Digital Comprehensive Summaries of Uppsala Dissertations
from the Faculty of Science and Technology 2663

Concurrency

Models, Algorithms, and Verification

SAMUEL GRAHN

ACTA UNIVERSITATIS
UPSALIENSIS
2026

UPPSALA
UNIVERSITET

Dissertation presented at Uppsala University to be publicly examined in 2001, Ångströmlaboratoriet, Lägerhyddsvägen 1, Uppsala, Tuesday, 2 June 2026 at 13:15 for the degree of Doctor of Philosophy. The examination will be conducted in English. Faculty examiner: Professor Constantin Enea (Ecole Polytechnique).

Abstract

Grahn, S. 2026. Concurrency. Models, Algorithms, and Verification. *Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology* 2663. 64 pp. Uppsala: Acta Universitatis Upsaliensis. ISBN 978-91-513-2810-2.

Concurrent programs are everywhere, as are the systems they run on. They can be found in everything, from watches to airplanes and medical devices. Ensuring that these programs and systems behave as intended is crucial, as any failures can have devastating consequences. Verifying programs on concurrent hardware is tricky, since the order in which operations are performed can be considered nondeterministic from the point of view of the programmer. Moreover, the intuitive notions of causality may or may not apply to memory accesses, due to sacrifices made by cache protocols designed to optimize these accesses. This thesis presents work on the verification of concurrent programs and systems.

In the first contribution, we focus on linearizability, a correctness condition for concurrent objects. In particular, we develop efficient monitoring algorithms --- checking whether single executions are linearizable --- for stacks, queues, and (multi)sets. Our stack algorithm runs in quadratic time, our queue algorithm runs in linearithmic time, and our (multi)set algorithm runs in linear time. We also implement these algorithms in a tool, LiMo, and mechanize the stack algorithm and its correctness proof in a theorem prover.

In the second contribution, we cover the consistency problem for event-driven traces. An event-driven trace, in addition to shared-memory concurrency, contains a message-passing component: handlers execute messages from their mailboxes and can send messages to each other. We expect these messages to be delivered and handled in an order consistent with queue semantics: first-in-first-out (FIFO). We check, for a given trace, whether there is an order of sent and received messages that satisfies the semantics. Modern model checking solutions like dynamic partial order reduction (DPOR) can use these results to extend their tools to the event-driven model.

We provide a restatement of the problem that is suitable for extending existing model checking algorithms to support event-driven models. We show that the problem is NP-complete. We also present and implement an algorithm.

In the third contribution, we focus on the concurrent systems themselves. Memory systems such as cache protocols or distributed memory protocols rarely implement the conceptually simple model of sequential consistency: that a concurrent program acts equivalently to executing one operation at a time, interleaving the operations of each thread. Rather, they implement much weaker memory models. We consider a family of models known from the C11 memory model: Release-Acquire. This family consists of a relaxed version named Weak Release-Acquire (WRA), the standard version (RA), and a strong version (SRA). We consider the system verification problem for these models: given a memory system modeled as a register machine, do all executions of all programs executing on this system satisfy the corresponding memory semantics? We show that the problem can be solved in polynomial time for WRA, and PSPACE-complete for RA and SRA.

Keywords: concurrency, event-driven, linearizability, monitoring, verification, weak memory models

Samuel Grahn, Department of Information Technology, Computer Systems, Box 337, Uppsala University, SE-75105 Uppsala, Sweden.

ISSN 1651-6214
ISBN 978-91-513-2810-2
URN urn:nbn:se:uu:diva-582954 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-582954)

List of papers

This thesis is based on the following papers, which are referred to in the text by their Roman numerals.

I **Efficient Linearizability Monitoring**
Parosh Aziz Abdulla, Samuel Grahn, Bengt Jonsson, Shankaranarayanan Krishna, Om Swostik Mishra
Proceedings of the ACM on Programming Languages, Volume 9, Issue PLDI, 2025, Article No.: 225, Pages 1937-1960, https://doi.org/10.1145/3729328, Extended version at https://arxiv.org/abs/2509.17795

II **Checking Consistency of Event-driven Traces**
Parosh Aziz Abdulla, Mohamed Faouzi Atig, R. Govind, Samuel Grahn, Ramanathan S. Thinniyam
Potanin, A. (eds) Programming Languages and Systems, APLAS 2025, Lecture Notes in Computer Science, vol 16201. Springer, Singapore, https://doi.org/10.1007/978-981-95-3585-9_9, Extended version at https://doi.org/10.48550/arXiv.2508.07855

III **Verification of the Release-Acquire Semantics**
Parosh Aziz Abdulla, Elli Anastasiadi, Mohamed Faouzi Atig, Samuel Grahn
Liu, Z., Saoud, A., Wehrheim, H. (eds) Theoretical Aspects of Computing, ICTAC 2025, Lecture Notes in Computer Science, vol 16237. Springer, Cham., https://doi.org/10.1007/978-3-032-11176-0_8, Extended version at https://doi.org/10.48550/arXiv.2506.08238

Reprints were made with permission from the publishers. The appendices of the extended versions are included in the thesis.

Contents

1. Introduction

> To err is human; to really foul things up requires a computer
>
> Bill Vaughan

Humans have convinced rocks to do math using domesticated lightning. In fact, we have become so adept at it that we do it with rocks several orders of magnitude smaller than grains of sand. These rocks (transistors) and the domesticated lightning (electricity) are now everywhere, in all forms of electronic devices, from the phones in our pockets to medical devices and airplanes. That these devices and their programs work as intended is crucial, yet this is often not the case. Software is often buggy, and incorrect behaviors can have devastating consequences: medical devices can overdose patients with gamma radiation [52] and airplanes can lose control mid-flight [9].

A lot has happened since the creation of the first microprocessors. Transistors got smaller, which means we could put more of them onto a single processor. Eventually, we got to a point where transistors were so small that quantum mechanics started to get in the way. Increasing the clock frequency — doing things faster — increases the power usage, and thus heat. Since humans generally prefer not to have their pockets, offices, and server rooms on fire, we had to be more clever: we constructed processors that could do many tasks simultaneously. If a task can be split evenly into two independent chunks, we can compute each chunk separately and simultaneously, in half the runtime, on pieces of the processor henceforth referred to as *cores*. Due to the laws of physics, this is much more energy efficient than doubling the frequency. Moreover, one can connect these processors to each other, e.g. via the internet, and have them communicate, forming complex systems.

As processors became more complicated, so did the programs we write for them. This inevitably leads to more bugs. The standard way of checking that a program is reliable is through testing: examining one execution of a program to determine whether that execution agrees with the intention. One such approach is *monitoring*: observing the system during use, which can be used to notify users when a program behaves incorrectly. However, the fact that some program runs are correct does not imply that *all* the possible program runs are correct. To make that guarantee, one needs to actually check *all* possible runs. While this is feasible for smaller programs, as soon as the input domain is unbounded (e.g. the integers) or sufficiently large (e.g. the 64-bit integers of

modern hardware), this approach quickly becomes infeasible. There are several verification techniques that avoid checking all executions, instead checking only subsets of them. These subsets can be chosen in many ways, including according to execution paths (as in symbolic execution), or as approximations of the full program behavior by considering only runs up to a particular size (as in bounded model checking).

To make maximum use of modern hardware, programmers must write programs that take advantage of this parallelism: a more difficult task than one might first think. In sequential software, instructions are executed predictably in the order they occur in the program. In order to reason about a program, it is thus sufficient to consider only this sequence of instructions, possibly parameterized over inputs. This is already challenging, and many problems of this form are undecidable: there is no algorithm that can solve them. In contrast, a concurrent program has an additional layer of complexity; the order in which instructions are executed is no longer fixed: each core acts in a specified order, but its operations may *interleave* with operations on other cores. To verify a concurrent program, one needs to consider all the possible interleavings of its instructions. Finding errors in such programs is difficult, as they may only appear in some of these interleavings. As the number of possible interleavings grows exponentially with the program length, so does the search space for finding bugs. This is known as the *state explosion problem*.

The state explosion problem often makes program verification computationally intractable. For this reason, testing approaches such as monitoring, which sacrifice completeness for efficiency, are still the primary choice for programmers when developing software.

In a concurrent program, the workload is divided among the cores, and to synchronize this work, the cores need some way of communicating. While this is implemented through shared memory on modern processors, there are several communication libraries that enable other forms of communication, such as using *shared data structures* (for example, queues, stacks, and sets) or sending *messages*. Using a library for all communication in a program reduces the part of the code where complex reasoning is needed. If all concurrency is handled correctly (under a chosen correctness condition) in the implementations of these shared data structures, the programmer can reason about the rest of the code as if writing a sequential program while still maintaining some notion of correctness. But what does it mean for such a data structure to be *correct*?

To illustrate, we use the queue data structure. The intuition for a queue in the real world is simple: when in line at a store, the order in which you are helped by the cashier is the same as the order in which you entered the queue. Specifically, there is a *first-in-first-out* (FIFO) order. The queue data structure has the same requirement: elements are added and removed in FIFO order. In the concurrent setting, consider what happens when you and somebody else

want to enter the queue simultaneously. Ideally, after some polite waving, one of you enters the queue before the other, keeping the queue as a line. This polite waving is hard to implement in hardware, but *some* communication needs to happen so that you agree on the end result: which of you is ahead of the other.

The standard correctness condition for such shared data structures is *linearizability*, which is an extension of a sequential specification to the concurrent setting. If one can pick a point in time between the call and return of an operation (in the queue example, the operations are enqueue and dequeue), such that the resulting total order of operations induced by these time points satisfies the sequential definition, we say that the execution is *linearizable*. One of the main reasons for the success of linearizability as a correctness criterion is that it is compositional: if one uses only linearizable data structures for communication between cores in a program, the combined history of operations on those structures is linearizable. Thus, when reasoning about a concurrent program using such structures as means of communication, the programmer can reason as if these operations took effect atomically, very similarly to how they would reason when writing a sequential program.

Another popular class of communication libraries consists of those that utilize message passing. Some programming languages, like Erlang, allow processes to communicate only via sending and receiving messages. Reactive systems, such as high-performance servers, smartphone applications, and web applications, often combine message passing with shared memory concurrency. We refer to this model as *event-driven concurrency*. In such a system, these messages can contain pieces of code to be executed. In a smartphone application, we want the click of a button to change something. In such a setting, the event of clicking is sent as a message to the *mailbox* of a *handler*. The handler executes the message, reading and updating the global shared memory state. While this approach makes programming reactive systems much more intuitive, reasoning about correctness becomes even more complicated. In this setting, not only do we have the interleaving behaviors of threads, but we also need to consider the behavior of the mailbox. Given such an execution, one can ask the question: is this execution consistent with the intended mailbox semantics? Solutions to this problem are important for several model checking algorithms, as they need to discard executions that could not have been produced by the system.

So far, we have described the simple interleaving-based model as formulated by Lamport: “the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program”. Systems satisfying this property are called *sequentially consistent*.

This model is not sufficient to fully understand the modern processor. Since accessing main memory to read and update data is a crucial step in programming, we need it to be quick. The physical distance between main memory

and the arithmetic units of the processor is (comparatively) large, which makes these accesses slow. To alleviate this, we designed processors with *caches*, small pieces of storage that mirror regions of main memory. A processor reading from memory loads the values into its cache. Future reads and updates are then only performed on this cache, until the cache is eventually synchronized back to memory. Another core reading that same memory region may obtain the old copy, and not see the updated values until after the first core's cache has synchronized with main memory.

These caches lead to memory updates not being immediately visible to all cores. When verifying concurrent programs, it is thus not always sufficient to consider only the interleavings; one must also consider when memory updates may propagate.

How, when, and in what order these propagations may occur is defined by hardware. When writing programs to be run on this hardware, one must take into account all possible ways the system can interleave instructions, as well as all possible memory behaviors. To make this easier for programmers, there are models for what is guaranteed by different hardware implementations. The behavior of most modern hardware can be modeled using *weak memory models*.

The aim for hardware designs, and thus of memory models, is to make guarantees about the possible executions. Such guarantees are often aimed at preserving certain intuitive ideas about how concurrent computation *should* work. For instance, many of these memory models aim to preserve some notion of causality; that causes happen before their effects. Translated to memory operations: if one thread changes memory through two writes, then any thread that sees the second write should ideally have seen the first as well. Such models are weaker than the interleaving-based model, but still strong enough to be useful for programmers. Models of this form include the *release-acquire* (**RA**) family of models, which is one component of the C11 memory model and one of the models studied in this thesis.

1.1 Research Challenges

Recall that concurrent verification is fundamentally difficult for two reasons. First, operations from different threads may interleave in many possible ways, and errors may only appear in a small subset of those interleavings. Second, modern hardware exposes weak memory behavior: updates are not necessarily visible to all cores immediately. Thus, correctness depends not only on instruction order but also on when writes propagate.

Event-driven concurrency adds a further, design-level source of complexity. The correctness of event-driven programs depends jointly on interleavings, memory behavior, and the semantics of message delivery and execution.

These observations lead to three research questions:

- When is an execution of a concurrent data structure consistent with its intended abstract behavior?
- When is an event-driven execution consistent with intended mailbox semantics?
- When does a memory implementation satisfy the guarantees of a weak memory model?

These questions are the three core challenges addressed in this thesis.

1.2 Brief Summary of Contributions

In this thesis, we present solutions to the three problems posed above. In Paper I, we present algorithms for checking linearizability of executions of the queue, stack, and (multi)set data structures. In Paper II, we present an algorithm for checking whether an execution respects the event-driven semantics with a queue mailbox. Finally, in Paper III, we present algorithms for checking whether a memory implementation satisfies the guarantees of the **RA** family of weak memory models. The papers in this thesis have been extended with appendices containing proofs.

1.3 Thesis Structure

The remainder of the thesis is organized as follows. Chapter 2 introduces the required background. In Chapter 3 we consider the linearizability monitoring problem and present efficient algorithms for monitoring stacks, queues, and (multi)sets. In Chapter 4 we consider the trace consistency problem for event-driven traces with queue mailbox semantics. In Chapter 5, we present algorithms and complexity results for verifying that a memory protocol implements memory models of the **RA** family. In Chapter 7 we present related work. In Chapters 8 and 9 we conclude and describe future research directions, respectively.

1.4 Funding

The work in this thesis has been partially supported by the Swedish Research Council (VR).

2. Background

> If you wish to make an apple pie from scratch,
> you must first invent the universe.
>
> Carl Sagan

This chapter covers the technical preliminaries for the contributions of this thesis.

2.1 Libraries and Linearizability

A concurrent library is a shared object coupled with operations that each thread may perform on that object. Such libraries often implement well-known data structures.

In Paper I we consider the data structures queue, stack, set, and multiset. The operations on these structures share a common feature: they support inserting and removing elements, but with different constraints on ordering. In stack and queue implementations, one generally calls the pop (and deq) operation without an argument, expecting to get a value back (i.e. $pop() \rightarrow a$), while in sets and multisets one supplies the value to the removal operation (i.e. $rmv(a)$). We unify the notations of these operations to two types of operations: *insertion* operations (denoted $!a$) and *removal* operations $?a$. We now define the *specifications* of each of these structures.

- A queue satisfies *first-in-first-out* order: $!a$ adds a value to the *back* of a queue, and $?a$ removes the value a at the *front* of the queue, i.e. the oldest value.
- A stack satisfies *last-in-first-out* order: $?a$ always removes the most recently added value a that has not yet been removed.
- A set only allows insertion of a value a if it is not in the set, and removal if it is. There is no constraint on the order of operations on different values.
- A multiset allows adding arbitrarily many copies of each value a, but only allows removal if there is at least one copy present.

The point of view taken in linearizability monitoring approaches, including in Paper I, is that of a user of a library implementing one of these structures. The user may call the available operations, and wait for them to return. If we

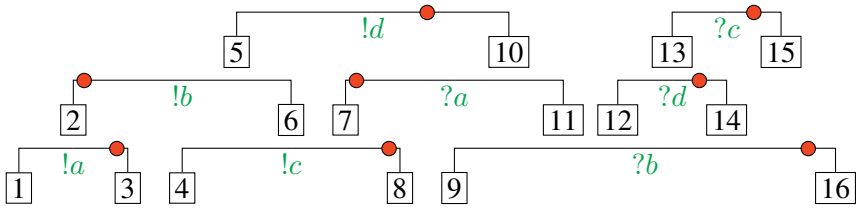


Figure 2.1. A linearizable stack history with corresponding linearization. The red dots form the total order $!b \cdot !a \cdot ?a \cdot !c \cdot !d \cdot ?d \cdot ?c \cdot ?b$ which is a linearization of the history since it is a valid sequential stack trace that respects the real-time ordering of the operations.

record the *real-time* of each such call and return, we obtain a total order of them. We refer to this total order of calls and returns of operations as a *history*. For instance, ignoring the red dots for now, Figure 2.1 is a visualization of a history

$$!a \cdot \mathsf{call}, !b \cdot \mathsf{call}, !a \cdot \mathsf{return}, \ldots, ?d \cdot \mathsf{return}, ?c \cdot \mathsf{return}, ?b \cdot \mathsf{return}$$

From this point of view, we want every history to correspond to some sequential history τ — i.e. a history in which every call is immediately followed by its return — that respects the semantics of the object we are executing.

We say a history is *linearizable* with respect to a sequential specification if we can pick a time point during the execution of each operation, such that the total order of operations induced by these time points corresponds to a sequential execution τ that respects the sequential specification.

When the intended specification is known for a history, we say it is a history of that type. For instance, we refer to a history consisting of stack operations as a stack history, regardless of whether it is linearizable. As an example, consider Figure 2.1. It is a stack history, since the operations correspond to the stack operations push and pop. Moreover, it is linearizable since we can pick time points — the red dots — during the execution of each operation, that form a sequential history satisfying the stack specification.

2.2 Read-Write Executions

In this section, we will consider executions of multithreaded shared-variable programs. Such programs read from and write to memory. Consider, for example, the program in Figure 2.2. It consists of two threads that interact with shared memory via variables x and y. If we consider the sequentially consistent (**SC**) model of concurrent executions, an execution of this program is an interleaving of the instructions of each thread that respects the ordering on each thread. As an example, $x := 1; y := 1; a := read(y); b := read(x)$ is such

Thread 1	Thread 2
1 $x := 1$	**1** $y := 1$
2 $a := read(y)$	**2** $b := read(x)$

Figure 2.2. A simple program example where x, y are memory locations, and a and b are thread-local registers. Initially all variables are set to 0.

an interleaving. The threads agree on this order, and in this interleaving, it is clear that the read $a := read(y)$ should read from the write $y := 1$.

What are the possible values for (a, b) after executing this program? In every interleaving, we can easily see that at least one of a, b must be 1. If both threads were to read 0, then both reads must occur before both writes in the interleaving, which violates the ordering of operations on the threads.

However, executing this program on modern hardware, the writes are not immediately visible to other threads. Specifically, both threads may execute their reads before the other thread's write is visible. Thus, the result that both a and b are 0 is possible on modern hardware. There is no interleaving that corresponds to this result: modern hardware generally does not satisfy sequential consistency. Modern hardware guarantees weaker properties than sequential consistency; for instance, that the effects of instructions executed on a thread are immediately visible to that thread. To reason about executions of multi-threaded programs on modern hardware, we introduce partial orders that represent relations between reads and writes that are related either causally or by time. These partial orders offer a context in which we will define memory models.

2.3 Execution Graphs

Partial orders are commonly represented via a directed graph. Specifically, for a partial order $\prec$ on a set A, one considers each object $x \in A$ as a vertex, and if $x \prec y$, then there is an edge (x, y) in the graph.

An execution graph is a specialization of this representation for executions of shared-memory programs. Rather than a single partial order (and thus edge type) $\prec$, we consider a collection of partial orders representing properties of the memory model. The program order po relates *events* — memory accesses or other synchronizing actions taken by a thread — that are ordered due to being executed in some order on the same thread. This gives us a piecewise linear order; we expect each thread to see the effects of its own actions immediately. We extend this graph with other relations representing different aspects of the behavior of a system. For instance, we know that if a thread reads a value written by some write, the write must have happened before the read. We encode

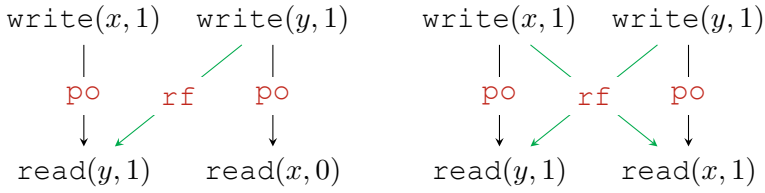


Figure 2.3. The execution graphs corresponding to two executions of Figure 2.2. The left execution graph corresponds to executions in which only the write on y reaches the other thread. The right execution graph corresponds to executions in which both writes reach the other thread.

this with the *reads-from* relation $\texttt{rf}$. Representing these relations as a graph, we obtain an *execution graph*. Each of the possible executions of the program in Figure 2.2 corresponds to some execution graph. For examples of execution graphs, see Figure 2.3.

We continue with some notation for these relations. First, given a binary relation $R \subseteq A^2$ on some set A, we denote $(e, e') \in R$ by $e \cdot R \cdot e'$. We define the inverse relation R^{-1} as the set $R^{-1} := \{(b, a) \mid (a, b) \in R\}$. We write $e \cdot R \cdot R' \cdot e'$ to denote that there exists an event e'' with $e \cdot R \cdot e''$ and $e'' \cdot R' \cdot e'$. Finally, we let $[R]^+$ denote the transitive closure of R: we have $e \cdot [R]^+ \cdot e'$ if $e \cdot R \cdots R \cdot e'$.

At baseline, an execution graph contains the following edges.

- $\texttt{po}$: The operations on any one core (or thread) are ordered by program-order, $\texttt{po}$, that relates two events $e \cdot \texttt{po} \cdot e'$ if e and e' are executed in that order by the same thread.

- $\texttt{rf}$: Every read event reads from a memory location. The $\texttt{rf}$ relation relates each read event r with a write event w that wrote the value that was read. Intuitively, the write must have happened before the read, which is encoded with a relation $w \cdot \texttt{rf} \cdot r$.

- $\texttt{hb}$: The happens-before relation is defined as $[\texttt{po} \cup \texttt{rf}]^+$. In other words, $e \cdot \texttt{hb} \cdot e'$ if there is a path $e \cdot [\texttt{po} \cup \texttt{rf}] \cdots [\texttt{po} \cup \texttt{rf}] \cdot e'$.

- $\texttt{co}_x$: The coherence order $\texttt{co}_x$ is a total order on the writes to the variable (memory location) x. In other words, if there is a total $\texttt{co}_x$-order, the threads agree on the order in which the writes reached memory: it is not necessarily the order in which they happened in real time, but the order in which their results were propagated. We also define $\texttt{co} := \cup_x \texttt{co}_x$.

- $\texttt{fr}$: The from-reads relation is defined as $\texttt{rf}^{-1} \cdot \texttt{co}$. It captures the notion that a read must precede any write that overwrites the value it read.

2.4 Weak Memory Models

The strongest memory model, **SC**, is defined as the requirement that all memory operations can be considered as being executed in some total interleaved order. This condition has an equivalent formulation in terms of execution graphs. Namely, an execution ρ satisfies sequential consistency if its corresponding execution graph G_ρ satisfies acyclicity of $[\mathtt{po} \cup \mathtt{rf} \cup \mathtt{fr} \cup \mathtt{co}]^+$ [11]. We denote this with $G_\rho \models \mathbf{SC}$. Weak memory models are often defined similarly, as acyclicity conditions on an execution graph. The edge types used in this execution graph, and the relations required to be acyclic, define the memory model.

In Paper III we consider the family of memory models that form one component of the C11 memory model [15]: Release-Acquire (**RA**) [50]. Memory accesses in the C11 memory model can be classified into three categories depending on their synchronization and ordering constraints. The least restrictive are *relaxed* accesses, for which there are no synchronization or ordering constraints. On the other end, there is support for sequential consistency, where memory operations are executed in a total order. The relaxed model makes reasoning about program correctness difficult, and sequential consistency is slow in hardware. For this reason, there is also a level in between the two: Release-Acquire (**RA**). The name Release-Acquire stems from two types of memory accesses: publishing accesses (writes) that publish the results of earlier actions on a thread (release), and acquiring accesses (reads) that acquire the published effects of other threads. In Paper III, we consider the **RA** model.

Release-Acquire consistency does not enforce a total order of memory events; it enforces only the existence of a total order per memory location. Each thread has its own program order, and each memory location has a coherence order describing how writes to that location are related. A read may observe any write not "hidden" by any other write through these relations. Specifically, any read that does not cause a $\mathtt{po} \cup \mathtt{rf} \cup \mathtt{co}_x$-cycle is permitted.

Since **RA** requires only the *existence* of a total order $\mathtt{co}_x$, we use the fact that for any partial order, there exists a total order that extends it. Specifically, we consider a minimal partial order $\mathtt{pco}_x$ respecting causality. If the $\mathtt{pco}_x$ relation is acyclic, then we know there exists a total order $\mathtt{co}_x$.

The relation $\mathtt{pco}_x$ can be deduced from $\mathtt{hb}$ on a single variable x. In fact, it is sufficient to consider the following scenario. Assume we have two write events on variable x: w and w', as well as a read event r that reads from w, i.e. $w \cdot \mathtt{rf} \cdot r$. Finally, assume we have $w' \cdot \mathtt{hb} \cdot r$. Then we know that both w and w' must happen before r. To respect causality, we need w to not be overwritten by w': we require that $w' \cdot \mathtt{pco}_x \cdot w$. Thus, for two writes w, w' on a variable x, we have $w' \cdot \mathtt{pco}_x \cdot w$ if there is a read event r such that $w \cdot \mathtt{rf} \cdot r$ and $w' \cdot \mathtt{hb} \cdot r$.

Given an execution graph without $\texttt{co}_x$-edges, to determine whether there exists some relation $\texttt{co}_x$ that yields acyclicity of $\texttt{po} \cup \texttt{rf} \cup \texttt{co}_x$, it is sufficient to check acyclicity of $\texttt{po} \cup \texttt{rf} \cup \texttt{pco}_x$, after generating $\texttt{pco}_x$ as described above.

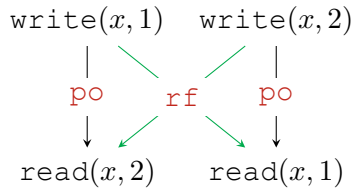


Figure 2.4. An execution that is consistent with **WRA** but not **RA** as there will be bidirectional $\texttt{pco}$-edges between the writes, forming a cycle.

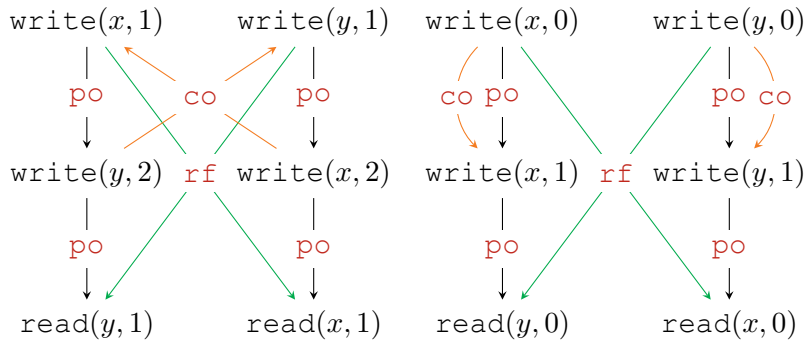
In addition to the **RA** memory model, we study two modifications of it. The first is Weak Release-Acquire (**WRA**), which allows for more behaviors. The second is Strong Release-Acquire (**SRA**), which allows fewer behaviors.

The $\texttt{co}_x$ edges are difficult to handle in model checking algorithms. Moreover, programs rarely express the patterns that **RA** uses these $\texttt{co}_x$-edges to detect [47]. For this reason, a relaxation of **RA**, which we refer to as Weak Release-Acquire (**WRA**), has emerged as a promising alternative [47]. Moreover, this model is equivalent to a model defined for distributed memory: causal consistency [10]. Memory implementations in distributed systems have the same trade-off between speed and consistency as hardware implementations do. This makes sequential consistency impractical. Many algorithms and implementations for distributed memory are designed to satisfy causal consistency. The **WRA** model allows "mutual justification" behaviors in which two reads observe values that depend on each other. To do so, **WRA** ignores $\texttt{co}$-edges, and instead explicitly forbids one type of causality cycle: cycles that are formed by following exactly one $\texttt{rf}$-edge backwards, connecting two writes to the same variable. For an example of an execution graph that satisfies **WRA** but not **RA**, see Figure 2.4.

Strong Release-Acquire (**SRA**) [50] is, as the name suggests, a strengthening of **RA**. It was developed to more accurately capture the intuitive notion of causality. Specifically, **RA** considers coherence only on the same variable through the relation $\texttt{co}_x$. The **SRA** model instead considers the union of these relations, $\texttt{co} := \cup_x \texttt{co}_x$. In fact, this model is what the compilation schemes of C11 compilers usually adhere to: the behaviors that **SRA** forbids are not observable on any known hardware under any known (sound) compilation scheme of C11 **RA** programs [50]. See Figure 2.5 for examples.

To summarize, the models we consider in Paper III are defined as the following acyclicity constraints on execution graphs.

- $G_\rho \models$ **WRA** if for any two write events w, w' on the same variable, there is no cycle of the form $w \cdot \texttt{hb} \cdot w' \cdot \texttt{hb} \cdot \texttt{rf}^{-1}$.
- $G_\rho \models$ **RA** if the relation $[\texttt{po} \cup \texttt{rf} \cup \texttt{pco}_x]^+$ is acyclic.



(a) An execution consistent with **WRA** and **RA**, but not **SRA**

(b) An execution consistent with **SRA** but not with **SC**.

Figure 2.5. Execution graphs exemplifying the difference between **RA**, **SRA** and **SC**.

- $G_\rho \models$ **SRA** if the relation $[\mathtt{po} \cup \mathtt{rf} \cup \mathtt{pco}]^+$ is acyclic.

Note that the requirement that the transitive closure of a relation is acyclic is equivalent to it being irreflexive.

2.5 Event-Driven Concurrency

Having established the linearizability correctness condition, one direction of study is the use of concurrent libraries, such as queues, to control the execution of programs. A common design pattern is that of message passing, used in e.g. OpenMPI [58]. One may also allow for both message passing and shared memory execution, such as in the Android concurrency model [57]. An execution of a program in this context has a number of *handlers* that are the computational units of the system. For each handler, there is an associated *mailbox* containing *messages*. The messages consist of sequences of instructions. These instructions are either shared memory operations (reads and writes) or *post* operations that send a new message to the mailbox of some handler. Each message starts with a special *get* event that corresponds to retrieving it from the mailbox.

Initially, one handler has an initial message. Each handler may, when not busy, retrieve a message from its *mailbox* and execute it until completion. These mailboxes are assumed to be of some specific data structure, commonly a queue or a multiset.

Modern model checking techniques, such as dynamic partial order reduction (DPOR) [30], avoid checking all possible executions by instead considering certain equivalence relations on executions, and exploring only one execution per equivalence class. We refer to these equivalence classes as *traces*. These tools need a consistency check. They need to verify that a given trace is real-

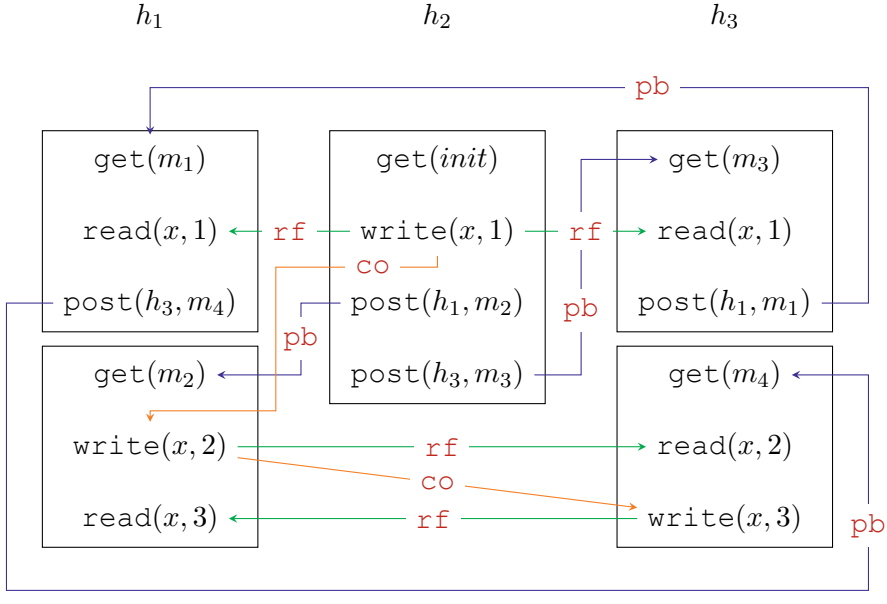


Figure 2.6. An event-driven trace with three handlers and five messages.

izable: that the system which we want the program to execute on can actually produce this trace. One straightforward equivalence relation is that of execution graphs. Different executions may generate the same execution graph, since the order of some operations need not change any of the edges in the execution graph. Thus, these tools need to check whether the current execution graph corresponds to some execution that can be produced by the system. To reuse this machinery for event-driven programs, we need to extend execution graphs and check whether they are realizable by an event-driven system.

Similarly to how `rf` encodes that a write must happen before any read that reads its value, we intuitively know that a message must be posted before it is executed. This relation is encoded via the relation `pb` that relates every post to the corresponding get. An example of an event-driven execution graph is shown in Figure 2.6. In Paper II we consider the problem of determining whether an event-driven execution is consistent with the intended mailbox semantics.

2.6 Data Independence

An implementation whose behavior is not affected by the particular values supplied to it is *data-independent*. Checking whether an implementation is data-independent is straightforward: it is sufficient to check whether any input value is ever used for branching behavior. Since a queue should always dequeue the *oldest* value, regardless of the actual value, most queue implementations

are data-independent. The same is true for many data structures, including stacks. Similar reasoning also applies to memory systems such as caches and distributed memory protocols: the contents of memory should not affect how memory is handled. For this reason, considering only data-independent implementations is often sufficient.

When considering only data-independent implementations, we can specialize our reasoning. For data-independent implementations of shared data structures, it has been shown that to determine linearizability of all executions of this implementation, it is sufficient to consider all *differentiated* executions [1]: executions where each input is used at most once. The observation is that for any linearizable stack or queue history h there is a differentiated history h', such that h' can be transformed into h by a function that maps values in h' to values in h. Similar reasoning applies to memory systems: it is sufficient to consider differentiated executions when analyzing data-independent implementations [18].

3. Efficient Linearizability Monitoring

> This was intended to be a short 3-week project.
> It was much harder than we thought.
>
> Parosh Aziz Abdulla
> Three years into the project

In this chapter, we consider the linearizability monitoring problem: is a given history linearizable? In other words, for a history h of some data structure, does there exist a sequential trace that respects the real-time ordering of events in h and satisfies the specification of the data structure? The general version of the linearizability monitoring problem is shown to be NP-complete [32], even for simple data structures such as registers.

Recall from Section 2.6 that implementations of shared data structures such as stacks and queues are often *data-independent*. For this reason, we consider only *differentiated* executions of stacks and queues, where each value may be used in at most one push operation (and thus also at most one pop operation). Thus, our work aims to solve the following problems.

Linearizability Monitoring

- Let h be a differentiated stack or queue history. Is h linearizable?
- Let h be a set or multiset history. Is h linearizable?

Note that the restriction to differentiated histories does not affect our ability to monitor real implementations. Executing an object with non-differentiated input can easily be changed to use differentiated input via a thin wrapper that uses the underlying structure, but replaces each input value a with an input (a, i), where i uniquely identifies this occurrence of a, e.g. as the count of earlier instances of a.

In the rest of this chapter, we will use *history* when we refer to an execution consisting of calls and returns, and *trace* for sequences of operations.

3.1 Stacks

In this section, we describe our algorithm for monitoring a concurrent stack. We assume a differentiated history as input, and the algorithm outputs whether the history is linearizable.

We start by defining two restrictions on stack histories. We will define our algorithm for histories satisfying these restrictions, and then extend our algorithm to not require these restrictions. The first assumption we make is that the history is *matched*: every push operation has a corresponding pop operation. The second assumption is that there are no $?-$-operations.

We note that the set of valid stack traces that are matched and contain no $?-$-operations can be defined inductively using three rules [17].

1. The empty trace, ϵ, is a valid stack trace.
2. If τ is a valid stack trace, then so is the trace $!a \cdot \tau \cdot ?a$; it is differentiated if τ is differentiated and a does not occur in τ.
3. If τ, γ are valid stack traces, their concatenation $\tau \cdot \gamma$ is a valid stack trace; it is differentiated if τ, γ are both differentiated and have no values in common.

Our algorithm recursively constructs a trace using these rules, while ensuring that it is a linearization of the input history. We define two properties of values in a history. We say that a value a is *minimal* if $!a$ is called before the return of all other operations. Similarly, we say that a is *maximal* if $?a$ returns after all operations have been called. A value that is both minimal and maximal is called *extreme*. Let $h \cdot \mathsf{extreme}$ denote the set of extreme values for a history h. Note that a value being minimal corresponds exactly to when its push operation can happen before every other operation. Similarly, a value being maximal corresponds to when its pop operation can happen after every other operation. A value that is extreme can thus "wrap" the rest of the history, as required by a in rule 2.

To use rule 3 — concatenation — we need to find a *partition* of the history h into two disjoint sets of operations h_L, h_R such that (i) they have no values in common, (ii) $h_L \cup h_R = h$, and (iii) no operation in h_R returns before any operation in h_L is called. If we find such a partitioning, and find linearizations τ of h_L and γ of h_R, we can show that $\tau \cdot \gamma$ is a linearization of h.

We define $h - S$ for a history h and set S of values to denote removing all operations on values in S from h. To use extreme values and partitions in an algorithm, we prove that a matched differentiated stack history h without $?-$-operations is linearizable if and only if one of the following three conditions applies.

1. h is empty.
2. h has a nonempty set of extreme values $h \cdot \mathsf{extreme}$, and removing all operations on these values from h yields a history $h - h \cdot \mathsf{extreme}$ that is linearizable.

3. h can be partitioned into two linearizable nonempty subhistories h_L and h_R.

At each step, we check if any of the above conditions apply, in the order they appear above. If the history is empty, we are trivially done. If there are any extreme values, we remove them from the history. Note that removing a value from a linearizable stack history cannot make it not linearizable. This is easily proven by structural induction on the rules for a stack trace, but the intuition is that any violation in a stack trace can be reduced to two values interacting in a bad way: as $!a \cdot !b \cdot ?a \cdot ?b$. Any linearization of a history h does not contain such a pattern, and it cannot be introduced by removal. We can thus safely conclude that the history h is linearizable if and only if $h - h \cdot \mathsf{extreme}$ is. Thus, we recursively call the algorithm on $h - h \cdot \mathsf{extreme}$.

This property — that a violation lies in the interaction of two values — does not hold for stack *histories*. In fact, in Paper I we show how to construct arbitrarily large histories that are not linearizable, that become linearizable when removing any one of its values. This leads to the conclusion that there is no *small model property* for stacks: you need to consider the full history to determine linearizability.

Finally, we check whether there is a partition of the history. If there is such a partition, we check each side of the partition for linearizability separately, and conclude linearizability if both are linearizable. While there may be many possible partitions, we show that the choice does not matter. If the history is linearizable, any such partition is linearizable. Similarly, if the history is not linearizable, at least one of the two sides of any partition must not be linearizable.

What remains to describe is how to (i) find extreme values and (ii) find partitions. As the history is a total order of call and return events, we can associate with each operation two indices (henceforth referred to as timestamps): the index of its call and the index of its return. We define the *inner* interval of a value as the interval $[!a \cdot \mathsf{return}, ?a \cdot \mathsf{call}]$. Intuitively, during this time, the value a *must* be present in the stack. When taking the union of any set of intervals, the result has an equivalent representation as a set of *non-overlapping* intervals. Nonoverlapping intervals are ordered, and as such, the union of a set of intervals can be seen as a *sequence* of intervals with increasing start and end timestamps. Taking the union of all inner intervals for all values in the history, we obtain a sequence of non-overlapping intervals we refer to as the *populated segments*. These are the intervals of time during which there must be *at least one* value present in the stack. Similarly, its complement is a sequence of non-overlapping intervals *during which the stack may be empty*. We refer to this set of intervals as the *deserted segments*.

Algorithm 1: Stack

```
Linearizable(h)
1 if h = ε then return true
2 if ∃a ∈ h · extreme then return Linearizable(h − h · extreme)
3 [d_1, ..., d_m] ← h · deserted
4 if m = 2 then return false
5 h_L, h_R ← Partition(h, d_2)
6 return Linearizable(h_L) ∧ Linearizable(h_R)
```

We show that a value is extreme if its push operation overlaps with the leftmost deserted segment and its pop operation overlaps with the rightmost deserted segment. Similarly, we show that a partitioning exists if and only if we can separate the history h around an *internal deserted segment* I — a deserted segment that is neither the first nor the last deserted segment — by assigning a value to h_L if its push operation returns before the start of I, and to h_R otherwise. Each push operation that returns before a deserted segment must have its corresponding pop operation called before that same segment: otherwise it would not be deserted. Similarly, any push operation that does *not* return before the deserted segment must return after it. This gives rise to a natural partitioning, in which there is a "gap" — a time point where the stack is empty — in the deserted segment. This is what ensures that linearizations of h_L and h_R can be concatenated to a linearization of h. The reason we require the deserted segment to be internal is that otherwise one of h_L and h_R is empty, and we do not reduce the size of the recursive call. Summing up, this gives us an algorithm, shown in Algorithm 1, for monitoring stack histories in $\mathcal{O}\left(n^2\right)$ time.

In Figure 3.1 we present the execution of Algorithm 1 on a history h_1. In the first step, we find that 0 and 1 are both extreme values. We know that a history is linearizable if and only if it is linearizable after removing extreme values, so we remove them and obtain h_2. There are no extreme values in h_2, so we conclude it is linearizable if and only if there is a partition. To that end, we find the populated and deserted segments, highlighted in yellow and green, respectively. For instance, the populated segment $[6, 12]$ is the union of the inner intervals for values 2 and 3. We arbitrarily choose the first internal deserted segment, $[12, 15]$, and partition the history around it. We obtain two histories: the left subhistory h_4, and the right subhistory h_5. In h_4, both values are extreme, so we remove them and obtain the empty history, concluding linearizability of h_4. In h_5, we find that 8 is extreme, so we remove it to obtain h_6. The history h_6 has no extreme values, and to check for partitionings we find the populated and deserted segments. There is only one internal deserted segment, $[22, 23]$, so we partition around it. We obtain h_7, h_8 as the left and right side of the partition, respectively. In each of h_7 and h_8, all values are extreme. We re-

move them and obtain the empty history, concluding linearizability of h_7 and h_8. Reasoning backwards from this point, we know h_7 and h_8 are linearizable, so we can conclude that h_6 is linearizable. Since h_6 is linearizable, so is h_5. Since h_4 and h_5 are linearizable, so is h_2. Finally, since h_2 is linearizable, so is h_1.

Let us now consider the general case, in which we allow pushes without corresponding pops, as well as $?-$-operations. If some set of push operations does not have matching pops, we note that in any valid linearization of the history, those values need to still be present in some order at the end. Thus, appending a set of matching concurrent pops will clear those values, and the new history is linearizable if and only if the original history is. Similarly, we note that a pop operation is allowed to return nothing ($?-$) if the stack is empty. The stack can be empty precisely during the deserted segments of the history. We show that a history with $?-$ operations is linearizable if and only if each of these $?-$ operations intersects a deserted segment, and the history without them is linearizable.

3.2 Queues

The queue algorithm is based on one key observation: that a queue history is linearizable if and only if **no** pair of values form a violation. In other words, in contrast to stacks, there is a small model property for queues. To establish linearizability, it is sufficient to consider all subhistories of size 2. If all subhistories of size 2 (i.e. with 2 values) are linearizable, the original history is linearizable. Note that for stacks, we do not have such a property. In fact, we show that there is no such property by constructing an unlinearizable history of arbitrary size, but each of its subhistories is linearizable. Specifically, a queue history is linearizable if and only if there is no pair of values a, b such that:

$$!b \cdot \mathsf{return} < !a \cdot \mathsf{call} < ?a \cdot \mathsf{return} < ?b \cdot \mathsf{call}$$

This condition corresponds exactly to when the *outer* interval of a value, defined as $[!a \cdot \mathsf{call}, ?a \cdot \mathsf{return}]$, is contained within the inner interval of some other value b. We call a pair of values satisfying this condition a *critical pair*. To check linearizability, it is sufficient to check the above condition for each pair of values.

Checking containment pairwise would incur a quadratic time complexity. To avoid this, we design a specialized data structure that we refer to as *queue tree*. A queue tree is an extension of a balanced binary interval tree [13] that encodes the set of inner intervals efficiently. Each node $n = \langle l, r, h, L, R \rangle$ in the tree represents one value, and has three numbers: the left timestamp $n \cdot \mathsf{left} = l$ of the inner interval (i.e. its enqueue-return timestamp), the right timestamp $n \cdot \mathsf{right} = r$ of the inner interval (i.e. its dequeue-call timestamp), and the

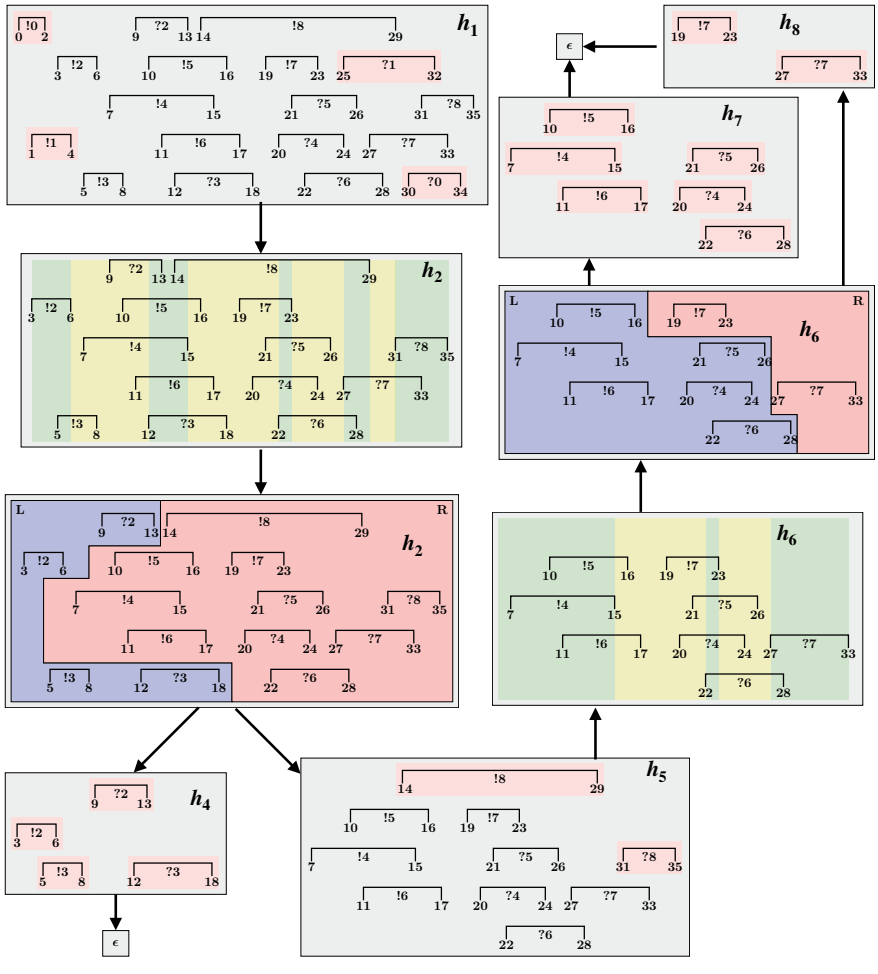


Figure 3.1. Execution of the stack algorithm on a history. The green and yellow areas are deserted and populated segments, respectively. Separations are marked by blue and red for L and R, respectively. Operations marked with red correspond to extreme values that are removed.

highest right timestamp occurring among its descendants $n \cdot \mathsf{high} = h$. Each node has two subtrees: the left subtree $n \cdot \mathsf{ltree} = L$ and the right subtree $n \cdot \mathsf{rtree} = R$. The tree is ordered by the left timestamps: the left and right subtrees of a node consist of values with smaller and larger left timestamps, respectively.

The tree can be populated efficiently, since insertion is logarithmic. To check the condition above, we need only check, for each value, whether its outer interval I is contained within any inner interval in the tree. This is done via the following recursive procedure, starting at the root node. If the inner interval

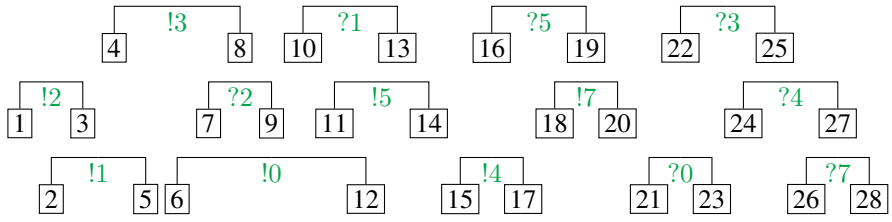


Figure 3.2. A queue history. The corresponding queue tree is presented in Figure 3.3.

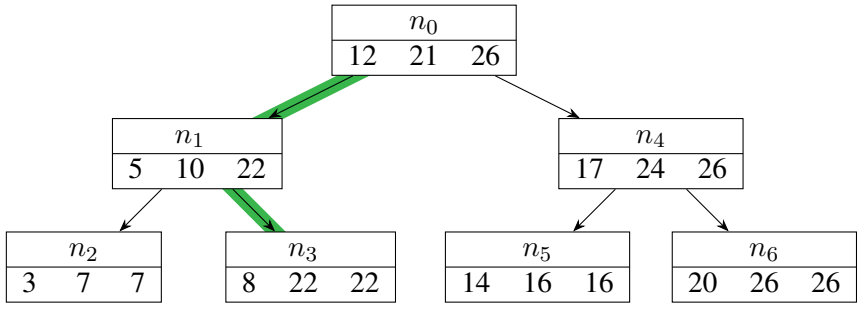


Figure 3.3. The queue tree for the history in Figure 3.2. Node n_i corresponds to value i in the history. The green edges correspond to the chosen branches when checking if the outer interval $[11, 19]$ of value 5 is contained within any inner interval.

$[l, r]$ of the current node n contains I, we return true. If we have $n \cdot \mathsf{left} \leq I \cdot \mathsf{left}$, we can conclude that $n' \cdot \mathsf{left} < I \cdot \mathsf{left}$ for all nodes n' in the subtree $n \cdot \mathsf{ltree}$. If the high key of the left subtree, $n \cdot \mathsf{ltree} \cdot \mathsf{high}$, is large enough to contain all of I, i.e. $I \cdot \mathsf{right} \leq n \cdot \mathsf{ltree} \cdot \mathsf{high}$, then we conclude that some interval in the left subtree $n \cdot \mathsf{ltree}$ contains I, and return true. Otherwise, if we have $n \cdot \mathsf{ltree} \cdot \mathsf{high} < I \cdot \mathsf{right}$, we conclude that I is not contained in any node in $n \cdot \mathsf{ltree}$, and we recursively search $n \cdot \mathsf{rtree}$ for I. The only case that remains is when $I \cdot \mathsf{left} < n \cdot \mathsf{left}$, at which point we search the left subtree $n \cdot \mathsf{ltree}$ for I.

As an example, consider the queue history in Figure 3.2 and its corresponding queue tree in Figure 3.3. The green edges indicate the path taken when checking whether the outer interval $[11, 19]$ for value 5 is contained within any inner interval. We start by noting that the left key of n_0 is too high, so we traverse to its left subtree. At n_1, we probe n_2 and see that its high key is too low to contain the interval $[11, 19]$, which leads us to follow the edge to n_3, where we detect containment: the outer interval $[11, 19]$ is contained within the inner interval $[8, 22]$. Finally, if instead $I \cdot \mathsf{left} < l$, we see that we need to pick a node whose interval starts further left, so we search the left subtree L. This containment check is logarithmic in time. Repeating it for all values yields a complexity of $\mathcal{O}(n \log n)$.

3.3 Sets

For (multi)sets, we note that a history is linearizable if and only if each of its *single-value projections* — the histories obtained by restricting the history to only one of its values — is linearizable. For these single-value projections onto a value x, we provide two algorithms. The first algorithm traverses the history in ascending time, and records the number of called and returned operations of each kind, up to that timestamp. For multisets, a necessary and sufficient condition is that the number of returned removes never exceeds the number of called adds that have occurred at that point. For sets, one must add the condition that the number of returned adds never exceeds the number of called removes by more than one, to obtain a necessary and sufficient condition. The second algorithm is a greedy algorithm for sets that linearizes operations *when needed*. At each timestamp, the algorithm keeps track of three things: the number of active operations of each type, the current state of the set (whether x is present or not), and the number of linearized active operations of each type. The number of linearized active operations represents how many operations we have chosen to linearize before they return. When we linearize an operation before its return, we increment this counter to mark that we can skip the next return. Whenever we see a call, we increment the corresponding active counter. Whenever we see the return of an add operation, we check whether the number of linearized active adds is positive. If so, we decrease both the active and linearized counters and continue. If the active linearized adds counter is zero, we see that we must linearize this operation. To do so, we first consider the state: to linearize an add operation, we need to be sure that the value is not in the set. If it is, we need to first linearize a remove operation by incrementing the number of active linearized removes. We then linearize this add operation and decrement the number of active add operations. The reasoning for removes is analogous.

We extend the second algorithm to additionally handle *membership queries*: operations $\gamma(x, \mathsf{true})$ and $\gamma(x, \mathsf{false})$. If the set contains the value x, an operation $\gamma(x, \mathsf{true})$ can be linearized. Similarly, if the set does not contain x, then $\gamma(x, \mathsf{false})$ can be linearized. We do this by tracking these queries. When a containment query is called, we check if it is requesting the current active state. If so, we can safely ignore it, as well as its return. Otherwise, we keep track of it in a separate set. Whenever we change states by linearizing an add or a remove, we clear this set, as the remaining queries can all be linearized directly after this switch. When a membership query returns, if the set is not empty, we know we need to switch states to satisfy this query. To do so, we linearize an appropriate operation, and clear the set. Each of these algorithms, including with the modification to handle membership queries, runs in linear time, $\mathcal{O}(n)$.

3.4 Evaluation

We implement our algorithms in a publicly available tool, LiMo [39], using the C++ programming language. We show that our implementations both outperform (mostly due to language choice) and scale better than the existing state-of-the-art tool Violin [28].

We also note an inherent difficulty of monitoring for linearizability. In order to test a given implementation, we need to record the times for call and return events. There are two main problems with recording these events. First, obtaining a total order of events requires synchronization of some sort. Second, the act of recording a call (or return) must occur strictly before (or after) the actual call and return of the operation, since some hardware instruction must be performed to acquire timing information. This means that each operation will inevitably become *stretched*, which might introduce new possible linearizations due to the lost precision, possibly turning an unlinearizable history into a linearizable one.

We chose to use a monotonic system clock, which guarantees that checking the current time will always yield monotonically increasing responses, possibly by sacrificing accuracy in terms of real-world time. This is a good match for our case, as we do not care about the specific times, merely the order between the events.

3.5 Mechanization of Correctness Proof

The stack algorithm and its correctness proof have been mechanized in the Lean 4 theorem prover [22]. The proof outline closely follows the pen-and-paper proof in the appendix of Paper I [6]. On the foundational level, we model intervals as pairs of integers, and define *interval sets* as sets of non-overlapping intervals. We provide the required operations on these interval sets, such as union, intersection, and complementation, and prove relevant properties. Values wrap intervals of push and pop operations, and we define inner and outer intervals accordingly. A history is modeled as an associative map with additional predicate constraints on entries, e.g. that no timestamps are shared. We implement the required operations of such a history, such as computing populated and deserted segments, removing values, and separating into L and R. The algorithm is defined as a recursive function in the same way it is defined in the paper, and we prove its correctness by induction on the size of the history.

4. Checking Consistency of Event-driven Traces

> If we knew what it was we were doing, it would not be called research, would it?
>
> Albert Einstein

Event-driven programming is a powerful paradigm for writing reactive applications, by augmenting shared memory concurrency with message passing. In addition to sharing memory, processes send messages to each other. These messages contain instructions for the receiver to execute. In this model, a process that receives messages is called a *handler*. Sent messages are placed into the recipient's *mailbox*. The mailboxes are usually modeled as queues or multisets.

Modern model checking techniques such as dynamic partial order reduction (DPOR) [30] explore only a subset of the possible executions of a program. They do so by exploiting notions of trace equivalence: that different but equivalent traces satisfy the same correctness properties. The most common form of trace equivalence is the equality of the corresponding Mazurkiewicz traces [56], which corresponds to equality of execution graphs. The DPOR algorithm reorders memory accesses in a previously explored trace, shifting `rf`- and `co`-edges, to obtain new traces. However, not all traces that can be obtained this way are feasible. If the program cannot produce a trace when executing on a system, any errors detected in that trace would be a false positive. To alleviate this, DPOR algorithms often require a consistency checking component that checks whether a given execution satisfies the restrictions of the employed programming model. Several works extend DPOR to the event-driven concurrency model [4, 53].

An execution graph in the event-driven setting contains the usual execution graph events and edges, along with *post* and *get* events. A post event represents the sending of a message, while a get event represents a handler retrieving a message from its mailbox. We define a relation `pb` (posted-by) on these events. The post event of a message is related via `pb` to the corresponding get event. To check whether this execution graph is feasible, we need to assign an *execution order* of the messages that conforms to the semantics of the mailboxes. This execution order is represented as a relation `eo` on the execution graph, connecting the last event of one message to the first event of the next executed

message on the same handler. Similarly, the order in which the post events reach a handler needs to be fixed; we need to assign a *message order* $\texttt{mo}$. In Paper II we present an algorithm that answers the following question.

Event-Driven Trace Consistency

Given an event-driven trace and mailbox semantics, are there relations $\texttt{mo}$ (message order) and $\texttt{eo}$ (execution order) that satisfy the following conditions:

1. $\texttt{mo}$ totally orders the post events to any one given handler
2. $\texttt{eo}$ connects the last event of a message to the first event of the next executed message on the same handler, such that $\texttt{eo} \cup \texttt{po}$ totally orders the events of each handler.
3. $\texttt{eo}$ and $\texttt{mo}$ are compatible with the given mailbox semantics
4. The resulting execution graph is acyclic

When there is an $\texttt{eo}$-edge from the last event of message m_1 to the first event of message m_2, we abuse notation and write $m_1 \cdot \texttt{eo} \cdot m_2$.

In Section 4.1, we present an algorithm for solving the trace consistency problem for queue and multiset mailboxes, as well as our implementation of this algorithm. In Section 4.2, we show that the problem is NP-complete.

4.1 Algorithm

In this section, we describe our algorithm, which is based on a formalization of condition 3 above. Specifically, we transform the execution graph by appending a new edge relation, such that the acyclicity of the obtained execution graph guarantees the third condition in addition to the fourth.

Consider the trace in Figure 2.6. Assume that we instantiate $\texttt{eo}$ as $m_1 \cdot \texttt{eo} \cdot m_2$ and $m_3 \cdot \texttt{eo} \cdot m_4$, and $\texttt{mo}$ as $m_2 \cdot \texttt{mo} \cdot m_1$ and $m_3 \cdot \texttt{mo} \cdot m_4$. Notice that in this particular example, the $\texttt{mo}$ relation is forced by $\texttt{hb} \cup \texttt{pb}$, otherwise we would get a cycle. This instantiation satisfies multiset semantics. However, this assignment does not satisfy the semantics of a queue mailbox: since $m_2 \cdot \texttt{mo} \cdot m_1$, the queue semantics requires that $m_2 \cdot \texttt{eo} \cdot m_1$. However, with $m_2 \cdot \texttt{eo} \cdot m_1$, we obtain a cycle. Thus, there is no feasible execution corresponding to this trace.

We present a formalization of condition 3 above: that $\texttt{eo}$ and $\texttt{mo}$ are *compatible with the given mailbox semantics*, as an additional edge type: $\texttt{do}$. This edge type represents orderings that the choice of mailbox semantics forces. For multisets, any ordering is permissible, so $\texttt{do}_{\texttt{multiset}} = \emptyset$. For queues, we define $\texttt{do}_{\texttt{queue}} = \texttt{pb}^{-1} \cdot \texttt{mo} \cdot \texttt{pb}$. To see why, we look at Figure 4.1. Intuitively, the path $\texttt{pb}^{-1} \cdot \texttt{mo} \cdot \texttt{pb}$ corresponds to a path from a message back to

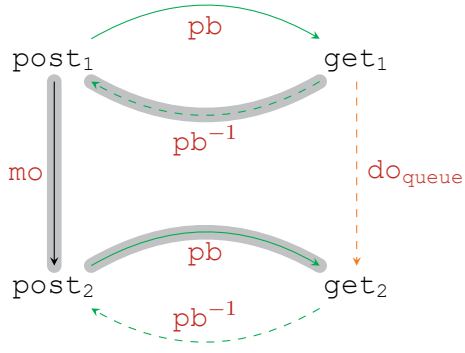


Figure 4.1. The highlighted path $\texttt{pb}^{-1} \cdot \texttt{mo} \cdot \texttt{pb}$ and the induced $\texttt{do}_{\texttt{queue}}$-edge.

its poster ($\texttt{pb}^{-1}$), to another later post (mo), and finally to the posted message pb. The existence of this path says that the messages are related by mo, i.e. (i) to the same handler and (ii) the first is posted before the second. Following the queue semantics, we now require that the first message executes before the second, and to enforce this, we add the $\texttt{do}_{\texttt{queue}}$-edge. The acyclicity of the execution graph endowed with do now additionally guarantees that eo and mo are compatible with the given mailbox semantics.

We return to the example in Figure 2.6. We will show that this trace is not compatible with queue mailbox semantics. Recall that mo is forced in this example. We see that we have a path

$$\texttt{get}(m_2) \cdot \texttt{pb}^{-1} \cdot \texttt{post}(h_1, m_2) \cdot \texttt{mo} \cdot \texttt{post}(h_1, m_1) \cdot \texttt{pb} \cdot \texttt{get}(m_1)$$

which adds $\texttt{get}(m_2) \cdot \texttt{do}_{\texttt{queue}} \cdot \texttt{get}(m_1)$. To avoid a cycle we thus need to assign $\texttt{read}(x, 3) \cdot \texttt{eo} \cdot \texttt{get}(m_1)$. However, we have now produced a cycle. Starting at $\texttt{get}(m_4)$, we can reach $\texttt{write}(x, 3)$ via po, and continue via rf to $\texttt{read}(x, 3)$ in m_2. We can now follow our newly inferred eo-edge to reach $\texttt{get}(m_1)$. Following po we reach $\texttt{post}(h_3, m_4)$. Finally, following the pb-edge we again reach $\texttt{get}(m_4)$. We can thus conclude that there is no assignment of eo that does not produce a cycle. In other words, this execution is not realizable with queue mailboxes.

We have implemented our algorithm as a publicly available tool [38] in the Rust programming language, using a library interface to the **Z3** SMT solver. Our tool uses the built-in partial order theory in **Z3** to model the problem. Specifically, from an input execution graph we create a set in **Z3** with elements corresponding to the events in the execution graph, and require a partial order $\prec$ on this set. Each po, co, rf and pb edge is then encoded as a fixed constraint on $\prec$. The condition for eo states that for every pair of messages m_1, m_2 on a handler, either $m_1 \cdot \texttt{eo} \cdot m_2$ or $m_2 \cdot \texttt{eo} \cdot m_1$. This is modeled as a disjunctive constraint on $\prec$. We make analogous constraints on $\prec$ for mo. Finally, the

$\texttt{do}_{\texttt{queue}}$-edges are modeled as implication constraints: when $\texttt{post}(h, m_i) \prec \texttt{post}(h, m_j)$, we require $\texttt{get}(m_i) \prec \texttt{get}(m_j)$. We then ask **Z3** to solve for $\prec$. If there is a $\prec$ satisfying all constraints, it is by definition acyclic and we conclude that the trace is feasible.

4.2 NP-completeness

In this section, we show that the problem of checking consistency of an event-driven trace under the queue semantics is NP-complete.

For NP-membership, we can guess $\texttt{eo}$ and $\texttt{mo}$, compute the relation induced by this guess, and check the resulting execution graph for acyclicity in polynomial time.

We show NP-hardness using a reduction from 3-BI-SAT: a restriction of the boolean satisfiability (SAT) problem, where (i) each clause contains two or three literals, (ii) each variable occurs in at most 3 clauses, and (iii) each variable appears at most once per clause. We start with a 3-BI-SAT formula φ, with variables $x_1, \ldots, x_n$ and clauses $C_1, \ldots, C_m$, and construct a trace (without $\texttt{eo}$ and $\texttt{mo}$) such that there exist orderings $\texttt{eo}, \texttt{mo}$ satisfying the queue semantics if and only if φ is satisfiable.

The reduction has two phases: variable assignment and clause satisfaction. Each phase consists of 4 handlers, and one additional handler h_W synchronizes the two phases.

Since the only source of nondeterminism is the ordering of messages via $\texttt{eo}$ and $\texttt{mo}$, we encode the assignment of a variable x as the execution order of two messages x_T and x_F. Specifically, $x_T \cdot \texttt{eo} \cdot x_F$ represents x being set to true, and if $x_F \cdot \texttt{eo} \cdot x_T$ then x is set to false. Due to technical details, we need to copy these assignments, once for each clause, to avoid making extra unintended cycles.

Variable Assignment

Initially, a single handler h_0 posts, for each variable x_i, messages $m_{i,T}, m_{i,F}$ to handlers h_1, h_2, respectively. These messages are sent in the following order.

$$m_{1,T}, m_{1,F}, \ldots, m_{i,T}, m_{i,F}, \ldots, m_{n,T}, m_{n,F}$$

Each such message $m_{i,b}$ posts the message $m_{i,1,b}$ to h_0. The 1 here corresponds to the fact that the *next clause* has index 1. The order in which $m_{i,1,T}$ and $m_{i,1,F}$ appear here corresponds to the value assigned to variable x_i, as mentioned above.

What remains is to create copies of these variables for each clause they are part of. Moreover, we need these copies to be synchronized, in the sense that the

clauses should agree on the assignment. To that end, we use handlers h_1, h_2 and a third handler h_3. The message $m_{i,j,b}$ posts $m_{i,j+1,b}$ to h_0, and if x_i is the variable in the lth position in clause C_j, it also posts $v_{i,j,b}$ to h_l. The messages $v_{i,j,b}$ on handler h_l proceed to post $w_{i,j,b}$ to handler h_W. This h_W will then contain some permutation of $w_{i,j,b}$ messages for clause C_j. Moreover, the ordering representing $x_{i,T}$ and $x_{i,F}$ is preserved. The messages on h_0 corresponding to the last clause — messages $m_{i,n,b}$ — do not send any messages to h_0.

At this point h_W contains, for each clause, a sequence of messages that encode the variable assignment, and these assignments match across the clauses. The contents of these messages will be described during clause satisfaction.

Clause Satisfaction

Next, we will need messages that encode clause satisfaction. We will construct messages for each clause such that, if the clause is false with the encoded assignment, there is a cycle. These messages are executed on new handlers $h_1^C, \ldots h_4^C$, so as not to induce any unnecessary hb-edges. Let k be the number of literals in C_j, $2 \leq k \leq 3$. We create messages $c_{j,m}$ for $1 \leq m \leq k+1$ on handler h_m^C. We will construct these messages such that $c_{j,m}$ is hb-connected to $c_{j,m+1}$ if and only if the assignment falsifies the m-th literal of C_j. Moreover, $c_{j,k+1}$ will be hb-connected to $c_{j,1}$.

We need the ordering encoding to carry over to the messages $c_{j,m}$. We do this by a sandwiching technique, which places the execution of the $c_{j,m}$ *within* the corresponding $m_{i,j,b}$ messages. To do this, we only need to induce two rf-edges; one from $m_{i,j,b}$ to $c_{j,m}$, and one in the other direction (in that order). This is easily done by writes to and reads from fresh memory locations used only for this purpose. We start with $c_{j,1}$. Assume x_i is the first variable of the clause C_j. This message contains one read $\texttt{read}(z_j, 1)$, used only to later connect it to message $c_{j,k+1}$ via hb. This read is followed by a sandwiching as described above. Specifically, if x_i appears positive, this sandwiching is to message $m_{i,j,F}$, and if it appears negative it is to $m_{i,j,T}$. The message $c_{j,2}$ starts with a sandwiching to the *other* message $m_{i,j,b}$: the one that was not used for sandwiching in the previous step. This gives the following property: if clause C_j is not satisfied by the first variable, there is a hb-path from $c_{j,1}$ to $c_{j,2}$. We repeat this for all variables, but for the final message, $c_{j,k+1}$, the final write is instead a write $\texttt{write}(z_j, 1)$, connecting back to $c_{j,1}$ via a rf-edge.

Correctness of the Encoding

We note the following:

- The construction has executions that correspond to all assignments, since the variable assignment phase allows arbitrary permutations.
- The copies of this assignment agree on valuation of each variable x_i.

- The messages corresponding to clauses form cycles if the clause is not satisfied by an assignment. Thus, if there is no satisfying assignment, there is no valid choice of eo and mo.

What remains is to show that if there is a valid choice of eo and mo, then there is an assignment satisfying the initial formula φ. The intuition is simple: if there exist choices of eo and mo, then there is an ordering of the assignment messages that does not create a cycle among any of the clause messages. This means that, for the assignment this ordering encodes, none of the clause gadgets contain a cycle. In other words, in every clause there is at least one satisfying variable that prevents the cycle. Thus, there is a satisfying assignment.

5. Verification of the Release-Acquire Semantics

Birri Birri

Elli Anastasiadi

In this chapter, we consider memory system implementations, such as cache protocols and distributed memory protocols. Such systems are the connection between the threads of a program and shared memory. When a thread reads or writes, it acts on its local cache rather than on shared memory. Subsequently, the effects are synchronized with other threads and with shared memory. This behavior is what gives rise to weak memory models.

Verification techniques for programs executing on a given weak memory model can at most prove correctness when the underlying system is guaranteed to satisfy the given weak memory model. To verify that programs behave correctly, it is thus not sufficient to consider only the programs; one must also consider the underlying memory system.

In this work, we consider the Release-Acquire subset of the C11 memory model. Recall the definitions of **WRA**, **RA** and **SRA** from Section 2.4.

- $G_\rho \models$ **WRA** if for any two write events w, w' on the same variable, there is no cycle of the form $w \cdot \mathtt{hb} \cdot w' \cdot \mathtt{hb} \cdot \mathtt{rf}^{-1}$.
- $G_\rho \models$ **RA** if the relation $[\mathtt{po} \cup \mathtt{rf} \cup \mathtt{pco}_x]^+$ is acyclic.
- $G_\rho \models$ **SRA** if the relation $[\mathtt{po} \cup \mathtt{rf} \cup \mathtt{pco}]^+$ is acyclic.

We want to verify that the implementation of a memory system, e.g. a cache protocol or distributed memory protocol, satisfies these semantics. Specifically, we consider the following problem.

Memory Implementation Consistency

Given a memory system, do all executions of all programs executing on this system satisfy the **WRA**, **RA** and **SRA** semantics?

To study this question, we first need to define what a memory system is. To analyze these memory protocols formally, we need a model for describing them. A memory system is loosely described as an interface for programs that interact with memory. When a thread writes, the memory system decides where

the data is stored. When a thread reads, the system decides from where to get the data.

The most basic model of an implementation is to model it as the set of executions it can produce. Since memory systems generally allow for arbitrarily long executions, this set needs a more practical representation. The internal state of any implementation is stored in some local memory. In real systems, this memory usage is not expected to grow indefinitely. This suggests that a finite-state model is suitable for modeling memory systems. To that end, one can consider the set of executions to be those accepted by some automaton. When modeling protocols as NFAs, the problem is shown to be undecidable for causal consistency, which is an equivalent model to **WRA** [18]. Adding the data-independence assumption, the problem is shown to be decidable for causal consistency [18], via a reduction to state reachability. As mentioned in Section 2.6, real cache implementations are often data-independent: they rarely change behavior depending on the values they store.

In this chapter, we will first describe register machines: our model for implementations of memory systems. In Section 5.2 we describe our polynomial algorithm for **WRA**. In Section 5.3 we show that the problem is PSPACE-complete for **RA** and **SRA**.

5.1 Register Machines

As a basis for reasoning about memory system implementations, we need a modeling language that is both sufficiently expressive to model cache- and distributed protocols, yet restrictive enough not to admit data-dependent behavior. We chose to model implementations by a form of *register machines*. Our register machine is a simplification of the more well-known register automata. Specifically, we remove equality checks from the model so as to allow only data-independent protocols. Yet it is sufficiently complex to allow modeling common features of memory protocols, such as vector clocks, broadcast communication, and store buffers [23].

Our register machine is a finite automaton that is equipped with finite sets of threads, variables, and registers. Recall that from the view of a program, a memory system is an interface to memory. A thread in this program may ask the memory system to store its writes, and provide values for its reads. When the thread θ wants to write a value v to a program variable x, the memory system sees this as a request. The memory system chooses some register a, to which the value v is written. Similarly, if θ wants to read from a program variable x, the system chooses a register, and returns the value stored in that register. The variables of the register machine correspond to these program variables: they are identifiers, or names, and do not store any data.

In a register machine, the request *and* the response are modeled as a single *event*. These events consist of writes, reads and copy operations.

- A write event is of the form $(\mathsf{W}, \theta, x, a)$, where thread θ asks the memory system to write some value v to variable x, and the system stores the value in register a. Note that we do not store this value anywhere in the model: we do not care for particular values, only from which write event they originate.
- A read event is of the form $(\mathsf{R}, \theta, x, a)$, where thread θ asks the memory system for the contents of variable x, and the system returns the value stored in register a.
- A copy event is of the form $a := b$, where the value contained in register b is copied to register a. This transition is internal to the memory system and is not a request from a thread.

Formally, we define a register machine as a tuple $\langle Q, q_0, \Delta \rangle$, where Q is a set of states, q_0 is the initial state, and Δ is a set of transitions of the form $q_i \xrightarrow{e} q_j$, where $q_i, q_j \in Q$ and e is an event of one of the forms described above. We assume Q and Δ to be finite sets.

Operational Semantics

We will describe the semantics of register machines through an example. The register machine in Figure 5.1 models the MSI cache protocol restricted to two threads θ_1, θ_2, and one variable x. In the MSI protocol, each thread has a local cache consisting of values associated with memory locations. The protocol tracks, for each thread and location, a local *status* of the cache for that thread and location. The local status of each thread and location is either invalid (I), indicating that the thread has no access to the variable; shared (S), indicating that the thread has shared (read-only) access to the variable; or modified (M), indicating that the thread has exclusive access to the variable. If one thread has exclusive access, all other threads have the I-status. In the register machine of Figure 5.1 there is only one variable, so there is only one local status per thread.

Each state of the register machine represents a "global status" of the implemented system. For instance, the states of the register machine in Figure 5.1 correspond to combinations of the local statuses of θ_1 and θ_2. In state MI, thread θ_1 has status M, and thread θ_2 has status I. Each outgoing transition $q_i \xrightarrow{e} q_j$ from a state q_i in the register machine describes one action, e, that the program or system is allowed to take when in that state. The state q_j corresponds to the internal state of the system after taking this action. In the register machine Figure 5.1, when thread θ_1 writes, it obtains exclusive access to x, so no other thread can read it. Thus, the resulting state after such a write transition is MI.

Note that the register machine presented in Figure 5.1 is an abstraction. For instance, in a hardware implementation of the MSI protocol, transitioning between internal states requires communication, using techniques such as snooping and cache invalidation. These details are not present in the register machine model, as we care only about global protocol behavior. For instance, in state IM, thread θ_1 may have a read instruction to execute next. In hardware, the processor would see this instruction as next in line and communicate in some way that the caches need to be synchronized, before allowing θ_1 to read.

An *run* of a register machine is a sequence of transitions between states, starting in the initial state. For instance, Figure 5.2 is a run of the register machine in Figure 5.1. For each run ρ of such a register machine, there is a corresponding execution graph G_ρ. We consider only the edge types and events relevant to Release-Acquire as defined in Section 2.4, namely read and write events, and the relations $\texttt{po}$, $\texttt{rf}$, and $\texttt{pco}_x$. The copy events do not appear in the execution graphs, but rather change from which write events a future read takes its value.

To obtain the execution graph G_ρ of the run ρ in Figure 5.2, we start from the beginning, with an empty execution graph. The first event is a write event $w = (\mathsf{W}, \theta_2, x, a_2)$, so we add w to the execution graph. The next event is a write event $w' = (\mathsf{W}, \theta_1, x, a_1)$, so we add w' to the execution graph. The next transition is a copy event and, as such, does not appear in the execution graph. Next, there is a read event $r = (\mathsf{R}, \theta_2, x, a_2)$, which we add to the execution graph. To find the source of the value in a_2, we scan backwards in ρ. The event before this read event was the copy $a_2 := a_1$. Thus, the value in a_2 comes from register a_1. We keep traversing backwards, and see the write w', that writes to a_1. Since this is the most recent write to a_1, this must be the write that r read from. Consequently, we add the edge $w' \cdot \texttt{rf} \cdot r$ to the execution graph. Finally, we see that r happens after the event w on the same thread, and as such we can add the edge $w \cdot \texttt{po} \cdot r$. To add $\texttt{pco}_x$, we can see that w is $\texttt{hb}$-related to r, while r reads from w'. Thus, we can add the $\texttt{pco}_x$-edge $w \cdot \texttt{pco}_x \cdot w'$. See Figure 5.3 for the resulting execution graph.

For a memory model M, run ρ, and its corresponding execution graph G_ρ, we write $\rho \models M$ to mean that $G_\rho \models M$. We call a run ρ with $\rho \not\models M$ a *violating* run.

5.2 Weak Release-Acquire

In this section, we outline our algorithm for checking that a memory system satisfies **WRA**. The algorithm is based on a saturation approach. In essence, we first note that we need only consider runs that end with a read event. We track suffixes of such runs that generate specific patterns in the corresponding execution graph. We use these representations to find runs that generate cycles

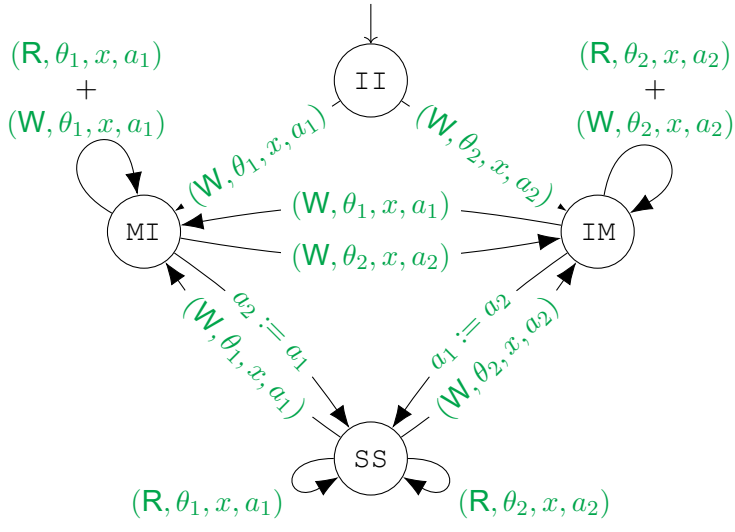


Figure 5.1. The MSI cache protocol modeled as a register machine. Each thread (in this case, θ_1 and θ_2) has its own register (a and b, respectively) for storing the variable x. A state XY of the register machine corresponds to thread θ_1 having status X and θ_2 having status Y. The statuses correspond to what type of access the thread has to the variable x. The first status is I, Invalid. This corresponds to a thread not having an up-to-date value of the variable. The status M, Modified, means a thread currently has exclusive (write) access to the variable. The status S, Shared, means a thread has shared access, i.e. read access, to the variable x.

$$II \xrightarrow{(\mathsf{W},\theta_2,x,a_2)} IM \xrightarrow{(\mathsf{W},\theta_1,x,a_1)} MI \xrightarrow{a_2:=a_1} SS \xrightarrow{(\mathsf{R},\theta_2,x,a_2)} SS$$

Figure 5.2. A run of the MSI register machine from Figure 5.1

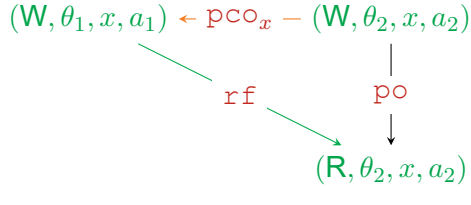


Figure 5.3. The execution graph obtained from the run in Figure 5.2, together with the inferred $\mathtt{pco}_x$-edge.

in the execution graph, and thus violate the requirements for **WRA**. We will now present the algorithm.

From the definition of **WRA**, in the execution graph corresponding to a violating run ρ there is a cycle $w \cdot \mathtt{hb} \cdot w' \cdot \mathtt{hb} \cdot \mathtt{rf}^{-1}$. We note that the $\mathtt{rf}^{-1}$ relation must be from a read event r. Thus, the cycle is of the following form.

$$w \cdot \mathtt{hb} \cdot w' \cdot \mathtt{hb} \cdot r \cdot \mathtt{rf}^{-1} \cdot w$$

We say the event w' *hides* the write w from r. Both of the involved write events must be strictly before r in ρ, as they are $\mathtt{hb}$-related to r. This means that when looking at a run ρ, adding one event at a time to the execution graph G_ρ in the order they appear in ρ, the point at which we can detect the violation is when we reach r. This is when the cycle appears in the execution graph. It is thus sufficient to only consider runs that end with a read event, i.e. runs of the form

$$\rho_1 \cdot w \cdot \rho_\beta \cdot w' \cdot \rho_\alpha \cdot r$$

where ρ_α relates w' to r via $\mathtt{hb}$, and ρ_β relates w to w' via $\mathtt{hb}$.

We search for such runs by constructing representations of them, backwards, from every transition in the register machine labeled with a read event. Specifically, we capture representations of $\mathtt{hb}$-paths that would be created in the execution graphs. We call these representations *tuples*, of which there are two types, corresponding to whether we have seen the event w' or not. The first tuple type, fragile, represents runs of the form $\rho_\alpha \cdot r$. The second tuple type, exposed, represents runs of the form $\rho_\beta \cdot w' \cdot \rho_\alpha \cdot r$.

To find a run ρ with $\rho \not\models$ **WRA**, we will first create fragile tuples from every read transition. These are propagated backwards, until we find a write event that could act as the event w' in a violating run. We then create a exposed tuple. This is also propagated backwards, until we find a write that completes the cycle in the corresponding execution graph, and conclude that we have found a violating run. For the formal saturation rules, we refer to Paper III.

We implement this algorithm in a publicly available tool, ccchecker [37], using the Rust programming language. We show the expressivity of the register machine model by modeling protocols. Specifically, we model one protocol that uses broadcast communication and message buffers [64], and one protocol that uses vector clocks [10]. Specifically, we generate the register machines corresponding to bounded versions of these protocols. Our implementation shows that these register machines satisfy **WRA** for small bounds on the size of the message buffers, and on the number of threads and variables. These results are not part of Paper III, but will be part of a future tool paper once we have implemented algorithms for **RA** and **SRA**.

5.3 Release-Acquire and Strong Release-Acquire

In Paper III, we show that the problem of checking whether a register machine satisfies **RA** and **SRA** is both NP- and coNP-hard. We also show that it is in PSPACE. After publication of Paper III, we found a reduction from DFA intersection non-emptiness that proves the problem is PSPACE-hard, thereby establishing PSPACE-completeness. This result will be included in a journal version of the paper. In this section we outline our PSPACE-hardness construction for the system consistency problem for **RA** and **SRA**. We also provide a PSPACE algorithm, thereby showing PSPACE-completeness.

5.3.1 PSPACE-Membership

Consider a register machine with a run ρ that violates **RA**. This run can be arbitrarily long, and so can the cycle in the corresponding execution graph. However, if such a chain enters and leaves a thread θ more than once, we can use the $\texttt{po}$-edges on that thread to "short-circuit", and produce a smaller cycle. Thus, it is sufficient to consider bounded runs: runs that only enter and leave each thread at most once. Our algorithm is nondeterministic. We guess such a bounded path, and we guess the transitions that correspond to adding edges that cross between threads. Finally, we guess an order for these events that produces the cycle. After making these guesses, we check if the machine can produce a run that takes these transitions, in the specified order, so that the correct $\texttt{rf}$ is created.

Whether the guessed transitions are reachable from each other, in the order they appear in the guess, is easy to check in polynomial time. The difficult part is to track from which write the value at each register originates, so as to produce the correct $\texttt{rf}$-edges. We can track these values as an array of size $\mathcal{O}(n)$. We run the register machine non-deterministically, and at each step guess whether we take one of the reserved events for the cycle, or if we change the configuration — the contents of the array corresponding to register values — of the register machine through some other transition.

If we can make such guesses, we can conclude that the register machine can produce a run that violates **RA**. Similarly, if we cannot make such guesses, there is no run that violates **RA**.

While we need to make exponentially many guesses, the amount of space required is polynomial. Thus, checking **RA** is in NPSPACE, which is equal to PSPACE.

5.3.2 PSPACE-Hardness

We reduce from DFA intersection non-emptiness, where, given k DFAs over a shared alphabet Σ, we want to know whether there is some word $\omega \in \Sigma^*$ such

that ω is accepted by each of the k DFAs. Note that k is part of the input: for fixed k the problem is solvable in polynomial time.

The construction is divided into three phases. First, there is an initialization phase, where we initialize registers corresponding to the acceptance (or rejection) of each automaton. Second, we encode the concurrent execution of each of the automata through a loop that takes one step for each automaton. Finally, there is an output phase that creates a cycle in the execution graph if the path chosen encodes a word accepted by all automata.

For our reduction to work, we need each automaton to have a unique final state. To that end, we start with an arbitrary instance of the DFA intersection non-emptiness problem. We are given k DFAs $D_0, \ldots, D_{k-1}$. We add a termination symbol # to Σ, and a new state $q_{i,f}$ to each automaton D_i, which we say are the new final states. For every final state $q_{i,j}$ in the input DFA D_i, we add a transition $q_{i,j} \xrightarrow{\#} q_{i,f}$. These new states are now the only final states of D_i. It is easy to see that these modified DFAs accept the same languages, except that each accepted word is appended with the termination symbol. Thus, intersection non-emptiness is preserved. This is crucial, since we can now ensure the configuration that corresponds to all automata accepting a word is unique.

Formally, using the traditional tuple notation for DFAs, the input after the above transformations is a set of DFAs $D_i = \langle Q_i, q_{i,0}, \Sigma \cup \{\#\}, \delta_i, q_{i,f} \rangle$, where Q_i is the set of states, $q_{i,0}$ is the initial state, $\Sigma \cup \{\#\}$ is the shared alphabet, δ_i is the transition function, and $q_{i,f}$ is the final state. We will refer to the states of D_i as $q_{i,j} \in Q_i$ for $0 \leq j < |Q_i|$. When indexing over such i, we implicitly assume the indices are modulo k. For instance $q_{-1,j} = q_{k-1,j}$.

We will construct a register machine $\mathcal{M}$, whose size is polynomial in the size of the k DFAs, that admits a **RA**-violation if and only if there exists some word accepted by each of the k DFAs. The register machine $\mathcal{M}$ will have:

- One thread θ_i per automaton D_i, and one extra thread θ_{nil}.
- A single variable x.
- For each automaton D_i and state $q_{i,j} \in Q_i$, a register $r_{i,j}$. In addition, there will be one register r_{nil}.

Informally, a "non-nil" value in a register $r_{i,j}$ during some execution of $\mathcal{M}$ models that the automaton D_i is in state $q_{i,j}$. To distinguish from states $q_{i,j}$ in the DFAs, we will denote the states of the register machine $\mathcal{M}$ with the letter m. Moreover, when we quantify over registers $r_{i,j}$, we do *not* include the register $r_{i,f}$.

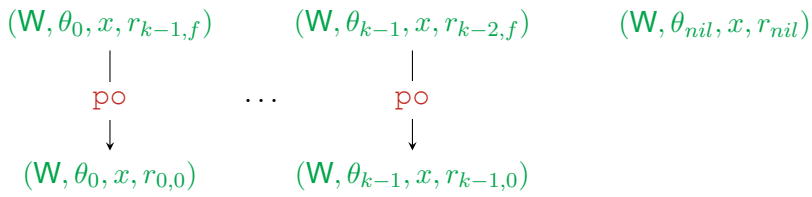


Figure 5.4. The execution graph at the end of the initialization phase in the PSPACE-hardness reduction for **RA**.

Initialization Phase

We begin by describing the initialization phase. This phase consists of each thread performing two writes. Locally, $\mathcal{M}$ will have states and transitions

$$m_{2i} \xrightarrow{(\mathsf{W},\theta_i,x,r_{i-1,f})} m_{2i+1} \xrightarrow{(\mathsf{W},\theta_i,x,r_{i,0})} m_{2i+2}$$

Note that state m_{2i+2} is the same state as $m_{2(i+1)}$. This means these writes are linked in a single chain, starting at m_0 and ending at state m_{2k}. We add a write transition on thread θ_{nil} that leads to a new state m_{nil}.

$$m_{2k} \xrightarrow{(\mathsf{W},\theta_{nil},x,r_{nil})} m_{nil}$$

Before we exit the initialization phase, we need to copy the nil value (written to r_{nil} by θ_{nil}) to each register not corresponding to the initial or final state, so that the register machine can never read from uninitialized memory. To do so, we insert a chain of copy events $r_{i,j} := r_{nil}$ for $j \neq 0$.

$$m_{nil} \xrightarrow{r_{0,1}:=r_{nil}} \cdots \to m_{nil,i,j} \xrightarrow{r_{i,j}:=r_{nil}} \cdots \to m_{main}$$

Since this part of the register machine only allows for a single execution, the corresponding part of the execution graph will always be the same; as presented in Figure 5.4. Note that the first write instruction of each θ_i writes to the "previous" thread's register, and the second to the "current" thread's register. Arranging these writes on neighboring threads in this way will enable us to induce the $\mathtt{pco}_x$-edges that end up producing cycles in the output phase.

Execution Phase

The goal of this phase is to simulate the execution of each DFA over a non-deterministically chosen string w. For each symbol a of $\Sigma \cup \{\#\}$, there is an outgoing transition from m_{main}. Following these transitions leads to paths that encode taking an a-labeled transition from the current state of each DFA. Specifically, from state m_{main}, there are transitions $m_{main} \xrightarrow{\epsilon} m_a$ for each $a \in \Sigma$. Here, ϵ is an empty operation that can be modeled using e.g. copy operations $r := r$ for some register r not used anywhere else. This corresponds to non-deterministically choosing the next symbol to be a.

From each state m_a there is a chain of transitions that correspond to taking the transitions labeled a of each DFA. Specifically, for each state m_a and each DFA D_i, there is a set of states $M_{\Sigma,i} = \{m_{a,i} \mid 0 \leq i \leq k\}$. Each such state $m_{a,i}$ initiates a chain that corresponds to taking an a-transition in D_i. To initiate this chain, there are transitions $m_a \xrightarrow{\epsilon} m_{a,0}$.

For each state in $M_{\Sigma,i}$, we create a set of states $M_{\Sigma,i,j} = \{m_{a,i,j} \mid 0 \leq j < |Q_i|\}$. Entering state $m_{a,i,j}$ corresponds to taking an a-transition from state $q_{i,j}$. For every symbol $a \in \Sigma$, DFA D_i and state $q_{i,j}$ in D_i with $\delta_i(q_{i,j}, a) = q_{i,j'}$, we create transitions $m_{a,i} \xrightarrow{r_{i,j'} := r_{i,j}} m_{a,i,j}$. If $r_{i,j}$ was storing a non-nil value, i.e. we encoded that DFA i was in state $q_{i,j}$, we see that we have now taken a transition and ended in state $q_{i,j'}$. However, after the copy, $r_{i,j}$ still has a non-nil value. To retain the connection between registers and states, we need to reset $r_{i,j}$ to a nil value. In fact, to prevent cycles from appearing when taking transitions from a state that D_i is not in, we need to reset *all* $r_{i,l}$ for $l \neq j'$. We do this with a chain of copy transitions of the form $r_{i,l} := r_{nil}$, starting from state $m_{i,j'}$, and ending at state $m_{a,i+1}$. Finally, after each automaton has taken a transition, we put a final ϵ-transition back to m_{main}, to allow selecting the next symbol from $\Sigma \cup \{\#\}$.

We make the following observations. When entering the execution phase, there are two registers for each thread with non-nil values: $r_{i,0}$ and $r_{i,f}$. Each path taken from m_{main} initiates a chain of transitions that correspond to taking a transition with the same label in each DFA. If we take a transition in this chain that corresponds to some $q_{i,j}$ that is *not* the current state (i.e. $r_{i,j}$ is nil), then the reset-chain described above would make *all* registers $r_{i,j}$ contain a nil value. The only way to keep the non-nil value in exactly one register $r_{i,j}$ is to take the paths that correspond to valid transitions and valid simulation of a word. The final transition, with the termination symbol #, overwrites the write to $r_{i,f}$. If the word corresponding to the taken execution is accepted by the automaton, the replacement will be from the write to $r_{i,0}$. Thus, whenever $\mathcal{M}$ is in state m_{main}, either we have taken a valid path and for each DFA D_i there is exactly one $r_{i,j}$ that has a non-nil value, or for at least one DFA D_i, all $r_{i,j}$ are nil. We are now ready to describe the final phase: the output phase.

Output Phase

In the output phase, we will perform output operations that produce a cycle in the execution graph if and only if the path taken corresponds to a word that is accepted by each DFA. It is reached with an ϵ-transition $m_{main} \xrightarrow{\epsilon} m_{out}$. Recall the execution graph at the end of the initialization phase, as shown in Figure 5.4. The execution phase does not change this execution graph as there are no read or write transitions in that phase. Thus, when arriving at the output phase, the execution graph is the same.

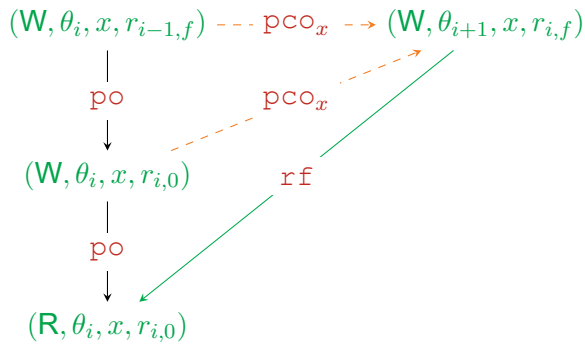


Figure 5.5. The local execution graph corresponding to an execution in which the path taken in the execution phase corresponds to a word accepted by DFA D_i.

From m_{out}, $\mathcal{M}$ contains a sequence of read operations.

$$m_{out,i} \xrightarrow{(\mathsf{R},\theta_i,x,r_{i,f})} m_{out,i+1}$$

If the path taken in the execution phase corresponds to an accepting word in automaton D_i, then the value written to the initial state by thread θ_{i+1} in the initialization phase will have been copied to $r_{i,f}$. Thus, we will get an $\mathtt{rf}$-edge, that induces two $\mathtt{pco}_x$-edges as shown in Figure 5.5. If all automata accept this word, we will get a cycle of $\mathtt{pco}_x \cup \mathtt{po}$-edges.

Correctness of Construction

We start by stating that the construction is indeed polynomial. For an instance of the problem with k DFAs, let n be the maximum number of states in a DFA D_i. The register machine $\mathcal{M}$ will have $\mathcal{O}(k)$ threads and $\mathcal{O}(k \cdot n)$ registers. To compute a bound on the number of states, we note that the execution phase contains the most states. First, there are $\mathcal{O}(|\Sigma|)$ states of the form m_a: one for each $a \in \Sigma$. For each such state, there are k states of the form $m_{a,j}$, i.e. $\mathcal{O}(|\Sigma| \cdot k)$. For each such state, there is a state $m_{a,i,j}$ for each a-transition in D_i, which leads to a bound of $\mathcal{O}(|\Sigma| \cdot k \cdot n)$. For each of these states, there is a chain that copies r_{nil} to registers $r_{i,j}$. The length of each such chain is bounded by n: there is one transition per state in D_i. Thus, there are $\mathcal{O}\left(|\Sigma| \cdot k \cdot n^2\right)$ such states, which is an upper bound on the number of states in the construction. The same reasoning applies to the number of transitions in $\mathcal{M}$, and yields a similar polynomial bound.

What remains to be shown is that $\mathcal{M}$ has an execution that violates **RA** if and only if there is some word w that is accepted by each DFA D_i. For a cycle to be formed in the execution graph, we first need to note that there cannot be any cycles involving the write by θ_{nil}, as there can never be a $\mathtt{pco}_x$-edge *from* its write: no read sees this write via $\mathtt{po}$, so it will always be safe to place it "last". Thus, this write cannot partake in a cycle. Moreover, any cycle must

go across threads. The $\texttt{pco}_x$-edges to and from writes on different threads can only be formed if the value read by θ_i in the output phase originates from the write by θ_{i+1}. Recall that during the execution phase, when there is exactly one register $r_{i,j}$ that is non-nil, this corresponds to the current state of DFA D_i being $q_{i,j}$. This non-nil value originates from the write by θ_{i+1}. Consider an execution that corresponds to a word not accepted by some DFA D_i. The value in register $r_{i,0}$ will never be copied to register $r_{i,f}$, since at some point the execution will have to take a transition that is not from its current state. This can be the transition corresponding to the terminal symbol. Taking such a transition will copy the value of r_{nil} to all registers $r_{i,j}$. Thus, for this $\texttt{pco}_x$-edge to be formed, the execution must correspond to a word accepted by D_i. Moreover, the only way for a cycle to form is if *all* of these $\texttt{pco}_x$-edges are formed. In other words, when all of the DFAs accept the word encoded by the path taken in the execution phase.

6. Personal Contributions

All work in this thesis is a joint effort with the coauthors. In this section, I outline my personal contributions to each of the papers.

6.1 Paper I

As the main author of this paper, I developed the algorithms, proved their correctness — both pen-and-paper and in Lean — and implemented them. I also wrote most of the text in the paper.

6.2 Paper II

All authors contributed equally to this work. I was responsible for the implementation and the experiments.

6.3 Paper III

All authors contributed equally to this work. I implemented the algorithm for **WRA** as a publicly available tool [37], and co-developed the hardness results and algorithm.

7. Related Work

Linearizability has been extensively studied since its inception by M. Herlihy and J. M. Wing [41]. The methodologies for verifying implementations range from fully automatic to semi-automatic and manual. Fully automatic verification faces scalability limits due to unbounded data and dynamic heaps [7]. When implementations are modeled as regular languages, checking linearizability is EXPSPACE-complete [12, 40]. There are several tools for verifying linearizability of an implementation, such as Line-Up [19], which builds on the static model checker CHESS [61]. Using similar techniques, and the model checker SPIN [42], the tool PARAGLIDER [69] aims to aid programmers in creating linearizable implementations.

Verifying implementations using semi-automatic approaches is less computationally expensive, at the cost of requiring more user input. For instance, by assigning linearization points to operations [1], verifying an implementation is PSPACE-complete when the implementation is modeled as a regular language [16], and EXPSPACE-complete when the implementation is a finite-shared-memory program [17].

Other works focus on manual approaches, proving data structure implementations to be linearizable using theorem provers [24]. Several libraries exist that enable formal reasoning about linearizability of implementations, such as Iris [59, 60] and fine-grained concurrent separation logic (FCSL) [66].

More recent work has analyzed the linearizability of implementations running under weak memory models [67, 36].

The linearizability monitoring problem for arbitrary data structures has been shown to be NP-complete [32]. Nevertheless, improvements to the naive exhaustive search approach have been developed in tools such as VeriLin [46] and Porcupine [14]. Specializing to specific structures, early work showed that without the assumption that histories are differentiated, monitoring stacks, queues, priority queues, and counters is NP-complete, that monitoring differentiated queues is $\mathcal{O}(n)$, and that monitoring differentiated stacks is polynomial [31]. Their queue algorithm assumes constant time access to array elements, which we do not assume in Paper I. We additionally handle (multi)sets and mechanize our results in a theorem prover.

Later work developed specialized algorithms for stacks, queues, and sets. In Paper I, we show that they were either incorrect [65], or that their correctness

proofs were unsound [28, 27]. These works highlight the need for formally verified monitoring algorithms, which we provide in Paper I. Later work has independently developed algorithms similar to those presented in Paper I [51].

There are several variants of linearizability. Examples include strong linearizability [35], interval-linearizability [20], set-linearizability [62], and more recent work on durable linearizability [25, 44]. Verification problems on these models are less studied, and no general complexity classification is known. Other correctness conditions for libraries include serializability [29], which removes the real-time constraint from linearizability. Serializability has been both extended and studied in several works [43, 63, 70].

Verification of programs with shared-memory concurrency has been studied extensively. A well-established technique is that of stateless model checking (SMC) [33]. SMC has been implemented in several tools, e.g. VeriSoft [33], CHESS [61], Concuerror [21], Nidhugg [2] and GenMC [49]. These tools have been used to verify real programs, such as telephone switches [34] and parts of the Linux kernel [48]. For event-driven programs, prior work developed partial-order reduction and stateless model checking techniques [45], some explicitly focusing on Android programs [54, 55]. SMC tools often reduce the set of examined traces by examining only traces that admit identical behavior, e.g. have the same corresponding Mazurkiewicz trace [56]. A successful approach for reducing the set of considered executions is dynamic partial order reduction (DPOR) [30].

DPOR techniques need to check that the executions they explore are consistent with the underlying program and memory model: that there exists some execution of the program that corresponds to a generated execution. This consistency check is an important component of several DPOR algorithms.

The consistency problem for a trace has been extensively studied for various memory models. Notable examples include the NP-hardness results for **SC** [32], causal consistency [18], the Release-Acquire model from C11 [68], and **TSO** [11]. The consistency problem for **WRA** has been shown to be decidable [18]. The trace consistency problem for the event-driven model where the mailboxes are modeled as multisets is also NP-hard [4]. In Paper II we prove NP-hardness for the trace consistency problem with queue mailboxes, and provide a restatement of the problem that is amenable to implementation in model checking tools like GenMC [49].

At the language level, the C11/C++11 weak memory semantics have been formalized in foundational works [15]. The central component of this model, **RA**, has been relaxed (**WRA** [47]) to enable more efficient verification [47], and strengthened (**SRA** [50]) to more precisely capture the guarantees provided by compilers [50].

Several distributed protocols exist in the literature that implement various memory models. Notable examples include [64, 10].

The verification of a memory system with respect to a memory model is undecidable in the case of **SC** [12] and causal consistency [18] when modeled as a regular language. However, requiring *data-independence* — that the particular values used do not change the behavior — the problem is solvable in polynomial time [18] for causal consistency. In Paper III we consider the memory system verification problem for the **RA** family of models. We develop an algorithm for **WRA**, and establish complexity results for **RA** and **SRA**.

8. Conclusion

This thesis studied three complementary verification problems for concurrent computation: correctness of concurrent objects from observed histories, consistency of event-driven traces, and consistency of memory system implementations with weak memory semantics. Together, these problems span software libraries, execution models, and the underlying memory systems, and they address a common challenge: reasoning about correctness when behavior is only partially ordered.

In Paper I, we considered the linearizability correctness criterion and presented monitoring algorithms for stacks, queues, and (multi)sets. Our algorithms are efficient and are accompanied by correctness proofs. The runtimes of our algorithms are $\mathcal{O}\left(n^2\right)$, $\mathcal{O}\left(n \log n\right)$, and $\mathcal{O}\left(n\right)$ for stacks, queues, and (multi)sets, respectively. We also implemented our algorithms in a tool, LiMo. Finally, we mechanized the stack algorithm and its correctness proof in a theorem prover.

In Paper II, we studied trace consistency for event-driven programs under queue mailbox semantics. We presented a restatement of the problem suitable for extending existing model checking algorithms, and showed NP-completeness. This contribution supports modular stateless model checking by providing a foundation for a consistency-checking component tailored to event-driven executions.

In Paper III, we considered the consistency problem for memory systems, such as cache protocols, under a register machine model that captures only data-independent implementations. We presented a polynomial-time algorithm for **WRA**. We also showed NP- and coNP-hardness for **RA** and **SRA**. We later established PSPACE-completeness for **RA** and **SRA**, which will be included in a journal version of the paper.

This thesis contributes to an established line of work on concurrency correctness by providing concrete checking algorithms and complexity results across three settings: histories of concurrent objects, event-driven traces, and memory system implementations.

9. Future Work

9.1 Linearizability

The most promising future direction is to extend our approach to other criteria and models. There are many other variants of linearizability, e.g. interval-linearizability [20] and set-linearizability [62], for which developing monitoring algorithms is a possible direction for future work. Work on persistent architectures has presented the *durable linearizability* criterion [25] — linearizability in the presence of crashes — and this is one direction for future monitoring algorithms. One may also extend model checking approaches, e.g. [36], by replacing the internal trace consistency check with our efficient algorithms to possibly obtain better performance.

9.2 Event-Driven Concurrency

One of the main use cases for this type of consistency check is its incorporation into model checking software. For instance, the model checker GenMC [49] is built to be modular and extensible, and our algorithm can be an effective component for enabling GenMC to handle event-driven programs. Similarly, there is an extension to the Nidhugg [3] model checker for event-driven programs with a multiset mailbox [4]. Implementing our algorithm to extend model checkers to event-driven programs with queue mailboxes is a main direction for future work. Another direction for future work is to extend this approach to other memory models, e.g. **RA**, coupled with the event-driven semantics.

9.3 Weak Memory Models

The first step for future work is to implement a PSPACE algorithm for **RA** and **SRA**. On a more theoretical level, one direction for future work is to consider modifications to the register machine model we used in Paper III. One such modification is to extend to parametric implementations, where e.g. the number of threads and registers is unbounded. There is substantial work on parameterized verification (e.g. [26, 5, 8]). It is possible that some techniques used in these works can be extended to the memory consistency problem. Other directions include finding minimal restrictions to the model to obtain better complexity results for **RA** and **SRA**, examining other memory models such as **PSI** and **TSO**, and extending the register machine model to capture, for example, *compare-and-swap* events that are used in many distributed protocols.

10. Acknowledgements

> Let us be dreamers, thinkers, speculative philosophers, or as our spouses would have it: Idiots
>
> Douglas Adams

I want to thank my supervisors Bengt Jonsson, Parosh Aziz Abdulla, and Mohamed Faouzi Atig for their support and guidance throughout my PhD. Bengt's pragmatic approach and thorough comments on my work have made it much more presentable, and I greatly appreciate his feedback. Special thanks to Parosh, with whom I've worked the most. His approach to research has helped me immensely. He has helped me learn how to simplify difficult problems: to check for special cases, to test theories, and to find out *why* they failed. Throughout my PhD, he has been supportive, inspiring, and a joy to be around.

I wish to thank my colleagues, many of whom have become my friends, for interesting discussions, board game nights, and staying late at the DoCS fika. Spending time with you, both in and out of the office, traveling for conferences, and playing board games, has made my PhD so much more enjoyable. Special thanks to those who have endured me the most: Fredrik, Stephan, Sarbojit, Elli, Nick, Paul, Frej, and Govind. This journey would not have been the same without all of you.

For most of my time as a PhD student, I shared an office with Fredrik. We tried to have plants, but it turns out they need water. So, instead, we thought we could get a cactus. This cactus, of course, could not just stand on the table. No, that would be boring. We decided we wanted to hang it above the door, as a trap for anyone who dared to enter and interrupt our discussions, which were obviously very science-related and definitely not about using the fire alarm to solve the aforementioned watering problem. We never did get the cactus, but whenever I see one, I think of our well-thought-out trap.

I want to thank my friends, who have helped me stay (at least somewhat) sane, listed in lexicographical order: Anna, Arvid, Elin, Gustav, Hacke, Joel, Joachim, Jonathan, Julia, Miranda, Perra, Sofia, and Timothy. Thank you for all the board game nights, quizzes, holidays and birthdays we have celebrated together! Perra has been my closest friend since we met on the first day of gymnasiet. We have watched countless bad movies (and a few good ones), talked about everything (and a lot about nothing), supported each other when

times were tough, and shared many bottles of wine. If not for you, and for how we encouraged each other, I do not think I would have done nearly as well during gymnasiet, and I probably would not have made it this far. I met Joel when I started studying mathematics at Uppsala in 2013. I still think the set of board games we have played together is countable, but I fully expect that set to become uncountable eventually.

I want to thank my family and relatives. Kristina, Joakim and Rebecka, Lukas and Py, Albin and Elin, Elton and Lykke, Barbro and Gösta, Ann-Sofi and Stefan, Lin, Elvira, and Michael. Your support has been endless, as has your interest in what I do. Celebrating holidays and birthdays and spending summers in Narvetorp has helped me stay grounded. An extra thank you to my mother, Kristina, for drawing the cover of this thesis (and almost every other painting in my home)!

My loving partner, Malin, has been my rock. We have traveled the world together and built a safe, loving home together. Her positive attitude and her encouragement have helped stave off impostor syndrome, and she has helped me transform from a stressed student without a sense of purpose, whose keys were permanently misplaced, into a somewhat functioning human being: one that admittedly still cannot find his keys. I look forward to sharing many more years of gardening, board games, laughter, and dinners with you!

And last, but not least, my cats: Alice and Louise. They have consistently reminded me to take breaks from work and helped me relax during times of stress. They have comforted me throughout my PhD, and they have sat in my lap all day whenever I worked from home.

Thank you to all who have supported me, all who have taught me, all who have challenged me, and all who have kept me grounded. I would not be where I am today if not for you!

11. Sammanfattning på Svenska

Datorer är en central del av våra samhällen och liv. När program innehåller buggar kan det få förödande konsekvenser. Sedan datorer började användas i säkerhetskritiska miljöer har medicinsk utrustning överdoserat vid strålningsterapi och flygplan har tappat kontroll mitt under pågående flygning. Att garantera att mjukvara inte innehåller buggar — att verifiera den — är mycket svårt.

Den intuitiva modellen för hur en dator fungerar är att den följer en lista med instruktioner och kör dem i ordning. Redan i denna modell är många problem bevisade att vara oavgörbara, det vill säga omöjliga att lösa systematiskt. Andra problem kan kräva flera tusen år av beräkningstid. Men den intuitiva modellen är en grov förenkling av hur datorer fungerar idag. I jakt på snabbare datorer har vi offrat mycket av garantierna som medföljer de förhållandevis enkla sekventiella processorerna. En processor har idag flera *kärnor*, var och en med sin egen instruktionslista. För att verifiera ett program som körs på sådana processorer räcker det därför inte att endast följa en sekvens av instruktioner, utan man måste ta hänsyn till alla möjliga sätt dessa listor av instruktioner kan sammanflätas. Det ger en exponentiell tillväxt av möjliga körningar, vilket leder till kraftigt ökad komplexitet för de flesta verifieringsproblem.

Men även det är en förenkling: instruktionerna som genomförs av en kärna är inte omedelbart synliga för andra kärnor; de kan ha olika uppfattning om vad minnet innehåller. Trots att verkligheten inte är så enkel, antar många programmerare den intuitiva modellen om sammanflätning. Detta kan orsaka svårhittade samtidighetsbuggar. Av den anledningen är det ofta viktigt att emulera den intuitiva modellen. Ett exempel på en sådan emulering är lineariserbarhet. För att illustrera betraktar vi den klassiska datastrukturen kö. Datastrukturen kö fungerar precis som köer i verkligheten: vill man gå till kassan ställer man sig sist i kön, och när en person betalat börjar kassören hjälpa den som står först i kön.

Vad händer när flera ställer sig i kö samtidigt? När två personer kommer från varsitt håll och ska ställa sig i kö kommer alltid den ena att hamna före den andra. Kön är en tydlig “linje”, ordningen är alltid entydig utifrån sett. I en dator är det inte lika enkelt. Eftersom bilden av minnet — själva kön — kan skilja sig mellan de olika kärnorna, kan de ha olika uppfattning om vem som kom först. Vad är sedan kassörskans bild? Vem av dem får hjälp först? Låt oss betrakta körningen i efterhand: när folk ställde sig i kö, och när de fick hjälp.

Om vi kan totalt ordna alla som ställer sig i en kö — att vi vet för varje par av personer, vem av dem som stod före den andra — så att kassören hjälpte dem en i taget i rätt ordning, kan vi kalla kön *lineariserbar*. Vad detta effektivt innebär är att vi, för en samtidig körning, har hittat en motsvarande sekventiell körning som uppfyller specifikationen för en kö.

I Artikel I utvecklar vi metoder för att verifiera lineariserbarhet hos datastrukturerna stapel (engelska: stack), kö och mängd. Det gör vi genom att analysera endast två tidpunkter för varje operation: anrop och retur.

I ett parallellt system kan man utnyttja sådana datastrukturer inte bara för att skicka runt data, utan också för att organisera själva programstrukturen. Varje kärna kan ha en kö av uppgifter att utföra. De kan skicka uppgifter till varandra, och när en kärna är klar med en uppgift kan den börja utföra nästa uppgift i sin kö. Denna modell kallas för händelsedriven (event-driven) programmering, och är en central del i till exempel Androids modell för samtidighet. I Androids modell blandas händelsedriven programmering med delat minne, vilket ökar komplexiteten ytterligare. I Artikel II utforskar vi denna modell och utvecklar lösningar för att se om en körning av ett sådant program är kompatibelt med kösemantik och *sekventiellt konsistent*: alltså att det finns en entydig bild av ordningen på händelser.

Att kärnor inte har en enad bild av minnet leder till så kallade *svaga minnesmodeller*. I sådana svaga minnesmodeller förutsätts inte att alla händelser är ordnade. Istället tillåts kärnor ha olika uppfattning om en sådan ordning. Olika minnesmodeller bevarar olika egenskaper hos dessa ordningar. Till exempel är det vanligt att man vill bevara sambandet mellan orsak och verkan. Mer specifikt vill man att om en kärna läser ett värde en annan kärna skrivit, kan den första kärnan se allt den andra kärnan gjorde innan den skrivningen. I Artikel III utvecklar vi metoder för att verifiera att ett minnessystem — till exempel distribuerat minne eller interaktionen mellan kärnors egna minnen — uppfyller tre minnesmodeller som används i programmeringsspråken C och C++.

Denna avhandling behandlar olika aspekter av samtidighet och verifiering av samtidiga program: delade datastrukturer, händelsedrivna program och svaga minnesmodeller. Resultaten ger praktiska metoder för att upptäcka buggar och att öka tillförlitligheten i samtidiga program.

11.0.1 Finansiering

Denna avhandling är delvis finansierad av Vetenskapsrådet (VR).

Bibliography

[1] P.A. Abdulla, F. Haziza, L. Holík, B. Jonsson, and A. Rezine. An integrated specification and verification technique for highly concurrent data structures. In *TACAS*, pages 324–338, 2013.

[2] Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. Stateless model checking for TSO and PSO. *Acta Inf.*, 54(8):789–818, 2017.

[3] Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Optimal dynamic partial order reduction. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 373–384. ACM, 2014. URL: http://doi.acm.org/10.1145/2535838.2535845, doi:10.1145/2535838.2535845.

[4] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Frederik Bønneland, Sarbojit Das, Bengt Jonsson, Magnus Lång, and Konstantinos Sagonas. Tailoring stateless model checking for event-driven multi-threaded programs. arXiv CoRR, July 2023. Extended Version with Proofs. doi:10.48550/arXiv.2307.15930.

[5] Parosh Aziz Abdulla, Mohamed Faouzi Atig, and Rojin Rezvan. Parameterized verification under TSO is pspace-complete. *PACMPL*, 4(POPL), 2020.

[6] Parosh Aziz Abdulla, Samuel Grahn, Bengt Jonsson, Shankaranarayanan Krishna, and Om Swostik Mishra. Efficient linearizability monitoring, 2025. URL: https://arxiv.org/abs/2509.17795, arXiv:2509.17795.

[7] Parosh Aziz Abdulla, Bengt Jonsson, and Cong Quy Trinh. Fragment abstraction for concurrent shape analysis. In *ESOP*, LNCS, 2018.

[8] Parosh Aziz Abdulla, A. Prasad Sistla, and Muralidhar Talupur. Model checking parameterized systems. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking.*, pages 685–725. Springer, 2018. doi:10.1007/978-3-319-10575-8_21.

[9] Federal Aviation Administration. Airworthiness directives; the boeing company airplanes, 2025. URL: https://www.federalregister.gov/documents/2025/11/14/2025-19891/airworthiness-directives-the-boeing-company-airplanes.

[10] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Comput.*, 9(1):37–49, 1995. doi:10.1007/BF01784241.

[11] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2):7:1–7:74, 2014.

[12] Rajeev Alur, Kenneth L. McMillan, and Doron A. Peled. Model-checking of correctness conditions for concurrent objects. *Inf. Comput.*, 160(1-2):167–188, 2000. URL: https://doi.org/10.1006/inco.1999.2847, doi:10.1006/INCO.1999.2847.

[13] Chuan-Heng Ang and Kok-Phuang Tan. The interval b-tree. *Inf. Process. Lett.*, 53(2), 1995.

[14] Anish Athalye. Porcupine: A fast linearizability checker in Go. https://github.com/anishathalye/porcupine, 2017.

[15] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing c++ concurrency. In *POPL*, pages 55–66. ACM, 2011.

[16] Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza. Verifying concurrent programs against sequential specifications. In *Proceedings of the 22nd European Conference on Programming Languages and Systems*, ESOP'13, page 290–309, Berlin, Heidelberg, 2013. Springer-Verlag. URL: https://doi-org.ezproxy.its.uu.se/10.1007/978-3-642-37036-6_17, doi:10.1007/978-3-642-37036-6_17.

[17] Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza. On reducing linearizability to state reachability. *Information and Computation*, 261:383–400, 2018. ICALP 2015. URL: https://www.sciencedirect.com/science/article/pii/S0890540118300178, doi:10.1016/j.ic.2018.02.014.

[18] Ahmed Bouajjani, Constantin Enea, Rachid Guerraoui, and Jad Hamza. On verifying causal consistency. In *Proceedings of POPL 2017, the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 626–638, 2017.

[19] Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan. Line-up: a complete and automatic linearizability checker. In Benjamin G. Zorn and Alexander Aiken, editors, *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, pages 330–340. ACM, 2010. doi:10.1145/1806596.1806634.

[20] Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. Unifying concurrent objects and distributed tasks: Interval-linearizability. *J. ACM*, 65(6), November 2018. doi:10.1145/3266457.

[21] Maria Christakis, Alkis Gotovos, and Konstantinos Sagonas. Systematic testing for detecting concurrency errors in erlang programs. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 154–163, 2013. doi:10.1109/ICST.2013.50.

[22] Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In *Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings*, page 625–635, Berlin, Heidelberg, 2021. Springer-Verlag. doi:10.1007/978-3-030-79876-5_37.

[23] Giorgio Delzanno. Constraint-based verification of parameterized cache coherence protocols. *Formal Methods Syst. Des.*, 23(3):257–301, 2003. doi:10.1023/A:1026276129010.

[24] Mike Dodds, Andreas Haas, and Christoph M. Kirsch. A scalable, correct time-stamped stack. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 233–246. ACM, 2015. doi:10.1145/2676726.2676963.

[25] Emanuele D'Osualdo, Azalea Raad, and Viktor Vafeiadis. The path to durable linearizability. *Proc. ACM Program. Lang.*, 7(POPL), January 2023. doi:10.1145/3571219.

[26] E. Allen Emerson and Vineet Kahlon. Exact and efficient verification of parameterized cache coherence protocols. In Daniel Geist and Enrico Tronci, editors, *Correct Hardware Design and Verification Methods, 12th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2003, L'Aquila, Italy, October 21-24, 2003, Proceedings*, volume 2860 of *Lecture Notes in Computer Science*, pages 247–262. Springer, 2003. doi:10.1007/978-3-540-39724-3_22.

[27] Michael Emmi and Constantin Enea. Sound, complete, and tractable linearizability monitoring for concurrent collections. *Proc. ACM Program. Lang.*, 2(POPL):25:1–25:27, 2018. doi:10.1145/3158113.

[28] Michael Emmi, Constantin Enea, and Jad Hamza. Monitoring refinement via symbolic reasoning. In David Grove and Stephen M. Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 260–269. ACM, 2015. doi:10.1145/2737924.2737983.

[29] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, November 1976. doi:10.1145/360363.360369.

[30] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. *SIGPLAN Not.*, 40(1):110–121, January 2005. doi:10.1145/1047659.1040315.

[31] Phillip B. Gibbons, John L. Bruno, and Steven Phillips. Black-Box Correctness Tests for Basic Parallel Data Structures. 35(4):391–432. doi:10.1007/s00224-002-1046-6.

[32] Phillip B. Gibbons and Ephraim Korach. Testing shared memories. *SIAM J. Comput.*, 26(4):1208–1244, 1997. doi:10.1137/S0097539794279614.

[33] P. Godefroid. Model checking for programming languages using VeriSoft. In *POPL*, pages 174–186, Paris, France, 1997. ACM Press.

[34] Patrice Godefroid, Robert S. Hanmer, and Lalita Jategaonkar Jagadeesan. Model checking without a model: an analysis of the heart-beat monitor of a telephone switch using verisoft. *SIGSOFT Softw. Eng. Notes*, 23(2):124–133, March 1998. doi:10.1145/271775.271800.

[35] Wojciech Golab, Lisa Higham, and Philipp Woelfel. Linearizable implementations do not suffice for randomized distributed computation. In *Proceedings of the Forty-Third Annual ACM Symposium on Theory of Computing*, STOC '11, page 373–382, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/1993636.1993687.

[36] Pavel Golovin, Michalis Kokologiannakis, and Viktor Vafeiadis. RELINCHE: automatically checking linearizability under relaxed memory consistency. *Proc. ACM Program. Lang.*, 9(POPL):2090–2117, 2025. doi:10.1145/3704906.

[37] Samuel Grahn. ccchecker. URL: https://github.com/grahnen/ccchecker.

[38] Samuel Grahn. edchecker. URL: https://github.com/grahnen/edchecker.

[39] Samuel Grahn. LiMo artifact. doi:10.5281/zenodo.15258056.

[40] Jad Hamza. On the complexity of linearizability. *Computing*, 101(9):1227–1240, September 2019. doi:10.1007/s00607-018-0596-7.

[41] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[42] Gerard Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 1st edition, 2011.

[43] Toshihide Ibaraki, Tiko Kameda, and Toshimi Minoura. Serializability with constraints. *ACM Trans. Database Syst.*, 12(3):429–452, September 1987. doi:10.1145/27629.214284.

[44] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In Cyril Gavoille and David Ilcinkas, editors, *Distributed Computing*, pages 313–327, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[45] Casper S. Jensen, Anders Møller, Veselin Raychev, Dimitar Dimitrov, and Martin Vechev. Stateless model checking of event-driven applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 57–73. Association for Computing Machinery. URL:

https://dl.acm.org/doi/10.1145/2814270.2814282, doi:10.1145/2814270.2814282.

[46] Qiaowen Jia, Yi Lv, Peng Wu, Bohua Zhan, Jifeng Hao, Hong Ye, and Chao Wang. Verilin: A linearizability checker for large-scale concurrent objects. In *International Symposium on Theoretical Aspects of Software Engineering*, pages 202–220. Springer, 2023.

[47] Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. Effective stateless model checking for C/C++ concurrency. *PACMPL*, 2:17:1–17:32, 2018.

[48] Michalis Kokologiannakis and Konstantinos Sagonas. Stateless model checking of the linux kernel's hierarchical read-copy-update (tree rcu). In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, SPIN 2017, page 172–181, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3092282.3092287.

[49] Michalis Kokologiannakis and Viktor Vafeiadis. Genmc: A model checker for weak memory models. In *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part I*, page 427–440, Berlin, Heidelberg, 2021. Springer-Verlag. doi:10.1007/978-3-030-81685-8_20.

[50] Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. Taming release-acquire consistency. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 649–662. ACM, 2016.

[51] Zheng Han Lee and Umang Mathur. Efficient decrease-and-conquer linearizability monitoring. *Proc. ACM Program. Lang.*, 9(OOPSLA2), October 2025. URL: https://doi-org.ezproxy.its.uu.se/10.1145/3763123, doi:10.1145/3763123.

[52] Nancy G. Leveson. Medical devices: the therac-25. 1985. URL: https://api.semanticscholar.org/CorpusID:13148715.

[53] Pallavi Maiya, Rahul Gupta, Aditya Kanade, and Rupak Majumdar. Partial order reduction for event-driven multi-threaded programs. In *Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9636*, page 680–697, Berlin, Heidelberg, 2016. Springer-Verlag. doi:10.1007/978-3-662-49674-9_44.

[54] Pallavi Maiya, Rahul Gupta, Aditya Kanade, and Rupak Majumdar. Partial Order Reduction for Event-Driven Multi-threaded Programs. In Marsha Chechik and Jean-François Raskin, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 680–697. Springer, 2016. doi:10.1007/978-3-662-49674-9_44.

[55] Pallavi Maiya, Aditya Kanade, and Rupak Majumdar. Race detection for android applications. *SIGPLAN Not.*, 49(6):316–325, June 2014. doi:10.1145/2666356.2594311.

[56] A. Mazurkiewicz. Trace theory. In *Advances in Petri Nets*, 1986.

[57] Zigurd Mednieks, Laird Dornin, G. Blake Meike, and Masumi Nakamura. *Programming Android*. "O'Reilly Media, Inc.", 2012.

[58] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 5.0*, June 2025. URL: https://www.mpi-forum.org/docs/mpi-5.0/mpi50-report.pdf.

[59] Met Office. *Iris: A powerful, format-agnostic, and community-driven Python package for analysing and visualising Earth science data*. Exeter, Devon, v3.6 edition, 2010 - 2023. URL: http://scitools.org.uk/, doi:10.5281/zenodo.7948293.

[60] Ike Mulder and Robbert Krebbers. Proof automation for linearizability in separation logic. *Proc. ACM Program. Lang.*, 7(OOPSLA1), April 2023. URL: https://doi-org.ezproxy.its.uu.se/10.1145/3586043, doi:10.1145/3586043.

[61] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 267–280, USA, 2008. USENIX Association.

[62] Gil Neiger. Set-linearizability. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '94, page 396, New York, NY, USA, 1994. Association for Computing Machinery. doi:10.1145/197917.198176.

[63] Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, October 1979. doi:10.1145/322154.322158.

[64] Matthieu Perrin, Achour Mostéfaoui, and Claude Jard. Causal consistency: beyond memory. In Rafael Asenjo and Tim Harris, editors, *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, March 12-16, 2016*, pages 26:1–26:12. ACM, 2016. doi:10.1145/2851141.2851170.

[65] Christina L. Peterson, Victor Cook, and Damian Dechev. Concurrent correctness in vector space. In Fritz Henglein, Sharon Shoham, and Yakir Vizel, editors, *Verification, Model Checking, and Abstract Interpretation - 22nd International Conference, VMCAI 2021, Copenhagen, Denmark, January 17-19, 2021, Proceedings*, volume 12597 of *Lecture Notes in Computer Science*, pages 151–173. Springer, 2021. doi:10.1007/978-3-030-67067-2_8.

[66] Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Mechanized verification of fine-grained concurrent programs. In David Grove and Stephen M. Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN*

Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015, pages 77–87. ACM, 2015. doi:10.1145/2737924.2737964.

[67] Graeme Smith, Kirsten Winter, and Robert J. Colvin. Linearizability on hardware weak memory models. *Formal Aspects of Computing*, 32(1):1–32, February 2020. doi:10.1007/s00165-019-00499-8.

[68] Hünkar Can Tunç, Parosh Aziz Abdulla, Soham Chakraborty, Shankaranarayanan Krishna, Umang Mathur, and Andreas Pavlogiannis. Optimal reads-from consistency checking for c11-style memory models. *Proc. ACM Program. Lang.*, 7(PLDI), June 2023. doi:10.1145/3591251.

[69] Martin Vechev and Eran Yahav. Deriving linearizable fine-grained concurrent objects. *SIGPLAN Not.*, 43(6):125–135, June 2008. URL: https://doi-org.ezproxy.its.uu.se/10.1145/1379022.1375598, doi:10.1145/1379022.1375598.

[70] K. Vidyasankar. Generalized theory of serializability. *Acta Inf.*, 24(1):105–109, February 1987. doi:10.1007/BF00290709.