for any cyclic program when (and if) it returns. Of course, it cannot be guaranteed to terminate in general. Finally, our implementation explores extensions due to different instruction sequences in parallel, rather than sequentially.

Our hypothesis is that when analyzing certain classes of programs lazy TSO reachability is better suited to find bugs than other existing approaches. We compare our prototype implementation against two other verifiers that support the TSO semantics: MEMORAX [Abd+12b] (revision 4f94ab6) and CBMC [CKL04] (version 4.7). MEMORAX implements a reachability checker that is both sound and complete by reducing it to checking coverability in a well-structured transition system. CBMC is an SMT-based bounded model checker for C programs. Consequently, it is sound, but not complete: it is complete only up to a given bound on the number of loop iterations in the input program.

5.1.1 Examples

We first tested our implementation on a small set of examples. Figure 5.1 summarizes characteristics of the examples taken from the initial TRENCHER tests: number of threads (T), states (St), and transitions (Tr). The first example is a model of the buggy Parker class from Java VM [Dic09]. The next three examples are mutual exclusion protocols implemented using shared variables. These protocols do not guarantee mutual exclusion under TSO. We tested the Dekker and Peterson's algorithms for two threads and Lamport's fast mutex [Lam87] for three threads. The last three tests from Figure 5.1 give statistics concerning reachability in robust test cases for the lock-free stack and for the MCS and CLH locking algorithms from [HS08].

#	Program	T	St	Tr	RQ	CPU	Real
1	Parker (non-robust)	2	11	10	4	8	5
2	Peterson (non-robust)	2	14	18	12	21	13
3	Dekker (non-robust)	2	24	30	30	171	70
4	Lamport (non-robust)	3	33	36	27	1839	694
5	MCS Lock	4	52	50	30	127	61
6	CLH Lock	3	43	41	70	10	7
7	Lock-Free Stack	4	46	50	14	9	7

Spin	Clang	CPU	Real
132	1987	10	1229
788	9861	100	1651
2270	38413	670	3600
4358	22297	330	3061
454	4498	20	1653
452	12083	90	1998
38	475	0	518

Figure 5.1: TRENCHER benchmarking results. Times are in milliseconds. The right-hand-side table shows the higher times needed when using SPIN.

We also performed three parametrized tests. First, we varied the number of threads in Lamport's fast mutex [Lam87] (see Figure 5.2). Next, inspired by examples of the fence-insertion tool MUSKETEER [Alg+14], the modified Dekker in Figure 5.3 adds a branching structure parametric over N to both program threads. Lastly, the program in Figure 5.4 places stores to address x on a length N loop in thread t_1: since t_1 expects to load the initial y value while t_2 expects to load 1 and then 0 from x, an execution that reaches the goal state goes through the length N loop twice.

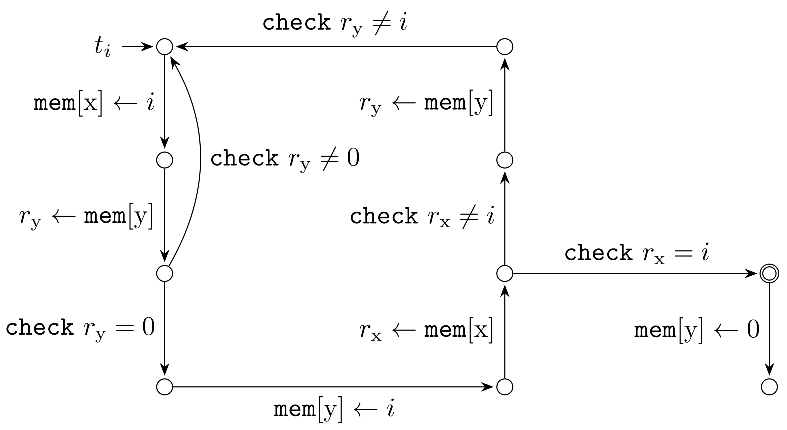


Figure 5.2: The i-th thread of Lamport's fast mutual exclusion protocol. The goal state is reachable under TSO if, e.g., the stores mem[x] $\leftarrow i$ are all buffered past the loads $r_y \leftarrow$ mem[y] that all load the initial 0 value at y.

5.1.2 Evaluation

We ran the tests on a QEMU @ 2.67GHz virtual machine (16 cores) with 8GB RAM running GNU/Linux. The table in Figure 5.1 summarizes the results of the TRENCHER benchmark tests. The RQ columns indicate the number of SC reachability queries raised by TRENCHER. The CPU and Real columns give the total CPU and wall-clock time that a test took.

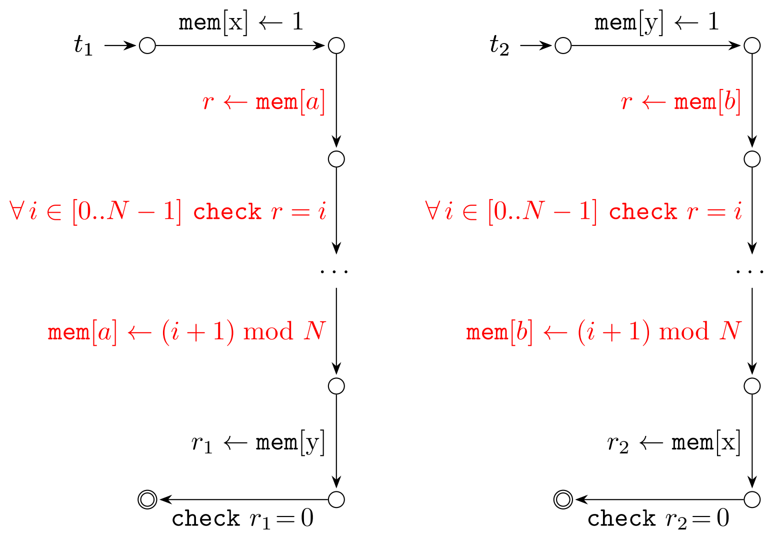


Figure 5.3: The Dekker algorithm modified to stress branching. The highlighted branching structures contain operations that manipulate distinct addresses $a, b \notin \{x, y\}$ between the accesses to x and y. The $\forall$-quantified notation indicates $2 \times N$ transitions in each thread that store $(i+1)$ mod N at the address that r previously loaded its value from (the notation spans N intermediary states marked by $\cdots$). Each intermediary state is the target of precisely one `check` $r = i$ labeled transition for some $i \in [0..N-1]$ as well as the source of one `mem`$[a] \leftarrow (i+1)$ mod N labeled transition for that same i. If the first store is delayed past the last load in either of the two threads then a goal state is TSO-reachable.

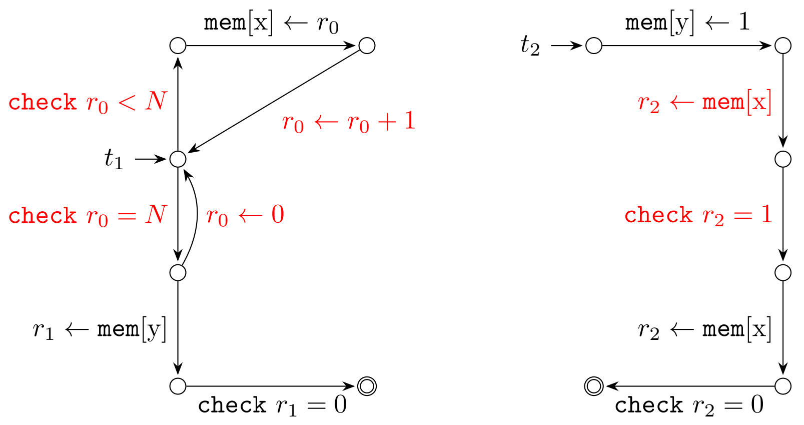


Figure 5.4: The Dekker algorithm modified to stress unwinding. The highlighted loops in thread t_1 write $0, 1, \ldots, N-1$ to x and non-deterministically reset r_0 to 0 or continue with reading from y. The unwinding needed to reach a goal state under TSO increases for larger values of N. Concretely, a goal state is TSO-reachable if t_1 goes through the (length N) loop two times: once to satisfy `check` $r_2 = 1$ and the second time to satisfy `check` $r_2 = 0$.

The graph in Figure 5.5 depicts the running times of the three tools on the non-robust examples from Figure 5.1. For CBMC, we used the mutual exclusion algorithms' versions that its authors provide. For MEMORAX, we hand-wrote `*.rmm` files for the first 4 test programs. We did not perform a comparison for robust programs: if SC reachability returns false on an input program, our implementation decides mutual exclusion as fast as TRENCHER is able to determine robustness. Moreover, CBMC implements strictly an under-approximative method where the number of loop iterations is bounded. Our robust tests, however, contain unbounded loops.

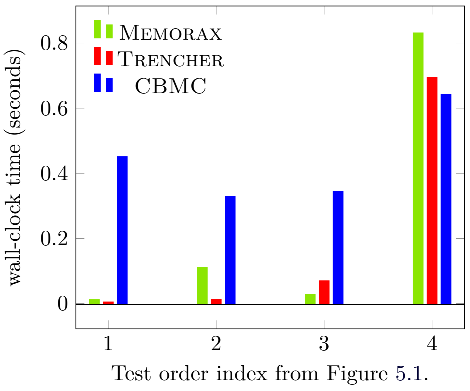


Figure 5.5: Side-by-side runtimes for the non-robust tests in Figure 5.1.

The high time needed to verify Lamport's mutex — compared to the other Figure 5.1 tests — is justified by the correlation between the program's data domain size and its number of threads. The graph in Figure 5.6 shows that CBMC is fastest for a larger number of threads. This is the case since, actually, the smallest unwind bound suffices for CBMC to conclude reachability. For MEMORAX and TRENCHER the system runs out of memory when $N = 5$. This underlines once again just how troublesome the state-space explosion is for TSO reachability. Although it is not easily noticeable in the picture, MEMORAX's exponential scaling is better than TRENCHER's: although TRENCHER is slightly faster than MEMORAX for $N \in \{2, 3\}$, MEMORAX clearly outperforms TRENCHER when $N = 4$.

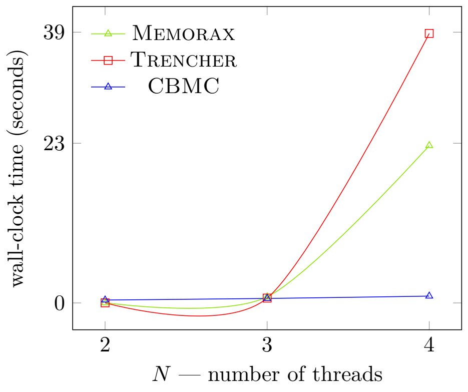


Figure 5.6: Runtimes for Lamport's fast mutex (depicted in Figure 5.2).

The graphs in Figure 5.7 show that for programs as in Figure 5.3 our prototype is faster than MEMORAX. It seems MEMORAX cannot cope well with the branching factor that the parameter N introduces. The right-hand-side graph in Figure 5.7 serves to show that TRENCHER's behavior with varying N is also exponential but less steep.

Similarly, the Figure 5.8 graphs show that for programs as in Figure 5.4 our prototype is faster than CBMC. Indeed, with increasing N, an ever larger number of constraints need to be generated by CBMC. For TRENCHER, regardless of the value of N, it takes three SC reachability queries to conclude TSO reachability. As before, from the right-hand-side graph in Figure 5.8 we see that TRENCHER's behavior is also exponential but less steep.

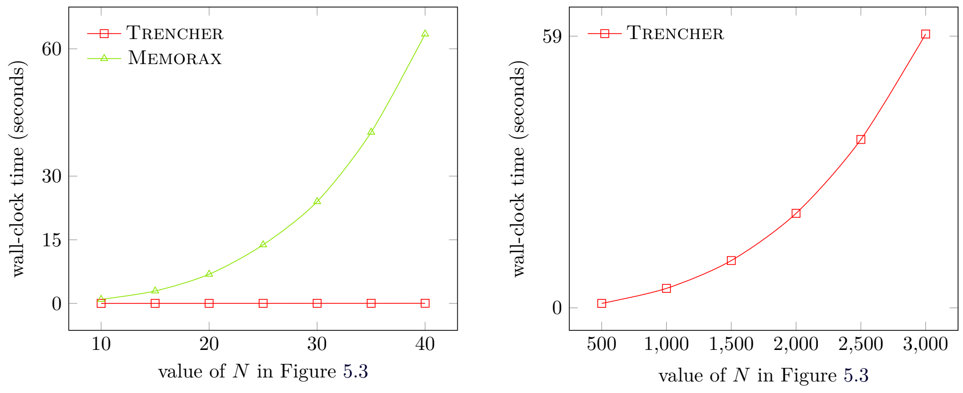


Figure 5.7: Side-by-side runtimes of TRENCHER and MEMORAX for Figure 5.3 programs with increasing values of N. MEMORAX takes $\simeq$ 1.5 minutes for $N = 50$.

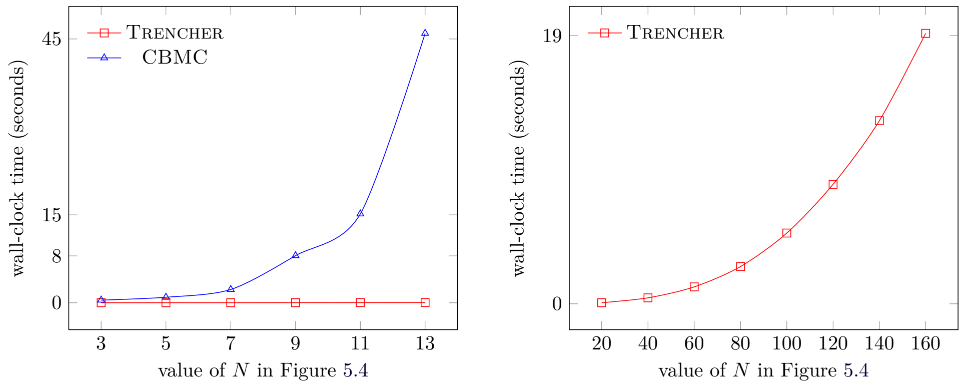


Figure 5.8: Side-by-side runtimes of TRENCHER and of CBMC for Figure 5.4 programs with increasing values of N. CBMC takes $\simeq$ 8.5 minutes for $N = 20$.

5.1.3 Discussion

Because we find several witnesses in parallel, throughout the experiments our implementation required up to 2 iterations of the loop in Algorithm 3.1. In the case of robust programs, one iteration is always sufficient. This suggests that robustness violations are really the critical behaviors leading to goal states unreachable under SC to become reachable under TSO.

The experiments indicate that, at least for some programs with a high branching factor, our implementation is faster than MEMORAX if a useful witness can be found within a small number of iterations of Algorithm 3.1. Similarly, our prototype is faster than CBMC for programs that require a high unwinding bound to make visible TSO behavior needed for reaching a certain goal state. Although the two programs by which we show this are rather artificial, we expect such characteristics to occur in actual code. Hence, our approach seems to be strong on an orthogonal set of programs. In a portfolio model checker, it could be used as a promising alternative to other existing techniques.

To evaluate the practicality of our method, more experiments are needed. In particular, we hope to be able to substantiate the above conjecture for concrete programs with behavior like that depicted in Figures 5.3 and 5.4. Unfortunately, there seems to be no clear way of translating (compiled) C programs into Assembly syntax without substantial abstraction. To handle C code, an alternative would be to re-implement our method within CBMC. This approach, however, would force us to determine a priory a good-enough unwinding bound. Moreover, we could no longer conclude safety of robust programs with unbounded loops.

5.2 Evaluation for Exploration Techniques

We implemented in Trencher [Der15] several new algorithms for state-space exploration and tested both how fast they are as well as how much state reduction they achieve.

First, motivated by a class of examples that Trencher's default DFS exploration cannot cope with, we tested an alternative BFS implementation. Our hypothesis is that DFS and BFS explorations are well suited for different classes of input programs. Furthermore, as already empirically shown in [Der15] for DFS exploration, we think that using Trencher's default live variable analysis and local-events-first state-space reduction is generally better than not using them — both for DFS and for BFS exploration.

Second, we implemented a non-stateless version of dynamic POR — essentially Algorithm 4.4 enhanced by tracking the *visited* set of states. We then tested it against the default local-events-first heuristics implemented in Trencher. Our hypothesis is that this dynamic POR implementation (with *visited* states tracked) is slower for standard examples while being faster for specific classes of examples like the Indexer algorithm from [FG05].

5.2.1 BFS and DFS Exploration for SC reachability

The example in Figure 5.10 made us recognize the need for an alternative BFS exploration to Trencher's standard DFS one: using BFS exploration it can be successfully checked that this Assembly program is non-robust.

To compare the DFS and BFS implementations of Trencher's SC reachability we evaluated the Figure 5.9 benchmark tests. These tests are variations of the Figure 5.1 ones (and a subset of the [Der15] Table 7.3 tests).

#	Program	T	St	Tr	CPU	Real	CPU	Real
1	Dekker (not fenced)	2	24	30	431	165	489	171
2	Dekker (fenced)	2	28	34	0	0	1	0
3	Peterson (not fenced)	2	14	18	168	56	161	55
4	Peterson (fenced)	2	16	20	1	0	0	0
5	Lamport (not fenced)	3	33	36	12572	3577	14916	4291
6	Lamport (fenced)	3	39	42	2	1	1	1
7	CLH Lock	3	42	41	94	29	103	31
8	MCS Lock	2	54	58	41	11	46	12
9	Cilk WSQ (incorrect use)	5	80	79	103267	28597	126098	36693
10	Cilk WSQ (correct use)	3	73	72	35	10	23	6
11	Lock-free stack	4	46	50	0	0	0	0

Figure 5.9: TRENCHER benchmarking results (DFS vs BFS exploration). Times are in milliseconds. The analysis reports on robustness-enforcing fence insertion as in [BDM13] and the examples are a subset of the test benchmark used in [Der15]. The right-hand-side table shows the mostly higher times for BFS exploration.

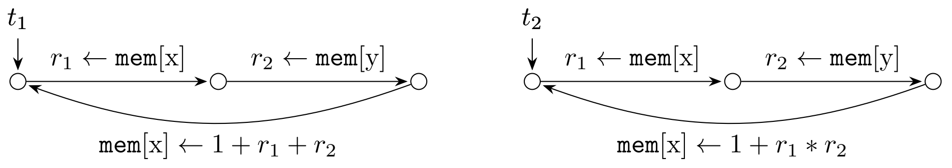

t_1
$r_1 \leftarrow \texttt{mem}[x]$
$r_2 \leftarrow \texttt{mem}[y]$
$\texttt{mem}[x] \leftarrow 1 + r_1 + r_2$
t_2
$r_1 \leftarrow \texttt{mem}[x]$
$r_2 \leftarrow \texttt{mem}[y]$
$\texttt{mem}[x] \leftarrow 1 + r_1 * r_2$

Figure 5.10: A non-robust program for which DFS analysis segfaults.

Evaluation and Discussion We ran tests on a computer with a 1.7 GHz Intel Core i7 processor (2 cores with 2 threads each) with 8GB RAM running OS X Yosemite. The table in Figure 5.9 summarizes the control results of TRENCHER on our selected benchmark tests. As in Figure 5.1, the columns CPU and Real give the total CPU time and the wall-clock time a test takes while columns T, St, and Tr describe how many threads, control states, and state-changing transitions each test has.

Figure 5.9 shows that, for several standard examples from the literature (the benchmark tests that take more than 10 milliseconds are depicted), the DFS exploration is generally faster than the BFS exploration. This holds even when using live variable analysis and/or the local-events-first reduction (as described in Section 4.3) under SC. One reason for this behavior is that, for each SC reachability query of the fence insertion benchmark, the BFS approach has to store the fringe of the explored state-space in a queue instead of only relying on the system stack as is the case when using DFS.

However, Figure 5.11 shows that no particular combination of DFS/BFS and/or live variable analysis and/or simple partial order reduction is a “best” silver bullet technique. Indeed, how fast each SC reachability query returns and how many states it explores depends, intuitively, on how deep (for BFS) or on how deep and how far to the right (for left-to-right DFS exploration) a goal state can be found in the state-space.

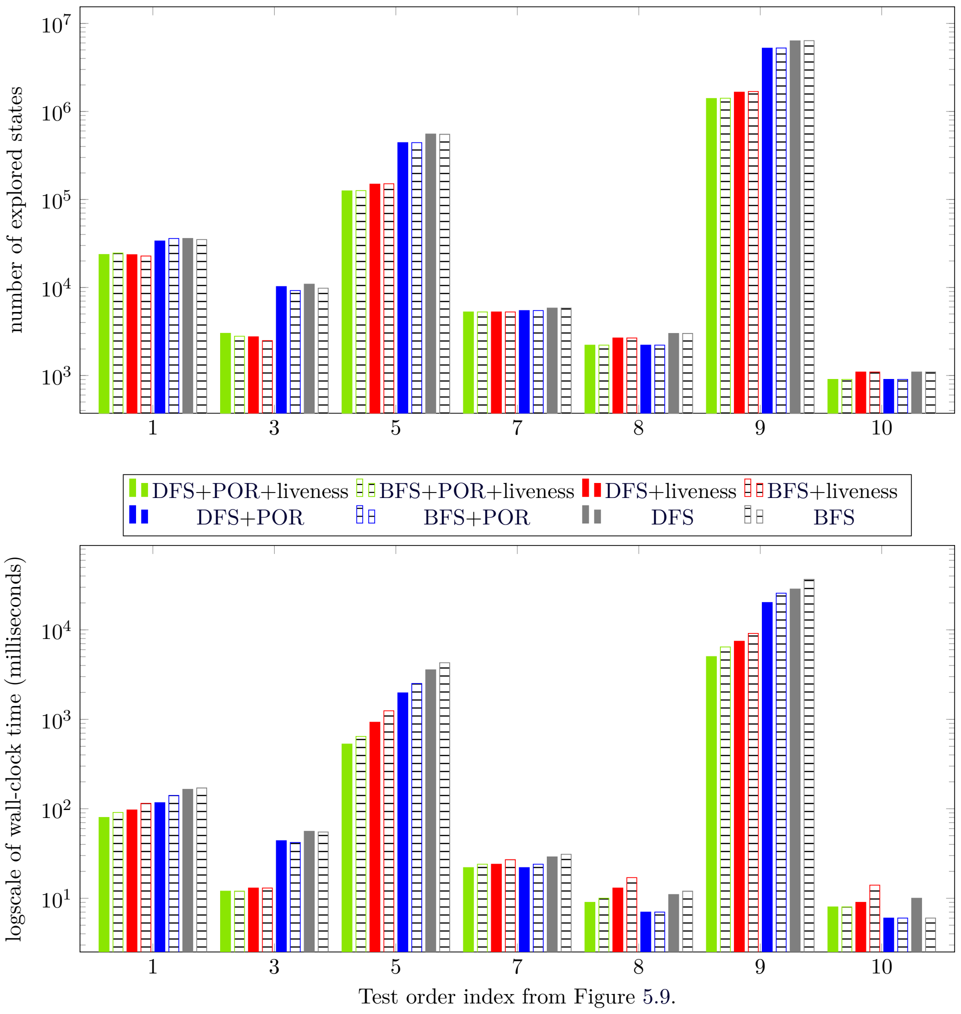


Figure 5.11: Trencher (DFS & BFS) results with two further reductions: live variable analysis and local-events-first POR (named POR in the legend).

5.2.2 POR Exploration for SC reachability

To compare dynamic POR exploration and the DFS implementation with local-event-first heuristics of Trencher's SC reachability (Algorithm 4.2 with M = SC) we once more evaluated the benchmark tests in Figure 5.9. As before, we used tests that take more than 10 milliseconds, namely, the unfenced Dekker, Peterson and Lamport algorithms, the algorithm models for the lock free stack and the CLH and MCS locks [HS08], and two use cases for Cilk-5's work-stealing queue [FLR98].

Furthermore, we used the Indexer program from [FG05] to demonstrate that, for certain classes of programs, dynamic POR outperforms the simpler local-events-first reduction. The Indexer program's i^{th} thread is depicted using the Assembly syntax in Figure 5.12.

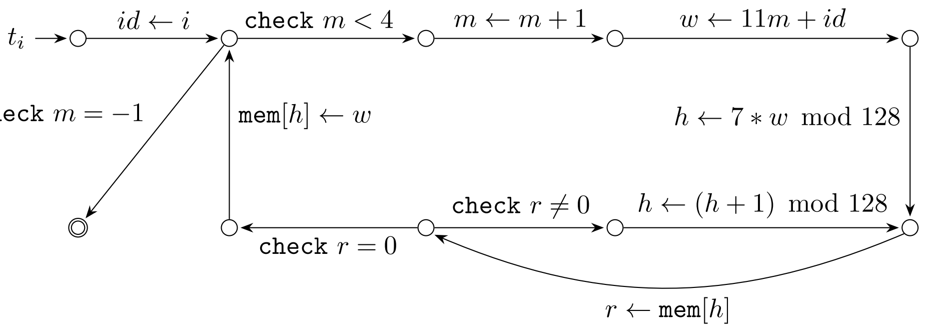


Figure 5.12: The i-th thread of the Indexer program that manipulates a shared hash table (the memory in our interpretation). Each thread seeks to insert into the table at hash index $h := 7 * w \mod 128$ a fixed (in the picture 4) number of messages w. If a hash table collision occurs then the next free entry in the table is used. We artificially inserted marked control states such that a goal state is not reachable — when no POR is used this enforces the full state space exploration.

Evaluation and Discussion As before, we ran the tests on a computer with a 1.7 GHz Intel Core i7 processor (2 cores with 2 threads each) with 8GB RAM running OS X Yosemite.

Figure 5.13 shows that for the Indexer program depicted in Figure 5.12 the local-events-first heuristics cannot cope with an increasing number of threads as well as dynamic POR exploration can.

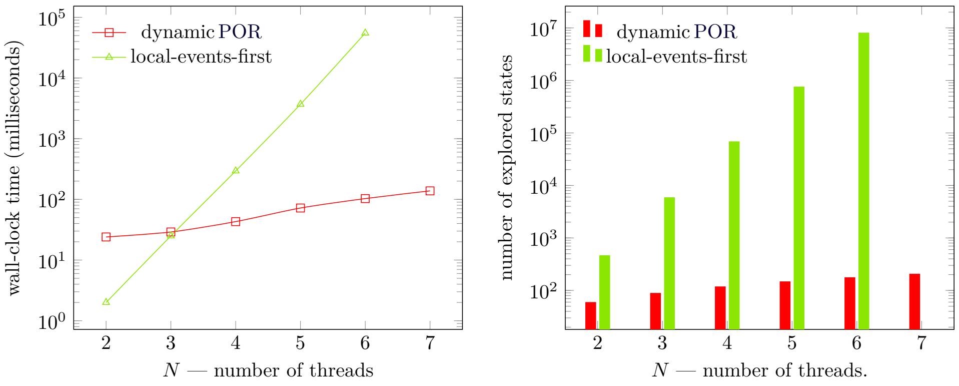


Figure 5.13: Logscales of runtimes and explored states for the N-threaded Indexer program while running Trencher with live variable analysis turned on.

Figure 5.14 shows that, for the Figure 5.9 benchmark tests that take more than 10 milliseconds, the DFS exploration with local-events-first reduction is faster than the dynamic POR exploration. Concerning explored states, since the tested exploration techniques are DFS-based they explore a similar number of states. Subtle differences in the number of explored states are the result of either (1) the order in the parallelization of the SC checks performed, or (2) the order in which transitions are explored, as dictated by the two (essentially different) algorithms.

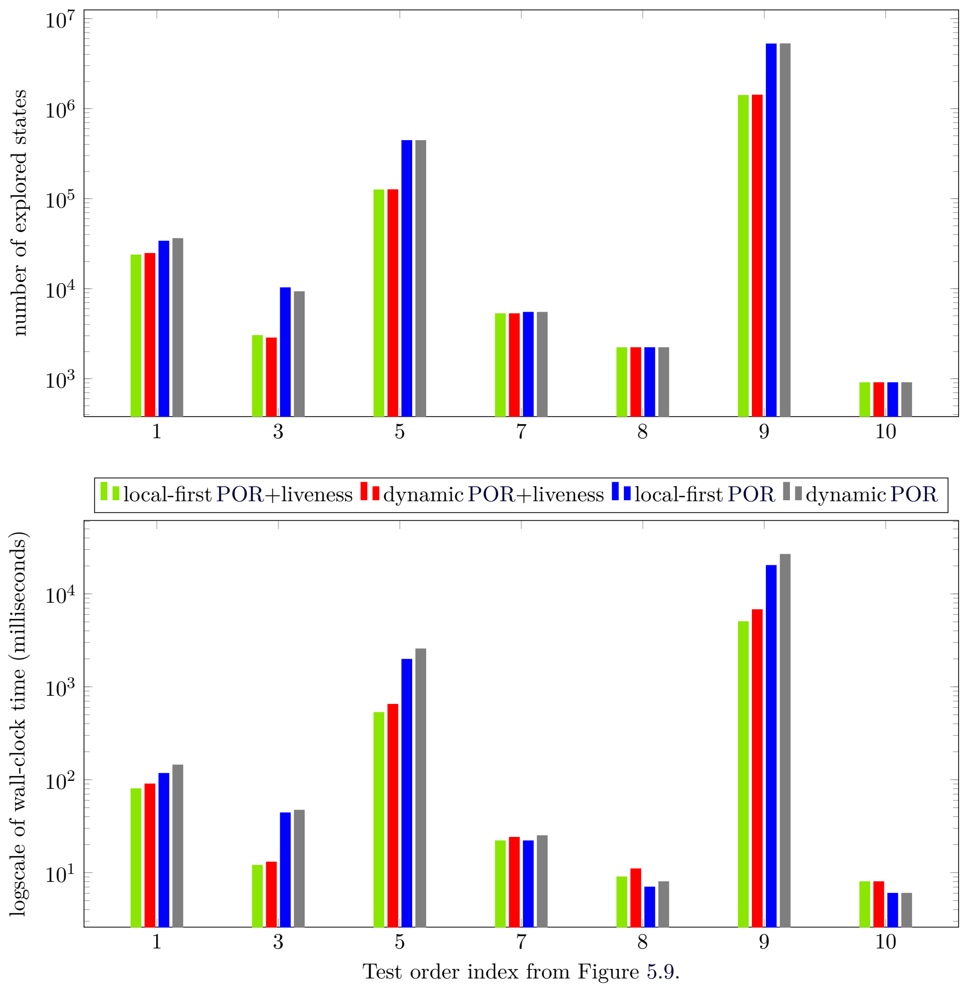


Figure 5.14: TRENCHER local-events-first (local-first POR in the legend) and dynamic POR comparison with and without live variable analysis.

To conclude, just like when offered the choice of either DFS or BFS exploration, there is no POR silver bullet technique — both the dynamic POR and the local-events-first reduction can, depending on the considered test, be faster and/or reduce more states than the other.

Chapter 6

Conclusion

All human knowledge begins with intuitions, proceeds from thence to concepts, and ends with ideas.

Immanuel Kant
Critique of pure reason

This thesis presents new algorithms to verify TSO-relaxed programs. The TSO memory model's core feature is that stores are buffered between a program's threads and the system's shared memory. While the TSO stores' de-atomization speeds up program execution, it also makes programs more difficult to understand and analyse. Consider, for instance, the reachability problem — the main problem that we target. While reachability under SC is *only* PSPACE-complete [Koz77], the TSO reachability problem is tougher: non-primitive-recursive-complete decidable to be precise [Ati+10].

6.1 Summary

Owing to TSO reachability's high complexity, we develop approximating techniques that target this problem.

To *under-approximate* TSO reachability we propose an algorithm to check it lazily [Bou+15]. Lazy TSO reachability simulates store buffering using thread-auxiliary variables between oracle-indicated control locations. The two natural properties of an oracle guarantee that:

- if the oracle indicates an empty sequence of instructions then the easier SC reachability task produces the same answer as TSO reachability.
- if the oracle outputs some non-empty sequence ι of instructions then the TSO delays that buffering ι stores produces is encoded using finitely many thread-auxiliary registers. By iteratively performing this refinement and by checking SC reachability in the program with the encoded delays TSO reachability can eventually be affirmatively answered.

To *over-approximate* TSO reachability we propose abstractions of TSO buffers and algorithms that exploit them. Furthermore, we determine that reachability of TSO reachability with multiset-abstracted buffers and per variable last-added-value information is decidable.

Finally, we show that partial order reduction approaches introduced for the model checking problem are generalizable to account for TSO memory. Intuitively, we stress that TSO stores consist of

- a machine-local operation whose precise interleaving with operations of the other machines is inconsequential to program correctness, and
- a non-local operation whose interleaving with non-local operations of the other machines might interfere with safe program behavior.

Using this insight and the trace-based POR perspective [Maz86; SS88] we develop and adapt several reduction techniques including dynamic and cartesian partial order reduction.

6.2 Future Work

The theory behind lazy TSO reachability seems not to need to be developed for the finite case and then extended via bounded model checking to general programs: through an efficient enumeration of oracle suggestions one should still achieve semi-decidability. Furthermore, it would be interesting to know how an enumeration-based oracle compares to our robustness-based oracle in terms of efficiency.

Both happens-before-based properties like robustness [BMM11] and persistence [AAN15] as well as set-based buffer approximations bring us closer to verifying a larger class of programs under TSO. It would be worthwhile to assess which other set-based abstractions might help enlarge this class of programs. Furthermore, tight complexity bounds for reachability in the multiset abstraction with per address last-added-values are not yet known. A possible way to find such bounds would be through a reduction from/to the ExpSpace-complete Petri net coverability problem [Lip76; Rac78]. This, however, might be non-trivial: our attempts to find a reduction from Petri net coverability led to the conclusion that the nets might need zero-test arcs.

For further advances on the topic of partial order reduction the recent study of Rodríguez et al. [Rod+15] that combines POR with net unfoldings seems to be the most promising starting point.

Finally, the experimental evaluation could be improved through more tests and repeating runs to provide a statistically sounder ζ-score for the compared implementations. Furthermore, more algorithms (including the over-approximating refinement and the cartesian POR one) could be implemented and evaluated. Concerning POR algorithms, it is likely more efficient to implement a scheduler-intensive approach similar to the one in [Abd+15] instead of explicitly storing partially-ordered computation traces.

Bibliography

[AAN15] P. A. Abdulla, M. F. Atig, and T. P. Ngo. “The best of both worlds: trading efficiency and optimality in fence insertion for TSO”. In: *24th European Symposium on Programming.* Vol. 9032. LNCS. Springer, 2015, pp. 308–332.

[Abd+12a] P. A. Abdulla, M. F. Atig, Y-F. Chen, C. Leonardsson, and A. Rezine. “Automatic fence insertion in integer programs via predicate abstraction”. In: *19th International Symposium on Static Analysis.* Vol. 7460. LNCS. Springer, 2012, pp. 164–180.

[Abd+12b] P. A. Abdulla, M. F. Atig, Y-F. Chen, C. Leonardsson, and A. Rezine. “Counter-example guided fence insertion under TSO”. In: *18th Tools and Algorithms for the Construction and Analysis of Systems.* Vol. 7214. LNCS. Springer, 2012, pp. 204–219.

[Abd+13] P. A. Abdulla, M. F. Atig, Y-F. Chen, C. Leonardsson, and A. Rezine. “Memorax, a precise and sound tool for automatic fence insertion under TSO”. In: *19th Tools and Algorithms for the Construction and Analysis of Systems.* Vol. 7795. LNCS. Springer, 2013, pp. 530–536.

[Abd+14] P. A. Abdulla, S. Aronis, B. Jonsson, and K. F. Sagonas. “Optimal dynamic partial order reduction”. In: *41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* ACM, 2014, pp. 373–384.

[Abd+15] P. A. Abdulla, S. Aronis, M. F. Atig, B. Jonsson, C. Leonardsson, and K. F. Sagonas. “Stateless model checking for TSO and PSO”. In: *21st Tools and Algorithms for the Construction and Analysis of Systems.* Vol. 9035. LNCS. Springer, 2015, pp. 353–367.

[Abd+96] P. A. Abdulla, K. Cerans, B. Jonsson, and Y.-K. Tsay. “General decidability theorems for infinite-state systems”. In: *11th*

Annual IEEE Symposium on Logic in Computer Science. IEEE Computer Society, 1996, pp. 313–321.

[ABP11] M. F. Atig, A. Bouajjani, and G. Parlato. “Getting rid of store-buffers in TSO analysis”. In: *23rd Computer Aided Verification.* Vol. 6806. Springer, 2011, pp. 99–115.

[Adv+91] S. V. Adve, M. D. Hill, B. P. Miller, and R. H. B. Netzer. “Detecting data races on weak memory systems”. In: *18th Annual International Symposium on Computer Architecture.* ACM, 1991, pp. 234–243.

[AG96] S. V. Adve and K. Gharachorloo. “Shared memory consistency models: a tutorial”. In: *IEEE Computer* 29.12 (1996), pp. 66–76.

[AJ96] P. A. Abdulla and B. Jonsson. “Verifying programs with unreliable channels”. In: *Information and Computation* 127.2 (1996), pp. 91–101.

[AK95] P. A. Abdulla and M. Kindahl. “Decidability of simulation and bisimulation between lossy channel systems and finite state systems (extended abstract)”. In: *6th International Conference on Concurrency Theory.* Vol. 962. LNCS. Springer, 1995, pp. 333–347.

[AKH03] J. H. Anderson, Y.-J. Kim, and T. Herman. “Shared-memory mutual exclusion: major research trends since 1986”. In: *Distributed Computing* 16.2-3 (2003), pp. 75–110.

[AKT13] J. Alglave, D. Kroening, and M. Tautschnig. “Partial orders for efficient BMC of concurrent software”. In: *25th Computer Aided Verification.* Vol. 8044. LNCS. Springer, 2013, pp. 141–157.

[Alg+13] J. Alglave, D. Kroening, V. Nimal, and M. Tautschnig. “Software verification for weak memory via program transformation”. In: *22nd European Symposium on Programming.* Vol. 7792. Springer, 2013, pp. 512–532.

[Alg+14] J. Alglave, D. Kroening, V. Nimal, and D. Poetzl. “Don’t sit on the fence - a static analysis approach to automatic fence insertion”. In: *26th Computer Aided Verification.* Vol. 8559. LNCS. Springer, 2014, pp. 508–524.

[Alg10] J. Alglave. “A shared memory poetics”. PhD thesis. University Paris 7, 2010.

[AM06] Arvind and J.-W. Maessen. “Memory model = instruction reordering + store atomicity”. In: *33rd International Symposium on Computer Architecture.* IEEE Computer Society, 2006, pp. 29–40.

[AM11] J. Alglave and L. Maranget. “Stability in weak memory models”. In: *22nd Computer Aided Verification*. Vol. 6806. LNCS. Springer, 2011, pp. 50–66.

[AMP96] R. Alur, K. L. McMillan, and D. Peled. “Model-checking of correctness conditions for concurrent objects”. In: *11th Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society, 1996, pp. 219–228.

[AMT14] J. Alglave, L. Maranget, and M. Tautschnig. “Herding cats: modelling, simulation, testing, and data mining for weak memory”. In: *ACM Transactions on Programming Languages and Systems* 36.2 (2014), 7:1–7:74.

[Ati+10] M. F. Atig, A. Bouajjani, S. Burckhardt, and M. Musuvathi. “On the verification problem for weak memory models”. In: *37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2010, pp. 7–18.

[Ati+12] M. F. Atig, A. Bouajjani, S. Burckhardt, and M. Musuvathi. “What’s decidable about weak memory models”. In: *21st European Symposium on Programming*. Vol. 7211. LNCS. Springer, 2012, pp. 24–46.

[BAM07] S. Burckhardt, R. Alur, and M. M. K. Martin. “CheckFence: checking consistency of concurrent data types on relaxed memory models”. In: *Programming Language Design and Implementation*. ACM, 2007, pp. 12–21.

[BBR10] J. Barnat, L. Brim, and P. Rockai. “Parallel partial order reduction with topological sort proviso”. In: *8th IEEE International Conference on Software Engineering and Formal Methods*. IEEE Computer Society, 2010, pp. 222–231.

[BDM13] A. Bouajjani, E. Derevenetc, and R. Meyer. “Checking and enforcing robustness against TSO”. In: *22nd European Symposium on Programming*. Vol. 7792. LNCS. Springer, 2013, pp. 533–553.

[BK08] C. Baier and J.-P. Katoen. *Principles of model checking*. MIT Press, 2008.

[BLL06] D. Bosnacki, S. Leue, and A. Lluch-Lafuente. “Partial-order reduction for general state exploring algorithms”. In: *12th SPIN Workshop*. Vol. 3925. LNCS. Springer, 2006, pp. 271–287.

[BM08] S. Burckhardt and M. Musuvathi. “Effective program verification for relaxed memory models”. In: *20th Computer Aided Verification*. Vol. 5123. LNCS. Springer, 2008, pp. 107–120.

[BMM11] A. Bouajjani, R. Meyer, and E. Möhlmann. “Deciding robustness against total store ordering”. In: *38th International Colloquium on Automata, Languages and Programming*. Vol. 6756. LNCS. Springer, 2011, pp. 428–440.

[BMP08] S. Baswana, S. K. Mehta, and V. Powar. “Implied set closure and its application to memory consistency verification”. In: *20th Computer Aided Verification*. Vol. 5123. LNCS. Springer, 2008, pp. 94–106.

[Bou+15] A. Bouajjani, G. Calin, E. Derevenetc, and R. Meyer. “Lazy TSO reachability”. In: *18th Fundamental Approaches to Software Engineering*. Vol. 9033. LNCS. Springer, 2015, pp. 267–282.

[BP09] G. Boudol and G. Petri. “Relaxed memory models: an operational approach”. In: *36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2009, pp. 392–403.

[BSS11] J. Burnim, C. Stergiou, and K. Sen. “Sound and complete monitoring of sequential consistency for relaxed memory models”. In: *17th Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 6605. LNCS. Springer, 2011, pp. 11–25.

[Bur+12] S. Burckhardt, A. Gotsman, M. Musuvathi, and H. Yang. “Concurrent library correctness on the TSO memory model”. In: *21st European Symposium on Programming*. Vol. 7211. LNCS. Springer, 2012, pp. 87–107.

[Cal+13] G. Calin, E. Derevenetc, R. Majumdar, and R. Meyer. “A theory of partitioned global address spaces”. In: *33rd IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*. Vol. 24. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2013, pp. 127–139.

[CE81] E. M. Clarke and E. A. Emerson. “Design and synthesis of synchronization skeletons using branching-time temporal logic”. In: *Logics of Programs, Workshop, Yorktown Heights, New York*. Vol. 131. LNCS. Springer, 1981, pp. 52–71.

[CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, 1999.

[CKL04] E. Clarke, D. Kroening, and F. Lerda. “A tool for checking ANSI-C programs”. In: *10th Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 2988. LNCS. Springer, 2004, pp. 168–176.

[CMM13] K. E. Coons, M. Musuvathi, and K. S. McKinley. "Bounded partial-order reduction". In: *Object-Oriented Programming, Systems, Languages & Applications*. ACM, 2013, pp. 833–848.

[Der15] E. Derevenetc. "Robustness against relaxed memory models". PhD thesis. University of Kaiserslautern, 2015.

[Dic09] D. Dice. *A race in LockSupport park() arising from weak memory models*. https://blogs.oracle.com/dave/entry/a_race_in_locksupport_park. 2009.

[Dij65] E. W. Dijkstra. "Solution of a problem in concurrent programming control". In: *Communications of ACM* 8.9 (1965), p. 569.

[DM14] E. Derevenetc and R. Meyer. "Robustness against Power is PSPACE-complete". In: *41st International Colloquium on Automata, Languages and Programming*. Vol. 8573. LNCS. Springer, 2014, pp. 158–170.

[DSD14] J. Derrick, G. Smith, and B. Dongol. "Verifying linearizability on TSO architectures". In: *11th Integrated Formal Methods*. Vol. 8739. LNCS. Springer, 2014, pp. 341–356.

[FG05] C. Flanagan and P. Godefroid. "Dynamic partial-order reduction for model checking software". In: *32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2005, pp. 110–121.

[Fin87] A. Finkel. "A generalization of the procedure of Karp and Miller to well structured transition systems". In: *14th International Colloquium on Automata, Languages and Programming*. Vol. 267. LNCS. Springer, 1987, pp. 499–508.

[FLR98] M. Frigo, C. E. Leiserson, and K. H. Randall. "The Implementation of the Cilk-5 Multithreaded Language". In: *Programming Language Design and Implementation*. ACM, 1998, pp. 212–223.

[FS01] A. Finkel and P. Schnoebelen. "Well-structured transition systems everywhere!" In: *Theoretical Computer Science* 256.1-2 (2001), pp. 63–92.

[Fur+14] F. Furbach, R. Meyer, K. Schneider, and M. Senftleben. "Memory model-aware testing - a unified complexity analysis". In: *14th International Conference on Application of Concurrency to System Design*. IEEE Computer Society, 2014, pp. 92–101.

[GHV09] J. Geldenhuys, H. Hansen, and A. Valmari. "Exploring the scope for partial order reduction". In: *7th International Symposium on Automated Technology for Verification and Analysis*. Vol. 5799. LNCS. Springer, 2009, pp. 39–53.

[GK97] P. B. Gibbons and E. Korach. “Testing shared memories”. In: *SIAM Journal of Computing* 26.4 (1997), pp. 1208–1244.

[GMY12] A. Gotsman, M. Musuvathi, and H. Yang. “Show no weakness: sequentially consistent specifications of TSO libraries”. In: *26th International Symposium on Distributed Computing.* Vol. 7611. LNCS. Springer, 2012, pp. 31–45.

[God96] P. Godefroid. “Partial-order methods for the verification of concurrent systems - an approach to the state-explosion problem”. PhD thesis. University of Liège, 1996.

[GRB06] G. Geeraerts, J.-F. Raskin, and L. Van Begin. “Expand, enlarge and check: new algorithms for the coverability problem of WSTS”. In: *Journal of Computer and System Sciences* 72.1 (2006), pp. 180–203.

[Gue+07] G. Gueta, C. Flanagan, E. Yahav, and M. Sagiv. “Cartesian partial-order reduction”. In: *14th SPIN Workshop.* Vol. 4595. LNCS. Springer, 2007, pp. 95–112.

[Han+04] S. Hangal, D. Vahia, C. Manovit, J.-Y. J. Lu, and S. Narayanan. “TSOtool: A program for verifying memory systems using the memory consistency model”. In: *31st International Symposium on Computer Architecture.* IEEE Computer Society, 2004, pp. 114–123.

[HKV97] L. Higham, J. Kawash, and N. Verwaal. “Defining and comparing memory consistency models”. In: *10th Parallel and Distributed Computing Systems.* ISCA, 1997, pp. 349–356.

[Hol97] Gerard J. Holzmann. “The Model Checker SPIN”. In: *IEEE Transactions on Software Engineering* 23.5 (1997), pp. 279–295.

[HR06] T. Q. Huynh and A. Roychoudhury. “A memory model sensitive checker for C#”. In: *14th International Symposium on Formal Methods.* Vol. 4085. LNCS. Springer, 2006, pp. 476–491.

[HS08] M. Herlihy and N. Shavit. *The art of multiprocessor programming.* Morgan Kaufmann, 2008.

[Int06] Intel. *Intel 64 and IA-32 architectures software developer's manual.* `http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`. 2006–.

[Koz77] D. Kozen. “Lower bounds for natural proof systems”. In: *Foundations of Computer Science.* IEEE Computer Society, 1977, pp. 254–266.

[KR08] H. Kastenberg and A. Rensink. “Dynamic partial order reduction using probe sets”. In: *19th International Conference on Concurrency Theory*. Vol. 5201. LNCS. Springer, 2008, pp. 233–247.

[KR88] B. W. Kernighan and D. Ritchie. *The C programming language, second edition*. Prentice-Hall, 1988.

[KVY10] M. Kuperstein, M. Vechev, and E. Yahav. “Automatic inference of memory fences”. In: *10th Formal Methods in Computer-Aided Design*. IEEE Computer Society, 2010, pp. 111–119.

[KVY11] M. Kuperstein, M. T. Vechev, and E. Yahav. “Partial-coherence abstractions for relaxed memory models”. In: *32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2011, pp. 187–198.

[KWG09] V. Kahlon, C. Wang, and A. Gupta. “Monotonic partial order reduction: an optimal symbolic partial order reduction technique”. In: *20th Computer Aided Verification*. Vol. 5643. LNCS. Springer, 2009, pp. 398–413.

[LA04] C. Lattner and V. S. Adve. “LLVM: A compilation framework for lifelong program analysis & transformation”. In: *2nd IEEE / ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society, 2004, pp. 75–88.

[Laa+13] A. Laarman, E. Pater, J. van de Pol, and M. Weber. “Guard-based partial-order reduction”. In: *20th SPIN Workshop*. Vol. 7976. LNCS. Springer, 2013, pp. 227–245.

[Lam79] L. Lamport. “How to make a multiprocessor computer that correctly executes multiprocess programs”. In: *IEEE Transactions on Computers* 28.9 (1979), pp. 690–691.

[Lam83] L. Lamport. “What good is temporal logic?” In: *IFIP Congress on Information Processing*. 1983, pp. 657–668.

[Lam87] L. Lamport. “A Fast Mutual Exclusion Algorithm”. In: *ACM Transactions on Computer Systems* 5.1 (1987), pp. 1–11.

[Lin14] A. Linden. “On the verification of programs on relaxed memory models”. PhD thesis. Université de Liège, 2014.

[Lip76] R. Lipton. *The reachability problem requires exponential space*. Tech. rep. 62. Yale University, 1976.

[Liu+12] F. Liu, N. Nedev, N. Prisadnikov, M. T. Vechev, and E. Yahav. “Dynamic synthesis for relaxed memory models”. In: *Programming Language Design and Implementation*. ACM, 2012, pp. 429–440.

[Luc01] V. Luchangco. “Memory consistency models for high performance distributed computing”. PhD thesis. Massachusetts Institute of Technology, 2001.

[LW11] A. Linden and P. Wolper. “A verification-based approach to memory fence insertion in relaxed memory systems”. In: *18th SPIN Workshop.* Vol. 6823. LNCS. Springer, 2011, pp. 144–160.

[LW13] A. Linden and P. Wolper. “A verification-based approach to memory fence insertion in PSO memory systems”. In: *19th Tools and Algorithms for the Construction and Analysis of Systems.* Vol. 7795. LNCS. Springer, 2013, pp. 339–353.

[Maz86] A. W. Mazurkiewicz. “Trace theory”. In: *Advances in Petri Nets 1986, Part II: Relationships to Other Models of Concurrency.* Vol. 255. LNCS. Springer, 1986, pp. 279–324.

[Mil89] R. Milner. *Communication and concurrency.* PHI Series in Computer Science. Prentice Hall, 1989.

[MP09] J. V. Meulen and C. Pecheur. “Combining partial order reduction with bounded model checking”. In: *32nd Communicating Process Architectures Conference.* Vol. 67. Concurrent Systems Engineering Series. IOS Press, 2009, pp. 29–48.

[MP92] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems - specification.* Springer, 1992.

[MP95] Z. Manna and A. Pnueli. *Temporal verification of reactive systems - safety.* Springer, 1995.

[OSS09a] S. Owens, S. Sarkar, and P. Sewell. “A better x86 memory model: x86-TSO”. In: *Theorem Proving in Higher Order Logics.* Vol. 5674. LNCS. Springer, 2009, pp. 391–407.

[OSS09b] S. Owens, S. Sarkar, and P. Sewell. *A better x86 memory model: x86-TSO (extended version).* Tech. rep. CL-TR-745. University of Cambridge, 2009.

[Owe10] S. Owens. “Reasoning about the implementation of concurrency abstractions on x86-TSO”. In: *24th European Conference on Object-Oriented Programming.* Vol. 6183. LNCS. Springer, 2010, pp. 478–503.

[PD95] S. Park and D. L. Dill. “An executable specification, analyzer and verifier for RMO (Relaxed Memory Order)”. In: *SPAA: Annual ACM Symposium on Parallel Algorithms and Architectures.* ACM, 1995.

[Pel93] D. Peled. “All from one, one for all: on model checking using representatives”. In: *5th Computer Aided Verification*. Vol. 697. LNCS. Springer, 1993, pp. 409–423.

[Pel96] D. Peled. “Combining partial order reductions with on-the-fly model-checking”. In: *Formal Methods in System Design* 8.1 (1996), pp. 39–64.

[Pnu77] A. Pnueli. “The temporal logic of programs”. In: *Foundations of Computer Science*. IEEE Computer Society, 1977, pp. 46–57.

[PW97] D. Peled and T. Wilke. “Stutter-Invariant Temporal Properties are Expressible Without the Next-Time Operator”. In: *Information Processing Letters* 63.5 (1997), pp. 243–246.

[QS82] J.-P. Queille and J. Sifakis. “Specification and verification of concurrent systems in CESAR”. In: *International Symposium on Programming, 5th Colloquium*. Vol. 137. LNCS. Springer, 1982, pp. 337–351.

[Rac78] C. Rackoff. “The covering and boundedness problems for vector addition systems”. In: *Theoretical Computer Science* 6 (1978), pp. 223–231.

[Rei85] W. Reisig. *Petri nets: an introduction*. Springer-Verlag New York, Inc., 1985.

[Rod+15] C. Rodríguez, M. Sousa, S. Sharma, and D. Kroening. “Unfolding-based Partial Order Reduction”. In: *26th International Conference on Concurrency Theory*. Vol. 42. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015, pp. 456–469.

[Roy+06] A. Roy, S. Zeisset, C. J. Fleckenstein, and J. C. Huang. “Fast and generalized polynomial time memory consistency verification”. In: *18th Computer Aided Verification*. Vol. 4144. LNCS. Springer, 2006, pp. 503–516.

[Sew+10] P. Sewell, S. Sarkar, S. Owens, F. Zappa-Nardelli, and M. O. Myreen. “x86-TSO: a rigorous and usable programmer’s model for x86 multiprocessors”. In: *Communications of ACM* 53.7 (2010), pp. 89–97.

[Sie12] S. F. Siegel. “Transparent partial order reduction”. In: *Formal Methods in System Design* 40.1 (2012), pp. 1–19.

[SKH12] O. Saarikivi, K. Kähkönen, and K. Heljanko. “Improving dynamic partial order reductions for concolic testing”. In: *12th International Conference on Application of Concurrency to System Design*. IEEE Computer Society, 2012, pp. 132–141.

[SN04] R. C. Steinke and G. J. Nutt. “A unified theory of shared memory consistency”. In: *Journal of the ACM* 51.5 (2004), pp. 800–849.

[SS88] D. Shasha and M. Snir. “Efficient and correct execution of parallel programs that share memory”. In: *ACM Transactions on Programming Languages and Systems* 10.2 (1988), pp. 282–312.

[Str04] J. Strejček. “Linear temporal logic: expressiveness and model checking”. PhD thesis. Masaryk University in Brno, 2004.

[Sur+05] Z. Sura, X. Fang, C.-L. Wong, S. P. Midkiff, J. Lee, and D. A. Padua. “Compiler techniques for high performance sequentially consistent Java programs”. In: *10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* ACM, 2005, pp. 2–13.

[Val90] A. Valmari. “Stubborn sets for reduced state space generation”. In: *10th Applications and Theory of Petri Nets.* Vol. 483. LNCS. Springer, 1990.

[VH10] A. Valmari and H. Hansen. “Can stubborn sets be optimal?” In: *31st International Conference on Applications and Theory of Petri Nets.* Vol. 6128. LNCS. Springer, 2010, pp. 43–62.

[VZ11] V. Vafeiadis and F. Zappa-Nardelli. “Verifying fence elimination optimisations”. In: *18th International Symposium on Static Analysis.* Vol. 6887. LNCS. Springer, 2011, pp. 146–162.

[WG94] D. Weaver and T. Germond, eds. *The SPARC architecture manual version 9.* PTR Prentice Hall, 1994.

[Yan+04] Y. Yang, G. Gopalakrishnan, G. Lindstrom, and K. Slind. “Nemos: A framework for axiomatic and executable specifications of memory consistency models”. In: *18th International Parallel and Distributed Processing Symposium.* IEEE Computer Society, 2004.

[Yan+08] Y. Yang, X. Chen, G. Gopalakrishnan, and R. M. Kirby. “Efficient stateful dynamic partial order reduction”. In: *15th SPIN Workshop.* Vol. 5156. LNCS. Springer, 2008, pp. 288–305.

[YWY06] X. Yi, J. Wang, and X. Yang. “Stateful dynamic partial-order reduction”. In: *8th International Conference on Formal Engineering Methods.* Vol. 4260. LNCS. Springer, 2006, pp. 149–167.

Appendices

Appendix A

Detailed Proofs of Lemmas 6 and 11

A bit of preparation is required prior to proving Lemma 6. Namely, in the proof of Lemma 6 we rely on computations that delay flush events locally the least. Lemma 24 explains what are these computations.

Lemma 24. *Let $\alpha \in \mathcal{C}_{TSO}(\mathcal{R})$ and $t \in \mathit{TID}$. There exists $\ddot{\alpha} \in \mathcal{C}_{TSO}(\mathcal{R})$ such that $\rightarrow_{hb}(\alpha) = \rightarrow_{hb}(\ddot{\alpha})$ and, for all events $\mathtt{e}_{\mathtt{store}} \leftrightarrow \mathtt{e}_{\mathtt{flush}}$ within thread t, if $\ddot{\alpha}{\downarrow}t := \alpha_{\mathit{prefix}} \cdot \mathtt{e}_{\mathtt{store}} \cdot \alpha' \cdot \mathtt{e}_{\mathtt{flush}} \cdot \alpha_{\mathit{suffix}}$ then either*

(1) $\alpha' := \beta \cdot \mathtt{e}_{\mathtt{load}} \cdot \beta'$ and all events $\mathtt{e} \in \beta'$ are flushes,

or (2) all events $\mathtt{e} \in \alpha'$ are local assignments or conditionals.

Proof. Intuitively, the theorem states that flush events of thread t delayed past same-thread local events, may be delayed less without changing the happens-before relation of the computation. Local events are assignments, conditionals, and store events in the same thread.

Let $\alpha := \alpha_1 \cdot \mathtt{e}_{\mathtt{store}} \cdot \alpha_2 \cdot \mathtt{e} \cdot \alpha_3 \cdot \mathtt{e}_{\mathtt{flush}} \cdot \alpha_4$ such that $\mathtt{e}_{\mathtt{store}} \leftrightarrow \mathtt{e}_{\mathtt{flush}}$ are events of thread t, $\mathtt{e}$ is a local event in t and $\mathit{thread}(\mathtt{e}') \neq t$ for all events $\mathtt{e}' \in \alpha_3$.

We denote by $\alpha_0 := \alpha_1 \cdot \mathtt{e}_{\mathtt{store}} \cdot \alpha_2 \cdot \alpha_3 \cdot \mathtt{e}_{\mathtt{flush}} \cdot \mathtt{e} \cdot \alpha_4$ the TSO computation that first performs the flush $\mathtt{e}_{\mathtt{flush}}$ and then the event $\mathtt{e}$. Notice that since α_3 contains no events $\mathtt{e}'$ with $\mathit{thread}(\mathtt{e}') = t$, feasibility of computation α_0 is ensured and $\rightarrow_{hb}(\alpha) = \rightarrow_{hb}(\alpha_0)$ holds.

Starting with the last flush event in α, we use the above reordering of events $\mathtt{e}$ to locally delay flush events less. In the end we obtain computation $\ddot{\alpha}$ in which no flush event of thread t can be locally delayed less. ☐

Furthermore, in order to reference instructions of $\mathcal{R} \oplus \iota$ that the extension adds we give an alternative, more concrete, description for some of the transition sequences in the main text. Recall that variable `count` keeps track of the number of store instructions processed along ι.

If $cmd(inst_i) = \texttt{mem}[e] \leftarrow e'$, we said $\texttt{count}$ is incremented and instructions that remember the value and address in $ar_{\texttt{count}}$ and $vr_{\texttt{count}}$ are added.

$$\begin{tikzcd} \overline{q}_{i-1} \circ \arrow[r, "ar_{\texttt{count}} \leftarrow e"] & \circ \arrow[r, "vr_{\texttt{count}} \leftarrow e'"] & \circ\ \overline{q}_i \end{tikzcd} \tag{A.1}$$

If $cmd(inst_i) = r \leftarrow \texttt{mem}[e]$ we said instructions are added that load from memory only when a load from the simulated buffer is not possible. More precisely, if some $j \in [1, \texttt{count}]$ such that $ar_j = e$ is found, r is assigned the value of vr_j. Otherwise, the register r receives its value from the address $\widehat{e}$.

$$\begin{tikzcd}
\overline{q}_{i-1} \circ \arrow[r, "\texttt{check}\ ar_{\texttt{count}} \neq e"] \arrow[dd, "\texttt{check}\ ar_{\texttt{count}} = e"] & \circ & \cdots & \circ \arrow[r, "\texttt{check}\ ar_1 \neq e"] \arrow[d, "\texttt{check}\ ar_1 = e"] & \circ \arrow[r, "r \leftarrow \texttt{mem}{[e]}"] & \circ\ \overline{q}_i \\
 & & \cdots & \circ \arrow[urr, bend right, "r \leftarrow vr_1"'] & & \\
\circ \arrow[uurrrrr, bend right, "r \leftarrow vr_{\texttt{count}}"'] & & & & &
\end{tikzcd}$$

Alternatively, assuming $\overline{q}_{\text{check},i,\texttt{count}} := \overline{q}_{i-1}$, this can be stated as adding

$$\{(\overline{q}_{\text{check},i,\texttt{count}},\ \texttt{check}\ ar_{\texttt{count}} = e,\ \overline{q}_{\text{buf},i,\texttt{count}})\} \tag{A.2}$$
$$\uplus\ \{(\overline{q}_{\text{check},i,\texttt{count}},\ \texttt{check}\ ar_{\texttt{count}} \neq e,\ \overline{q}_{\text{check},i,\texttt{count}-1})\} \tag{A.3}$$
$$\uplus\ \{(\overline{q}_{\text{buf},i,\texttt{count}},\ r \leftarrow vr_{\texttt{count}},\ \overline{q}_i)\} \tag{A.4}$$
$$\vdots$$
$$\uplus\ \{(\overline{q}_{\text{check},i,1},\ \texttt{check}\ ar_1 = e,\ \overline{q}_{\text{buf},i,1})\} \tag{A.5}$$
$$\uplus\ \{(\overline{q}_{\text{check},i,1},\ \texttt{check}\ ar_1 \neq e,\ \overline{q}_{\text{mem},i})\} \tag{A.6}$$
$$\uplus\ \{(\overline{q}_{\text{buf},i,1},\ r \leftarrow vr_1,\ \overline{q}_i)\} \tag{A.7}$$

$$\uplus\ \{(\overline{q}_{\text{mem},i},\ r \leftarrow \texttt{mem}[e],\ \overline{q}_i)\} \tag{A.8}$$

We said that out of control state $\overline{q}_n$ we create a sequence of stores to flush the contents of the auxiliary registers and return to the code of the original thread.

$$\begin{tikzcd} \overline{q}_n \circ \arrow[r, "\texttt{mem}{[ar_1]} \leftarrow vr_1"] & \circ \cdots \circ \arrow[r, "\texttt{mem}{[ar_{\texttt{max}}]} \leftarrow vr_{\texttt{max}}"] & \circ\ dst(inst_n) \end{tikzcd}$$

Alternatively, we could have stated it as adding

$$\{(\overline{q}_n,\ \texttt{mem}[ar_1] \leftarrow vr_1,\ \overline{q}_{\text{flush},1})\} \tag{A.9}$$
$$\vdots$$
$$\uplus\ \{(\overline{q}_{\text{flush},\texttt{max}-1},\ \texttt{mem}[ar_{\texttt{max}}] \leftarrow vr_{\texttt{max}},\ dst(inst_n))\} \tag{A.10}$$

Furthermore, for all instructions $\mathit{inst} \in I_t$ with $\mathit{src}(\mathit{inst}) = \mathit{src}(\mathit{inst}_i)$ for some $i \in [1..n]$ and for which $\mathit{inst} \neq \mathit{inst}_i$ we added instructions that flush the stores buffered in the auxiliary registers and return to $\mathit{dst}(\mathit{inst})$.

$$\begin{tikzcd}
\overline{q}_i\ \circ \arrow[r, "{\texttt{mem}[ar_1] \leftarrow vr_1}"] & \circ\ \cdots\ \circ \arrow[r, "{\texttt{mem}[ar_{\texttt{count}}] \leftarrow vr_{\texttt{count}}}"] & \circ \arrow[r, "{\mathit{cmd}(\mathit{inst})}"] & \circ\ \mathit{dst}(\mathit{inst})
\end{tikzcd}$$

Alternatively, we could have stated it as adding

$$\{(\overline{q}_i,\ \texttt{mem}[ar_1] \leftarrow vr_1,\ \overline{q}_{\text{next},i,1})\} \tag{A.11}$$

$$\vdots$$

$$\uplus\ \{(\overline{q}_{\text{next},i,\texttt{count}-1},\ \texttt{mem}[ar_{\texttt{count}}] \leftarrow vr_{\texttt{count}},\ \overline{q}_{\text{next},i,\texttt{count}})\} \tag{A.12}$$

$$\uplus\ \{(\overline{q}_{\text{next},i,\texttt{count}},\ \mathit{cmd}(\mathit{inst}),\ \mathit{dst}(\mathit{inst}))\} \tag{A.13}$$

Finally, for all load instructions inst_i, where $i < n$, as well as out of $\overline{q}_1$ we added instructions that flush and fence the pair (ar_1, vr_1), make the remaining buffered stores in the auxiliary registers visible, and return to q. Here $q := \mathit{src}(\mathit{inst}_i)$ in the load case and $q := \mathit{dst}(\mathit{inst}_1)$ otherwise.

$$\begin{tikzcd}
\overline{q}_i\ \circ \arrow[r, "{\texttt{mem}[ar_1] \leftarrow vr_1}"] & \circ \arrow[r, "\texttt{mf}"] & \circ\ \cdots\ \circ \arrow[r, "{\texttt{mem}[ar_{\texttt{count}}] \leftarrow vr_{\texttt{count}}}"] & \circ\ q
\end{tikzcd}$$

Alternatively, we could have stated it as adding

$$\{(\overline{q}_i,\ \texttt{mem}[ar_1] \leftarrow vr_1,\ \overline{q}_{\text{fence},i})\} \tag{A.14}$$

$$\uplus\ \{(\overline{q}_{\text{fence},i},\ \texttt{mf},\ \overline{q}_{\text{orig},i,2})\} \tag{A.15}$$

$$\uplus\ \{(\overline{q}_{\text{orig},i,2},\ \texttt{mem}[ar_2] \leftarrow vr_2,\ \overline{q}_{\text{orig},i,3})\} \tag{A.16}$$

$$\vdots$$

$$\uplus\ \{(\overline{q}_{\text{orig},i,\texttt{count}},\ \texttt{mem}[ar_{\texttt{count}}] \leftarrow vr_{\texttt{count}},\ q)\} \tag{A.17}$$

We can now turn to the actual proof of Lemma 6.

Proof of Lemma 6. Assume that t is the thread of $\iota := \mathit{inst}_1 \cdot \ldots \cdot \mathit{inst}_n$, $\mathrm{X}_{\mathrm{TSO}}(\mathcal{R} \oplus \iota) := (E_\oplus, S_\oplus, \Delta_{\mathrm{X,TSO}}, s_\oplus, F_\oplus)$, I and Q are the instructions and states of $\mathcal{R}$, ADR and REG are registers and addresses used by $\mathcal{R}$, and $I_\oplus$ are the instructions I'_t of $\mathcal{R} \oplus \iota$ as described in Section 3.1.

A direct result of Lemmas 24 and 4 is that TSO computations of $\mathcal{R}$ that delay flushes of t locally the least reach all the states in the set $\mathit{Reach}_{\mathrm{TSO}}(\mathcal{R})$. Assume $\alpha \in \mathcal{C}_{\mathrm{TSO}}(\mathcal{R})$ is a computation where flushes of t are delayed locally the least as Lemma 24 describes and let $s_0, \ldots, s_m \in S_{\mathrm{TSO}}$ for some $m \in \mathbb{N}$ be all the states along the transition sequence $s_0 \xrightarrow{\alpha} s$, i.e., $s_0 := s_0$ and $s_m := s$. Also, for all $k \in [0..m]$, let α_k denote prefixes of α with $s_0 \xrightarrow{\alpha_k} s_k$.

We prove by induction over state indexes $k \in [0..m]$ that there exist prefixes β_k of $\beta \in \mathcal{C}_{\mathrm{TSO}}(\mathcal{R} \oplus \iota)$ and states $s'_0, \ldots, s'_m \in S_\oplus$ along $s_\oplus \xrightarrow{\beta} s' \in \Delta^*_{\mathrm{X,TSO}}$ with $s'_0 := s_\oplus$ and $s'_m := s'$ such that the following invariants hold:

I-0 $s_0 \xrightarrow{\alpha_k} (\mathsf{pc}, \mathsf{val}, \mathsf{buf})$ and $s_\oplus \xrightarrow{\beta_k} (\mathsf{pc}', \mathsf{val}', \mathsf{buf}')$.

I-1 If pc and pc' differ, they only differ for thread t. If $\mathsf{pc}(t) \neq \mathsf{pc}'(t)$, then $\mathsf{pc}(t) = \mathit{dst}(\mathit{inst}_i)$ and $\mathsf{pc}'(t) = \overline{q}_i$ for some $i \in [1..n-1]$.

I-2 $\mathsf{val}'(a) = \mathsf{val}(a)$ for all $a \in \mathsf{ADR} \cup \mathsf{REG}$.

I-3 buf and buf' differ at most for t. If $\mathsf{buf}(t) \neq \mathsf{buf}'(t)$, then $\mathsf{pc}'(t) = \overline{q}_i$ for some $i \in [1..n-1]$ and $\mathsf{buf}(t) = (\widehat{ar_{\mathtt{count}}}, \widehat{vr_{\mathtt{count}}}) \cdots (\widehat{ar_1}, \widehat{vr_1}) \cdot \mathsf{buf}'(t)$ where $\mathtt{count}$ stores are seen along ι from $\mathit{src}(\mathit{inst}_1)$ to $\mathit{dst}(\mathit{inst}_i)$.

For the **induction base case** $k = 0$, $\alpha_0 = \epsilon$, $s_0 = s_0$, $\mathsf{pc} = \mathsf{pc}_0$, $\mathsf{val} = \mathsf{val}_0$, and $\mathsf{buf} = \mathsf{buf}_0$. Then, for $\beta_0 := \epsilon$ and $s'_0 = s_\oplus$, invariants **I-0...3** hold.

For the **induction step case**, assume that invariants **I-0...3** hold for $k < m$ and that $s_k \xrightarrow{\mathsf{e}} s_{k+1} := (\mathsf{pc}_+, \mathsf{val}_+, \mathsf{buf}_+)$ for some $\mathsf{e} \in E$. We use a case distinction over possible events e to define the prefix β_{k+1} such that $s'_0 \xrightarrow{\beta_{k+1}} s'_{k+1} := (\mathsf{pc}'_+, \mathsf{val}'_+, \mathsf{buf}'_+)$ and invariants **I-0...3** hold for $k+1$.

If $\underline{\mathit{thread}(\mathsf{e}) := t' \neq t}$ it means $\mathit{inst}(\mathsf{e}) \in I_\oplus$ is enabled in $\mathsf{pc}'(t')$, so there exist $\mathsf{e}' \in E_\oplus$ and $s'_{k+1} \in S_\oplus$ such that $\mathit{inst}(\mathsf{e}') := \mathit{inst}(\mathsf{e})$ and $(s'_k, \mathsf{e}', s'_{k+1}) \in \Delta_{\mathrm{X,TSO}}$ in $\mathrm{X}_{\mathrm{TSO}}(\mathcal{R} \oplus \iota)$. We define $\beta_{k+1} := \beta_k \cdot \mathsf{e}'$ and find that, by the $\Delta_{\mathrm{X,TSO}}$ semantics (Figure 2.19) and under the assumption that invariants **I-0...3** hold for k, invariants **I-0...3** also hold for $k+1$.

If $\underline{\mathit{thread}(\mathsf{e}) = t}$ we make the following case distinction over e and $\mathsf{pc}'(t)$.

$\boxed{1}$ "e is a flush event." This first case deals with the possibility that a store operation is flushed. Depending on whether $\mathsf{buf}'(t) \neq \epsilon$, we either flush the oldest address-value pair of $\mathsf{buf}'(t)$ or the first address-value auxiliary registers pair. By Lemma 24, the later case can only happen when $\mathsf{pc}'(t) = \overline{q}_i$ for some $i \in [2..n-1]$ and inst_i performs a load or $i = 1$.

If $\mathsf{buf}'(t) \neq \epsilon$ we flush the oldest write access buffered. Let $\mathsf{e}_{\mathtt{flush}} \in E_\oplus$ and $s'_{k+1} \in S_\oplus$ such that, according to rule (WM), $(s'_k, \mathsf{e}_{\mathtt{flush}}, s'_{k+1}) \in \Delta_{\mathrm{X,TSO}}$. We define $\beta_{k+1} := \beta_k \cdot \mathsf{e}_{\mathtt{flush}}$ and invariants **I-0...3** hold for $k+1$ since

(0) **I-0,3** hold for k so $s_0 \xrightarrow{\alpha_{k+1}} s_{k+1}$ and $s'_0 \xrightarrow{\beta_{k+1}} s'_{k+1}$, implying invariant **I-0** holds for $k+1$.

(1) **I-1** holds for k, $\mathsf{pc}_+(t) = \mathsf{pc}(t)$, and $\mathsf{pc}'_+(t) = \mathsf{pc}'(t)$, so invariant **I-1** holds for $k+1$.

(2) **I-2,3** hold for k, so events e and $\mathsf{e}_{\mathtt{flush}}$ update the same address by a same value and invariant **I-2** holds for $k+1$.

(3) **I-3** holds for k and events e and $\mathsf{e}_{\mathtt{flush}}$ remove one address-value pair from both $\mathsf{buf}(t)$ and $\mathsf{buf}'(t)$, so invariant **I-3** holds for $k+1$.

Otherwise, $\mathsf{buf}'(t) = \epsilon$ and $\mathtt{count}$ stores are encountered from $\mathit{src}(\mathit{inst}_1)$ to $\mathsf{pc}'(t) = \overline{q}_i$ for some $i \in [1..n-1]$. Then $\mathsf{buf}(t) = (\widehat{ar_{\mathtt{count}}}, \widehat{vr_{\mathtt{count}}}) \cdot \ldots \cdot (\widehat{ar_1}, \widehat{vr_1})$ and, by Lemma 24, we know inst_i is either the first store inst_1 of ι or a load. Either way, let $\mathsf{e}_1, \ldots, \mathsf{e}_{\mathtt{count}}, \mathsf{e}_{\mathtt{flush}}, \mathsf{e}_{\mathtt{fence}} \in E_\oplus$ match equations (A.14–A.17) in the extension and $s'_{k+1} \in S_\oplus$ such that events e_j are, for all $j \in [1..\mathtt{count}]$, the buffering events for the stores (A.14,A.16–A.17), $\mathsf{e}_{\mathtt{flush}}$ is the flush event for the store (A.14), $\mathsf{e}_{\mathtt{fence}}$ is the event for the fence (A.15), and $s'_k \xrightarrow{\mathsf{e}_1 \cdot \mathsf{e}_{\mathtt{flush}} \cdot \mathsf{e}_{\mathtt{fence}} \cdot \mathsf{e}_2 \cdot \ldots \cdot \mathsf{e}_{\mathtt{count}}} s'_{k+1} \in \Delta^*_{\mathrm{X,TSO}}$ according to rules (ST,MEM,F) in Figure 2.19. We then define $\beta_{k+1} := \beta_k \cdot \mathsf{e}_1 \cdot \mathsf{e}_{\mathtt{flush}} \cdot \mathsf{e}_{\mathtt{fence}} \cdot \mathsf{e}_2 \cdot \ldots \cdot \mathsf{e}_{\mathtt{count}} \cdot \mathsf{e}$ and find that invariants **I-0...3** hold for $k+1$ since

(0) **I-0,3** hold for k so $s_0 \xrightarrow{\alpha_{k+1}} s_{k+1}$ and $s'_0 \xrightarrow{\beta_{k+1}} s'_{k+1}$, i.e. invariant **I-0** holds for $k+1$.

(1) **I-1** holds for k and $\mathsf{pc}_+(t) = q = \mathsf{pc}'_+(t)$, where $q := \mathit{src}(\mathit{inst}_i)$ if inst_i is a load and $q := \mathit{dst}(\mathit{inst}_1)$ otherwise, so invariant **I-1** holds for $k+1$.

(2) **I-2,3** hold for k, events e and $\mathsf{e}_{\mathtt{flush}}$ update the same address by the same value and, since the other events do not update any address, invariant **I-2** holds for $k+1$.

(3) **I-3** holds for k, and events $\mathsf{e}_2, \ldots, \mathsf{e}_{\mathtt{count}}$ place the corresponding address-value pairs that match $\mathsf{buf}_+(t)$ into $\mathsf{buf}'_+(t)$, so invariant **I-3** holds for $k+1$.

$\boxed{2}$ "e is not a flush event, $\mathsf{pc}'(t) = \overline{q}_i$ for $i \in [1..n-1]$, $\mathit{inst}(\mathsf{e}) \neq \mathit{inst}_{i+1}$." Event e corresponds to an instruction that does not follow ι. Then, events for instructions (A.11–A.13) place the auxiliary address-value pairs into $\mathsf{buf}'_+(t)$ and then perform $\mathit{cmd}(\mathit{inst}(\mathsf{e}))$. Let $\mathsf{e}_1, \ldots, \mathsf{e}_{\mathtt{count}}, \mathsf{e}' \in E_\oplus$ and $s'_{k+1} \in S_\oplus$ such that e_j are, for $j \in [1..\mathtt{count}]$, the buffering events for stores (A.11–A.12), e' is the event of instruction (A.13), and $s'_k \xrightarrow{\mathsf{e}_1 \cdot \ldots \cdot \mathsf{e}_{\mathtt{count}} \cdot \mathsf{e}'} s'_{k+1} \in \Delta^*_{\mathrm{X,TSO}}$, according to the Figure 2.19 rules. We define $\beta_{k+1} := \beta_k \cdot \mathsf{e}_1 \cdot \ldots \cdot \mathsf{e}_{\mathtt{count}} \cdot \mathsf{e}'$ and find that invariants **I-0...3** hold for $k+1$ since

(0) **I-0** holds for k so $s_0 \xrightarrow{\alpha_{k+1}} s_{k+1}$ and $s'_0 \xrightarrow{\beta_{k+1}} s'_{k+1}$, i.e. invariant **I-0** holds for $k+1$.

(1) **I-1** holds for k and $\mathsf{pc}_+(t) = \mathit{dst}(\mathit{inst}(\mathsf{e})) = \mathsf{pc}'_+(t)$, so invariant **I-1** holds for $k+1$.

(2) **I-2** holds for k and the events e and e' update at most one REG register by the same value, so invariant **I-2** holds for $k+1$.

(3) **I-3** holds for k, the buffering store events $\mathsf{e}_1, \ldots, \mathsf{e}_{\mathtt{count}}$ make the address-value pairs of the auxiliary registers explicit in $\mathsf{buf}'_+(t)$, and if events e and e' are buffering events for stores then they add the same address-value pair, so invariant **I-3** holds for $k+1$.

$\boxed{3}$ "$\mathit{inst}(\mathsf{e})$ performs a store and $\boxed{2}$ fails." We analyze the following two subcases depending on the value of $\mathsf{pc}'(t)$.

$\boxed{3a}$ "$\mathsf{pc}'(t) = \overline{q}_{i-1}$ for some $i \in [1..n-1]$." Since $\boxed{2}$ does not hold, $\mathit{inst}(\mathsf{e}) = \mathit{inst}_i$ and auxiliary registers track the store inst_i. Let $\mathsf{e}_a, \mathsf{e}_v \in E_\oplus$ be events for the instructions in (A.1) and $s'_{k+1} \in S_\oplus$ such that $s'_k \xrightarrow{\mathsf{e}_a \cdot \mathsf{e}_v} s'_{k+1} \in \Delta^*_{\mathrm{X,TSO}}$ according to the $\Delta_{\mathrm{X,TSO}}$ rule for local assignments. We define $\beta_{k+1} := \beta_k \cdot \mathsf{e}_a \cdot \mathsf{e}_v$ and find that invariants **I-0...3** hold for $k+1$ since

(0) **I-0** holds for k so $s_0 \xrightarrow{\alpha_{k+1}} s_{k+1}$ and $s'_0 \xrightarrow{\beta_{k+1}} s'_{k+1}$, i.e. invariant **I-0**

holds for $k+1$.

(1) **I-1** holds for k, $\mathsf{pc}_+(t) = \mathit{dst}(\mathit{inst}_i)$, and $\mathsf{pc}'_+(t) = \overline{q}_i$, so invariant **I-1** holds for $k+1$.

(2) **I-2** holds for k and no memory changes occurred outside of auxiliary registers, so invariant **I-2** holds for $k+1$.

(3) **I-3** holds for k and $(\widehat{ar_{\mathtt{count}}}, \widehat{vr_{\mathtt{count}}})$ matches the address-value pair added by e to $\mathsf{buf}_+(t)$, so invariant **I-3** holds for $k+1$.

$\boxed{\text{3b}}$ "$\mathsf{pc}'(t) = \mathsf{pc}(t) \neq \mathit{src}(\mathit{inst}_1)$." This case is similar to having $\mathit{thread}(\mathsf{e}) \neq t$ since $\mathit{inst}(\mathsf{e}) \in I_\oplus$. Then there exist $\mathsf{e}' \in E_\oplus$ and $s'_{k+1} \in S_\oplus$ such that $\mathit{inst}(\mathsf{e}') = \mathit{inst}(\mathsf{e})$ and $(s'_k, \mathsf{e}', s'_{k+1}) \in \Delta_{\mathrm{X,TSO}}$ in $\mathrm{X}_{\mathrm{TSO}}(\mathcal{R} \oplus \iota)$. We define $\beta_{k+1} := \beta_k \cdot \mathsf{e}'$ and find that, by the $\Delta_{\mathrm{X,TSO}}$ semantics (Figure 2.19), invariants **I-0...3** continue to hold for $k+1$.

$\boxed{4}$ "$\mathit{inst}(\mathsf{e})$ performs a load and $\boxed{2}$ fails." We analyze the following subcases depending on the value of $\mathsf{pc}'(t)$.

$\boxed{\text{4a}}$ "$\mathsf{pc}'(t) = \overline{q}_{i-1}$ for some $i \in [1..n-1]$." Since $\boxed{2}$ does not hold, $\mathit{inst}(\mathsf{e}) = \mathit{inst}_i$ and we use (A.4–A.7,A.8) to load from e only when no register ar_j matches e for any $j \in [1..\mathtt{count}]$.

If there exists a largest $j \in [1..\mathtt{count}]$ such that $ar_j = e$ then r will take its value from the auxiliary register vr_j. Let $\mathsf{e}_{\mathtt{count}}, \ldots, \mathsf{e}_j, \mathsf{e}_{\mathtt{assign}} \in E_\oplus$ and $s'_{k+1} \in S_\oplus$ such that e_k are, for all $k \in [j+1..\mathtt{count}]$, the events for negative conditional checks (A.3,A.6), e_j is the event for the earliest positive conditional check (A.2,A.5), $\mathsf{e}_{\mathtt{assign}}$ is the event for an instruction (A.4,A.7), and $s'_k \xrightarrow{\mathsf{e}_{\mathtt{count}} \cdot \ldots \cdot \mathsf{e}_j \cdot \mathsf{e}_{\mathtt{assign}}} s'_{k+1} \in \Delta^*_{\mathrm{X,TSO}}$ according to the rules for conditionals and local assignments in $\Delta_{\mathrm{X,TSO}}$. We define $\beta_{k+1} := \beta_k \cdot \mathsf{e}_{\mathtt{count}} \cdot \ldots \cdot \mathsf{e}_j \cdot \mathsf{e}_{\mathtt{assign}}$ and find that the invariants **I-0...3** hold for $k+1$ since

(0) **I-0** holds for k so $s_0 \xrightarrow{\alpha_{k+1}} s_{k+1}$ and $s'_0 \xrightarrow{\beta_{k+1}} s'_{k+1}$, i.e. invariant **I-0** holds for $k+1$.

(1) **I-1** holds for k, $\mathsf{pc}_+(t) = \mathit{dst}(\mathit{inst}_i)$, and $\mathsf{pc}'_+(t) = \overline{q}_i$, so invariant **I-1** holds for $k+1$.

(2) **I-2** holds for k, both e and $\mathsf{e}_{\mathtt{assign}}$ update r by the same value, and no other event $\mathsf{e}_{\mathtt{count}}, \ldots, \mathsf{e}_j$ changes any address, so invariant **I-2** holds for $k+1$.

(3) **I-3** holds for k and no event alters buffer contents, so invariant **I-3** holds for $k+1$.

Otherwise, $ar_j \neq e$ holds for all $j \in [1..\mathtt{count}]$ and the register r will take its value from the address indicated by e. Namely, let $\mathsf{e}_{\mathtt{count}}, \ldots, \mathsf{e}_1, \mathsf{e}_{\mathtt{load}} \in E_\oplus$ and $s'_{k+1} \in S_\oplus$ such that e_k are, for all $k \in [1..\mathtt{count}]$, the events for negative conditional checks (A.3,A.6), $\mathsf{e}_{\mathtt{load}}$ is the event for instruction (A.8), and $s'_k \xrightarrow{\mathsf{e}_{\mathtt{count}} \cdot \ldots \cdot \mathsf{e}_1 \cdot \mathsf{e}_{\mathtt{load}}} s'_{k+1} \in \Delta^*_{\mathrm{X,TSO}}$ according to the rule for conditionals in $\Delta_{\mathrm{X,TSO}}$ and (LB/LM). We define $\beta_{k+1} := \beta_k \cdot \mathsf{e}_{\mathtt{count}} \cdot \ldots \cdot \mathsf{e}_1 \cdot \mathsf{e}_{\mathtt{load}}$ and find that invariants **I-0...3** hold for $k+1$:

(0) **I-0** holds for k so $s_0 \xrightarrow{\alpha_{k+1}} s_{k+1}$ and $s'_0 \xrightarrow{\beta_{k+1}} s'_{k+1}$, i.e. invariant **I-0** holds for $k+1$.

(1) **I-1** holds for k, $\mathsf{pc}_+(t) = dst(inst_i)$, and $\mathsf{pc}'_+(t) = \overline{q}_i$, so invariant **I-1** holds for $k+1$.
(2) **I-2** holds for k, both e and $\mathsf{e}_{\mathtt{load}}$ update r by the same value, and no other event $\mathsf{e}_{\mathtt{count}}, \ldots, \mathsf{e}_1$ changes any address, so invariant **I-2** holds for $k+1$.
(3) **I-3** holds for k and no event alters buffer contents, so invariant **I-3** holds for $k+1$.
$\boxed{4\text{b}}$ "$\mathsf{pc}'(t) = \overline{q}_{n-1}$." Since $\boxed{2}$ does not hold, $inst(\mathsf{e}) = inst_n$. Furthermore, because $\mathtt{count} = \mathtt{max}$, additionally to performing the events that simulate the load behavior as in subcase $\boxed{4\text{a}}$, the extension returns to the original program flow using events for (A.9–A.10) and makes the auxiliary registers address-value pairs explicit in $\mathsf{buf}'_+(t)$.

Let $\mathsf{e}'_1, \ldots, \mathsf{e}'_{\mathtt{max}} \in E_\oplus$ and $s'_{k+1} \in S_\oplus$ such that e'_k are, for all $k \in [1..\mathtt{max}]$, the buffering events for stores (A.9,A.10), and $s''_{k+1} \xrightarrow{\mathsf{e}'_1 \cdot \ldots \cdot \mathsf{e}'_{\mathtt{max}}} s'_{k+1} \in \Delta^*_{\mathrm{X,TSO}}$ according to (LS) from Figure 2.19, with s''_{k+1} being notation for s'_{k+1} from $\boxed{4\text{a}}$. We define $\beta_{k+1} := \beta'_{k+1} \cdot \mathsf{e}'_1 \cdot \ldots \cdot \mathsf{e}'_{\mathtt{max}}$, where β'_{k+1} is notation for β_{k+1} from $\boxed{4\text{a}}$, and find that the invariants **I-0...3** hold for $k+1$ since

(0) **I-0** holds for k so $s_0 \xrightarrow{\alpha_{k+1}} s_{k+1}$ and $s'_0 \xrightarrow{\beta_{k+1}} s'_{k+1}$, i.e. invariant **I-0** holds for $k+1$.
(1) **I-1** holds for k and $\mathsf{pc}_+(t) = dst(inst_n) = \mathsf{pc}'_+(t)$, so invariant **I-1** holds for $k+1$.
(2) **I-2** holds for k, both events e and $\mathsf{e}_{\mathtt{load}}$ update r by the same value, and no other event $\mathsf{e}_{\mathtt{count}}, \ldots, \mathsf{e}_1, \mathsf{e}'_1, \ldots, \mathsf{e}'_{\mathtt{max}}$ changes any address, so invariant **I-2** holds for $k+1$.
(3) **I-3** holds for k and events $\mathsf{e}'_1, \ldots, \mathsf{e}'_{\mathtt{max}}$ place the corresponding address-value pairs that match $\mathsf{buf}_+(t)$ into $\mathsf{buf}'_+(t)$, so invariant **I-3** holds for $k+1$.
$\boxed{4\text{c}}$ "$\mathsf{pc}'(t) = \mathsf{pc}(t)$." This case is similar to $\boxed{3\text{b}}$. Let $\mathsf{e}' \in E_\oplus$ and $s'_{k+1} \in S_\oplus$ such that $inst(\mathsf{e}') = inst(\mathsf{e})$ and $(s'_k, \mathsf{e}', s'_{k+1}) \in \Delta_{\mathrm{X,TSO}}$ in $\mathrm{X}_{\mathrm{TSO}}(\mathcal{R} \oplus \iota)$. We define $\beta_{k+1} := \beta_k \cdot \mathsf{e}'$ and find that, by the $\Delta_{\mathrm{X,TSO}}$ semantics (Figure 2.19), the invariants **I-0...3** hold for $k+1$.

$\boxed{5}$ "$inst(\mathsf{e})$ is an assignment, conditional, or memory fence and $\boxed{2}$ fails." We analyze the following subcases.
$\boxed{5\text{a}}$ "$\mathsf{pc}'(t) = \overline{q}_{i-1}$ for $i \in [1..n-1]$." Since $\boxed{2}$ does not hold, $inst(\mathsf{e}) = inst_i$ is either a conditional or an assignment.

If $cmd(inst_i) = r \leftarrow e$ let $\mathsf{e}' \in E_\oplus$ and $s'_{k+1} \in S_\oplus$ such that $inst(\mathsf{e}') = (q_{i-1}, r \leftarrow e, q_i)$ and $(s'_k, \mathsf{e}', s'_{k+1}) \in \Delta_{\mathrm{X,TSO}}$ by the $\Delta_{\mathrm{X,TSO}}$ rule for local assignments. We define $\beta_{k+1} := \beta_k \cdot \mathsf{e}'$ and find that the invariants **I-0...3** hold for $k+1$ since

(0) **I-0** holds for k so $s_0 \xrightarrow{\alpha_{k+1}} s_{k+1}$ and $s'_0 \xrightarrow{\beta_{k+1}} s'_{k+1}$, i.e. invariant **I-0** holds for $k+1$.
(1) **I-1** holds for k, $\mathsf{pc}_+(t) = dst(inst_i)$, and $\mathsf{pc}'_+(t) = \overline{q}_i$, so invariant **I-1** holds for $k+1$.

(2) **I-2** holds for k and e is evaluated the same by both e and e', so the register r is updated by the same value and invariant **I-2** holds for $k+1$.
(3) **I-3** holds for k and no event alters buffer contents, so invariant **I-3** holds for $k+1$.

Otherwise, $\mathit{cmd}(\mathit{inst}_i) = \texttt{check}\ e$. Let $\mathsf{e}' \in E_{\oplus}$ and $s'_{k+1} \in S_{\oplus}$ such that $\mathit{inst}(\mathsf{e}') = (q_{i-1}, \texttt{check}\ e, q_i)$ and $(s'_k, \mathsf{e}', s'_{k+1}) \in \Delta_{\mathrm{X,TSO}}$ by the $\Delta_{\mathrm{X,TSO}}$ rule for conditionals. We define $\beta_{k+1} := \beta_k \cdot \mathsf{e}'$ and find that the invariants **I-0...3** hold for $k+1$ since
(0) **I-0** holds for k so $s_0 \xrightarrow{\alpha_{k+1}} s_{k+1}$ and $s'_0 \xrightarrow{\beta_{k+1}} s'_{k+1}$, i.e. invariant **I-0** holds for $k+1$.
(1) **I-1** holds for k, $\mathsf{pc}_+(t) = \mathit{dst}(\mathit{inst}_i)$, and $\mathsf{pc}'_+(t) = \overline{q}_i$, so invariant **I-1** holds for $k+1$.
(2) **I-2** holds for k and both e and e' do not change any address, so invariant **I-2** holds for $k+1$.
(3) **I-3** holds for k and no event alters buffer contents, so invariant **I-3** holds for $k+1$.
$\boxed{\text{5b}}$ "$\mathsf{pc}'(t) = \mathsf{pc}(t)$." This case covers the remaining possibilities when e is an assignment, conditional, or memory fence. Similar to cases $\boxed{\text{3b}}$ and $\boxed{\text{4c}}$, let $\mathsf{e}' \in E_{\oplus}$ and $s'_{k+1} \in S_{\oplus}$ such that $\mathit{inst}(\mathsf{e}') = \mathit{inst}(\mathsf{e})$ and $(s'_k, \mathsf{e}', s'_{k+1}) \in \Delta_{\mathrm{X,TSO}}$ in $\mathrm{X}_{\mathrm{TSO}}(\mathcal{R} \oplus \iota)$. We define $\beta_{k+1} := \beta_k \cdot \mathsf{e}'$ and find that, by the $\Delta_{\mathrm{X,TSO}}$ semantics (Figure 2.19), invariants **I-0...3** hold for $k+1$.

The above case distinction covers all possibilities for events e that α may perform from s_k. Hence, by complete induction, the extension does not remove TSO-reachable states: if $s = (\mathsf{pc}, \mathsf{val}, \mathsf{buf})$ is reachable by α then there exists $s' = (\mathsf{pc}', \mathsf{val}', \mathsf{buf}')$ and $\beta \in \mathcal{C}_{\mathrm{TSO}}(\mathcal{R} \oplus \iota)$ such that s' is reachable by β in $\mathcal{R} \oplus \iota$, $\mathsf{pc} = \mathsf{pc}'$, $\mathsf{val}(a) = \mathsf{val}'(a)$ for all $a \in \mathsf{ADR} \cup \mathsf{REG}$, and $\mathsf{buf} = \mathsf{buf}'$ are empty.

For the reverse direction, let $f_\tau \colon \mathcal{C}_{\mathrm{TSO}}(\mathcal{R}) \to \mathcal{C}_{\mathrm{TSO}}(\mathcal{R} \oplus \tau)$ be the map $\alpha \mapsto \beta$ that the inductive proof implies, respectively $f_\tau \colon E \to E^*_{\oplus}$ its restriction to events matching the different inductive cases. Furthermore, consider computations $\beta \in \mathcal{C}_{\mathrm{TSO}}(\mathcal{R} \oplus \iota)$ that do not interleave events of other threads within the events of sequences $f_\tau(\mathsf{e})$. Such computations reach the entire set $\mathit{Reach}_{\mathrm{TSO}}(\mathcal{R} \oplus \iota)$. E.g., since local events $\mathsf{e}_{\texttt{count}}, \ldots, \mathsf{e}_1$ as in case $\boxed{\text{4a}}$ that precede $\mathsf{e}_{\texttt{load}}$ can be performed right before $\mathsf{e}_{\texttt{load}}$, the above restriction does not change the set of TSO-reachable states in $\mathcal{R} \oplus \iota$. Note that f_τ is a bijection between such computations β and computations $\alpha \in \mathcal{C}_{\mathrm{TSO}}(\mathcal{R})$ that delay flushes locally the least wrt t.

Another induction can show that for each computation β as described above there exists a computation $\alpha \in \mathcal{C}_{\mathrm{TSO}}(\mathcal{R})$ such that invariants **I-0...3** hold for prefixes of β and α. This implies that the extension by ι does not add TSO-reachable states. $\square$

Proof of Lemma 11. Let $\alpha \in \mathcal{C}_{\text{TSO}}(\mathcal{P})$ be any computation of an arbitrary ASSEMBLY program $\mathcal{P}$. Furthermore, assume that $s_k := (\mathsf{pc}_k, \mathsf{val}_k, \mathsf{buf}_k) \in S_{\text{TSO}}$ — for some $m \in \mathbb{N}$ and all $k \in [0..m]$ — are all the states along the transition sequence $s_0 \xrightarrow{\alpha} s \in \Delta^*_{\text{X,TSO}}$, i.e., state s_0 in the state sequence is $\text{X}_{\text{TSO}}(\mathcal{P})$'s initial state s_0 and $s_m := s$. Also, for all $k \in [0..m]$, let α_k denote prefixes of α with $s_0 \xrightarrow{\alpha_k} s_k \in \Delta^*_{\text{X,TSO}}$.

We will prove by induction over $k \in [0..m]$ that $\alpha \in \mathcal{L}_F(A_{set}(\mathcal{P}))$. More precisely, we show by induction that there exist (abstract) states $s'_0, \ldots, s'_m \in {}^AS$ in $A_{set}(\mathcal{P})$ for which the following invariants hold:

I-7 ${}^As_0 \xrightarrow{\alpha_k} s'_k := (\mathsf{pc}'_k, \mathsf{val}'_k, {}^s\mathsf{buf}_k)$ is a valid computation prefix in $A_{set}(\mathcal{P})$.

I-8 $\mathsf{pc}_k = \mathsf{pc}'_k$ and $\mathsf{val}_k = \mathsf{val}'_k$.

I-9 for all threads t and addresses a, $last(a, \mathsf{buf}_k(t)) = last(a, {}^s\mathsf{buf}_k(t))$ and $(a, v) \in \mathsf{buf}_k(t) \cap (\mathsf{ADR} \times \mathsf{DOM})$ iff $(a, v) \in {}^s\mathsf{buf}_k(t)$.

For the induction **base case** $k = 0$, $\alpha_0 = \epsilon$, $s_0 = s_0$, $s'_0 = {}^As_0$, and invariants **I-7...2** hold.

For the **induction step case**, assume that invariants **I-7...2** hold for $k < m$ and that $s_k \xrightarrow{\mathsf{e}} s_{k+1}$ for some $\mathsf{e} \in E$. We use a case distinction over possible events e with $t := thread(\mathsf{e})$ to show that there is some state s'_{k+1} such that invariants **I-7...2** hold for $k + 1$.

$\boxed{1}$ "e is a flush event." We distinguish the following cases depending if the pair (a, v) that e flushes is the last $\{a\} \times \mathsf{DOM}$ or (a, v) pair in $\mathsf{buf}_k(t)$.
$\boxed{1a}$ "e flushes the last $\{a\} \times \mathsf{DOM}$ pair (a, v) in $\mathsf{buf}_k(t)$." Then, by picking rule (WM-D) for $s'_k \xrightarrow{\mathsf{e}} s'_{k+1}$ in $A_{set}(\mathcal{P})$, invariants **I-7..9** hold for $k+1$ since

(0) **I-7** holds for k so ${}^As_0 \xrightarrow{\alpha_k} s'_k \xrightarrow{\mathsf{e}} s'_{k+1}$ is a valid computation prefix, i.e. invariant **I-7** holds for $k + 1$.

(1) **I-8** holds for k and $\mathsf{pc}_{k+1} = \mathsf{pc}_k = \mathsf{pc}'_k = \mathsf{pc}'_{k+1}$ while val_{k+1} and val'_{k+1} update the same address a by the same value v, so invariant **I-8** holds for $k + 1$.

(2) **I-9** holds for k, the last-added values in both the concrete buf_k and abstract ${}^s\mathsf{buf}_k$ buffers for address a are no longer defined after being flushed by e, hence, $last(a, \mathsf{buf}_{k+1}(t)) = \perp = last(a, {}^s\mathsf{buf}_{k+1}(t))$, and all other buffer values stay unchanged, hence, $(a, v) \in \mathsf{buf}_{k+1}(t)$ iff $(a, v) \in {}^s\mathsf{buf}_{k+1}(t)$, so invariant **J-3** holds for $k + 1$.

$\boxed{1b}$ "e flushes the last (a, v) pair with $v \neq last(a, \mathsf{buf}_k(t))$ in $\mathsf{buf}(t)$." Like in case $\boxed{1a}$ above, the preconditions for picking rule (WM-D) are fulfilled, and by choosing it for $s'_k \xrightarrow{\mathsf{e}} s'_{k+1}$ in $A_{set}(\mathcal{P})$, invariants **I-7..9** hold for $k + 1$: **I-7** and **I-8** for the same reasons (0) and (1) as above, and **I-9** since $last(a, \mathsf{buf}_{k+1}(t)) = last(a, \mathsf{buf}_k(t)) = last(a, {}^s\mathsf{buf}_k(t)) = last(a, {}^s\mathsf{buf}_{k+1}(t))$ and since removing the last occurrence of an (a, v) pair additionally preserves $(a, v) \in \mathsf{buf}_{k+1}(t)$ iff $(a, v) \in {}^s\mathsf{buf}_{k+1}(t)$.

$\boxed{1c}$ "neither $\boxed{1a}$ nor $\boxed{1b}$ hold." Then the (a, v) pair that e flushes is neither the last added for address a in the thread nor the last added of its kind. Therefore, by picking rule (WM-ND) for $s'_k \xrightarrow{\mathsf{e}} s'_{k+1}$ in $A_{set}(\mathcal{P})$, invariants **I-7..9** hold for $k+1$ since

(0) **I-7** holds for k so ${}^{A}s_0 \xrightarrow{\alpha_k} s'_k \xrightarrow{\mathsf{e}} s'_{k+1}$ is a valid computation prefix, i.e. invariant **I-7** holds for $k+1$.

(1) **I-8** holds for k and $\mathsf{pc}_{k+1} = \mathsf{pc}_k = \mathsf{pc}'_k = \mathsf{pc}'_{k+1}$ while val_{k+1} and val'_{k+1} update the same address a by the same value v, so invariant **I-8** holds for $k+1$.

(2) **I-9** holds for k, the last-added values for both the concrete buf_k and abstract ${}^{s}\mathsf{buf}_k$ stay unchanged so $last(a, \mathsf{buf}_{k+1}(t)) = last(a, {}^{s}\mathsf{buf}_{k+1}(t))$ as in case $\boxed{1b}$ above, and all existing buffer values are still present in both the concrete and abstract buffers, i.e., $(a, v) \in \mathsf{buf}_{k+1}(t)$ iff $(a, v) \in {}^{s}\mathsf{buf}_{k+1}(t)$, so invariant **J-3** holds for $k+1$.

$\boxed{2}$ "e performs a store." Then, by following rule (LS), $s'_k \xrightarrow{\mathsf{e}} s'_{k+1}$ in $A_{set}(\mathcal{P})$ and invariants **I-7..9** hold for $k+1$ since

(0) **I-7** holds for k so ${}^{A}s_0 \xrightarrow{\alpha_k} s'_k \xrightarrow{\mathsf{e}} s'_{k+1}$ is a valid computation prefix, i.e. invariant **I-7** holds for $k+1$.

(1) **I-8** holds for k and rule (LS) advances the program counter in the same way in both $\mathrm{X_{TSO}}(\mathcal{P})$ and $A_{set}(\mathcal{P})$ so $\mathsf{pc}_{k+1} = \mathsf{pc}'_{k+1}$ while $\mathsf{ADR} \cup \mathsf{REG}$ stay unchanged, i.e., $\mathsf{val}_{k+1} = \mathsf{val}_k = \mathsf{val}'_k = \mathsf{val}'_{k+1}$, so invariant **I-8** holds for $k+1$.

(2) **I-9** holds for k, the last-added values for the concrete buf_k and abstract ${}^{s}\mathsf{buf}_k$ are updated only for address a in the (LS) precondition for which $last(a, \mathsf{buf}_{k+1}(t)) = v = last(a, {}^{s}\mathsf{buf}_{k+1}(t))$ and all existing buffer values are still present in both the concrete and abstract buffers, i.e., $(a, v) \in \mathsf{buf}_{k+1}(t)$ iff $(a, v) \in {}^{s}\mathsf{buf}_{k+1}(t)$, so invariant **J-3** holds for $k+1$.

$\boxed{3}$ "e performs a load from the buffer." Then, by following rule (RB), $s'_k \xrightarrow{\mathsf{e}} s'_{k+1}$ in $A_{set}(\mathcal{P})$ and invariants **I-7..9** hold for $k+1$ since

(0) **I-7** holds for k and $\exists v = last(addr(\mathsf{e}), {}^{s}\mathsf{buf}(t))$ so ${}^{A}s_0 \xrightarrow{\alpha_k} s'_k \xrightarrow{\mathsf{e}} s'_{k+1}$ is a valid computation prefix, i.e. invariant **I-7** holds for $k+1$.

(1) **I-8** holds for k and rule (RB) advances the program counter in the same way in both $\mathrm{X_{TSO}}(\mathcal{P})$ and $A_{set}(\mathcal{P})$ so $\mathsf{pc}_{k+1} = \mathsf{pc}'_{k+1}$ while $\mathsf{ADR} \cup \mathsf{REG}$ is changed the same way since **I-9** holds for k, i.e., $\mathsf{val}_{k+1} = \mathsf{val}'_{k+1}$, so invariant **I-8** holds for $k+1$.

(2) **I-9** holds for k, the last-added values for the concrete buf_k and for the abstract ${}^{s}\mathsf{buf}_k$ are not changed and all existing buffer values are still present in both the concrete and abstract buffers, i.e., $(a, v) \in \mathsf{buf}_{k+1}(t)$ iff $(a, v) \in {}^{s}\mathsf{buf}_{k+1}(t)$, so invariant **J-3** holds for $k+1$.

$\boxed{4}$ "e performs a load from memory." Then, by following rule (RM), $s'_k \xrightarrow{\mathsf{e}} s'_{k+1}$ in $A_{set}(\mathcal{P})$ and invariants **I-7..9** hold for $k+1$ since

(0) **I-7** holds for k and $\nexists v = last(addr(\mathsf{e}), {}^{s}\mathsf{buf}(t))$ so ${}^{A}s_0 \xrightarrow{\alpha_k} s'_k \xrightarrow{\mathsf{e}} s'_{k+1}$ is a valid computation prefix, i.e. invariant **I-7** holds for $k+1$.

(1) **I-8** holds for k and rule (RM) advances the program counter in the same way in both $\mathrm{X}_{\mathrm{TSO}}(\mathcal{P})$ and $A_{set}(\mathcal{P})$ so $\mathsf{pc}_{k+1} = \mathsf{pc}'_{k+1}$ while $\mathsf{ADR} \cup \mathsf{REG}$ is changed the same way since **I-9** holds for k, i.e., $\mathsf{val}_{k+1} = \mathsf{val}'_{k+1}$, so invariant **I-8** holds for $k+1$.

(2) **I-9** holds for k, the last-added values for the concrete buf_k and for the abstract ${}^s\mathsf{buf}_k$ are not changed and all existing buffer values are still present in both the concrete and abstract buffers, i.e., $(a, v) \in \mathsf{buf}_{k+1}(t)$ iff $(a, v) \in {}^s\mathsf{buf}_{k+1}(t)$, so invariant **J-3** holds for $k+1$.

$\boxed{5}$ "e is determined by a memory fence." Then, by following rule (LF), $s'_k \xrightarrow{\mathsf{e}} s'_{k+1}$ in $A_{set}(\mathcal{P})$ and invariants **I-7..9** hold for $k+1$ since

(0) **I-7** holds for k and ${}^s\mathsf{buf}(t) = \emptyset$ (since **I-9** holds for k) so ${}^A s_0 \xrightarrow{\alpha_k} s'_k \xrightarrow{\mathsf{e}} s'_{k+1}$ is a valid computation prefix, i.e. invariant **I-7** holds for $k+1$.

(1) **I-8** holds for k and rule (RM) advances the program counter in the same way in both $\mathrm{X}_{\mathrm{TSO}}(\mathcal{P})$ and $A_{set}(\mathcal{P})$ so $\mathsf{pc}_{k+1} = \mathsf{pc}'_{k+1}$ while $\mathsf{ADR} \cup \mathsf{REG}$ is changed the same way since **I-9** holds for k, i.e., $\mathsf{val}_{k+1} = \mathsf{val}'_{k+1}$, so invariant **I-8** holds for $k+1$.

(2) **I-9** holds for k, the last-added values for the concrete buf_k and for the abstract ${}^s\mathsf{buf}_k$ are not changed and all existing buffer values are still present in both the concrete and abstract buffers, i.e., $(a, v) \in \mathsf{buf}_{k+1}(t)$ iff $(a, v) \in {}^s\mathsf{buf}_{k+1}(t)$, so invariant **J-3** holds for $k+1$.

$\boxed{6}$ "e performs an assignment or a conditional check." Then, by following either rule (LA) or rule (LC), $s'_k \xrightarrow{\mathsf{e}} s'_{k+1}$ in $A_{set}(\mathcal{P})$ and invariants **I-7..9** hold for $k+1$ since

(0) **I-7** holds for k so ${}^A s_0 \xrightarrow{\alpha_k} s'_k \xrightarrow{\mathsf{e}} s'_{k+1}$ is a valid computation prefix, i.e. invariant **I-7** holds for $k+1$.

(1) **I-8** holds for k and the program counter advances in the same way in both $\mathrm{X}_{\mathrm{TSO}}(\mathcal{P})$ and $A_{set}(\mathcal{P})$ so $\mathsf{pc}_{k+1} = \mathsf{pc}'_{k+1}$ while $\mathsf{ADR} \cup \mathsf{REG}$ is changed the same way since **I-9** holds for k, i.e., $\mathsf{val}_{k+1} = \mathsf{val}'_{k+1}$, so invariant **I-8** holds for $k+1$.

(2) **I-9** holds for k, the last-added values for the concrete buf_k and for the abstract ${}^s\mathsf{buf}_k$ are not changed and all existing buffer values are still present in both the concrete and abstract buffers, i.e., $(a, v) \in \mathsf{buf}_{k+1}(t)$ iff $(a, v) \in {}^s\mathsf{buf}_{k+1}(t)$, so invariant **J-3** holds for $k+1$.

The above case distinction covers all possibilities for events e that may be performed from s_k along α. Therefore, by complete induction, invariants **I-7..9** hold for s_m. Then, since **I-9** in particular implies ${}^s\mathsf{buf}_m(t) = \emptyset$ for all threads t, by **I-7** and **I-9** we conclude that $\alpha \in \mathcal{L}_F(A_{set}(\mathcal{P}))$. □

Appendix B

$A_{mset}(\mathcal{P})$ Reachability is Decidable

As one should understand by now, the story behind program correctness overlaps multiple formalisms like logic, automata, and transition systems. Well-structured transition systems (WSTS) [FS01] specialize the latter and, in doing so, they generalize many other infinite state modeling formalisms like Petri nets [Rei85] or lossy channel systems [AJ96].

The WSTS framework is the outcome of generalizing decision procedures for Petri net termination and boundedness [Fin87] as well as lossy channel system control state reachability and simulation [AK95; Abd+96]. WSTS decidability results rely on the existence of a Well-quasi-ordering (WQO) between states that is (simulation-wise) compatible with the transitions. Using the WSTS framework we show that checking $A_{mset}(\mathcal{P})$ reachability is decidable. Therefore, we recall a few well-structured transition systems results that, in combination with showing that $A_{mset}(\mathcal{P})$ is a WSTS with decidable WQO, lets us conclude decidability.

Well-quasi-orderings A preorder or *quasi-order (qo)* over some set X is any binary relation $\leq$ over X that is both transitive and reflexive. We use $x < y$ to denote $x \leq y \not\leq x$ and $(X, \leq)$ to denote qo $\leq$ over X. Furthermore, we say that the set $Y \subseteq X$ is an antichain if $x \not\leq y$ for all $x, y \in Y$.

A *well-quasi-ordering* (WQO) over X is any qo $(X, \leq)$ such that, for any infinite sequence $x_0, x_1, \ldots$ in X, there exist indexes $i < j$ such that $x_i \leq x_j$.

By definition, every WQO over X is well-founded (Noetherian), i.e., there is no infinite strictly decreasing chain $x_0 > x_1 > \ldots$ in X.

Lemma 25. *If $(X, \leq)$ is a WQO then any infinite sequence $x_0, x_1, \ldots$ in X contains an infinite increasing subsequence $x_{i_0} \leq x_{i_1} \leq \ldots$ with $i_0 < i_1 < \ldots$*

Proof. Consider an infinite sequence $(x_k)_{k\in\mathbb{N}}$ in X and let $(x_{nd(k)})_{k\in\mathbb{N}}$ be its subsequence of elements that are not dominated by successors, i.e., for all $x_{nd(i)}$ there is no x_j with $x_{nd(i)} \leq x_j$ and $nd(i) < j$.

If the sequence $(x_{nd(k)})_{k\in\mathbb{N}}$ were infinite then, by the WQO definition, it would contain two comparable elements. However, this would contradict the assumption that $(x_{nd(k)})_{k\in\mathbb{N}}$ elements are not dominated by successors. Hence, the sequence $(x_{nd(k)})_{k\in\mathbb{N}}$ must be finite.

Now, let max be the maximum index in the sequence $(x_{nd(k)})_{k\in\mathbb{N}}$. Since every x_n with $n >$ max is dominated by at least one successor, one can start an infinite increasing subsequence of $(x_{nd(k)})_{k\in\mathbb{N}}$ from any such x_n. □

A straightforward corollary of the previous lemma is that there are no infinite antichains in $(X, \leq)$.

Lemma 26. *If $(X, \leq)$ is a WQO then there is no infinite antichain in X.*

Proof. To the contrary, assume that $Y \subseteq X$ is an infinite antichain and let $x_0, x_1, \ldots$ be an infinite sequence in Y. Since $x_0, x_1, \ldots$ is also a sequence in X, by Lemma 25, it contains a subsequence of increasing elements. This contradicts the assumption that Y is an infinite antichain. □

Upward-closed sets Given qo $\leq$, an *upward-closed set* is any set $I \subseteq X$ such that $x \in I$ and $x \leq y$ entail $y \in I$. The *upward closure* of a set $\mathcal{S} \subseteq X$ is $\uparrow\mathcal{S} := \{x \in X \mid x \geq y \text{ for some } y \in \mathcal{S}\}$.

Let $(X, \leq)$ be a WQO and let $\mathcal{S} \subseteq X$ be a subset of X. A set of minimal elements of $\mathcal{S}$ is any subset $min(\mathcal{S}) \subseteq \mathcal{S}$ such that (1) $min(\mathcal{S})$ is an antichain, and (2) for every $x \in \mathcal{S}$ there exists $m \in min(S)$ such that $m \leq x$.

The above definition does not say how many minimal elements are in an arbitrary set. We can actually show that, given a WQO $(X, \leq)$, every set $\mathcal{S} \subseteq X$ contains finitely many minimal elements $min(\mathcal{S})$.

Lemma 27. *Let $(X, \leq)$ be a WQO and let $\mathcal{S} \subseteq X$. A set $min(\mathcal{S})$ of minimal elements exists and this set is finite.*

Proof. To the contrary, assume there is no finite set of minimal elements. We construct a sequence $(x_k)_{k\geq 0}$ in $\mathcal{S}$ such that $x_i \not\leq x_j$ for all $0 \leq i < j$, i.e., such that no element in the sequence is dominated by any successor.

If, at some point, no $j + 1$ such that $x_i \not\leq x_{j+1}$ for all $0 \leq i \leq j$ exists, then we can construct a set of minimal elements from $\{x_0, \ldots, x_j\}$. This contradicts the proof's initial assumption.

If, on the other hand, the sequence $(x_k)_{k\in\mathbb{N}}$ is infinite, then its elements form an infinite antichain. This contradicts Lemma 26. □

Note that $min(\mathcal{S})$ need not be unique since antisymmetry is not required for the WQO $(X, \leq)$. Intuitively, each set $min(\mathcal{S})$ is a good representation for the (potentially infinite) set $\mathcal{S}$. Furthermore, the sets captured precisely by their minimal elements are upward closed.

Lemma 28. *Let $(X, \leq)$ be a WQO and let $I \subseteq X$ be an upward-closed set. If $min(I)$ is a set of minimal elements then $I = \uparrow min(I)$.*

Proof. We prove both $I \subseteq \uparrow min(I)$ and $I \subseteq \uparrow min(I)$ to draw the conclusion.

Let $x \in I$. By definition of $min(I)$ there is $m \in min(I)$ such that $m \leq x$. So, by definition of the upward closure, $x \in \uparrow min(I)$.

Let $x \in \uparrow min(I)$. By definition of the upward closure, there exists $m \in min(I)$ such that $m \leq x$. Then, since $min(I) \subseteq I$, $m \in I$. Hence, by definition of an upward-closed set, $x \in I$. □

The decision procedure for well-structured transition system control state reachability relies on increasingly growing sequences of upward closed sets. The WQO assumption guarantees that these sequences stabilize, thus ensuring that the algorithm terminates.

Lemma 29. *Let $(X, \leq)$ be a WQO. Any infinite increasing sequence $I_0 \subseteq I_1 \subseteq \ldots$ of upward-closed sets eventually stabilizes, i.e., there exists $k \in \mathbb{N}$ such that $I_k = I_{k+1} = \ldots$.*

Proof. To the contrary, assume there exists an infinite increasing sequence $I_0 \subseteq I_1 \subseteq \ldots$ that does not stabilize, i.e., for each $i \in \mathbb{N}$ there exists $j > i$ such that $I_i \subsetneq I_j$.

We can then extract an infinite subsequence $I_{n_0} \subsetneq I_{n_1} \subsetneq \ldots$ out of $(I_k)_{k \in \mathbb{N}}$ and we can construct an infinite sequence $x_0, x_1, \ldots$ such that $x_i \in I_{n_i} \setminus I_{n_{i-1}}$ for all $i > 0$.

Since $(X, \leq)$ is a WQO the sequence $(x_k)_{k \in \mathbb{N}}$ contains a comparable pair, i.e., $x_i \leq x_j$ for some $i < j$. Then, since $x_i \in I_{n_i}$ and I_{n_i} is upward-closed, also $x_j \in I_{n_i}$. And, since $I_{n_i} \subsetneq I_{n_{j-1}}$, this means that $x_j \in I_{n_{j-1}}$. This, however, contradicts $x_j \in I_{n_j} \setminus I_{n_{j-1}}$. □

Well-structured transition systems A *transition system* is a structure $(\Gamma, \rightarrow, \ldots)$ where Γ is a set of configurations and $\rightarrow \subseteq \Gamma \times \Gamma$ is any set of transitions.[1]

We write $\rightarrow^*$ for the reflexive and transitive closure of $\rightarrow$ and $\rightarrow^i$ with $i \in \mathbb{N}$ for the i-step iteration of $\rightarrow$. Furthermore, we use $succ(\gamma)$ to denote the set $\{\gamma' \in \Gamma \mid \gamma \rightarrow \gamma'\}$ of immediate γ *successors* and $pred(\gamma)$ to denote the set $\{\gamma' \in \Gamma \mid \gamma' \rightarrow \gamma\}$ of immediate γ *predecessors*. We restrict our attention to finitely branching transition systems, i.e., transition systems whose successor sets $succ(\gamma)$ are all finite.

A *well-structured transition system* (WSTS) $(\Gamma, \rightarrow, \leq)$ is a transition system $(\Gamma, \rightarrow)$ equipped with a WQO $\leq \subseteq \Gamma \times \Gamma$ that is also *compatible*[2] with the transition relation, i.e., for all $\gamma_1, \gamma_2, \gamma_1' \in \Gamma$ with $\gamma_1 \rightarrow \gamma_2$ and $\gamma_1 \leq \gamma_1'$ there exists $\gamma_2' \in \Gamma$ with $\gamma_1' \rightarrow^* \gamma_2'$ and $\gamma_2 \leq \gamma_2'$.

[1]Transition systems can have additional structure like initial states, transition/state labels, etc. For example, process graphs, the transition semantics of IMP programs, and automata are all transition systems.

[2]Compatibility states that $\leq$ is a weak simulation relation à la Milner [Mil89].

Let $(\Gamma, \rightarrow, \leq)$ be a WSTS and let $I \subseteq \Gamma$ be a set of configurations. Backward reachability analysis seeks to compute

$$\mathit{pred}^*(I) := \{\gamma \in \Gamma \mid \gamma \rightarrow^* \gamma' \text{ for some } \gamma' \in I\}$$

as the limit of $I_0 \subseteq I_1 \subseteq \ldots$ with $I_0 := I$ and $I_{i+1} := I_i \cup \mathit{pred}(I_i)$. Although the approach does not work in general, it works for upward-closed sets I since, by Lemmas 29 and 30, such a sequence stabilizes.

Lemma 30. *Let $(\Gamma, \rightarrow, \leq)$ be a WSTS and $I \subseteq \Gamma$ an upward-closed set. Then, $\mathit{pred}^*(I)$, $\mathit{pred}(I)$ and $I \cup \mathit{pred}(I)$ are upward-closed.*

Proof. Let $\gamma_1 \in \mathit{pred}^*(I)$ and assume $\gamma_1 \rightarrow^* \gamma_2$ for some $\gamma_2 \in I$. If $\gamma_1 \leq \gamma_1'$ then (iterated) compatibility entails $\gamma_1' \rightarrow^* \gamma_2'$ for some $\gamma_2' \geq \gamma_2$. Since $\gamma_2 \in I$ and I is upward-closed it means $\gamma_2' \in I$. Hence, $\gamma_1' \in \mathit{pred}^*(I)$ which proves $\mathit{pred}^*(I)$ is upward-closed.

Let $\gamma_1 \in \mathit{pred}(I)$ and assume $\gamma_1 \rightarrow \gamma_2$ for some $\gamma_2 \in I$. If $\gamma_1 \leq \gamma_1'$ then compatibility entails $\gamma_1' \rightarrow^* \gamma_2'$ for some $\gamma_2' \geq \gamma_2$. Since $\gamma_2 \in I$ and I is upward-closed it means $\gamma_2' \in I$. Hence, $\gamma_1' \in \mathit{pred}(I)$ which proves $\mathit{pred}(I)$ is upward-closed.

The fact that $I \cup \mathit{pred}(I)$ is upward-closed is a direct implication of upward-closed sets being closed under union. ☐

Theorem 31. *Let $(\Gamma, \rightarrow, \leq)$ be a WSTS with $\gamma \in \Gamma$ and let $I \subseteq \Gamma$ be an upward-closed set. Then $\mathit{pred}^*(I) = \bigcup_{i \in \mathbb{N}} I_i = I_k$ for some $k \in \mathbb{N}$ such that $I_k = I_{k+1}$. Furthermore, I is reachable from γ iff $\gamma \in \mathit{pred}^*(I)$.*

Proof. From Lemma 30 we know that the sets I_i are upward-closed. Then, from Lemma 29 we know the sequence $I_0, I_1, \ldots$ stabilizes, i.e., there is $k \in \mathbb{N}$ with $I_k = I_{k+1}$, so $\mathit{pred}(I_k) \subseteq I_k$. This implies that $I_k = I_{k+1} = I_{k+2} = \ldots$, hence, $\bigcup_{i \in \mathbb{N}} I_i = I_k$. Moreover, since $\mathit{pred}^*(I) = \mathit{pred}(\ldots \mathit{pred}(I)) = \bigcup_{i \in \mathbb{N}} I_i$, we also have $\mathit{pred}^*(I) = I_k$.

Assume that I is reachable from γ, i.e., there exists $\gamma' \in I$ such that $\gamma \rightarrow^i \gamma'$ for some $i \in \mathbb{N}$. Then, $\gamma \in I_i \subseteq I_k = \mathit{pred}^*(I)$.

If, on the other hand, $\gamma \in \mathit{pred}^*(I)$ then $\gamma \in I_k$ so there exists $\gamma' \in I = I_0$ and $i \in \mathbb{N}$ ($i \leq k$) such that $\gamma \rightarrow^i \gamma'$. Hence, I is reachable from γ. ☐

Intuitively, a decision procedure for backward reachability that checks whether some upward-closed I is reachable from $\gamma \in \Gamma$ would

(1) generate the sequence of upward-closed sets $I_0 \subseteq I_1 \subseteq \ldots$,
(2) check for stabilization $I_k = I_{k+1}$, and
(3) check for membership $\gamma \in I_k$.

The problem with this approach is that each of the sets $I_0, I_1, \ldots$ is infinite. A solution for it is to reason in terms of minimal elements M_i of the sets I_i. Indeed, one can show that $\gamma \in I_k$ with $I_k = I_{k+1}$ iff, for appropriately chosen $M_0, M_1, \ldots$, $\gamma \geq \gamma'$ with $\gamma' \in {\uparrow}M_k$ and ${\uparrow}M_k = {\uparrow}M_{k+1}$.

Formally, the decision procedure for WSTS backward reachability will compute a sequence of minimal elements such that

$$M_0 := \mathit{min}(I) \text{ and } M_{i+1} := \mathit{min}(M_i \cup \bigcup_{\gamma \in M_i} \mathit{minpred}(\gamma)) \text{ for all } i \in \mathbb{N}.$$

The above definition relies on $\mathit{minpred}(\,)$ returning a set of minimal elements $\mathit{min}(\mathit{pred}(\uparrow\{\gamma\}))$ for the predecessors of any $\uparrow\{\gamma\}$. Surprisingly, the finite sets M_i are precisely minimal elements for the upward-closed I_i.

Lemma 32. *If I is an upward-closed set, $I_0 = I$ and $I_{i+1} := I_i \cup \mathit{pred}(I_i)$, then $I_i = \uparrow M_i$ for all $i \in \mathbb{N}$.*

Proof. We prove the statement by induction over $i \in \mathbb{N}$.

For the induction **base case** we invoke Lemma 28 with $I = I_0$.

For the induction **step case** assume $I_i = \uparrow M_i$. We then find that

$$\begin{aligned}
I_{i+1} &= I_i \cup \mathit{pred}(I_i) \\
\{ \text{ induction hypothesis } \} &= \uparrow M_i \cup \mathit{pred}(\bigcup_{\gamma \in M_i} \uparrow\{\gamma\}) \\
\{ \text{ distributivity of } \mathit{pred}(\,) \text{ over } \cup \ \} &= \uparrow M_i \cup \bigcup_{\gamma \in M_i} \mathit{pred}(\uparrow\{\gamma\}) \\
\{ \text{ Lemma 28 for } \mathit{pred}(\uparrow\{\gamma\}) \ \} &= \uparrow M_i \cup \bigcup_{\gamma \in M_i} \uparrow \mathit{min}(\mathit{pred}(\uparrow\{\gamma\})) \\
\{ \text{ distributivity of } \uparrow \text{ over } \cup \ \} &= \uparrow\Big(M_i \cup \bigcup_{\gamma \in M_i} \mathit{min}(\mathit{pred}(\uparrow\{\gamma\}))\Big) \\
\{ \text{ definition of minimal elements } \} &= \uparrow \mathit{min}(M_i \cup \bigcup_{\gamma \in M_i} \mathit{min}(\mathit{pred}(\uparrow\{\gamma\})))
\end{aligned}$$

Since $\mathit{min}(\mathit{pred}(\uparrow\{\gamma\})) = \mathit{minpred}(\gamma)$ we conclude that $I_{i+1} = \uparrow M_{i+1}$. □

From Lemma 27 we know that $\mathit{min}(\,)$ is computable for all finite input sets if $\leq$ is decidable. As for $\mathit{minpred}(\,)$, deciding backward reachability by constructing the sequence $M_0, M_1, \dots$ requires that $\mathit{minpred}(\,)$ is effectively computable. In the following, we say that a WSTS has computable minimal predecessors if $\mathit{minpred}(\gamma)$ is computable for any $\gamma \in \Gamma$.

Theorem 33. *Let WSTS $(\Gamma, \rightarrow, \leq)$ have computable minimal predecessors and decidable $\leq$. Consider $\gamma \in \Gamma$ and let $I \subseteq \Gamma$ be an upward-closed set such that $\mathit{min}(I)$ is known. Then it is decidable whether I is reachable from γ.*

Proof. The algorithm computes the sequence $M_0, M_1, \dots$ described above until it finds $\uparrow M_k = \uparrow M_{k+1}$. The equality is decidable since $\leq$ is decidable and the sets M_i are finite. Then, by Theorem 31, I is reachable from γ iff $\gamma \geq \gamma'$ for some $\gamma' \in M_k$. The latter can be checked since M_k is finite. □

To prove that $A_{mset}(\mathcal{P})$ reachability is decidable we first show that, for the standard WQO over multiset-abstracted states, $A_{mset}(\mathcal{P})$ is a WSTS. Afterward, we show that *minpred*( ) is effectively computable and invoke Theorem 33 to conclude.

Given two states $(\mathsf{pc}, \mathsf{val}, \mathsf{buf})$ and $(\mathsf{pc}', \mathsf{val}', \mathsf{buf}')$ of $A_{mset}(\mathcal{P})$ we define $(\mathsf{pc}, \mathsf{val}, \mathsf{buf}) \leq (\mathsf{pc}', \mathsf{val}', \mathsf{buf}')$ iff $\mathsf{pc} = \mathsf{pc}'$, $\mathsf{val} = \mathsf{val}'$, and $\mathsf{buf} \leq \mathsf{buf}'$.

The comparison of buffer contents $\mathsf{buf} \leq \mathsf{buf}'$ in $A_{mset}(\mathcal{P})$ requires that, for all addresses $a \in \mathsf{ADR}$, values $v \in \mathsf{DOM}$ and threads $t \in \mathsf{TID}$, $\mathit{last}(a, \mathsf{buf}(t)) = \mathit{last}(a, \mathsf{buf}'(t))$ and $\mathsf{buf}(t)((a, v)) \leq \mathsf{buf}'(t)((a, v))$.

Using Lemmas 34 and 35 we can deduce that the $A_{mset}(\mathcal{P})$ comparison defined above is a WQO.

Lemma 34. *Both $(\mathbb{N}, \leq)$ and $(X, =)$ for finite sets X are WQO.*

Proof. Since $\leq \subseteq \mathbb{N} \times \mathbb{N}$ is both transitive and reflexive $\leq$ is a qo. Furthermore, for any infinite sequence $(x_i)_{i\in\mathbb{N}}$ of natural numbers there exists $j \in \mathbb{N}$ such that $0 < j$ and $x_0 \leq x_j$. Hence, $(\mathbb{N}, \leq)$ is a WQO.

Since $= \subseteq X \times X$ is both transitive and reflexive $=$ is a qo. Furthermore, since X is finite, for any infinite sequence $(x_i)_{i\in\mathbb{N}}$ in X there exists some x element of X that is repeating. Hence, there exist $i < j$ such that $x_i = x_j = x$, proving that $(X, =)$ is a WQO. □

Lemma 35. *If $(X, \leq)$ and $(Y, \sqsubseteq)$ are WQO then $(X \times Y, \leq \times \sqsubseteq)$ is a WQO.*

Proof. Let $(x_i, y_i)_{i\in\mathbb{N}}$ be an infinite sequence of $X \times Y$ elements where the pair components are explicit. Since $(x_i)_{i\in\mathbb{N}}$ is an infinite sequence in X and $(X, \leq)$ is a WQO, by Lemma 25, there exists a subsequence $x_{n_0}, x_{n_1}, \ldots$ such that $x_{n_i} \leq x_{n_{i+1}}$ for all $i \in \mathbb{N}$.

Consider the sequence $y_{n_0}, y_{n_1}, \ldots$ with the same indexes as the ones in the $(x_{n_i})_{i\in\mathbb{N}}$ subsequence above. Since and $(Y, \sqsubseteq)$ is a WQO, again by Lemma 25, there exists a subsequence $y_{n'_0}, y_{n'_1}, \ldots$ of such that $y_{n'_i} \leq y_{n'_{i+1}}$ for all $i \in \mathbb{N}$.

But then the $(x_i, y_i)_{i\in\mathbb{N}}$ subsequence $(x_{n'_0}, y_{n'_0}), (x_{n'_1}, y_{n'_1}), \ldots$ is ordered, thus proving that $(X \times Y, \leq \times \sqsubseteq)$ is a WQO. □

To show that $A_{mset}(\mathcal{P})$ is a WSTS we show that the above defined WQO is compatible with the transition relation. Assume s_1, s_2, s'_1 are states of $A_{mset}(\mathcal{P})$ such that $s_1 \to s_2$ and $s_1 \leq s'_1$. We show that there exists s'_2 such that $s'_1 \to^* s'_2$ and $s_2 \leq s'_2$ through the following case distinction over the event e that $s_1 \to s_2$ performs:

(flush) If e performs a flush of some address-value pair (a, v) in a thread t then, since $\mathsf{buf}_1(t) \leq \mathsf{buf}'_1(t)$, this flush can also be performed out of s'_1 as the last in a sequence of a-modifying flushes meant to ensure $\mathit{last}(a, \mathsf{buf}_2(t)) = \mathit{last}(a, \mathsf{buf}'_2(t))$. Then, since the valuation of address a changes the same from s_1 to s_2 as it changes from s'_1 to s'_2 and since the WQO $\mathsf{buf}_1 \leq \mathsf{buf}'_1$ is preserved, it must hold that $s_2 \leq s'_2$.

(store) If e buffers some (a, v) store of some thread t then, since $\mathsf{val}_1 = \mathsf{val}'_1$ the same address-value pair can be performed from s'_1. Moreover, since the event e changes the control in the same way and since the WQO $\mathsf{buf}_1 \leq \mathsf{buf}'_1$ is preserved, it must hold that $s_2 \leq s'_2$.

(fence) If e performs a memory fence in thread t it means $\mathsf{buf}_1(t)$ is empty. Since this is reflected through $last(a, \mathsf{buf}_1(t)) = \bot = last(a, \mathsf{buf}'_1(t))$ for all $a \in \mathsf{ADR}$, the memory fence can also be performed from s'_1 and the control change would be the same, hence $s_2 \leq s'_2$.

(load) If e performs a load in some thread t from some address a then, since $\mathsf{val}_1 = \mathsf{val}'_1$ and $last(a, \mathsf{buf}_1(t)) = last(a, \mathsf{buf}'_1(t))$, the same load can be performed from s'_1. Moreover, since $s_1 \leq s'_1$ and the event e only changes the control and register valuation, these changes are the same from s_1 to s_2 as from s'_1 to s'_2. Hence, it must hold that $s_2 \leq s'_2$.

(assignment) If e performs an assignment then, since $\mathsf{val}_1 = \mathsf{val}'_1$, the same assignment can be performed out of s'_1 with the same (and only) control flow and register changes. Hence, it must hold that $s_2 \leq s'_2$.

(conditional) If e performs a conditional then, since $\mathsf{val}_1 = \mathsf{val}'_1$, the same conditional can be performed out of s'_1 with the same (and only) control flow change. Hence, it must hold that $s_2 \leq s'_2$.

What remains to be shown is (Lemma 36) that $minpred(s)$ is effectively computable for any $A_{mset}(\mathcal{P})$ state $s = (\mathsf{pc}, \mathsf{val}, \mathsf{buf})$. Intuitively, this is the case since $A_{mset}(\mathcal{P})$ is finitely branching and, for each backward transition from s, one can determine the necessary conditions on the state s as well as the minimal changes needed in s_{pre} such that $s_{\text{pre}} \to s$.

Concretely, we define $minpred(s) := min(\mathcal{S})$, with $\mathcal{S}$ the smallest set so that $s_{\text{pre}} \in \mathcal{S}$ if $s_{\text{pre}} \xrightarrow{\mathsf{e}} s$ and $0 \wedge (1 \vee 2 \vee 3 \vee 4 \vee 5)$ where:

0: if e doesn't perform a flush and $thread(\mathsf{e}) = t$ then $\mathsf{pc}_{\text{pre}}(t)$ contains the pre-$inst(\mathsf{e})$ state in t

1: if e performs a flush of some address-value pair (a, v) of thread t then $\mathsf{buf}_{\text{pre}}(t)(a) = \mathsf{buf}(t)(a) + 1$ and $last(a, \mathsf{buf}_{\text{pre}}) = \bot$ if $\mathsf{buf}_{\text{pre}}(t)(a) = 1$

2: if e buffers some (a, v) store of some thread t then $\mathsf{buf}_{\text{pre}}(t)(a) + 1 = \mathsf{buf}(t)(a)$ with $last(a, \mathsf{buf}_{\text{pre}}) = v$

3: if e performs a fence in thread t then $\mathsf{buf}(t) = [\,]$

4: if e performs a load from a in thread t then this is reflected in the register change between $\mathsf{val}_{\text{pre}}$ and val

5: if e performs an assignment in thread t then this is reflected in the register change between $\mathsf{val}_{\text{pre}}$ and val

Lemma 36. *If s is an $A_{mset}(\mathcal{P})$ state then $minpred(s) = min(pred(\uparrow\{s\}))$.*

Proof (sketch). We prove the equality by showing double inclusion.

Let $s_{\text{pre}} \in minpred(s)$. Since we defined $minpred(\,)$ as the least fixed point satisfying $0 \wedge (1 \vee 2 \vee 3 \vee 4 \vee 5)$ we must show that $s_{\text{pre}} \in pred(\uparrow\{s\})$. Equivalently, we must show $s_{\text{pre}} \rightarrow s'$ for some $s' \in \uparrow\{s\}$. The latter holds by definition of upward-closure and $minpred(\,)$.

Now, let $s_{\text{pre}} \in min(pred(\uparrow\{s\}))$. To show that $s_{\text{pre}} \in minpred(s)$ we must show that $s_{\text{pre}} \rightarrow s$. This can be done using a case analysis for the event that changes s_{pre} to s. □

Appendix C

A More Concise TSO Semantics

Figure C.1 depicts a more concrete TSO semantics that makes explicit the event counters of the partially-ordered structure of computation events and (as, e.g., [OSS09b]) also takes locks into consideration. We abstracted away from these details for a clearer presentation of our findings. Including event counters and locks does not change our results.

In terms of syntax this means that the commands Com_t of a thread t may consist — additionally to loads, stores, memory fences, assignments, and conditional checks as in Section 2.3 — of `lock` and `unlock` commands. Their intuitive behavior is the following: if no thread holds the program's global lock and some thread t executes the `lock` instruction then all other threads can only execute instructions producing local events until thread t releases the global lock by executing a matching `unlock` instruction.

As far as semantics is concerned, up to the incomplete definition of the initial state, the rules in Figure C.1 effectively describe a more complete TSO semantics $\mathrm{X}_{\mathrm{TSO}}(\mathcal{P})$. In the initial state $s_0 := (\mathsf{ec}_0, \mathsf{pc}_0, \mathsf{val}_0, \mathsf{buf}_0, \mathsf{lock}_0)$, the event counter holds the initial values of per-thread event counters, $\mathsf{ec}_0(t) := 0$ for all $t \in \mathsf{TID}$, no thread holds the global lock, $\mathsf{lock}_0 := \bot$, and — as per the Section 2.3.1 semantics — the program counter holds initial control states, $\mathsf{pc}_0(t) := q_{0,t}$ for all $t \in \mathsf{TID}$, all registers and addresses contain value 0, and all buffers are empty, $\mathsf{buf}_0(t) := \varepsilon$ for all $t \in \mathsf{TID}$. The SC semantics can, again, be derived by atomically performing rules (LS) and (WM) for stores.

Atomic instructions $\texttt{atomic}\,\{\, cmd_1;\, \ldots;\, cmd_n \,\}$ can be implemented by having their inner commands performed, in sequence, within a `lock`-`unlock` environment. Semantically,

$$s \xrightarrow{\texttt{atomic}\,\{\, cmd_1;\, \ldots;\, cmd_n \,\}} s' \text{ is defined as } s \xrightarrow{\texttt{lock}}\xrightarrow{cmd_1} \cdots \xrightarrow{cmd_n}\xrightarrow{\texttt{unlock}} s',$$

where the intermediary states between s and s' are omitted.

$$\frac{cmd = r \leftarrow \texttt{mem}[e_a], \quad a = \widehat{e_a}, \quad \mathsf{buf}(t){\downarrow}(\mathbb{N} \times \{a\} \times \mathsf{DOM}) = (id, a, v) \cdot \beta, \quad \mathsf{lock} \in \{t, \bot\}}{s \xrightarrow{(t, \mathsf{ec}(t), inst, a)} (\mathsf{ec}', \mathsf{pc}', \mathsf{val}[r := v], \mathsf{buf}, \mathsf{lock})} \text{(RB)}$$

$$\frac{cmd = r \leftarrow \texttt{mem}[e_a], \quad a = \widehat{e_a}, \quad \mathsf{buf}(t){\downarrow}(\mathbb{N} \times \{a\} \times \mathsf{DOM}) = \varepsilon, \quad v = \mathsf{val}(a), \quad \mathsf{lock} \in \{t, \bot\}}{s \xrightarrow{(t, \mathsf{ec}(t), inst, a)} (\mathsf{ec}', \mathsf{pc}', \mathsf{val}[r := v], \mathsf{buf}, \mathsf{lock})} \text{(RM)}$$

$$\frac{cmd = \texttt{mem}[e_a] \leftarrow e_v, \quad a = \widehat{e_a}, \quad v = \widehat{e_v}, \quad id = \mathsf{ec}(t)}{s \xrightarrow{(t, id, inst, a)} (\mathsf{ec}', \mathsf{pc}', \mathsf{val}, \mathsf{buf}[t := (id, a, v) \cdot \mathsf{buf}(t)], \mathsf{lock})} \text{(LS)}$$

$$\frac{\mathsf{buf}(t) = \beta \cdot (id, a, v), \quad \mathsf{lock} \in \{t, \bot\}}{s \xrightarrow{(t, id, \text{flush}, a)} (\mathsf{ec}, \mathsf{pc}, \mathsf{val}[a := v], \mathsf{buf}[t := \beta], \mathsf{lock})} \text{(WM)}$$

$$\frac{cmd = \texttt{mf}, \quad \mathsf{buf}(t) = \varepsilon}{s \xrightarrow{(t, \mathsf{ec}(t), inst, \bot)} (\mathsf{ec}', \mathsf{pc}', \mathsf{val}, \mathsf{buf}, \mathsf{lock})} \text{(LF)}$$

$$\frac{cmd = r \leftarrow e, \quad v = \widehat{e}}{s \xrightarrow{(t, \mathsf{ec}(t), inst, \bot)} (\mathsf{ec}', \mathsf{pc}', \mathsf{val}[r := v], \mathsf{buf}, \mathsf{lock})} \text{(LA)}$$

$$\frac{cmd = \texttt{check}\ e, \quad \widehat{e} \neq 0}{s \xrightarrow{(t, \mathsf{ec}(t), inst, \bot)} (\mathsf{ec}', \mathsf{pc}', \mathsf{val}, \mathsf{buf}, \mathsf{lock})} \text{(LC)}$$

$$\frac{cmd = \texttt{lock}, \quad \mathsf{buf}(t) = \varepsilon, \quad \mathsf{lock} = \bot}{s \xrightarrow{(t, \mathsf{ec}(t), inst, \bot)} (\mathsf{ec}', \mathsf{pc}', \mathsf{val}, \mathsf{buf}, t)} \text{(Lock)}$$

$$\frac{cmd = \texttt{unlock}, \quad \mathsf{buf}(t) = \varepsilon, \quad \mathsf{lock} = t}{s \xrightarrow{(t, \mathsf{ec}(t), inst, \bot)} (\mathsf{ec}', \mathsf{pc}', \mathsf{val}, \mathsf{buf}, \bot)} \text{(Unlock)}$$

Figure C.1: Transition rules for $\mathrm{X}_{\mathrm{TSO}}(\mathcal{P})$ assuming $s = (\mathsf{ec}, \mathsf{pc}, \mathsf{val}, \mathsf{buf}, \mathsf{lock})$ with $\mathsf{pc}(t) = q$ and $inst = q \xrightarrow{cmd} q'$ in thread t and such that $\mathsf{lock} \in \{\bot\} \cup \mathsf{TID}$ indicates the thread that holds the global lock. The event and program counters are updated by $\mathsf{ec}' = \mathsf{ec}[t := \mathsf{ec}(t)+1]$ and $\mathsf{pc}' = \mathsf{pc}[t := q']$. We use $\widehat{e}$ for the result of atomically evaluating expression e under val and $\mathsf{buf}(t){\downarrow}(\mathbb{N} \times \{a\} \times \mathsf{DOM})$ for the projection of $\mathsf{buf}(t)$ to store operations that access address a.

Curriculum Vitae

Personal Data

Name	Georgel Ionuţ Călin
Birthdate	April 1986
Birthplace	Filiaşi, Romania
Email	calin@cs.uni-kl.de

Education

Oct. 10 – Jan. 16	**Computer Science PhD**. University of Kaiserslautern, Kaiserslautern, Germany
Oct. 08 – Sep. 10	**Computer Science MSc**. Saarland University, Saarbrücken, Germany
Sep. 05 – Jun. 08	**Mathematics BSc**. Jacobs University Bremen, Bremen, Germany
Sep. 01 – Jun. 05	**Secondary School**. Colegiul Naţional Fraţii Buzeşti, Craiova, Romania
Sep. 93 – Jun. 01	**Elementary School**. Şcoala cu Clasele I - VIII, Filiaşi, Romania